A complementary approach for testing system robustness based on passive testing and fault injection techniques

Fayçal Bessayah

Ecole Doctorale EDITE

Thèse présentée pour l'obtention du diplôme de Docteur de Télécom & Management SudParis

Doctorat conjoint Télécom & Management SudParis et Université Pierre et Marie Curie

Spécialité :

Informatique

Par

Fayçal Bessayah

Titre

Une Approche Complémentaire de Test de Robustesse Basée sur l'Injection de Fautes et le Test Passif

Soutenue le 3 décembre 2010 devant le jury composé de :

Nina Yevtushenko	Université d'Etat de Tomsk	Rapporteur
Ismael Rodríguez Laguna	Université Complutense de Madrid	Rapporteur
Fatiha Zaidi	Université Paris-Sud XI	Examinateur
Sébastien Tixeuil	Université Pierre et Marie Curie	Examinateur
Eliane Martins	Université d'Etat de Campinas	Co-Encadrant
Ana Cavalli	IT/Telecom SudParis	Directrice de thèse

Thèse n° 2010TELE0030

Doctoral School EDITE

Thesis submitted for obtaining the

PHD DEGREE IN COMPUTER SCIENCE

Doctorate jointly delivered by

Telecom & Management SudParis and Pierre et Marie Curie University- Paris 6

Speciality:

COMPUTER SCIENCE

Presented by

Fayçal Bessayah

Title

A Complementary Approach for Testing System Robustness Based on Passive Testing and Fault Injection Techniques

Committee in charge :

Nina Yevtushenko	Reviewer	Tomsk State University
Ismael Rodríguez Laguna	Reviewer	Complutense University of Madrid
Fatiha Zaidi	Examiner	Universisty of Paris-Sud XI
Sébastien Tixeuil	Examiner	Pierre et Marie Curie University
Eliane Martins	Co-advisor	State University of Campinas
Ana Cavalli	Co-advisor	IT/Telecom SudParis

Acknowledgments

I would like to thank Professor Ana Cavalli for her excellent support and dedication during all the time I spent working on this PhD thesis. I am very grateful for the time she spent helping me and guiding my researches. I would like also to thank her for granting me the freedom of developing my ideas and for her suggestions and advices throughout this work.

A special thanks also to my co-adviser, Professor Eliane Martins from the State University of Campinas, for her advices and guidance in preparing this thesis. I really learned a lot of things by working with her and benefited from her experience as well as her excellent research and technical skills.

I would like also to thank Doctor Amel Mammar a lecturer at Telecom SudParis who helped me and encouraged me a lot. A particular thanks to her for having read my manuscript and for all the suggestions she made to improve the quality of this document.

More thanks go to my thesis evaluation committee consisting of Professor Nina Yevtushenko, Professor Ismael Rodríguez Laguna, Professor Sébastien Tixeuil, Doctor Fatiha Zaidi, Professor Eliane Martins and Professor Ana Cavalli.

Many thanks also to all my friends and colleagues, in no particular order, Bakr Sarakbi, Wissam Mallouli, Willy Jimenez, Mounir Lallali, Felipe lalanne, Mazen Al Maarabani, Anis laouiti, Anderson Morais, Farouk Aissanou, Jose Pablo Escobar, Mohamed Ahmed Mohamed Sidi and Bachar Wehbi. Thanks also to Mme. Brigitte Laurent and Mme. Jocelyne Vallet for their help and support completing the non technical part of my work.

Last and not least, I would like to thank my mother, my grandmother and all my brothers and sisters Nassim, Souad and Radia for their encouragement and support. Thanks a lot.

Résumé

Que ce soit dans le domaine des transports, des énergies ou des banques, les systèmes informatiques sont immanquablement présents. Nous confions ce que nous avons de plus cher, à savoir nos vies et nos biens, à des programmes informatiques.

Parallèlement, cela va sans dire que ces systèmes sont de plus en plus complexes. Une complexité due essentiellement à une expansion sans précédent de systèmes largement distribués et hétérogènes. Sans parler de l'utilisation d'Internet comme principal réseau de transport de données, partagé par un nombre colossal de services et d'applications Web. Face à cette complexité croissante, tout dysfonctionnement, même temporaire, de ces systèmes peut avoir de lourdes conséquences économiques, voire dans certains cas, humaines. Afin de s'assurer de la fiabilité de tels systèmes, il importe donc de vérifier leurs comportements de la manière la plus rigoureuse possible.

L'utilisation des méthodes formelles pour le test de logiciels est probablement ce qu'il y a de plus sûr en matière de techniques de vérification. Ceci s'explique sans doute par les fondements mathématiques sur lesquels se basent ces méthodes, ce qui permet de développer un raisonnement plus rigoureux et de ce fait, plus fiable.

On peut requérir aux méthodes formelles pour spécifier les propriétés importantes du système testé, mais aussi pour vérifier ces propriétés sur l'implantation finale. L'utilisation de ces méthodes a permis de développer une théorie du test de conformité dont l'objectif est de réaliser un test fonctionnel qui permet de vérifier si le produit fini correspond à la spécification de référence. La recherche académique a publié de nombreux travaux sur le test de conformité. Globalement, on peut classifier l'ensemble de ces travaux en deux grandes catégories: les méthodes de test actif et les méthodes de test passif.

Le test actif consiste à appliquer au système sous test un ensemble de tests et à comparer le comportement observé avec la spécification de référence. De nombreuses méthodes de génération automatique de tests de conformité ont été proposées dans la littérature. Elles traitent généralement des systèmes protocolaires et applicatifs réactifs en faisant l'hypothèse de pouvoir interagir avec l'implantation sous test.

Le principe étant de stimuler le système testé en émettant des entrées particulières pour le faire réagir et de collecter les sorties produites pour les comparer avec celles attendues.

Ce type de test n'est malheureusement pas toujours possible à exécuter. Dans les systèmes de protocoles en couches par exemple, il est rare qu'on puisse bénéficier d'un accès direct pour interagir avec une couche particulière du système et ainsi appliquer les séquences de test. Aussi dans certains cas, la phase de test qui monopolise complètement le système, peut être très couteuse pour les industriels. Dans ce genre de situations, le test passif s'avère particulièrement intéressant.

En effet, le test passif ne requiert pas une interaction directe avec le système testé. Il consiste à observer et à collecter les entrées et les sorties produites par l'implantation sous test, et à analyser cette séquence par rapport à la spécification de référence. On vérifie alors si le comportement de l'implantation est conforme à celui prévu par la spécification.

La réalisation d'un test de conformité suppose que le système sous test s'exécute dans des conditions environnementales normales. On estime que dans de telles conditions, le comportement du système testé doit être conforme à sa spécification fonctionnelle. Cependant, lorsqu'un système informatique est susceptible d'évoluer dans un contexte hostile où les conditions environnementales sont plus ou moins stressantes, le test de conformité n'est plus suffisant. En effet dans ce genre de situations, on doit étudier le comportement du système en tenant compte de ces contraintes contextuelles. Ceci définit un autre type de test qu'on appelle : test de robustesse. L'objectif principal du test de robustesse est d'étudier le comportement d'une implantation s'exécutant dans un environnement hostile. L'implantation testée est considérée robuste si elle continue à avoir une exécution correcte en présence de fautes [1].

Les approches de test de robustesse peuvent être empiriques ou formelles. Les approches empiriques déterminent le niveau de robustesse du système étudié, tandis que les approches formelles s'intéressent à la vérification des propriétés de robustesse [2]. Les techniques d'injection de fautes sont couramment utilisées pour

l'évaluation empirique de la robustesse d'une implantation. L'injection de fautes consiste à introduire de façon délibérée, des erreurs dans un système lors de son exécution et d'observer sa réaction. Cela permet, lors de la réalisation d'un test de robustesse, de simuler un environnent hostile. Par ailleurs, les approches de test de robustesse formelles ont pour but de déterminer formellement la robustesse d'une implantation en vérifiant la satisfiabilité d'un ensemble de propriétés de robustesse sur cette implantation. Ces dernières s'inspirent fortement des méthodes de test de conformité actives à la différence près que le domaine d'entrées est ici augmenté par l'introduction d'un ensemble d'aléas (fautes). Ainsi, au lieu de stimuler le système sous test par des entrées valides, le testeur de robustesse, génère et exécute des séquences d'entrées corrompues pour perturber le fonctionnement du système testé.

Contributions

Le test de robustesse est très important pour assurer la sécurité et la fiabilité des systèmes logiciels. Les techniques d'injection de fautes appliquées au test de robustesse ont montré des résultats très intéressants. Elles souffrent cependant de ne pas disposer d'oracles de tests performants leurs permettant d'évaluer la robustesse du système testé de manière plus rigoureuse. En effet, ces techniques ne vérifient pas formellement la robustesse d'un système. Une implantation est considérée robuste si elle peut continuer son exécution en présence de fautes. En d'autres termes, si le système testé ne se bloque pas, il est considéré comme robuste. On sait cependant, qu'un système peut très bien continuer son exécution sans pour autant fournir le comportement attendu. De ce fait, nous avons besoins de requérir à des approches plus rigoureuses pour évaluer la robustesse d'un système.

En outre, les techniques d'injection de fautes ne contrôlent pas efficacement le processus d'injection. Les fautes sont injectées de manière plus ou moins aléatoire et il n'y a aucun moyen de s'assurer de la bonne exécution des campagnes d'injections (est ce que toutes les fautes ont été injectées correctement ?).

D'autre part, les techniques formelles de test de robustesse définissent formellement toutes les étapes du test. Les fautes sont générées à partir d'un modèle formel et les propriétés de robustesse sont vérifiées sur la base d'un oracle de test bien défini.

Toutefois, deux grandes questions peuvent être soulevées au sujet de ces méthodes. Tout d'abord, l'ensemble des fautes injectées est limité par le domaine d'entrées de l'application testée. A l'opposé des approches d'injection de fautes empiriques qui peuvent injecter n'importe quel type de fautes, les techniques formelles existantes créent le modèle de fautes en se référant au modèle fonctionnel du système testé. Ceci à l'avantage de permettre une injection mieux ciblée et plus adaptée au système testé, mais les types de fautes considérées sont limitées par le modèle fonctionnel. Si ce dernier ne prend pas en compte les aspects temporels par exemple, on ne pourra pas injecter de fautes temporelles. En plus, le modèle fonctionnel d'une implantation n'est pas toujours disponible.

Enfin, les méthodes formelles existantes appliquées au test de robustesse reprennent la même architecture que celle utilisée par les méthodes actives de test de conformité. Cette architecture impose que le testeur interagisse directement avec le système testé. Par conséquent, ces méthodes ne peuvent pas être utilisées pour tester des composants systèmes qui n'offrent pas d'interfaces d'interactions directes, ou lorsque le système testé ne peut pas être monopolisé par le testeur pour une durée importante.

Le travail que nous présentons dans ce document, consiste en un ensemble de propositions qui ont pour objectif de répondre aux défis auxquels font face les approches de test de robustesse existantes. Nous contribuons sur quatre principaux axes :

En premier lieu, nous nous intéressons aux techniques d'injection de fautes et plus particulièrement au problème de contrôle du processus d'injection. Nous proposons de formaliser les fautes injectées en utilisant une extension temporelle de la logique de Hoare [42]. Notre étude étant plus portée sur les systèmes communicants, nous proposons de spécifier chaque opération d'injection par un triplet de Hoare décrivant les pré-conditions qui doivent être satisfaites par les messages de communication interceptés avant l'exécution de l'opération d'injection, ainsi qu'un ensemble de post-conditions spécifiant comment l'exécution de cette opération devrait modifier les états de ces messages. Nous utiliserons ensuite cette formalisation comme

oracle de test pour vérifier la bonne exécution du processus d'injection. Ainsi, nous proposons un algorithme de test passif qui vérifie la conformité de l'ensemble des fautes injectées (spécifiées comme un ensemble de triplets de Hoare), sur une trace d'injection. De cette manière, nous pourrons contrôler les campagnes d'injections et ainsi apporter plus de fiabilité à nos expérimentations.

Notre seconde contribution concerne la spécification et la vérification des propriétés de robustesse. Nous proposons de formaliser les propriétés de robustesse en utilisant une extension de la logique temporelle linéaire qui permet la spécification de contraintes temps réel. Il s'agit de la logique temporelle à horloge explicite, XCTL (eXplicit Clock Temporal Logic) [32], dont l'expressivité permet à la fois de spécifier des propriétés simples et complexes avec une aisance particulière.

Pour la vérification de ces propriétés, nous proposons un algorithme de test passif qui vérifie la conformité des formules XCTL sur une trace d'événements. Le choix d'une approche basée sur le test passif permet de s'affranchir des limitations du test actif, mentionnées précédemment.

Nous contribuons aussi par une nouvelle approche de test de robustesse. Nous proposons une approche hybride basée sur l'injection de fautes et le test passif. L'injection de fautes est utilisée pour créer des conditions environnementales stressantes, et le test passif permet de vérifier la satisfiabilité des propriétés de robustesse sur les traces d'exécution collectées. Les fautes injectées ainsi que les propriétés de robustesse sont formellement spécifiées. Nous utilisons la logique de Hoare pour la spécification des fautes et la logique XCTL pour la formalisation des propriétés de robustesse. Ce qui nous permet de vérifier à la fois le processus d'injection et les exigences de robustesse en appliquant les approches de test passif proposées dans nos contributions précédentes.

Finalement, nous proposons une plateforme de modélisation et de vérification de la robustesse des services Web. Les services Web sont une technologie émergente qui tend progressivement à s'imposer comme un standard du paradigme de communication programme-à-programme. Ils fournissent aussi un excellent exemple de systèmes hétérogènes fortement distribués. Les services Web peuvent être simples

ou composés et ils sont largement utilisés pour la création d'applications e-commerce et de systèmes d'information distribués. Par conséquent, ils constituent un très bon exemple de systèmes critiques où le test de robustesse prend toute sa dimension.

La plateforme de test que nous proposons ici, est en réalité une instanciation de notre approche de test de robustesse, adaptée aux services Web. Cette plateforme intègre un injecteur de fautes innovant (WSInject) que nous avons conçu et développé pour pouvoir simuler un environnement d'exécution hostile. WSInject [36] est un injecteur de fautes pour services Web capable d'injecter des fautes d'interfaces et de communications, ou même de combiner les deux types de fautes en une seule injection. Il peut être utilisé pour le test de services simples ou composés.

Nous avons aussi implanté et intégré les algorithmes de test passif proposés pour la vérification du processus d'injection et des exigences de robustesse et nous avons conduit des expérimentations sur deux cas d'études pour illustrer l'utilisation de notre plateforme de test.

Organisation du manuscrit

Le présent manuscrit de thèse est organisé comme suit :

1. Dans le second chapitre, nous présentons l'état de l'art des approches de test de conformité et de robustesse. Pour le test de conformité, nous introduisons d'abord l'utilisation des méthodes formelles pour le test des systèmes logiciels. Ensuite, nous décrivons les approches les plus importantes des deux grandes familles de test : le test actif et le test passif. La deuxième partie de ce chapitre est consacrée aux méthodes de test de robustesse. Nous classons ces méthodes en deux grandes catégories. D'abord, nous exposons les techniques empiriques basées sur l'injection de fautes et ensuite nous abordons les techniques formelles.

2. Le troisième chapitre présente notre première contribution. Il s'agit de la formalisation et la vérification de l'injection de fautes. L'idée de base est de spécifier les fautes injectées par un ensemble de triplets de Hoare, puis d'utiliser

cette spécification comme oracle de test pour vérifier la bonne exécution du processus d'injection. Nous définissons pour cela un algorithme de test passif qui vérifie la satisfiabilité des spécifications de fautes sur une trace d'injection. Nous présentons aussi quelques exemples de spécification pour illustrer notre approche.

3. Dans le quatrième chapitre, nous présentons notre approche de test de contraintes temps réel. Nous discutons en premier, les travaux existants qui traitent des méthodes formelles pour le test de propriétés temps réel. Ensuite, nous présentons les formalismes permettant de spécifier ce type de propriétés et justifions notre choix de XCTL [32]. Nous présentons aussi notre algorithme de test passif pour la vérification de formules XCTL sur des traces d'exécutions et discutons les résultats obtenus au terme d'une évaluation expérimentale de l'algorithme.

4. Dans le chapitre cinq, nous décrivons notre approche de test de robustesse. Il s'agit d'une approche complémentaire, basée sur l'injection de fautes et le test passif. Nous étudions d'abord les travaux existants sur le test de robustesse. Ensuite, nous présentons l'architecture générale de notre approche et détaillons chacune de ses composantes. Nous utilisons dans cette approche, la logique de Hoare pour la spécification et la validation des campagnes d'injection et la logique temporelle à horloge explicite (XCTL) pour le test des propriétés de robustesse.

5. Finalement, dans le chapitre six, nous présentons notre plateforme de test de robustesse pour les services Web. Cette plateforme est une instanciation de notre approche de test appliquée aux services Web. Nous décrivons son architecture générale et chacun de ses composants, plus particulièrement l'injecteur de fautes WSInject. Pour ce dernier, nous motivons notre choix de développer un injecteur de fautes pour les services Web et présentons son architecture et ses fonctionnalités.

   Nous présentons aussi dans ce chapitre, l'application de notre plateforme de

test sur deux cas d'études et montrons comment cela a permis de détecter certains modes de défaillances que nous n'aurions pas pu déceler avec les méthodes de test traditionnelles.

6. Le dernier chapitre conclut notre travail. Nous rappelons nos principales contributions, que ce soit dans le domaine du test de conformité, de l'injection de fautes ou du test de robustesse ; et nous présentons quelques perspectives potentielles qui vont dans la continuité de notre travail.

Abstract

Robustness is a specialized dependability attribute, characterizing a system reaction with respect to external faults. Accordingly, robustness testing involves testing a system in the presence of faults or stressful environmental conditions to study its behavior when facing abnormal conditions.

Testing system robustness can be done either empirically or formally. Fault injection techniques are very suitable for assessing the robustness degree of the tested system. They do not rely however, on formal test oracles for validating their test. On the other hand, existing formal approaches for robustness testing formalize both the fault generation and the result analysis process. They have however some limitations regarding the type of the handled faults as well as the kind of systems on which they can be applied.

The work presented in this thesis manuscript aims at addressing some of the issues of the existing robustness testing methods. First, we propose a formal approach for the specification and the verification of the fault injection process. This approach consists in formalizing the injected faults as a set of Hoare triples and then, verifying the good execution of the injection campaigns, based on a passive testing algorithm that checks the fault specification against a collected injection trace.

Our second contribution focuses on providing a test oracle for verifying real time constraints. We propose a passive testing algorithm to check real time requirements, specified as a set of XCTL (eXplicit Clock Temporal Logic) formulas, on collected execution traces.

Then, we propose a new robustness testing approach. It is a complementary approach that combines fault injection and passive testing for testing system robustness. The injected faults are specified as a set of Hoare triples and verified against the injection trace to validate the injection process. The robustness requirements are formalized as a set of XCTL formulas and are verified on collected execution traces. This approach allows one to inject a wide range of faults and can be used to test both simple and distributed systems.

Finally, we propose an instantiation of our robustness testing approach for Web

services. We chose Web services technology because it supports widely distributed and heterogeneous systems. It is therefore, a very good application example to show the efficiency of our approach.

Keywords: Robustness Testing, Formal Specification, Fault Injection, Passive Testing, Trace Analysis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Context

Nowadays, software systems are everywhere : transportation, health, banking, energy, etc. We are actually entrusting our lives and our goods to programs and machines. On the other hand, the increasing complexity of those systems as well as their widely distributed architectures make them more difficult to control and/or to manage. Moreover, the introduction of modular and reusable components in communication systems creates new challenges. It is possible now and relatively easy, to build complex distributed systems based on a set of several heterogeneous components (as Web services for example). It is however, more painful to have a complete control on those systems. Sometimes, developers do not even know where some of their system components are hosted not to mention the environment conditions where they are running in.

Parallel to this, every single bug or failure that can be raised in such systems, may lead to serious financial or even human damages. Therefore, the testing of software systems during and after the development process is essential and must be undertaken with the greatest possible care. This testing step aims at guarantying the correctness of a system behavior and at ensuring its reliability and its conformance with respect to the expectations made by its developers.

Probably, the most rigorous approach for performing testing activities is to rely

on formal methods. Formal methods allow one to reason about system correctness based on mathematical foundations. They can be used to formalize the system requirements (as expected by its administrators), as well as to verify their correct implementation in the final product. The use of formal methods in testing permitted the emergence of a testing theory called: conformance testing. The goal of conformance testing, is to ensure that a given implementation verifies its expected functional requirements. The literature of the testing community has produced a huge number of contributions dealing with this theory. Basically, we can classify the set of existing conformance testing approaches into two main categories: *active* and *passive* testing approaches. This classification is due to the way the test process is performed. In active testing, the tester interacts directly with the tested system to issue a verdict about the conformance of the system behavior with respect to the specified requirements. In passive testing however, the tester does not communicate directly with the tested system. Instead, an execution trace is collected during the system execution and then, the passive tester checks on this trace the conformance of the specified requirements. Usually, we rely on passive testing when the tested implementation does not provide any interface to interact with the tester or when we are testing a system component that we could not access directly.

In conformance testing, we assume that the tested system is running in its normal environmental conditions. We expect that in such situations, the functional requirements should be verified. However, when a given system or one of its components is likely to run in a hostile environment or stressful environmental conditions, conformance testing is no more sufficient to validate its behavior. In such situations, we need also to check the behavior of the tested system when facing abnormal environmental contexts. This kind of test is known as *robustness testing*. The goal here is to study the system behavior when running in a hostile environment. The system is considered as robust if it continues to have a correct execution in disturbed conditions [1].

Robustness testing approaches can be either empirical or formal. Empirical approaches usually aim at evaluating the degree of robustness of a given system; while

formal approaches focus on the verification of robustness properties [2]. For empirical evaluation, fault injection techniques are very commonly used. Fault injection consists in introducing deliberate errors in a system and observe its reaction. This technique is used in robustness testing to create stressful environmental conditions. Then, we observe if the tested system is robust enough to keep running. Formal robustness verification techniques however, aim at formally assessing the robustness of a system by checking the satisfiability of a set of robustness requirements on this system. These techniques usually inspire from conformance testing approaches, particularly from active testing. The main difference with respect to active testing, is the fault dimension of the input domain. Instead of stimulating the tested system with the valid inputs, robustness methods generate and execute invalid entries to disturb the system behavior.

1.2 Contributions

Robustness testing is very important to ensure the safety and the reliability of software systems. Most existing approaches however, still present some limitations regarding their consistency and their capabilities. Fault injection techniques applied for robustness testing have shown interesting results, yet they are suffering from a lack of soundness, mainly because they rely exclusively on empirical analysis. In this kind of approaches, we do not specify formally the robustness requirements that the tested system must guarantee. A system is considered robust simply if it continues its execution in presence of faults. In other words, if the tested system does not hang or crash, it is considered as robust. We know however, that a system may well continue its execution without providing the expected behavior. Therefore, we need a more rigorous way to check the robustness of the tested implementations. Also, fault injection techniques do not control efficiently the injection process. Faults are injected in a more or less random manner and we have no feedback about the good execution of the injection campaigns (did all faults have been injected correctly or not?).

On the other hand, formal robustness testing techniques define formally all the

testing steps. Faults are generated from a formal model and the robustness requirements are verified based on a formal test oracle. As far as we know, all the existing formal approaches for testing system robustness follow the active testing architecture [46, 40, 37]. Two main issues can be raised regarding this kind of methods. First, the set of injected faults is limited by the set of the input domain. At the opposite to fault injection approaches which can inject any kind of faults, existing formal techniques are constrained by the behavioral model of the tested system which they use to generate the set of faults to inject (usually, a set of invalid inputs). Thus, if the behavioral model does not support time specification for example, there will be no temporal faults! Also, formal active testing techniques for robustness verification present some limitations when applied on composed systems. These techniques require direct interactions with the tested system components whereases, it is not always possible to have a direct access to those components. It is therefore, difficult to inject faults or to disturb communication between the different modules of a composed application.

The work we present in this PhD thesis, is a set of propositions which aim at solving the main issues facing the existing robustness testing techniques. Our contributions are then spread over four main axes:

First, we are interested in fault injection techniques because they can improve the faults detection power of the testing methods. To address the problem of soundness in fault injection, we propose a formal approach to specify and verify the injection process. We propose to formalize the fault injection using a timed extension of Hoare logic [42]. We focus here on fault injection for communication systems. Therefore, each injection operation is specified as a Hoare triple describing a set of preconditions that must be satisfied by the intercepted communication messages before the injection and a set of postconditions which specify how the executed injection operations should modify the state of those messages. This formalization is then used as a test oracle. We propose a passive testing algorithm to verify the good execution of the injection process by checking the specification of the injected faults (given as a set of Hoare triples) against the injection trace, collected during experimentations.

This way, one can control the injection process by verifying whether the injection experiments were well performed or not.

Our second contribution concerns the specification and the verification of robustness requirements. We believe that robustness requirement could be different from the functional ones. Therefore, instead of relying on a functional model, we propose to model system robustness as a set of real-time safety and liveness properties. We believe also that some requirements can be rather complex. Thus, we propose to specify those requirements using a real-time extension of linear temporal logic, called XCTL (eXplicit Clock Temporal Logic) [32], which can handle both simple and complex properties. For the verification, we propose a passive testing algorithm to check XCTL properties on execution traces, and we study its efficiency.

We also contribute by a new robustness testing approach. We propose an hybrid approach for testing system robustness, combining fault injection and passive testing techniques. Fault injection is used to simulate the stressful environmental conditions. Then, we use a passive testing technique to check the satisfiability of the robustness requirements against the collected execution traces. The injected faults as well as the robustness properties are formally specified. We use Hoare triples for fault specification and XCTL for robustness requirements. The specification of the injected faults is then used to validate the injection process and the specification of robustness requirements allows to assess formally the system robustness.

Finally, we propose a robustness testing framework for modeling and verifying Web services robustness. Web services are an emerging technology which tends progressively to become a standard for program-to-program communication paradigm. They are also a very good example of widely distributed systems. Web services can be either simple or very complex, integrating heterogeneous service components. They are widely used for building business process and distributed information systems. Therefore, they provide a very interesting illustration of critical distributed applications. The framework we propose is actually an instantiation of our robustness testing approach for Web services. It integrates an innovative Web services fault injector (WSInject [36]) which we developed to simulate hostile environments.

We also implemented the proposed passive testing algorithms to verify both the injection process and the robustness requirements, and we tested our framework on two case studies to show its capabilities.

1.3 Thesis plan

This thesis manuscript is organized as follows:

1. In the second chapter, we present the state of the art of both conformance and robustness testing techniques. For conformance testing, we first introduce the use of formal methods for system testing. Then, we describe the most relevant existing approaches for both active and passive testing. The second part of this chapter presents robustness testing. We classify robustness testing approaches into two main categories. First, we expose those which rely on fault injection techniques and then, we present the formal robustness testing methods.

2. The third chapter presents our first contribution. It describes our formal approach for the specification and the verification of fault injection process. The basic idea consists in formalizing the injected faults as a set of Hoare triples and then, to use this specification to verify the good execution of the injection experiment. This verification is based on a proposed passive testing algorithm which checks the specified injection operations on a collected injection trace. A set of examples of injection rules is also presented as matter of illustration.

3. In the fourth chapter, we present our passive testing approach for checking real-time constraints. We first discuss the related work tackling with formal approaches for testing temporal properties. Then, we present the existing real-time formalisms and justify our choice of the XCTL [32] language. We also present our passive testing algorithm for checking XCTL properties on execution traces and discuss the obtained results of an experimental evaluation of the proposed algorithm. This evaluation consisted in calculating the necessary execution time for checking a set of real-time patterns on traces of different lengths.

4. In the chapter five, we describe our robustness testing approach. It is a complementary approach based on fault injection and passive testing techniques. We first discuss the related work and the existing robustness testing approaches. Then, we present the general architecture of our approach and detail each step of the testing process. In this approach, Hoare logic is used to specify the injected faults; while the robustness requirements are specified as a set of safety and liveness properties formalized as XCTL formulas.

5. Finally in the sixth chapter, we present our framework for testing Web services robustness. This framework is an instantiation of the proposed robustness testing approach, for Web services. We first introduce Web services technology and its main features. Then, we present the framework architecture and describe each of its components. This chapter also presents WSInject which is a fault injection tool for Web services. We motivate our choice of developing such tool and describe its architecture and its capabilities. We show in this chapter also, how the abstract concepts presented in the previous chapters are instantiated for Web services (specification of the injection process and the robustness requirements) and we carry out two case studies to illustrate the use of our framework. We describe for each case study all the testing phases and discuss the obtained results. We show particularly how our framework was able to detect important failures that could not be revealed by traditional testing methods.

6. The last chapter of this manuscript concludes our work. We summarize our contributions in the fields of both conformance and robustness testing, and present some perspectives and possible future directions to extend our work.

Chapter 2

State of the Art

Contents

2.1 Formal Testing

The use of formal methods for software testing is motivated by the fact that, performing mathematical analysis can contribute efficiently to the reliability and the consistency of any testing approach. The main advantage of using formal languages is to be able to automate the verification process of any software system based on dedicated tools.

We can rely on formal methods at different system development phases, as follows:

- The system behavior (i.e. what the system is supposed to do) can be modeled using a formal system. This model, called also *system specification*, is in fact

a mathematical representation of the studied system.

- The *Verification* step consists to check that the system specification does not contain any errors. For example, we can check that some specific system properties are correctly represented by the formal model.

- In the *Implementation* phase, the system becomes real. In this step, we do not rely on any abstract model. The system developers are in charge of coding the system behavior using the most suitable programming language.

- *Testing* is usually the last step in the development process. It consists to check whether the implemented system is conform to its formal specification.

We can classify the set of existing formal testing methods into two main categories: the active testing methods and the passive testing methods. Each category contains various approaches and each approach can use different techniques. In the following we present the basic concepts of each testing family and introduce the most known approaches from each class.

2.1.1 Active testing

Active testing consists at executing a set of test scenarios on an Implementation Under Test (IUT) and check whether its behavior is conform to the specified requirements. In this kind of test, the tester interacts directly with the IUT via its external interfaces. Its provides the IUT with a set of inputs (test cases) and collects the returned outputs which it analyzes to issue a verdict about the conformance of the IUT with respects to its requirements.

Conformance testing

Conformance testing aims at verifying whether the behavior of a given system corresponds to its specification. This kind of test can be performed following either a *black-box*, a *white-box* or a *gray-box* strategy.

- Black-box testing, also called functional testing, consists at observing the exchanged inputs and outputs between the tester and the IUT without considering the internal actions. The verdict is issued based on the analysis of the observed events.

- White-box or *structural* testing considers the test of the implementation code. Here, we do not observe only the exchanged messages but also internal actions, data structures, loops, etc. There exist specific tools for this kind of test which are able to generate and execute test cases accounting the implementation structure.

- Gray-box testing corresponds to an intermediate approach between the black and the white box techniques. The idea here, is to consider some internal actions and other implementation features while observing the exchanged messages, without necessarily having access to all implementation code details.

A typical active testing approach proceeds in two steps. First, an automatic generation of a set of test cases from the system specification is performed. Then, the tester runs these test cases on the IUT and deduces a conformance verdict based on the analysis of the system reaction to the stimulation (test inputs). Figure 2.1 describes the general active testing architecture.

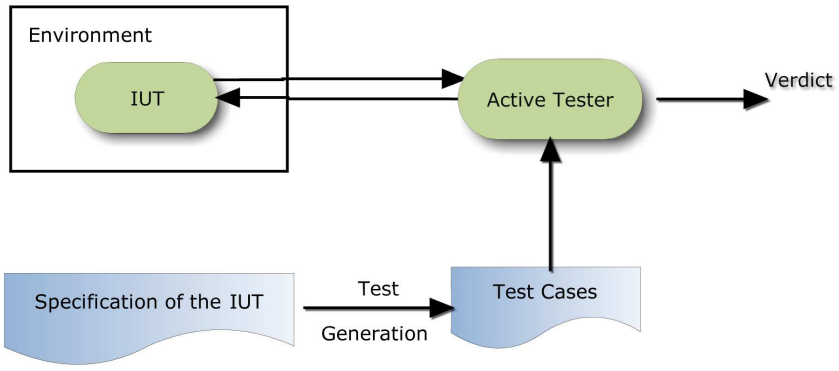

Figure 2.1: Active Testing Methodology

The standard `ISO/IEC 9646` [3] suggests some useful definitions for different conformance testing concepts. Thus, the issued verdict can be either *Pass*, *Fail* or

Inconclusive. The verdict *Pass* is returned when the IUT outputs are the same as the specified ones. In this case, we say that the IUT is conform to its specification as regard to the applied test cases. However, if the IUT outputs are different from the specified ones, the issued verdict would be *Fail* which means that the IUT is not conform to its specification. In the case where the execution of a test sequence does not lead to a *Pass* or a *Fail* verdict. The tester deduces an *Inconclusive* verdict. This verdict does not reveal an IUT failure, rather the execution of the test cases do not allow the verification of the test purpose. This could be due, for example, to a non-deterministic specification where a single input can lead to different paths. We need in this case to rerun the test cases for a better analysis.

This same standard [3] also introduces a set of terms to describe the tests applied on an IUT. A *test case* is defined as an elementary test. For a reactive system, a test case describes a set of interactions between the tester and the IUT which leads to a validation of a particular property of the tested system. This property is called a *test purpose* and is usually extracted from the system specification.

A test case is generally composed of a *preamble*, a *test body*, an *identification sequence* and a *postamble*.

The *preamble* is the initial part of a test case. It is a set of interaction sequences used to bring the implementation in a particular state where the test body can be executed. The *test body* is the part of the test case used to verify the *test purpose*.

The *identification sequence* is an interaction sequence which allows the tester to identify the state in which the IUT is, after the application of the test body. The *postamble* is used to bring the IUT to a well identified state (usually the initial state) to be able to apply another test case. Finally, we define a *test suite* as a set of test cases.

Overview of active testing approaches

A wide set of active testing techniques use Finite State Machines (FSMs) as a reference specification for modeling the behavior of the tested system. A finite state machine is a behavioral model with a finite number of states, transitions between

those states and actions. It is formally defined as follows:

Definition 2.1 *A finite state machine is a 6-tuple* $< S, I, O, \sigma, \lambda, s_0 >$ *where:*

- S *is a finite set of states, where* $s_0 \in S$ *is the initial state;*
- I *is a finite set of input events;*
- O *is a finite state of output events;*
- $\sigma : S \times I \rightarrow S$ *is the state transition function. We can extend* σ *to* $\sigma^* : S \times I^* \rightarrow S$ *where* I^* *is the set of all finite input sequences including the empty sequence* ε*;*
- $\lambda : S \times I \rightarrow O$*. We can extend* λ *to* $\lambda^* : S \times I^* \rightarrow O^*$ *where* I^* *is the set of all finite input sequences including the empty sequence* ε *and* O^* *is the set of all finite output sequences including the empty sequence* ε*;*

FSM-based testing methods suppose that we have a complete specification model `Spec` and that we can observe all inputs/outputs (I/O) of the implementation machine `Imp`. The specification machine must be minimal, complete and strongly connected. Since the implementation is tested as black-box, the strongest conformance relation that can be considered is the trace-equivalence.

Definition 2.2 *Two FSMs are trace-equivalents if they cannot be told apart by any input sequence. That is, both the specification and the implementation will generate the same outputs (a trace) for all specified input sequences.*

To check whether two machines are equivalents, one needs to show that:

- There is a set of implementation states that are isomorphic to the states of the specification.
- Every transition in the specification has a corresponding isomorphic transition in the implementation.

To check for isomorphic states, one needs to characterize each state of the machine. Thus, the main difference between the various FSM-based active testing approaches lies in the way they characterize the machine states. [30] discusses the most relevant FSM-based techniques. We can for example characterize machine states using transition tours [51], distinguishing sequences [38], characteristic sequences [28] or unique I/O sequences [64]. The algorithms proposed for these methods are all polynomial in time and memory consumption.

There is also another class of active testing approaches which do not rely on equivalence relation between the specification and the implementation. This kind of approaches consider that a system *Imp* can implement a system *Spec* while the two systems are not necessarily equivalents. For example, it is commonly acceptable that a system implementation would be more deterministic than its specification. In fact, in this case, the abstract specification does not represent all implementation details.

Therefore, in this kind of approaches, we need first to define a formal conformance relation between the implementation and its specification. Then, the tester would be able to check the conformance of an implementation with respect to its specification, based on this conformance relation.

E. Brinskma defines in [27] a conformance relation *conf* based on *Labeled Transition System* (LTS) which can check whether an implementation contains non-expected locks. This conformance relation does not distinguish between system events which are controllable by the environment (the inputs) and those which can be only observed (the outputs). The difference is however very important in practice as the tester needs to choose a set of inputs to stimulate the `IUT` so that it can observe the system outputs. Therefore, more expressiveness models were proposed to be able to reason about inputs and outputs such as *Input Output State Machine* `IOSM` in [54] and *Input Output Transition Systems* `IOTS` in [67]. In this kind of models, transitions represent either an input, an output or an internal action.

In [67], the behavior of the specification and the implementation is formalized as `IOTS`. The authors defined a conformance relation *ioco* which consider specification

traces as well as locks. An implementation `Imp` is conform to its specification *Spec* for *ioco* if after every trace σ from *Spec*, the set of outputs of *Imp* (including locks) is included in the set of outputs of *Spec*. The author considers three kind of locks: the *deadlock*, the *outputlock* and the *livelock*. The *deadlock* occurs when the tested system cannot progress; the *outputlock* occurs when the system is blocked while it is waiting for an input from its environment, and the *livelock* occurs when the system loops for an infinite sequence of internal actions.

A work inspired from [27] was proposed in [54] and uses a specification formalism based on `IOSM`. The author defines five implementation relations denoted by R_i as follows:

- The relation R_1 guarantees that all implementation outputs are expected by the specification. However, it accepts that the implementation does not response even if the specification expect an output.
- The relation R_2 refines the relation R_1 by considering lock situations.
- The relation R_3 is based on the inclusion of specification traces into implementation traces.
- The relation R_4 consider that the tested system must implement at least all the behavior expected by its specification. The tested system can however present more complex functionalities.
- The relation R_5 requires that the implementation behaves exactly as it is expected by its specification. R_5 is in fact a trace equivalence relation.

2.1.2 Passive testing

Passive testing (also called monitoring) consists at observing input and output events of a running application without disturbing its execution. The recorded observation is called an event trace. It will be analyzed by the passive tester according to the system specification to determine the conformance relation between the application and its specification. It is important to note here, that when an event trace is

conform to the specification, it does not mean that the whole application is conform to the specification. However, in the case where the trace does not conform to the specification, we can affirm that the application does not conform also.

Unlike active testing, passive testing does not influence the system under test. This has the huge advantage of not troubling the application execution. Thus, we can test a system running in its natural environmental condition. Also, passive testing can be run during all system life time in the opposite of active testing test campaigns which must be run for a specific system development phases.

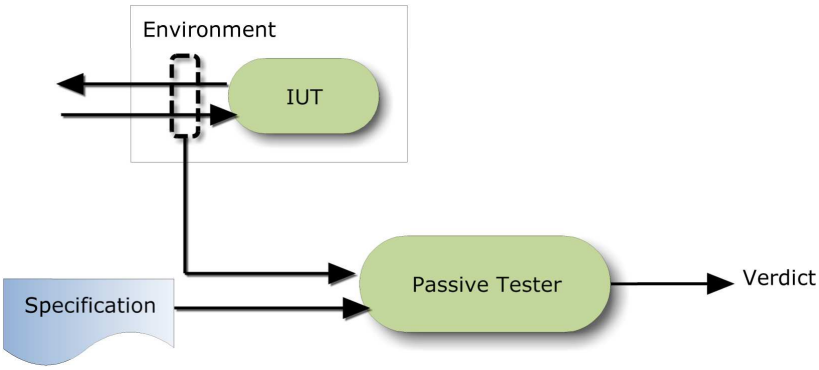

Figure 2.2: Passive Testing Methodology

Figure 2.2 describes the passive testing methodology. The trace analysis produces either a *PASS*, a *FAIL* or an *INCONCLUSIVE* verdict. A *PASS* verdict is issued if the trace is conform to the system specification (or properties) otherwise, a *FAIL* verdict is produced. In the case where the trace is not long enough to allow a complete analysis, the tester provides an *INCONCLUSIVE* verdict.

Several passive testing approaches were developed for different testing purposes. We present in the following the main important ones.

Passive testing by value determination

The *Extended Finite State Machine* (EFSM) model is an evolution of the classical FSM model which offers more specification possibilities. It is formally defined as follows:

Definition 2.3 *An Extended Finite State Machine M is a 7-tuple $M = (S, s_0, S_f, I, O, \vec{x}, T)$ where:*

- *S is a finite non empty set of states;*
- *s_0 is the initial state;*
- *S_f is a finite state of final states;*
- *I is a finite set of input symbols, with or without parameters;*
- *O is a finite set of output symbols, with or without parameters;*
- *$\vec{x} = (x_1, ..., x_k)$ is a vector denoting a finite set of variables;*
- *T is a finite set of transitions.*

Each transition t is defined as a 6-tuple $t = (s_t, f_t, i_t, o_t, P_t, A_t)$ where:

- *s_t is a starting state;*
- *f_t is an ending state;*
- *i_t is an input symbol;*
- *o_t is an output symbol;*
- *$P_t(\vec{x})$ is a predicate on the variables (boolean formula);*
- *$A_t(\vec{x})$ is a sequence of actions.*

Thus, each transition of the EFSM can contain:

- input and output events eventually with parameters,
- a predicate (or a guard) to satisfy,
- a sequence of actions to perform.

Using EFSM, passive testing approaches must not only check the correctness of event sequences (appearing in the collected trace), but also the variables and the parameter values. This first passive testing method is based on the deduction of variable and parameter values from an event trace considering an EFSM model. The schema in figure 2.3 shows an example of this deduction process.

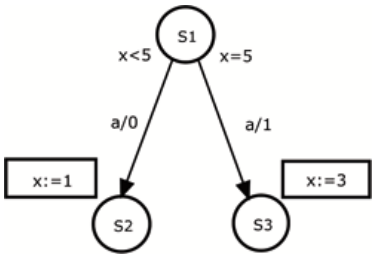

Figure 2.3: Deduction of variable values

Assume that we know the current state S_1 but not the value of variable x. If the next input/output couple from the trace is $a/1$ then, we can deduce that after the transition is fired, the current state becomes the state S_3 and x will be equal to 0. Based on this property, a passive testing algorithm was proposed in [66]. It considers that a transition is fired if :

1. the input/output couple of the trace matches with the input/output couple of the transition,

2. either the transition predicate is true or it cannot be evaluated due to a lack of information (values are not yet known).

The problem of information loss Consider the example presented in figure 2.4 If we assume that the current state is S_1 and that variable x has been identified with the value 3. If we consider that y is unknown, we must for any case fire the two transitions $S_1 \rightarrow S_2$ and $S_1 \rightarrow S_3$ because the I/O on both transitions are identical. Now that the two transitions give different values of x; x becomes UNDEFINED! We note here that undefined variables (y in this example) can lead to losing already found values of other variables (x in this example).

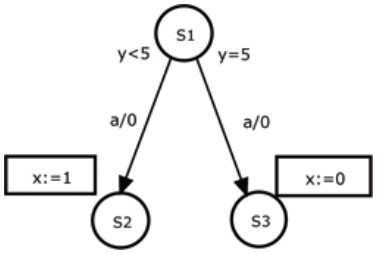

Figure 2.4: Information loss

The testing algorithm The testing algorithm proceeds in two main steps. The first step is called **homing phase** of the current state and the variable values. In this step, the following rules are considered:

- for a given I/O couple, if there exists a set of possible transitions producing different values for a same variable, then this variable becomes UNDEFINED,
- the predicates involving the UNDEFINED variables are considered to be true.

The second step is called **fault detection phase** and concerns the conformance checking of the remaining trace with respect to the specification.

Passive testing by interval determination

We saw that the algorithm presented previously suffers from an information loss phenomenon. A more efficient passive testing algorithm was propose in [29]. It is based on three main concepts.

1. **Intervals** to refer to the set of variable values such as $R(v) = [a; b]$ for variable v.
2. **Assertions** which are defined as predicates on variables denoted by $asrt(\vec{x})$ where $\vec{x}$ is the variable vector.
3. **Candidate Configuration Sets (CCS)** to formalize the analyzed environment of the system under test. A CCS is a triplet $(s, R(\vec{x}), asrt(\vec{x}))$ where s is the current state of the specification.

This algorithm aims to determine the values of variables by using a set (in the form of interval) of possible values for each variable. Intervals in which variables take their values are then, progressively refined.

The intervals The intervals are a beginning answer to the information loss problem. In the previous algorithm, a variable could not have more than one possible value. In the case where several values were possible, the variable becomes UNDEFINED. Using intervals, a variable v whose value is between two integers a and b will be defined by an interval $R(v)$ such as $R(v) = [a; b]$. If v has a constant value a, we will have $R(v) = [a; a]$. The variable v is then said **decided**. Three operation on intervals are possible:

- The sum of two intervals: $[a; b] + [c; d] = [a + c; b + d]$
- The subtraction of two intervals:$[a; b] - [c; d] = [a - c; b - d]$
- The multiplication of an interval by an integer:
  $w \times [a; b] = [w \times a; w \times b]$ if $w \geq 0$
  $w \times [a; b] = [w \times b; w \times a]$ if $w < 0$

The assertions An assertion $asrt(\vec{x})$ is a boolean formula on the variables vector $\vec{x}$ which must be true at the current state of the verification. Assertions are used to record constraints on variables, built based on transition predicates and actions. When a transition is fired, its predicate is added to the assertion as well as the actions that contain undecided variables in the right member of the equality. For example, if the action $x_2 \leftarrow x_1 + 1$ updates the variable x_2; every term of $asrt(\vec{x})$ containing x_2 must be deleted and the term $x_2 \leftarrow x_1 + 1$ must be added to $asrt(\vec{x})$. Thus, as soon as we discover x_2 we can deduce easily the value of x_2.

The Candidate Configuration Sets A Candidate Configuration Set (CCS) is a triplet $(s, R(\vec{x}), asrt(\vec{x}))$ where:

- s is the current specification state,

- $R(\vec{x})$ is the set of intervals,
- $asrt(\vec{x})$ is an assertion on the vector of variables $\vec{x}$.

Candidate configurations are used to model the states where the system under test is. They specify for each state, the related set of variable constraints. For example, the configuration $(S_1, R(x) = [2;6], (x < 4) \wedge (x > 4))$ means that the system is in the state S_1 and that the value of the variable x is contained between 2 and 6 but not equal to 4.

The algorithm defines two lists $Q1$ and $Q2$, where $Q1$ is the set of current possible CCS and $Q2$ is the set of possible CCS of the previous step. Thus, given $Q1$ and an event e, we should be able to obtain the corresponding transition. A transition t will be fired if it exists a configuration in $Q1$ whose constraints (the intervals of variables and the assertions) are compatible with the predicate p of t.

Passive testing by backward checking

This technique has been proposed in [26]. The presented algorithm is widely inspired from the one presented in [29]. However, in this work, the trace is checked backwardly. The authors built their algorithm based on the fact that the end of the trace corresponds to a system state. Therefore, starting from the end of the trace, it is more efficient and easier to get correct information about variable values by looking to the past of the trace.

This backward checking algorithm proceeds in two phases. The first step consists in tracking a trace ω starting from its end and going back to its beginning while mapping ω to the specification machine. The goal is to reach all possible configurations X that can generate the trace ω. In other words, the algorithm looks for all CCS from which ω could begin.

In the second phase, the algorithm verifies the past of the trace in order to validate at least one configuration from the set X. This validation consists in exploring all possible paths from a given configuration to verify that ω is reachable from the initial configuration of the specification. The algorithm looks for a path p that con-

nects a configuration c and an element of X. p validates the trace ω if there exists a set of predicates and actions that can confirm the correction of the element of X.

The complexity of this approach is at worst equal to the total parsing of the system specification i.e. the complete exploration of its accessibility graph.

Passive testing by invariant checking

All passive testing techniques discussed previously are based on the same concept which consists to compare a collected execution trace with the formal specification of the system under test. The major problem with this kind of approaches lies on the high complexity of the used algorithm, particularly when considering non-deterministic specification. The verification of each trace necessitates a partial (or a total) exploration of the whole specification.

To address this problem, an invariant-based approach was proposed in [45] and improved in [31]. The basic idea of invariant-based testing consists in extracting from the system specification a set of properties to verify on the trace. These properties must be satisfied at any moment, hence the name of invariants.

An input/output invariant is composed of two parts:

- The *test*, which is an input or an output symbol.
- The *preamble*, which is the sequence that must be found in the trace before checking the *test*.

Based on this definition, three types of invariants are introduced.

- Output invariants; defined when the test is an output symbol. These invariants are used to specify properties of the form : "immediately after the sequence preamble we must always have the output test". For example, consider the following output invariants:
  - $(\underbrace{i_1}_{preamble} / \underbrace{o_1}_{test})$ meaning that "i_1 is always followed by o_1".

  - $\underbrace{(i_1/o_1)(i_2}_{preamble}\underbrace{/o_2)}_{test}$ meaning that "immediately after the sequence (i_1/o_1) and the input i_2, we must have the output o_2". This invariant is said to be an invariant of length 2 because its preamble contains two I/O couples.

- Input invariants; defined when the test is an input symbol. This kind of invariants is used to specify properties of the form "immediately before the sequence preamble we must always have the input test". For example, consider the following input invariants.

  - $(\underbrace{i_1}_{test}/\underbrace{o_1}_{preamble})$ meaning that "o_1 is always preceded by i_1".
  - $\underbrace{(i_1}_{test}\underbrace{/o_1)(i_2/o_2)}_{preamble}$ meaning that "immediately before the sequence $o_1(i_2/o_2)$ we must have the input i_1".

- Succession invariants; used to specify complex properties such as loop problems. For example, the following set of invariants constitutes a succession invariant.

  - $\underbrace{(i_1/o_1)(i_2}_{preamble}\underbrace{/o_2)}_{test}$
  - $\underbrace{(i_1/o_1)(i_2/o_2)(i_2}_{preamble}\underbrace{/o_2)}_{test}$
  - $\underbrace{(i_1/o_1)(i_2/o_2)(i_2/o_2)(i_2}_{preamble}\underbrace{/o_3)}_{test}$

  This invariant forces the transition (i_2/o_2) to hold twice before the transition (i_2/o_3) must be fired. This kind of sequences is used to limit the number of attempts for a given protocol operation before returning a failure. In this example, the number of attempts is limited to two and the output o_3 can represent a failure event.

The invariant-based approach is a powerful passive testing technique though the extraction of the invariants from the system specification is still a hard task to perform. If we delegate this task to a human it is likely to take a big amount of time

and can lead to erroneous extractions. On the other hand, automatic extraction algorithms such as the one presented in [31] are very sensitive to the non-determinism of the specification when the invariant length is greater than one. Also, this approach cannot detect all types of errors and it is more likely designed to be used complementarity with other methods.

2.2 Robustness Testing: Techniques and Tools

Robustness testing aims to determine whether a software system or a component can have an acceptable behavior in the presence of faults or stressful environmental conditions. This definition covers a large spectrum of approaches, which can be classified according to two viewpoints.

The first viewpoint determines the input domain of interest. The input domain can be split into two main dimensions: the activity (workload) and the faults (faultload). The workload and the faultload can be given more or less emphasis, depending on the approaches. Workload-based approaches extend usual testing efforts by submitting the system to higher load tests while Faultload-based approaches focus on the fault dimension and the behavior of the system subjected to a given set of faults.

The two dimensions of the input domain can combine their effects on a system. The so-called mixed workload- and faultload-based approaches, explicitly consider such combined effects.

The second viewpoint concerns the classification of robustness testing approaches according to the target objective: testing for verification or evaluation purposes.

The verification of robustness is most often on the lineage of classical testing approaches, where a model of the system (e.g., a behavioral model) is used as a guide for selecting test cases (e.g., transition coverage is required). The evaluation of robustness rather builds on fault injection and load testing approaches, for which the first-class citizens are models of the input domain. For example, the workload is selected according to a probabilistic model of the operational profile and the faultload is based on a model of faults that are deemed representative of actual

faults in operation. Recent effort to standardize this kind of evaluation-oriented testing has yielded the emergence of the concept of dependability benchmarking.

This second classification is used to build the structure of this section. We first present work dealing with fault injection as a robustness testing technique. Then, we describe relevant robustness testing approaches based on system modeling and test case generation.

2.2.1 Fault injection approaches

Fault injection consists to introduce deliberate errors in a system and observe its behavior. This technique has been widely used for robustness testing because it allows one to evaluate the behavior of a given system when running in a hostile environment. In the following, we present most relevant fault injection tools for testing robustness of communication protocols and distributed systems.

DOCTOR

DOCTOR (integrateD sOftware Fault injeCTiOn enviRonment) [62] is a fault injection tool for distributed application. It can synthesize the workload and emulate the occurrence of faults in real time systems. It supports mainly three types of faults (processor, memory and communication faults) and can run three injection mode: permanent, transient and intermittent. During experimentations, DOCTOR collects performance and reliability information providing testers with significant evaluation data.

ORCHESTRA

ORCHESTRA [61] is a script-driven fault injection tool designed for testing the reliability and the liveness of distributed protocols. A fault injection layer is inserted between the tested protocol layer and the lower layers to filter and manipulate messages exchanged between the protocol participants.

Messages can be delayed, lost, reordered, duplicated and modified. Also, new messages can be spontaneously introduced into the tested system to bring it into a

particular global state.

The reception script and the sending script are written in TCL language and determine which operations are to be performed on received/sent messages. These scripts are specified with state machines. Transitions in these machines are driven by the type of the message, its contents, the history of received messages or other information that was previously collected during the test execution (e.g. local time, number of received messages, etc.).

Message modifications are however, specified using a user-defined script. The resulting message is passed to the next layer of the protocol stack.

ORCHESTRA is a "Message-level fault injector" because a fault injection layer is inserted between two layers in the protocol stack. This kind of fault injector allows injecting faults without requiring the modification of the protocol source code. However, the user has to implement his fault injection layer for each protocol he wants to test. The expressiveness of the fault scenario is limited as there is no communication between the various state machines executed on every node. Also, because the fault injection is based on exchanged messages, the knowledge of the type and the size of these messages is required [63].

NFTAPE

The NFTAPE project [65] arose from the double observation that no tool is sufficient to inject all fault models and that it is difficult to port a particular tool to different systems. NFTAPE provides mechanisms for fault-injection, triggering injections, producing workloads, detecting errors, and logging results. Unlike other tools, NFTAPE separates these components so that the user can create his own fault injectors and injection triggers using the provided interfaces.

NFTAPE is a Lightweight Fault Injector (LWFI). LWFIs are simpler than traditional fault injectors as they do not need to integrate triggers, logging mechanisms, and communication support. This way, NFTAPE can inject faults using any fault injection method and any fault model. Interfaces for the other components are also defined to facilitate portability to new systems.

In NFTAPE, the execution of a test scenario is centralized. A particular computer, called the control host, takes all control decisions. This computer is generally separated from the set of computers that execute the test. It executes a script written in Jython (Jython is a subset of the Python language) which defines the faults scenario. All participating computers are attached to a process manager which in turn communicates with the control host. The control host sends commands to process managers according to the fault scenario. When receiving a command, the process manager executes it. At the end of the execution or if a crash occurs, the process manager notifies the control host by sending a notification message.

All decisions are taken by the controller, which implies that every fault triggered at every node induces a communication with the controller. Then, according to the defined scenario, the controller sends a fault injection message to the appropriate process manager which can then inject the fault [63].

DEFINE

DEFINE (DistributEd Fault Injection and moNitoring Environment) [48] is a fault injector designed to evaluate system dependability, investigate fault propagation and validate fault tolerant mechanisms of distributed systems. This tool can inject software faults as well as hardware-induced software errors in any process running in distributed systems either in user mode or supervisor mode. The injected faults can be correlated or independents.

DEFINE is extended from it antecedent FINE [47], with additional distributed capability and injection mechanisms. It uses two fault injection techniques:

1. using hardware clock interrupts to control the time of fault injection and activation which allows injecting intermittent CPU/Bus faults in order to ensure their activation,

2. using software traps to inject faults and monitor fault activation in order to assist monitor whether the faults are activated and were they are activated.

Experiments using DEFINE were successfully conducted on SUN NFS-distributed

file system.

FAIL-FCI

FAIL-FCI [41] is a fault injection tool developed by INRIA (Institut National de Recherche en Informatique et Autimatique). First, FAIL (for FAult Injection Language) is a language that permits to easily described fault scenarios. Second, FCI (for FAIL Cluster Implementation) is a distributed fault injection platform whose input language for describing fault scenarios is FAIL. Both components aims at emulating large-scale networks on smaller clusters or grids.

The FAIL language allows defining fault scenarios. A scenario describes, using a high-level abstract language, state machines which model fault occurrences. The FAIL language also describes the association between these state machines and a computer (or a group of computers) in the network. The FCI platform is composed of several building blocks:

1. The FCI compiler: The fault scenarios written in FAIL are pre-compiled by the FCI compiler which generates C++ source files and default configuration files.

2. The FCI library: The files generated by the FCI compiler are bundled with the FCI library into several archives, and then distributed across the network to the target machines according to the user-defined configuration files. Both the FCI compiler generated files and the FCI library files are provided as source code archives, to enable support for heterogeneous clusters.

3. The FCI daemon: The source files that have been distributed to the target machines are then extracted and compiled to generate specific executable files for every computer in the system. Those executables are referred to as the FCI daemons. When the experiment begins, the distributed application to be tested is executed through the FCI daemon installed on every computer, to allow its instrumentation and its handling according to the fault scenario.

FCI is a Debugger-based Fault Injector because the injection of faults and the

instrumentation of the tested application is made using a debugger. This makes it possible not to have to modify the source code of the tested application, while enabling the possibility of injecting arbitrary faults (modification of the program counter or the local variables to simulate a buffer overflow attack, etc.). From the user point of view, it is sufficient to specify a fault scenario written in FAIL to define an experiment. The source code of the fault injection daemons is automatically generated. These daemons communicate between them explicitly according to the user-defined scenario. This allows the injection of faults based either on a global state of the system or on more complex mechanisms involving several machines (e.g. a cascading fault injection). In addition, the fully distributed architecture of the FCI daemons makes it scalable, which is necessary in the context of emulating large-scale distributed systems. FCI daemons have two operating modes: a random mode and a deterministic mode. These two modes allow fault injection based on a probabilistic fault scenario (for the first case) or based on a deterministic and reproducible fault scenario (for the second case). Using a debugger to trigger faults also permits to limit the intrusion of the fault injector during the experiment. Indeed, the debugger places breakpoints which correspond to the user-defined fault scenario and then runs the tested application. As long as no breakpoint is reached, the application runs normally and the debugger remains inactive.

2.2.2 Model-based approaches

Testing system robustness based on behavioral models can be seen as a conformance testing problem. Compared to traditional conformance testing, the only difference is the explicit fault dimension in the input domain, since faults are key inputs that the resilience mechanism is expected to deal with.

It is important, however, to note that the fault dimension has a strong impact on the implementation of the testbed. The experiments may necessitate the development of complex test platforms to be able to inject faults, synchronize them with the activity, and observe their effect.

In the following, we present most relevant contributions on model-based robust-

ness testing approaches.

The work presented in [46], builds a robustness testing approach based on the conformance testing of correctness properties. Thus, given a robustness property P, a system implementation is robust iff the property P is satisfied in presence of faults. This approach is based on the following elements:

- A formal model S describing the *nominal* system behavior. That is, the expected behavior of the tested system when running in normal environmental conditions. In this work, authors formalized S as a set of LTS's (Labeled Transition Systems).
- A fault model F representing the set of faults that may affect the tested system and cause failures. This fault model must be a set of mutations from the model S obtained by modifying exchanged parameter values, system transitions, etc.
- A robustness property P which specifies the expected system behavior in presence of faults. P is a linear property describing the set of robust execution sequences of the tested implementation.

Test cases are then generated as follows:

- Generation of a *degraded* model S_d by deriving a mutation of S based on the fault model F.
- Generation of an observer O. This observer is a Rabin automata [58] describing the set of sequences of P. It identifies the set of non robust sequences of S_d
- Generation of test cases from S_d and O: non robust execution sequences are extracted from S_d and transformed to test cases by computing an asymmetric product with the observer O.

Another model-based approach is proposed in [40] and concerns specifically embedded real time systems. In this work, we consider also two system specifications: a *nominal* and a *degraded* one. The degraded specification describes critical system actions that must be handled in stressful and/or unexpected environmental conditions. The robustness testing process proceeds as follows:

1. Generation of test sequences from the nominal specification;
2. Application of magnetic radiations on the system under test;
3. Running the generated test sequences;
4. End of magnetic radiations;
5. Result analysis and partial verdict;
6. Generation of mutant test sequences;
7. Running mutant test cases;
8. Result analysis and final verdict.

Authors also proposed another testing process based on test cases generation and execution based on the degraded specification.

In [37], authors presented a robustness testing framework based on a different model-based approach. This framework proceeds in two phases:

1. First, an increased specification is built by integrating hazards in the nominal specification;
2. Then, robustness test cases are generated from the increased specification and executed on the implementation.

Hazards denote any events not expected in the nominal specification of the system. Authors identified three kinds of controllable and representable hazards: invalid inputs, inopportune inputs (actions belonging to the specification alphabet but not expected in the given state) and unexpected outputs.

The first phase consists to integrate the representable hazards in the model of the nominal specification. The obtained model is called *increased* specification. Then, the robustness of the implementation is evaluated with respect to this increased specification by generating and executing test cases as follows:

1. Definition of a Robustness Test Purpose (RTP). RTP is a part of the total specification. It allows one to focus on a precise behavior of the system.

2. Computation of the synchronous product $S_A \otimes RTP$ where S_A is the increased specification.

3. Building a Robustness Test Graph (RTG) based on the result of the previous computation. This graph describes all tests corresponding to a given RTP. It is then reduced to a Reduced Robustness Test Graph RRTG which contains only paths describing acceptable behaviors (according to the RTP).

4. Generation and execution of robustness test cases from the RRTG.

Chapter 3

Specification and Verification of Fault Injection Process

Contents

3.1 Introduction

Fault Injection consists in introducing deliberate errors in a system and observe its behavior. This technique is usually used to assess error recovery and fault tolerant mechanisms, to perform some dependability measures such as availability, integrity and performance or simply to understand the effect of real faults. Fault injection can be applied both to hardware systems (HWIFI: Hardware Implemented Fault Injection) and software systems (SWIFI: Software Implemented Fault Injection) but there has been more research on SWIFI based tools, mostly because they do not require any expensive hardware.

SWIFI approaches are categorized into several classes according to the type of the injected faults and the injection level. Two of these categories are particularly interesting in the context of our work : interface faults and communication faults. At interface level, faults affect functions input/output parameters or protocol messages fields. The values attributed to these parameters are generated differently from an approach (or tool) to another: some fault injectors provide generic inputs to all parameters whatever their types, others generate type-specific inputs (like Ballista [52] which assigns a set of values to each parameter type) and there are also some tools like Fuzz [35] which generate random inputs for each parameter. For communication faults, the injection concerns the message exchanges between system components. The injector can corrupt, delay or replicate messages. It can also perform other operations according to the fault model specified by the tester.

The main goal of fault injection is experimental validation. As mentioned before, a fault injection test experiment lies in the introduction of faults from a given scenario into an implementation under test (IUT), the target, to observe how it behaves under the presence of such faults. However, relying only on experimental methods may be insufficient and in some cases can be seen as a lack of thoroughness and soundness, mainly during results analysis and validation. This can be widely avoided using formal methods.

The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate

mathematical analyses can contribute to the reliability and robustness of a design. In software testing, we rely on formal specifications at various stages during the test process. We specify the behavior of the implementation under test, the properties that must be analyzed and all the needed operations to achieve the test purposes. This allows us to avoid any ambiguity or conflict that may appear when depending only on experimental methods. By using formal methods, we can clearly specify the test purposes and the test methodology. Hence, results analysis will be based on mathematical concepts avoiding any false interpretations and/or verdict issues.

If we choose to rely on fault injection to perform any kind of test (robustness, security or even functional testing), we need, not only to specify the tested properties (robustness, security or functional properties) but also the fault injection itself. We should provide a formal description of the injected faults and the way they are injected. Because the verdict which will be issued, will strongly be dependent of the injected errors. Also, if researchers specify formally their entire injection methodology, then it can be easily studied, analyzed and eventually reproduced and/or extended by other testers in future time. Therefore, it can be the best way to study the effects of errors on real systems.

In this chapter, we propose a formal method for fault injection specification and verification. We aim to provide a generic and formal system for fault modeling to allow more rigor in error description and to avoid specification ambiguities. The main contributions we bring in this work are the following:

- First, we propose a fault injection specification formalism based on Hoare logic [42] and time constraints. The proposed formalism allows specification of several types of faults and can be used to test both communication protocols and distributed systems. It is formal as it is based on a mathematical logic. This avoids specification ambiguities and allows fault injection validation. It is also a generic formalism because it uses a high level abstract syntax. Therefore, it is well appropriate for the specification and the verification of various injection operations.

- Then, we propose a passive testing approach to verify the correctness of the

injection process. The idea is to exploit the formal specification of faults as a test oracle to check the good execution of the injection process.

3.2 Fault injection specification

3.2.1 Preliminaries

Hoare logic

Hoare logic [42] is a formal system which provides a set of logical rules based on mathematical logic. Its central feature is the Hoare Triple which describes how an execution of a set of actions changes the state of some variables. A Hoare triple is of the form $\{P\}C\{Q\}$ where C is a program (a set of actions) and P and Q are assertions expressed in a first-order logic. Informally, a triple $\{P\}C\{Q\}$ has the following meaning: if C is executed in a state satisfying precondition P and if C terminates then the final state satisfies postcondition Q. Hoare logic has also axioms and inference rules that can be used to reason about the correctness of computer programs. However, in this paper we are mostly interested by the formalization. Therefore, we focus only on Hoare Triples (a complete presentation of Hoare axioms and inference rules can be found in [42]).

Fault injector location and capabilities

We can rely on SWIFI approaches to test various aspect of a given system. Depending on the test purpose, fault can be injected at different system locations : memory, hard disk driver, communication interfaces, etc. In this work, we address the case of communication and interface faults applied on distributed systems. Therefore, we assume that the fault injector would be placed between two agents of a global system: $A1$ and $A2$; and that is able to perform the following actions:

- Intercept every message exchanged between A_1 and A_2.
- Apply some operations on the intercepted message.
- Resend the faulty message.

We note that the injection process is performed during a finite period of time. Therefore, the messages exchanged between $A1$ and $A2$ during the injection process are of a finite number.

3.2.2 Fault injection formalism

Based on the above assumptions, we propose to define a fault injection operation with a Hoare triple as follows.

Definition 3.1: (Injection operation) *an injection operation is a Hoare triple* $\{P\}C\{Q\}$ *where :*

- P *specifies a precondition on the intercepted message (its state before the execution of the injection operation);*
- C *denotes the operation itself (identified by its name and eventually a set of parameters);*
- *and* Q *is a postcondition which states the effect of the operation execution on the intercepted message.*

A communication message can be considered as a finite set of elements. Each element describes a part or a field of this message. Therefore, we can specify formally a communication message as a finite collection (a set where replicates are permitted) of elements $S = \{elt_1, ..., elt_n\}$. We specify also the set of all injection operations executed during an injection experiment as a finite set of injection rules R such as each injection rule $r \in R$ specifies a Hoare triple describing an injection operation applied on an intercepted message, as follows.

$$\{P(S)\}\ OperationName(param_1, ..., param_n)\ \{Q(S)\}$$

3.2.3 Time extension

The fault injection formalization presented above can be used to specify many injection operations. However, as it is based on the basic definition of Hoare logic

as it was introduced in [42], it does not support time specification. Thus, we are unable to specify timed injection operations like for example the delaying of messages; whereas time is probably one of the most important properties that must be considered when testing system reliability. Therefore, instead of using the classical Hoare logic, we propose to rely on an extended version which supports real-time specification.

In [44], the authors extended Hoare logic to real-time. They defined special variables and some primitives to allow time reasoning and illustrated their model with many specification examples. The proof of soundness and completeness of this extended model is given in [43].

Based on this extension, we propose to specify each fault injection operation as a Hoare triple where preconditions and postconditions are expressed in first-order logic with the following primitives.

- We assume that the timing behavior of the fault injector is described from the viewpoint of an external observer with his own clock,
- we define a time domain $TIME = \{\tau \in \Re | \tau \geq 0\}$ and logical time variables ranging over $TIME \cup \{\infty\}$, such as $t, t_0, t_1, ...$
- We define a special variable *now* which ranges over $TIME \cup \{\infty\}$ and refers to the global notion of time presented in the first point.

3.2.4 Specification language

We propose here a common and generic specification language to be used for pre/-postcondition specifications.
As we consider the captured messages as sets of elements, we define a set of functions and predicates inspired from the set theory so that we will be able to specify all kinds of pre-and postconditions related to sets.
Definition 3.2: (Specification primitives) *given two sets A and B and a set element elt, we define the following primitives.*

- *A.isEmpty(): returns true iff A is an empty set;*

- *A.size(): returns the size (number of elements) of the set A;*
- *A.has(elt): tells whether the given element elt belongs to the set A;*
- *A.count(elt): tells how many time a given element elt occurs in the current set A;*
- *A.remove(elt): returns a set containing the items in the current set (A) except for one of the given element elt.*
- *A.equals(B): returns true iff set A is equal to set B (they have the same size and the same elements);*
- *A.isSubSet(B):returns true iff every element of A is contained in B.*

We also define a modifier $new(SetName)$ to refer to the set $SetName$ after the execution of an injection operation. For example $new(S)$ refers to the state of the set S after the injection.

3.3 Specification examples

We present in this section several examples to illustrate our specification formalism. Each example describes a possible injection operation and provides its corresponding Hoare triple. As defined in the specification formalism, we will refer to each intercepted message as a set of elements S.

3.3.1 Operation Delete

The first operation which we specify is used to delete intercepted messages. We express it by a Hoare triple as follows.

$$\{\neg S.isEmpty()\}\ Delete(S)\ \{new(S).isEmpty()\}$$

We can also specify the deletion of one message element as follows.

$$\{S.has(elt)\}\ Delete(S, elt)\ \{new(S).equals(S.remove(elt))\}$$

3.3.2 Operation Delay

This operation is used to delay the forwarding of intercepted messages. A parameter $n \in TIME$ specifies the period of delay, which means that the captured message will be kept for n time units before it is resent. The corresponding Hoare triple is of the form:

$$\{\neg S.isEmpty() \wedge now = Val,\ Val \in TIME\}$$
$$Delay(S, n)$$
$$\{new(S).equals(S) \wedge now = Val + n + \varepsilon, \varepsilon \in TIME\}$$

In the precondition, we specify the time value before the execution of operation *Delay*. Then, in the postcondition, we ensure that this value has been exceeded by n time units. ε specifies the very short extra delay that we may accept due to the density of the time domain.

3.3.3 Operation Replicate

This operation is used to replicate message elements. The number of replication is specified by an argument $n \in N^+$.

$$\{S.has(elt)\}\ Replicate(S, elt, n)\ \{new(S).count(elt) = n * S.count(elt)\}$$

We can also specify a replication of all elements of the captured message as follows.

$$\{\neg S.isEmpty()\}$$
$$Replicate(S, n)$$
$$\{\forall\ elt :\ S.has(elt) \Rightarrow\ new(S).count(elt) = n * S.count(elt)\}$$

We verify in the postcondition that operation *Replicate* creates n copies of each element contained in S. The universal quantifier expression is true if for all elements elt such as $S.has(elt)$ is true, $new(S).count(elt) = n * S.count(elt)$ is also true.

3.3.4 Operation Insert

$$\{true\}\ Insert(S, elt)\ \{S.equals(new(S).remove(elt))\}$$

This injection operation inserts extra data in the captured message. It can be either a malicious element or just a huge bloc of insignificant data in order to disturb the communication.

3.3.5 Operation Corrupt

This is a content corruption operation which modifies the content of intercepted messages before their forwarding. We specify it with the following Hoare triple.

$$\{\neg S.isEmpty()\}$$
$$Corrupt(S)$$
$$\{new(S).size() = S.size() \wedge \neg new(S).equals(S)\}$$

In the postcondition, we check whether the message S keeps the same number of elements, with a different content.

3.4 Passive testing approach

If we want to include a fault injection mechanism as a part of a complete testing methodology, we have to verify and validate its behavior within the test context. This is a very important step, because it allows us to ensure that the specified injection operations are properly implemented and performed. Otherwise, some confusion may occur during the test execution. For example, if we are testing a security protocol using a fault injector that we configured to delete some specific messages. Then, after the test execution, how can we be sure that the lost messages have been effectively deleted by the fault injector and not lost due to a protocol vulnerability or a system failure? This confusion can be omitted if we had a mean to verify the good execution of the performed injection actions.

It is also very important to note that this verification step must be performed after each experiment. The fact that the used fault injector may have been already

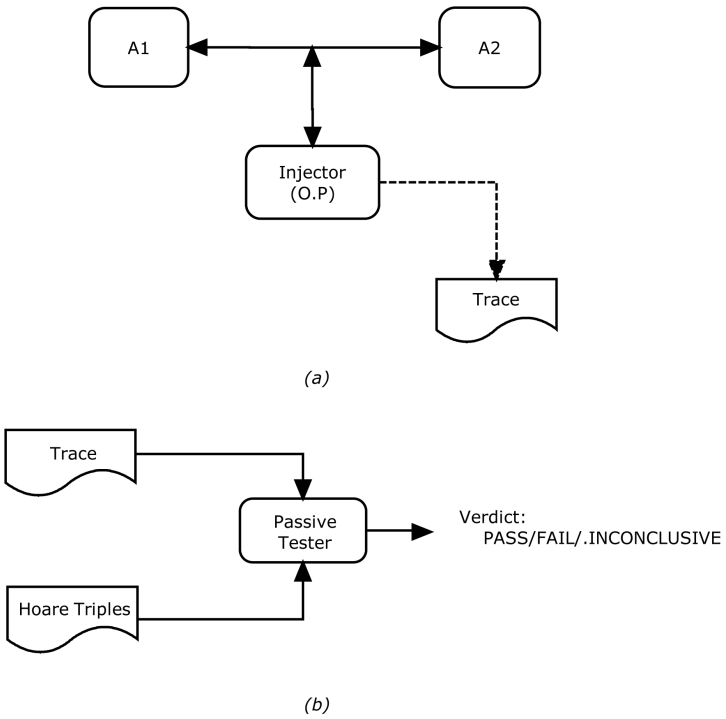

Figure 3.1: The passive testing approach: (a) Collecting the trace. (b) Checking trace conformance w.r.t. injection rules specification.

tested and validated before, does not mean that it will behave correctly in all situations and contexts. The fault injector is an extern element that we include in our testing environment. Therefore, depending on this environment (which may be a hostile or an experimental platform) it may work correctly or not.

In this section, we present a passive testing approach to perform this kind of verification. This approach allows one to check the conformance of a fault injection process with respect to its formal specification given as a set of Hoare triple rules. Figure 3.1 gives an overview of the proposed technique.

First, we put some observation points (O.P.) at the fault injector core to collect an execution trace during the injection process. We assume here that we have access to the fault injector source code so that we can log all executed operations or that the used fault injector provides a log file containing all necessary information. Otherwise, we can put the O.P. at the fault injector interface (to collect input/output messages), but in this case we can just verify the pre/postconditions independently

of the executed operations, which is not conform to Hoare logic semantics.

After the injection experiment terminates, we analyze the collected trace to check its conformance with respect to the specified fault injection model (figure 3.1.b). We note that this approach does not validate completelty the used fault injector but it allows testers to ensure if for a given experiment, the fault injection has been well performed or not.

The specification file provided to the passive tester contains a set of injection rules specified as Hoare triples using the specification language presented in section 3.2.4. The passive tester will then execute Algorithm 1 to check whether the collected trace is conform to the specified injection operations.

Algorithm 1 Trace checking

```
1:  Require: HT[r]: Hoare triple rules + Tr[l]: trace file;
2:  Ensure: Verdict[v]: Verdict table;
3:  Initialization :
4:  for each rule r ∈ HT do
5:     Verdict[r]:=INCONCLUSIVE;
6:  for each line l of Tr do
7:     if ∃r ∈ HT : (l ⊨ r.precond) and (r.operation ≡ l.Operation) then
8:        if ¬(l ⊨ r.postcond) then
9:           Verdict[r]:=FAIL;
10:          (log the current line which violates the current injection rule)
11:       else
12:          if Verdict[r] ≠ FAIL then
13:             Verdict[r]:=PASS;
```

The trace file is formatted as follows. For each executed operation, the following information are logged:

- `Operation` : the name and parameters of the executed operation ;
- `StartTime`: the time at which it starts its execution;
- `InMsg`: the input message (the captured message on which the current operation should be applied);
- `OutMsg`: the output message (the message returned by the current operation);
- `EndTime`: the time at which the current operation finishes its execution.

Based on that trace format, Algorithm 1 starts by an initialization step where it associates an INCONCLUSIVE verdict to all injection rules. INCONCLUSIVE verdict means that we are unable to verify the correct implementation of a given rule; either because no line from the trace satisfies the rule precondition or that the executed operation is different from the specified one.

After this first step, the algorithm verifies for each trace line l , if there exists a rule r from the specification file whose precondition is satisfied by l ($l \models r.precond$) and if the executed operation ($l.Operation$) matches with the specified one ($r.operation$). If it does, the rule verdict is updated with a PASS/FAIL verdict according to the conformance of the specified postcondition ($r.postcondition$) w.r.t. the observed trace line.

We note that each injection rule may be tested several times (each time a trace line satisfies its preconditon). However, if the test failed once then the final verdict associated to this rule will be FAIL. The complexity of the algorithm is straightforward. At worst, an injection operation might be concerned by all lines from the trace. Therefore, the complexity is $O(n.m)$, where n is the number of the specified injection rules (size of the table HT) and m is the trace length.

In the case of a black box testing, where we cannot log the executed operations, we can only observe the input/output messages with their relative input/output times. Therefore, even if we can modify Algorithm1 to check whether the pre-and postconditions related to a given message are respected, nothing can be said about the real implemented operations.

3.5 Conclusion

Fault injection is a powerful strategy to test fault-tolerant protocols and distributed systems. The first step in building a complete fault injection process is the specification of a fault scenario for the test experiment. This includes the specification of the fault injector location and the type and time of injected faults. In this chapter, we presented a generic fault injection formalism based on Hoare logic and time specification. We detailed its syntax and semantics and provided some specification

examples to illustrate its use.

Once faults are specified, one can easily control the injection process by verifying its execution. We proposed a passive testing approach which uses the injection specification to check the injection process. This way, we would be able to ensure that the injection is well performed and thus, we will avoid any ambiguity during result analysis.

The proposed fault injection formalism could also be exploited in other manners. For example, it would be interesting to study the possibility of automatic generation of faults from the abstract Hoare specification, or to propose some fault injection patterns for different testing purposes.

Chapter 4

A Formal Approach for Checking Real Time Constraints

Contents

4.1 Introduction

The high complexity and the large variety of actual implemented systems as well as the high degree required for their global performance, lead to increasingly challenging issues on developing approaches and techniques for verification and validation of correctness properties.

System requirements (also called *correctness properties*) are a set of rules which describe how data and other critical resources of a given system should be managed. Usually, such requirements are defined by network and/or system administrators which are in charge of implementing and controlling the critical mechanisms of their organizations. Since this set of rules can be more or less complex, any specification ambiguity can lead to conflicts or create security threats. To avoid these dangerous situations, administrators and test experts often rely on formal specification languages to describe their requirements. The choice of such formalism is crucial as it determines the type of correctness properties that can be carried and the reliability of the testing approach.

Defining time constraints as a way of controlling system behaviors may be an efficient technique to avoid temporal vulnerabilities. However, to ensure that a system respects the defined constraints, we need first to specify them using the most suitable formalism (which in this case must support time specifications). Then, we may rely on formal testing methods which offer more rigor and efficiency in verification process, to study the conformance of the system behavior with respect to the specified properties.

Formal testing techniques can be categorized into two main classes: (i) active testing approaches and (ii) passive testing approaches. In active testing, system implementations are checked by applying a set of test cases (generated from a global requirement model) and analyzing their behavior; while in passive testing, we do not interact directly with the tested system. Instead, we collect system execution traces and verify their conformance with respect to correctness properties.

As active testing requires direct interaction with the tested system, it is not always possible to rely on it in all situations. For example, when the tested imple-

mentation does not provide any interfaces or when the tested system is built upon a set of components running in their own environments and where there is no direct way to access them (like composed Web services for example). In such cases, there is a particular interest of using passive testing techniques where the verification process does not need any direct interaction with the tested system as it only analyzes collected execution traces.

In this chapter, we propose a formal and generic framework for specification and verification of real time requirements on execution traces. Our main contributions are:

- A formal specification of real time properties. We formalize temporal properties as XCTL(eXplicite Clock Temporal Logic) [32] formulas to be able to specify both simple and complex real time constraints.
- A passive testing algorithm to verify the conformance of such requirements against execution traces;
- A proposition of real time patterns and an experimental study to show the performance of the proposed algorithm.

4.2 Related work

Linear Temporal Logic (LTL) [55] is a well known mathematical logic which has been widely used in several testing domains. Broadly, we can rely on LTL to specify two types of critical system requirements: *safety* properties and *liveness* properties. Safety properties state that nothing bad ever happens in the system. For example: an intruder never gets user or administrative privileges on the network or a controller does not allow the boiler temperature to rise above a certain level. On the other hand, liveness requirements specify the active tasks that a system is designed to do i.e. they assert that "something good will eventually happen". For example, a banking system might have a liveness requirement stating that if a check is deposited and sufficient funds are available, the check eventually clears.

These features made LTL strong enough to build frameworks for different testing purposes such as security, reliability and robustness. However, as it appeared that LTL is very suitable for modeling security issues, most researches focused on providing LTL-based approaches for testing system security. In [23] for example, authors proposed a model checking approach for security protocols based on the set-rewriting formalism coupled with a subset of LTL. Their model allows specifications of assumptions on principals and communication channels as well as other security requirements. However, this approach does not support real time specifications and it only validates security properties with respect to the protocol specification and not against its real implementation (a model checking approach).

In [53], authors used temporal logic to build general intrusion detection framework. They based their approach on a runtime monitoring algorithm to automatically verify temporal specifications against a system execution and raise intrusion alarms whenever the specification is violated. They used the EAGLE formalism to specify temporal requirements. EAGLE [39] is a temporal logic formalism supporting recursively defined temporal formulas parameterizable by both logical formulas and data expressions. Although it is possible to specify some kind of real time properties using this formalism (time interval specifications), it is practically impossible to address complex properties which refer to correlated time constraints (temporal constraints defined with respect to other temporal constraints in the same formula).

Another LTL-based framework for testing security properties is presented in [68]. In this paper, authors proposed to test security policies of a given system based on test generation and execution of security rules from temporal logic specifications. This approach suffers from two main drawbacks. First, they addressed a very limited set of security patterns as they restricted the syntax and semantics of their formalism to a small subset of linear temporal logic. Second, their approach does not support real time specifications.

It is important also to highlight other work which aimed at providing real time frameworks not based on LTL. For example in [69], authors proposed a general framework for testing timed security properties based on deontic logic and linear

time specification. The same formalism was also used in [25] for monitoring Web services. However, this formalism supports only specification of simple temporal constraints like those we can specify using bounded temporal operators (section 4.3.1). Moreover, the specification language is very complex (which makes it hard to use in practice) and the used deontic logic is highly security oriented (which makes it difficult to apply on other testing purposes).

The approach we propose in this work aims at providing a generic and formal framework for testing real time properties. We want to be as generic as possible so that our approach can be applied not only for security testing (which is a widely known application of LTL), but also for safety, robustness and other testing purposes.

We propose to formalize real time requirements as XCTL formulas. This way, we would be able to specify more complex temporal constraints than those carried by the above cited approaches. Then, we propose an efficient monitoring algorithm based on passive testing to check such properties on execution traces.

4.3 LTL and real time logics

Linear Temporal Logic (LTL) [55] is a specific branch of temporal logic which allows one to reason about both causal and temporal properties based on linear time semantics.

An LTL formula consists of atomic propositions, Boolean operators and temporal operators. The operator $\bigcirc$ refers to the next state. E.g., $\bigcirc a$ expresses that a has to be true in the next state. $\cup$ is the until operator, where $a \cup b$ means that a has to hold from the current state up to a state where b is true. $\Box$ is the always operator, stating that a condition has to hold at all states of a path, and $\Diamond$ is the eventually operator that requires a certain condition to eventually hold at some time in the future. The syntax of LTL is given as follows, where AP denotes the set of atomic propositions:

Definition 4.1 (LTL syntax) *The BNF definition of LTL formulas is given as follows:*

$\phi := true|false|a \in AP|\neg\phi|\phi_1 \wedge \phi_2|\phi_1 \vee \phi_2|\phi_1 \rightarrow \phi_2|\phi_1 \equiv \phi_2|\phi_1 \cup \phi_2|\bigcirc \phi|\Box\phi|\Diamond\phi$

The semantics of LTL is expressed for infinite traces. However, as in this work we are dealing with "off-line" testing using a pre-collected set of traces, we will consider the finite LTL semantics as presented in [60].

We define a trace as a finite list of events. Assume two partial functions defined for nonempty traces $head : Trace \rightarrow event$ and $tail : Trace \rightarrow Trace$ for taking the head and tail respectively of a trace, and a total function *length* returning the length of a finite trace . That is, $head(e\ t) = e$, $tail(e\ t) = t$, $length(end) = 0$ and $length(e\ t) = 1 + length(t)$ where t is a trace, e is an event and *end* denotes the empty trace. Assume further that for any trace t that t_i for some natural number i, denotes the suffix trace that starts at position i, which position starting at 1. The finite LTL semantics can be defined as follows:

Definition 4.2 (LTL semantics) *The satisfaction relation* $\models \subseteq Trace \times Formula$ *which defines when a trace* t *satisfies a formula* ϕ *(written* $t \models \phi$*) is defined inductively over the structure of the formulas as follows (where* p *is an atomic proposition and* ϕ_1 *and* ϕ_2 *are any formulas):*

$t \models true$ iff $true$,
$t \models false$ iff $false$,
$t \models p$ iff $t \neq end$ *and* $head(t) = p$
$t \models \neg p$ iff $t \not\models p$
$t \models \phi_1 \wedge \phi_2$ iff $t \models \phi_1$ *and* $t \models \phi_2$
$t \models \phi_1 \vee \phi_2$ iff $t \models \phi_1$ *or* $t \models \phi_2$
$t \models \phi_1 \rightarrow \phi_2$ iff $t \not\models \phi_1$ *or* $t \models \phi_2$
$t \models \phi_1 \equiv \phi_2$ iff $t \models \phi_1$ *iff* $t \models \phi_2$
$t \models \Box\phi$ iff $(\forall i \leq length(t))\ t_i \models \phi$
$t \models \Diamond\phi$ iff $(\exists i \leq length(t))\ t_i \models \phi$
$t \models \phi_1 \cup \phi_2$ iff $(\exists i \leq length(t))\ (t_i \models \phi_2$ *and* $(\forall j < i)\ t_j \models \phi_1)$
$t \models \bigcirc\phi$ iff $t \neq end$ *and* $tail(t) \models \phi$

4.3.1 Real time extensions

Although LTL can be used to specify a wide range of temporal properties, it still presents some limitations regarding specifications of real time systems as it does not provide means to formalize real time constraints [56]. Therefore, several approaches have been proposed to extend LTL formulas in order to support real time specifications. These approaches can be classified into three main categories based on how time values are specified. In the following, we present and discuss these extensions and justify our choice of XCTL.

Bounded temporal operators

A common way of introducing real time in the syntax of LTL is to replace the unrestricted temporal operators by time-bounded versions. For example, the bounded operator $\Diamond_{[2,4]}$ is interpreted as "eventually within 2 to 4 time units". Based on this extension, one can specify properties like "every event p is followed by an event q within 3 time units" as follows.

$$\Box(p \rightarrow \Diamond_{[0,3]} q)$$

However, the bounded-operator notation can relate only adjacent temporal contexts. Consider, for instance, the property that "every request p is followed by a response q and, then, by another response r such that r is within 5 time units of the request p". While this kind of properties is very important, there is actually no direct way of expressing this "nonlocal" timing requirement using time-bounded operators.

This shortcoming of bounded temporal operators can be remedied by extending temporal logic with explicit references to the times of temporal contexts. We discuss in the following paragraphs two of such methods: one based on freeze quantification and the other using a dynamic state variable.

Freeze quantification

The idea of freeze quantification is based on the use of a *freeze* quantifier "x" inside an LTL formula to bind the associated variable x to the time of the current temporal context: the formula $x.\phi(x)$ holds at the time t iff $\phi(t)$ does. Thus, in the formula $\Diamond y.\phi$ the time variable y is bound to the time of the state at which ϕ is "eventually" true. By admitting atomic formulas that relate the times of different states, we can write the nonlocal property that "every request p is followed by a response q and, then, by another response r such that r is within 5 time units of the request p" as follows.

$$\Box x.(p \rightarrow \Diamond(q \wedge \Diamond z.(r \wedge z \leq x + 5)))$$

Explicit clock variable

Another way to specify real time requirements is based on standard first order temporal logic. The syntax uses a dynamic state variable T (the clock variable) and first order quantification for global variables overs the time domain. The clock variable assumes, in each state the value of the corresponding time. For example, the property "every request p is followed by a response q within 3 time units" can be specified as follows.

$$\Box((p \wedge T = x) \rightarrow \Diamond(q \wedge T \leq x + 3))$$

Here, the global variable x is bound to the time of every state in which p is observed. We refer to the use of a clock variable as the "explicit-clock" notation.

The linear time logic which is based on this technique is called XCTL [32](eXplicit Clock Temporal Logic). It is a real time logic whose assertion language for atomic timing constraints allows the primitives of *comparisons* and *addition*. Thus, the timing constraints of XCTL are richer than those of the previous logics, which prohibit the addition of time variables. Also, the definition of the clock variable T allows one to refer to the global time of the system, which is not possible with freeze quantification for example (as there is no global time reference) [56].

These features make XCTL very suitable for specification and verification of complex real time properties. In fact, by using XCTL, one can specify both simple and correlated time constraints and the use of a single global time variable makes the specifications easier. Therefore, we will rely on this logic to specify real time constraints and propose a passive testing algorithm to check this kind of constraints on events traces.

4.4 Passive testing algorithm

In this section we present our passive testing algorithm for verification and validation of real time properties. The algorithm inputs are a set of requirements specified as XCTL formulas and an event trace. The aim is to provide a verdict about the conformance of the trace with respect to the specified properties.

4.4.1 XCTL and passive testing

Formally, we specify the trace file as a finite set of couple $\{(e_1, t_1), ..., (e_i, t_i), ..., (e_n, t_n)\}$ where each couple represents an event e_i that occurs at a time t_i such as $\forall i \in [1, n], t_i < t_{i+1}$.

As all time values in the trace represent event occurrence times, some type of formulas cannot be checked directly. For example, a formula like $P \cup (T = val)$ cannot be verified because $T = val$ might not be observable on the trace (as it does not relate to an event occurrence). Therefore, we formally introduce the following sub-grammar of XCTL which allows to build only formulas where temporal constraints are connected to propositional variables with logical conjunctions.

Definition 4.3: *The BNF definition of XCTL formulas addressed by our approach is given as follows.*

$\phi := true|false|p \in AP|p \wedge TC|\neg\phi|\phi_1 \wedge \phi_2|\phi_1 \vee \phi_2|\phi_1 \rightarrow \phi_2|\phi_1 \equiv \phi_2|\phi_1 \cup \phi_2| \bigcirc \phi|\Box\phi|\Diamond\phi$

$TC := T \sim ax + c$

Where $\sim \in \{<, \leq, >, \geq, =\}$, *$T$ is the global clock variable, x is a static time variable and a, c are constants.*

The definitions of the time domain and the set of constants are given in [32], as well as the XCTL semantics which we rely on, in this work.

4.4.2 Test algorithm

Our algorithm is based on the idea that LTL properties can be checked backwards by updating the verdict at each step based on our knowledge of the future (as the trace is traversed from its end) [60]. We will first start by an example of a simple LTL formula (without temporal constraints) to show how it can be checked on a trace. Consider, for instance, the following formula.

$$\phi = \Box(P \rightarrow \Diamond Q)$$

The Breadth First Search (BFS) order of this formula gives the following set of subfomulas.

$\phi 1 = \Box(P \rightarrow \Diamond Q)$
$\phi 2 = P \rightarrow \Diamond Q$
$\phi 3 = P$
$\phi 4 = \Diamond Q$
$\phi 5 = Q$

Now, consider a finite trace of events $trace = \{e_1, ..., e_n\}$ (we will address time constraints later). One can define recursively a boolean matrix $mat[1..n, 1..m]$ where n is the length of the trace and m is the number of subformulas with the meaning that $mat[i, j] = true$ iff $trace_i \models \phi_j$. In our example it will be $mat[1..n, 1..5]$ such as.

$mat[i, 5] := (Q \in e_i)$
$mat[i, 4] := mat[i, 5] \vee mat[i + 1, 4]$
$mat[i, 3] := (P \in e_i)$
$mat[i, 2] := mat[i, 3] \rightarrow mat[i, 4]$

$mat[i,1] := mat[i,2] \wedge mat[i+1,1]$

for all $i < n$, where $\vee, \wedge, \rightarrow$ are ordinary boolean operators. For $i = n$, we need to initialize the matrix as follows.

$mat[n,5] := (Q \in e_n)$
$mat[n,4] := mat[n,5]$
$mat[n,3] := (P \in e_n)$
$mat[n,2] := mat[n,3] \rightarrow mat[n,4]$
$mat[n,1] := mat[n,2]$

An important observation is that, for each event from the trace, we may need at worst informations about the previous event (the next one when addressed backwards). Therefore, instead to keep all the table $mat[1..n, 1..m]$ which would be quite large in practice, one needs only to keep two rows $mat1[i, 1..m]$ and $mat2[i+1, 1..m]$ handling informations about the actual step and the next one. We will call this vectors *now* and *next*, respectively. We can now present the passive testing algorithm which address all kind of LTL properties as in Algorithm 2 [60]. Given an LTL formula ϕ and an event trace $Tr = \{e_1, ..., e_n\}$, this algorithm consists of three main phases:

1. First we generate the set of subformulas in the BFS order of the tested LTL formula. Let $\{\phi_1, \phi_2, ..., \phi_m\}$ be the list of all generated subformulas. The semantics of finite trace LTL allows us to determine the truth value of $Tr_i \models \phi_j$ from the truth values of $Tr_i \models \phi_{j'}$ for all $j < j' \leq m$ and the truth values of $Tr_{i+1} \models \phi_{j'}$ for all $j \leq j' \leq m$. This recurrence justify the backward checking order of the algorithm.

2. The second step is an initialization loop. Before the main loop, we should first initialize the vector $next[1..m]$. According to the semantics of LTL, the vector *next* is filled backwards. For a given $1 \leq j \leq m$, $next[j]$ is calculated as follows:

- If ϕ_j is a variable then $next[j] \leftarrow (\phi_j \in e_n)$; Here, we only verify if the atomic proposition satisfies the last event from the trace;
- If ϕ_j is $\neg\phi_{j'}$ for some $j < j' \leq m$ then $next[j] \leftarrow \ not\ next[j']$, where not is the negation operator on Booleans;
- If ϕ_j is $\phi_{j_1}\ Op\ \phi_{j_2}$ for some $j < j_1, j_2 \leq m$ then $next[j] \leftarrow next[j_1]\ op\ next[j_2]$, where Op is any propositional operation and op is its corresponding Boolean operation;
- If ϕ_j is $\bigcirc\phi_{j'}$, $\Box\phi_{j'}$ or $\Diamond\phi_{j'}$ then clearly $next[j] \leftarrow next[j']$ due to the stationary semantics of the finite trace LTL;
- If ϕ_j is $\phi_{j_1} \cup \phi_{j_2}$ for some $j < j_1, j_2 \leq m$ then $next[j] \leftarrow next[j_2]$ for the same reason as above.

3. The last step is the main loop. Considering the dependences in the recursive definition of finite trace LTL satisfaction relation, one must visit the remaining of the trace backwards, so the loop index will vary from $n-1$ down to 1. The loop body will update the vector now and at the end it will move it into the vector $next$ to serve as basis for the next iteration. At a certain iteration i, the vector now is updated backwards as follows:

   - If ϕ_j is a variable then $now[j] \leftarrow (\phi_j \in e_n)$;
   - If ϕ_j is $\neg\phi_{j'}$ for some $j < j' \leq m$ then $now[j] \leftarrow \ not\ now[j']$
   - If ϕ_j is $\phi_{j_1}\ Op\ \phi_{j_2}$ for some $j < j_1, j_2 \leq m$ then $now[j] \leftarrow now[j_1]\ op\ now[j_2]$, where Op is any propositional operation and op is its corresponding Boolean operation;
   - If ϕ_j is $\bigcirc\phi_{j'}$ then $now[j] \leftarrow next[j']$ since ϕ_j holds now iff $\phi_{j'}$ held at the previous step (which processed the next event, the $(i+1)^{th}$);
   - If ϕ_j is $\Box\phi_{j'}$ then $now[j] \leftarrow now[j'] \wedge next[j]$ because ϕ_j holds now iff $\phi_{j'}$ holds now and ϕ_j held at the previous iteration;
   - If ϕ_j is $\Diamond\phi_{j'}$ then $now[j] \leftarrow now[j'] \vee next[j]$ for similar reason as above;

- If ϕ_j is $\phi_{j_1} \cup \phi_{j_2}$ for some $j < j1, j2 \leq m$ then $now[j] \leftarrow now[j_2] \vee (now[j_1] \wedge next[j])$.

After each iteration $next[1]$ says whether the initial LTL formula is validated by the trace. Therefore desired output is $next[1]$ after the last iteration. The truth value of this vector element gives the final verdict ($true \equiv PASS$ and $false \equiv FAIL$).

Algorithm 2 Checking LTL properties

Require: An LTL formula ϕ and an event execution trace $Tr = \{e_1, ..., e_n\}$
Ensure: A verdict about the conformance of ϕ w.r.t. Tr
Generate a set of subformulae in BFS order $(\phi_1, ..., \phi_m)$
$/* Initialization */$
for j=m **downto** 1 **do**
  if (ϕ_j is a variable) **then**
    $next[j] := (\phi_j \in e_n)$;
  if ($\phi_j == !\phi_{j'}$) **then**
    $next[j] := (not\ next[j'])$;
  if ($\phi_j == \phi_{j_1}\ Op\ \phi_{j_2}$) **then**
    $next[j] := (next[j_1]\ op\ next[j_2])$;
  if (($\phi_j == \bigcirc\phi_{j'}$) || ($\phi_j == \Box\phi_{j'}$) || ($\phi_j == \Diamond\phi_{j'}$)) **then**
    $next[j] := next[j']$;
  if ($\phi_j == \phi_{j_1} \cup \phi_{j_2}$) **then**
    $next[j] := next[j_2]$;
/* Main loop */
for i=n-1 **downto** 1 **do**
  for j=m **downto** 1 **do**
    if (ϕ_j is a variable) **then**
      $now[j] := (\phi_j \in e_i)$;
    if ($\phi_j == !\phi_{j'}$) **then**
      $now[j] := (not\ now[j'])$;
    if ($\phi_j == \phi_{j_1}\ Op\ \phi_{j_2}$) **then**
      $now[j] := (now[j_1]\ Op\ now[j_2])$;
    if ($\phi_j == \bigcirc\phi_{j'}$) **then**
      $now[j] := next[j']$;
    if ($\phi_j == \Box\phi_{j'}$) **then**
      $now[j] := now[j'] \wedge next[j]$;
    if ($\phi_j == \Diamond\phi_{j'}$) **then**
      $now[j] := now[j'] \vee next[j]$;
    if ($\phi_j == \phi_{j_1} \cup \phi_{j_2}$) **then**
      $now[j] := now[j_2] \vee (next[j_1] \wedge next[j])$;
  $next := now$
$Verdict := next[1]$;

The analysis of this algorithm is straightforward. Its complexity is $\bigcirc(n.m)$ where n is the trace length and m is the number of subformulas generated from the LTL formula in the BFS order.

We now present our algorithm for checking an XCTL formula ϕ on a trace $Tr = \{(e_1, t_1), ..., (e_n, t_n)\}$. This algorithm is a extension of Algorithm 2 to support real time. It consists of the following steps:

- **Initialization**

  1. First, we link each variable appearing in ϕ to a table containing the set of its temporal constraints. We define therefore, the Temporal Constraints Table (TCT) such as: $TCT[var, tc_i]$ returns the i^{th} temporal constraint of the variable var;
  2. Then, we create a list ES which contains all the temporal constraints of the XCTL formula. Actually, this list represents an equation system. Initially, all temporal constraints are marked as $NOT\_INSTANTIATED$;
  3. After that, we generate the set of formulas in the BFS order of ϕ without accounting the temporal constraints parts. It results for example in a set of formulas $\{\phi_1, \phi_2, ..., \phi_m\}$.

- **Initialization loop:** This step is very similar to the initialization loop of Algorithm 2. We start by calculating the truth value of the vector $next[j]$ for $1 \leq j \leq m$, based on the last event from the trace (e_n, t_n)

  – If ϕ_j is a variable var then $next[j] \leftarrow (\phi_j \in e_n)$. Then, if this variable satisfies the current event ($next[j] \equiv true$), we instantiate its temporal constraint with the timestamp of e_n i.e t_n and we mark this temporal constraint as $INSTANTIATED$ in the equation system ES. This is due to the considered XCTL grammar presented in Definition 4.3, where we suppose that atomic propositions can only be connected to temporal constraints by conjunctions.

  – the rest of cases of ϕ_j is addressed exactly as in Algorithm 2.

- **Main loop:** The main loop is also similar to Algorithm 2 except for the case where ϕ_j is a variable that satisfies the current event trace e_i (i.e $now[j] \equiv$

true) in which case, it is addressed as above i.e we update its current temporal constraint with the timestamp t_i of corresponding to the current event.

At the end of each loop iteration, we update the final verdict based on the value of $next[1]$, which tells about the satisfiability of the tested formula without its temporal constrains and the resolvability of the equation system ES. The equation system ES is resolvable iff all temporal constraints that it contains are instantiated and that the system is correct. The detailed algorithm is given in Algorithm 3.

For illustration, let us take an example to show how this algorithm proceeds. Suppose we have an XCTL formula ϕ and a trace Tr such as:

$\phi = \Box((P \wedge T = x) \rightarrow \Diamond(Q \wedge T \leq x + 3))$
$Tr = \{(P, 5), (Q, 6)\}$

The BFS order of formula ϕ without its temporal constraints gives the following set of subformulas:

$\phi 1 = \Box(P \rightarrow \Diamond Q)$
$\phi 2 = P \rightarrow \Diamond Q$
$\phi 3 = P$
$\phi 4 = \Diamond Q$
$\phi 5 = Q$

The Temporal Constraint Tables of variables P and Q and the equation system ES are initialized as follows:

$TCT[P, 0] = \{T = x\}$
$TCT[Q, 0] = \{T \leq x + 3\}$
$ES = \{(T = x, not\_instantiated), (T \leq x + 3, not\_instantiated)\}$

Algorithm 3 Checking XCTL properties

Require: An XCTL formula ϕ and an event execution trace $Tr = \{(e_1, t_1), ..., (e_n, t_n)\}$
Ensure: A verdict about the conformance of ϕ w.r.t. Tr
Create a temporal constraint table TCT ($TCT[var, tc_i]$ returns the i^{th} temporal constraint related to variable var);
Create a list ES containing all temporal constraints (This is for the equation system);
Generate a set of subformulas in BFS order (without accounting temporal constraint parts, i.e only LTL) $(\phi_1, ..., \phi_m)$
/* Initialization */
for $j = m$ **downto** 1 **do**
  $tc := 0$; /*To access temporal constraints*/
  if (ϕ_j is a variable var) **then**
    $next[j] := (\phi_j \in e_n)$;
    if ($next[j]$) **then**
      $index := TCT[var, tc]$;
      $tc := (tc + 1)\ mod\ NbTc(var)$;
      /*NbTc(var) returns the number of temporal constraints related to variable var */
      $INSTANTIATE(ES, index, t_n)$;
      /*instantiates the current temporal constraint based on the actual time value t_n and mark it as "INSTANTIATED"*/
  ...
  /* The rest of the initialization is like in Algorithm 2 */
if (all temporal constraints in ES are instantiated) **then**
  $verdict := next[1] \wedge Resolve\_ES(ES)$;
  /*Resolve_ES(ES) returns true if the equation system is correct*/
else
  $verdict := next[1]$
if ($Resolve\_ES(ES)$) **then**
  $tc := 0$;
  $INIT(ES)$;
  /* INIT(ES) Undo all temporal constraints instantiations in ES and mark them NOT_INSTANTIATED */
/* Main loop */
for $i = n - 1$ **downto** 1 **do**
  for $j = m$ **downto** 1 **do**
    if (ϕ_j is a variable var) **then**
      $now[j] := (\phi_j \in e_i)$;
      if ($now[j]$) **then**
        $index := TCT[var, tc]$;
        $tc := (tc + 1)\ mod\ NbTc(var)$;
        $INSTANTIATE(ES, index, t_i)$;
    ...
    /* The rest of cases is like in Algorithm 2 */
  $next := now$;
  if (all temporal constraints in ES are instantiated) **then**
    $verdict := next[1] \wedge Resolve\_ES(ES)$;
  else
    $verdict := next[1]$
  if ($Resolve\_ES(ES)$) **then**
    $tc := 0$;
    $INIT(ES)$;

For the initialization loop, we would have $e_n = Q$ and $t_n = 6$. Therefore, the result will be:

$next[5] := true$
$next[4] := next[5] \equiv true$
$next[3] := false$
$next[2] := (next[3] \rightarrow next[4]) \equiv true$
$next[1] := next[2] \equiv true$

The temporal constraint of Q would be instantiated with the value of t_n. Therefore, the equation system would be updated as follows:

$ES = \{(T = x, not\_instantiated), (6 \leq x + 3, instantiated)\}$

For the main loop, we would have $j = m = 5$ and $i = n-1 = 1$ which means that the current event would be $(P, 5)$. Thus, we would update the vector now as follows:

$now[5] := false$
$now[4] := (now[5] \vee next[4]) \equiv true$
$now[3] := true$
$now[2] := (now[3] \rightarrow now[4]) \equiv true$
$now[1] := (now[2] \wedge next[1]) \equiv true$

The equations system would be:

$ES = \{(5 = x, instantiated), (6 \leq x + 3, instantiated)\}$

We can see here, that all equations are instantiated and the system is correct $((x = 5) \wedge (x \geq 3))$. Therefore, the final verdict would be : **PASS**.

$next := now$;

$verdict := (next[1] \wedge Resolve\_ES(ES)) \equiv true$

4.4.3 Correctness

We now argue that Algorithm 3 is correct with respect to the checking of an XCTL formula on a timed trace.

Theorem 4.1: *For a given trace* $Tr = \{(e_1, t_1), ..., (e_n, t_n)\}$ *and a given XCTL formula* ϕ*, Algorithm 3 issues a verdict PASS iff* $Tr \models \phi$.

Proof: Algorithm 3 is an improvement of Algorithm 2 for checking real time constraints. It follows exactly the same logic and structure of Algorithm 2 for checking the formula ϕ without its temporal parts **(1)**. The correctness of Algorithm 2 is proven in [60] **(2)**, therefore, we will focus here, on the treatment of the temporal constraints of ϕ.

According to the XCTL grammar presented in Definition 4.3, each propositional variable can be connected to a temporal constraint of the form of $T \sim ax + c$ where $\sim \in \{<, \leq, >, \geq, =\}$. Algorithm 3 starts by allocating a table TCT where it links each propositional variable from ϕ to the list of its temporal constraints. Then, each time a propositional variable from ϕ is validated on the trace Tr (according to Algorithm 2), Algorithm 3 instantiates its temporal constraint with the current timestamp from the trace and updates the table ES. The equation system table ES gathers all temporal constraints of formula ϕ. Each temporal constraints is initially marked as *not_instantiated* and is updated to *instantiated* by Algorithm 3. The instantiation consists of replacing the global time variable T of a given temporal constraints by the timestamp t_i of the trace event e_i which validates the current propositional variable. At the end, the algorithm checks if all temporal constraints in ES are instantiated and if the equation system is correct i.e: $\forall tc \in ES : tc\ is\ instantiated \wedge (\bigwedge_{i=1,n} tc_i \equiv true)$ (tc is a temporal constraint).

All temporal constraints are instantiated from the trace Tr itself (based on the satisfiability relation of Algorithm 2) and the equation system is resolved based on

these real timestamps values. Therefore, there cannot be any contradiction between the addressed temporal values and those who appear really in the trace **(3)**.

Consequently, we conclude from **(1)**, **(2)** and **(3)**, that Algorithm 3 issues a verdict **PASS** iff $Tr \models \phi$.

4.5 Real time patterns and experimental results

In this section, we present an experimental study of our approach. First, we identify a set of real time requirements which we formalize as XCTL formulas. Then, we test an implementation of the presented algorithm and evaluate its performances.

For more consistency, we propose to describe these requirements as abstract patterns specified in XCTL. In the following, we introduce four of such patterns and illustrate them with real examples.

4.5.1 Periodicity

The first pattern we identify relates to events that must be hold periodically to prevent eventual security/safety issues. Given a proposition P, we can specify the periodic occurrence of P by the following XCTL formula.

$$\Box((P \wedge T = x) \rightarrow \Diamond(P \wedge T = x + c))$$

where the constant c represents the period duration. An example of this property can be illustrated by a system which sends periodically a liveness message to inform administrators about eventual crashes.

4.5.2 Response

This pattern is usually used to specify a simple request/response paradigm. Given two propositions P and Q, the following XCTL formula specifies that each occurrence of P must be followed by Q within (resp. in exactly or after) c time units.

$$\Box((P \wedge T = x) \rightarrow \Diamond(Q \wedge T \sim x + c))$$

where $\sim \in \{<, \leq, >, \geq, =\}$. For illustration, we can specify for example that a connection establishment must not exceed 5 seconds.

$$\Box((ConnectReq \wedge T = x) \rightarrow \Diamond(ConnectResp \wedge T \leq x + 5))$$

4.5.3 Correlation

This pattern is an example of correlated temporal constraints that we are able to specify in XCTL. It corresponds to the following situation. Given three propositions P, Q and S; when P holds at a time x, it will be followed by Q at a time y and later by S which must hold within (resp. in exactly or after) $x + y$ time units. This situation can be specified by the following formula.

$$\Box((P \wedge T = x) \rightarrow \Diamond((Q \wedge T = y) \rightarrow \Diamond(S \wedge T \sim x + y)))$$

where $\sim \in \{<, \leq, >, \geq, =\}$.

4.5.4 Alternative

This last pattern is used to specify alternative situations. Given three propositions P, Q and S, the XCTL formula bellow specifies the following statement : " Q holds if S does not respond to P within (resp. in exactly or after)c time units".

$$\Box(\neg((P \wedge T = x) \rightarrow \Diamond(S \wedge T \sim x + c)) \rightarrow \Diamond Q)$$

where $\sim \in \{<, \leq, >, \geq, =\}$.

We can consider, as an example of this pattern, a reliable system where each request must be followed by a acknowledgment. In the case where no acknowledgment is received within 10 seconds, a cancellation message must be sent to abort the request.

To study the performances of our approach, we relied on these patterns to test an implementation of Algorithm 3. Experiment results are shown in figure 4.1.

In this figure, we vary the trace length and study the evolution of execution time of our algorithm with respect to the type of the used pattern. The figure represents evolution time curves of the four patterns presented above (periodicity and response

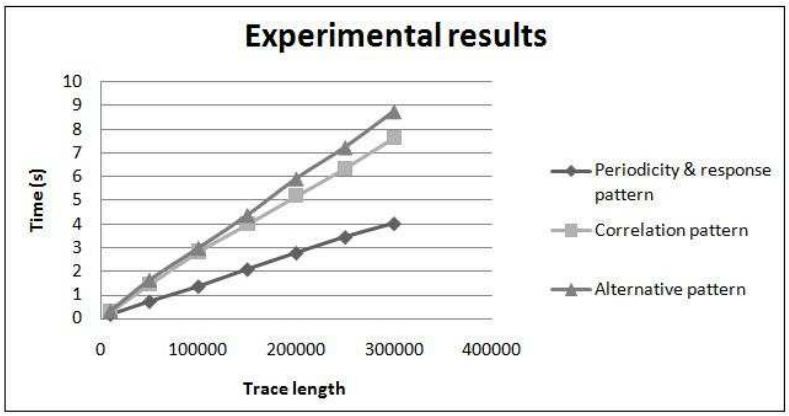

Figure 4.1: Experimental results

pattern are represented by a single curve as we consider that periodicity pattern can be seen as a particular case of response).

The three curves are growing in approximately a linear manner with a slight interval difference between them due to the complexity of the addressed pattern. Periodicity and response patterns are less complex, therefore, their curve is the lowest one. The performances shown by correlation and alternative patterns are almost the same. Alternative curve is higher because we chose a complex alternative formula (which includes a response formula), otherwise it would be much lower.

The approximative linearity of curves confirms the theoretical analysis of Algorithm 3. Indeed, in this algorithm each state (event) from the trace is visited only once but it is used to check the satisfiability of all derived formulas in BFS order. That is why the execution time is proportional to the length and the complexity of the tested formula (in addition to the trace length).

4.6 Conclusion

In this chapter, we proposed a formal approach to test real time properties specified as XCTL formulas. One of the main results we got in this work, is to be able to specify and check complex correctness properties with correlated temporal constraints i.e properties which contain temporal constraints defined with respect to other temporal constrains in the same formula.

We also tested the proposed passive testing algorithm on a set of real time patterns and discussed the obtained results. These patterns are probably not exhaustive and must be taken only as examples to illustrate the efficiency and reliability of our approach.

As future work, we are expecting to upgrade our algorithm for runtime checking so that we could deploy it as an *online* monitor. This way we could detect violations as soon as they happen and thus, avoid eventual attacks and/or dangerous scenarios.

Chapter 5

A Complementary Approach for Testing System Robustness

Contents

5.1 Introduction

The increasing complexity of current software system requires more rigorous testing and validation techniques as any failure of such systems may lead to catastrophic financial or human consequences.

The main purpose of the various existing testing techniques is to find defects on system implementations. Formal methods for conformance testing, for example, have been widely used to test distributed system and communication protocols.

These techniques aim at providing a verdict about the conformance of a system implementation with respect to its formal behavior specification when the system is running in its normal (proper) environment. Approaches for conformance testing can be either active or passive. In active testing, the tester interacts directly with the implementation under test (IUT). It provides inputs and collects the returned outputs which it analysis to issue the conformance verdict. In passive testing however, the tester does not interact directly with the tested system. It only observes its behavior (as execution traces) and verifies its conformance with respect to a given formal specification.

A software system may behave correctly when running in its proper environmental conditions. However, if the system environment is disturbed by external or non-expected events, the system behavior may be abnormal and unpredictable. This non-expected behavior can reveal many system failures and dangerous scenarios. Therefore it is very important for a tester to study this kind of situations, particularly for critical systems and applications.

Testing the behavior of a system running in stressful environmental conditions is known as *Robustness Testing*. At the opposite of conformance testing, robustness testing techniques consider that the tested system is running in an hostile environment. Therefore, they do not look for a correct behavior but an acceptable one [57]. The acceptable behavior can be assessed either empirically (the system does not crash or hang for example) or formally (robustness requirements are formally specified and checked against the system).

In this chapter, we propose a complementary approach for testing system robustness based on passive testing and fault injection techniques. We use fault injection as a perturbation mechanism to create stressful environmental conditions. Then, we rely on a passive testing technique to check the satisfiability of robustness requirements on system execution traces. The injected faults and the robustness properties are formally specified. The specification of the injected faults is used to validate the injection process and the specification of robustness requirements is to formally asses the system robustness.

5.2 Related work

As we presented in section 2.2, robustness testing approaches can be categorized into two classes: fault injection approaches and model-based approaches.

Fault injection approaches are based on deliberate introduction of errors in a running system and an observation of its behavior. Such techniques are very useful for simulating hostile environments as they can inject various kind of faults (interface faults, communication faults, etc.). There exist several fault injection tools for different kind of systems [62, 61, 65, 48]. In section 2.2.1, we gave an overview of the most relevant ones for distributed systems.

The major issue with the existing fault injection techniques, is that they do not rely on any efficient test oracle. The evaluation of system robustness is based simply on basic observations. Faults are injected during system execution and if the system does not crash or hang, it is considered as robust. Also, the injection process is not controlled. The injected faults are usually not formally specified and there is actually no way to validate the injection i.e. to ensure that the injector really injects the faults that it is supposed to inject (Chapter 3).

Model-based techniques for robustness testing proceed differently. They inspire from conformance testing approaches and particularly from active testing. The basic idea is to introduce a fault dimension in the input domain of traditional conformance active testing approaches. This way, it would be possible to generate faulty inputs which can eventually lead to system failures. This kind of techniques for robustness testing is relatively recent. We exposed in section 2.2.2, the most relevant ones.

Probably, the greatest advantage of model-based techniques is their formal aspect. At the opposite of fault injection techniques, model-based approaches formalize the injected faults as well as the expected robust behavior. This way, robustness testing experiments are completely controlled. Therefore, there is actually no possibility to issue incorrect verdicts. Also, as one can specify formally the robust behavior, results analysis is much deeper than a simple observation of a crash or a hang. In fact, with model-based approaches, one can specify a set of robustness requirements to verify. This is very important because some system failures may

not be revealed as a crash or a hang. They could be for example, a violation of critical safety and/or a liveness requirements.

However, model-based approaches suffer from two main shortcomings. First, the set of injected fault is related to the nominal input domain. In fact, faults are created as mutants of the input symbols from the original system specification. Thus, the set of faults is limited by the set of mutants that can be generated and depend strongly on the used specification formalism. For example, if we rely on a non temporal specification formalism to describe the system behavior, we would be unable to generate temporal faults.

The second issue with model-based approaches is due to the active testing architecture on which they rely on. As far as we know, all existing model-based approaches for testing system robustness follow the active testing architecture which imposes direct interactions with the tested system. This architecture presents some limitations when the tested system is built upon a set of components that could not be accessed directly. In this case, it is difficult to inject faults or to disturb communication between these different components of the tested system.

The approach we propose in this chapter is an hybrid approach combining fault injection and formal techniques. This way, we can take advantage of fault injection technique which we use to inject faults simultaneously on different application components and rely on formal passive testing as a test oracle to analyze the global system behavior.

5.3 Proposed approach

We introduce in this section our robustness testing approach. Its general architecture is presented in figure 5.1.

We can see in this figure that the robustness testing process involves three main stages. The first step (figure 5.1 (a)) focuses on experimentations. During this phase, faults are injected while the system under test (SUT) is running and execution traces of both the fault injector and the SUT are collected.

In the second step (figure 5.1 (b)), we verify the injection process. The exe-

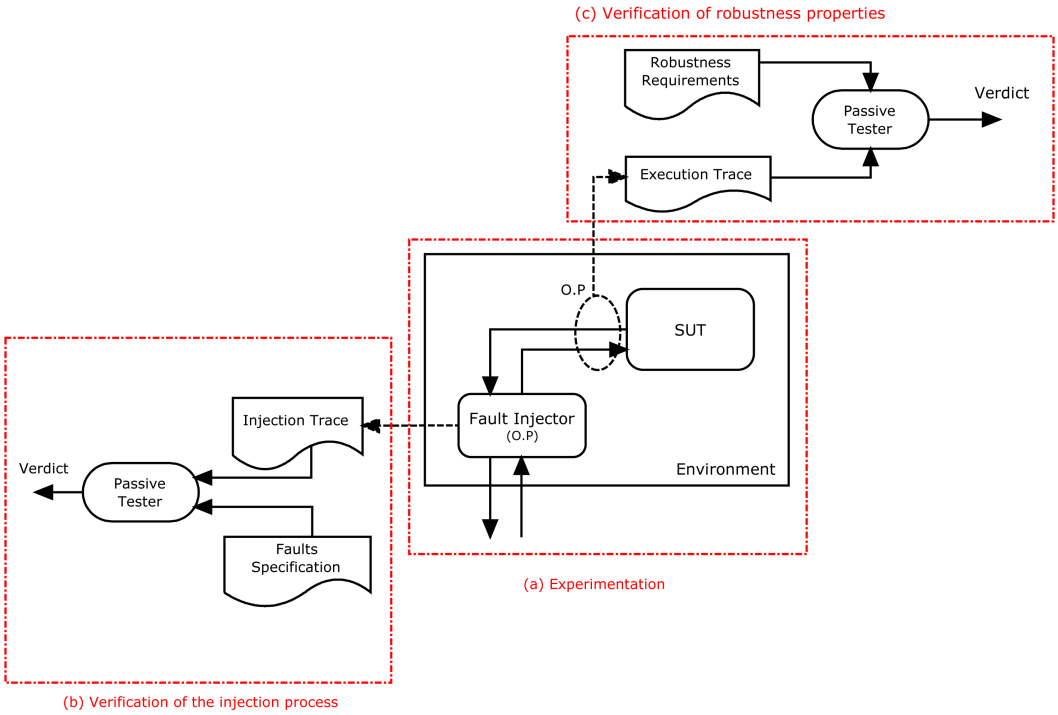

Figure 5.1: Architecture of the proposed robustness testing approach

cution trace of the fault injector is verified against the formal specification of the injected faults and a conformance verdict is issued. This step tells whether the injection process has been well performed i.e. if all specified faults have been correctly injected.

Finally, the last step (figure 5.1 (c)) concerns the verification of robustness requirements. In this step, we rely on passive testing to issue a verdict about the conformance of the collected SUT's execution trace with respect to the provided formal specification of robustness requirements. In what follows, we detail each of these steps.

5.3.1 Experimentation phase

We introduce a fault injection mechanism into the SUT environment to simulate stressful environmental conditions. The fault injection tool should be able to intercept all messages exchanged between the SUT and its external environment. It

represents, in fact, the *faultload* entity which is responsible for the generation and the injection of different kind of faults. In the case of a distributed system, the external environment of the SUT could be any communication partner such as a client application, a system component or any other entity that could stimulate the SUT. This entity represents the main source of the *workload* in our testing architecture.

The experiment consists to run simultaneously the fault injector and the SUT. According to a pre-specified injection campaign, the fault injector will intercept and corrupt some of the exchanged messages. The way the injection campaign is specified is usually proper to the used fault injector. Some tools are *script-driven* i.e. faults are specified using a dedicated script language while other ones are more user friendly providing a GUI (Graphical User Interface) to help the tester to create its injection campaign.

This difference in the way fault campaigns are specified brings us to propose a formal and a *tool-independent* specification language for fault description. Thus, in addition to the tool-specific description of the injection campaign, one needs to provide its equivalent using a formal language. This formal specification of faults will then be used to verify the injection process as it is explained in chapter 3.

During the experimentations execution, we collect traces from both the SUT and the fault injector. We define for that a set of Observation Points (O.P) at different application levels. As we have discussed it in section 3.4, the observation points for the fault injector must be defined inside the injection tool and not at its interface level. This is important because we need, for the verification of the injection process, not only information about the states of messages before and after the injection, but also the injection operations that were executed. In the case where we rely on a third-party fault injector which does not offer any possibility to insert observation points, we can simply use its log files as an injection trace. As far as we know, all of the most relevant existing fault injector provides such traces.

For the SUT, the observation points are implemented at interface level as shown in figure 5.1 (a). This way, we are able to collect a trace of all input/output messages of the SUT. This configuration is usually the most proper one for several types

of applications. However, in the case of a distributed system, it would be also interesting to collect a trace from the external communication partners to have a global view of the system behavior (figure 5.2). What is important in both those configurations, is that erroneous messages must also appear in the collected trace as they are important for robustness evaluation.

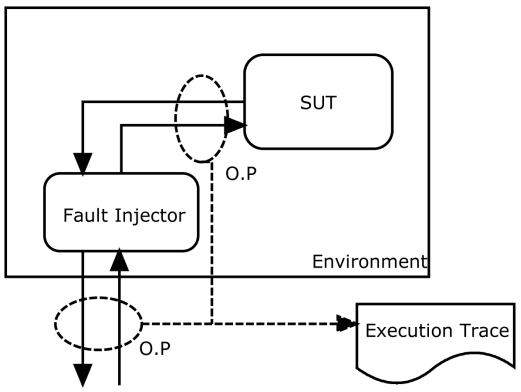

Figure 5.2: Observation points for distributed systems

5.3.2 Verification of the injection process

As we have already motivated it in section 3.4, it is important to verify, after each injection experiment, that the injection process has been correctly performed. This is due to the fact that the fault injector is an external mechanism that we introduce into the SUT environment to disrupt its behavior. The robustness of the SUT is then evaluated based on how the tested system reacts to the injected faults. Therefore, any failure in the behavior of the fault injector can seriously affect the robustness analysis and may lead to an erroneous verdict. Suppose for example that we are testing a communication protocol using a fault injector that we configured to delete some specific messages. Then, after the test execution, how can we be sure that the lost messages have been effectively deleted by the fault injector and not lost due to a protocol vulnerability or a system failure? This confusion can be omitted only if we have a mean to verify the good execution of the performed injection actions.

For our robustness technique, we propose to rely on the formal approach we proposed in chapter 3 to verify the good execution of the injection process. Thus,

we propose to formalize the set of the injection operations that we want to inject as a set of Hoare triples [42]. Then, we use the proposed passive testing algorithm to check the conformance of this formal specification against the injection trace that we collect during the experiment (Algorithm 1). This way, we can avoid any verdict ambiguity due to an eventual erroneous injection behavior.

5.3.3 Verification of robustness requirements

We define robustness requirements as the set of properties that the tested system must satisfy when running in stressful conditions. Some model-based approaches consider these properties as a subset or a variant (mutants) of the nominal functional model of the tested system [37, 40] while others, like in [46], propose to formalize the robustness observation model independently from the behavioral model.

In our approach, we will also consider that robustness requirements can be independent from the nominal functional ones, as we believe that critical systems may behave quite differently when they are disrupted. Nevertheless, we accept that in some situations, the robust behavior could be a variant of the functional one. For example, a nominal functional property of a server application is to response the received requests within a relatively short period of time. In abnormal environmental conditions however, the server could be configured to react differently. For instance, to avoid a server crash, the administrators can configure the server to close all its external connections when it receives a huge number of requests within a very short time interval. This could be seen as a robustness property.

In [40], authors used timed automata for modeling both the *nominal* and the *degraded* behavior of the tested systems; while in [37], the authors relied on the Input Output Labeled Transition System (IOLTS) to model the *nominal* and the *increased* specification (chapter 2). Timed automata and IOLTS are both very known formalisms for the specification of functional properties. Therefore, it is quite understandable that if we consider robustness requirements as different from the functional ones, we need to rely on another specification formalism. In [46] for example, the authors propose to specify each robustness requirement as an LTL

formula. The set of all robustness requirements is then represented as a Rabin automaton [58] such that the language generated by this automaton represents the robust behavior.

LTL is a very suitable formalism for the specification of *safety* and *liveness* properties. Safety and liveness are both very important requirements for any critical system. A safety property specifies that something bad never happen while a liveness property specifies that something good will eventually happen.

We believe that robustness requirements can be specified as *safety* and *liveness* properties. A safety robustness requirement describes how the robust system must avoid a dangerous scenario and a liveness robustness property specifies how the system must react to a stressful situation. Therefore, we propose for our approach, to model robustness requirements as a set of safety and liveness properties.

However, as we mentioned in chapter 4, LTL is not expressive enough to model complex requirements. We saw that several extensions have been proposed to evolve LTL expressiveness and we argued about the expressiveness of XCTL. Therefore, in our approach, we will rely on XCTL as a mathematical formalism for modeling robustness properties. We specify robustness requirements of the tested system as a set of XCTL formulas according to the grammar defined in Definition 4.3. Then, we use Algorithm 3 to check the conformance of such formulas against the collected execution trace.

5.4 Conclusion

We presented in this chapter a complementary approach for checking system robustness. Our approach uses fault injection and passive testing techniques to assess the ability of a given system to behave correctly in presence of faults.

The robustness testing technique we proposed, takes advantages from both fault injection and model-based approaches. The use of fault injection allows one to define a huge set of faults independently from the behavioral model of the tested system. On the other hand, relying on formal specification and passive testing help the testers to verify the good execution of the injection process and to evaluate the

robustness of their system.

In the same way, the proposed approach avoids some weaknesses of fault injection and model-based techniques. By providing a test oracle, we can formally assess the robustness requirements of the tested system instead of just an empirical evaluation of the injection results. Also, by using fault injection techniques, we are able to inject a larger set of faults and thus, we are not limited by the behavioral model of the SUT.

Chapter 6

A Framework for Modeling and Testing Web Services Robustness

Contents

6.1 Introduction

Web services are becoming increasingly widespread technology and tend to emerge as a standard paradigm for program-to-program interactions over Internet. The strength of this technology comes probably from its ability to manage communication between heterogeneous applications and systems with a dramatically lower cost. Consequently, Web services have been widely used for building all kind of distributed systems for different areas: business, multimedia, security, etc.

However, these inherent and powerful characteristics of Web services (widely distributed and heterogeneous applications) are paradoxically, also their main weakness points. This is due primarily to the problem of reusing and integrating older and/or third-party service components which may lead to several interoperability, security and/or performance issues.

Testing Web services is therefore, a very important process which has to be performed, not only during the development of new Web service applications, but also before and after deployment.

In this chapter, we propose a framework for modeling and testing robustness requirements of Web services. It is actually an instantiation of the robustness testing approach proposed in the previous chapter, adapted for Web services. The framework we propose here can be used to test both composed and single services. It includes an innovative fault injection tool (WSInject) and uses a monitoring approach based on passive testing for checking robustness requirements. Also, our framework can be used to test both experimental and real world services as it does not require the source code of the tested system (black box testing).

6.2 Web services technology

In this section, we will present the Web services technology and the Service Oriented Architecture. We will describe the main standard protocols used by those technologies and introduce to the most widespread services composition techniques: service orchestration and service choreography.

6.2.1 Service Oriented Architecture

Service Oriented Architecture (SOA) [34, 33] is a software architectural paradigm that aims to achieve loose coupling among interacting software agents. The goal is to allow organizing and utilizing distributed capabilities that may be under the control of different ownership domains and implemented using various technology stacks. An SOA architecture allows the use of existing service applications as well as the deployment of new service components. The deployed services can be used either by other services (composed services) or client applications.

Figure 6.1 shows the functional process of an SOA architecture. The service providers publish their hosted services in a service directory. This directory can be then accessed by users (other services or client applications) looking for services that verify a set of specific criteria or correspond to a certain description. If the service directory finds the requested services, it sends back the service contracts (containing all the needed information to exploit the services) to the client which can then, select the desired services and invoke the respective providers.

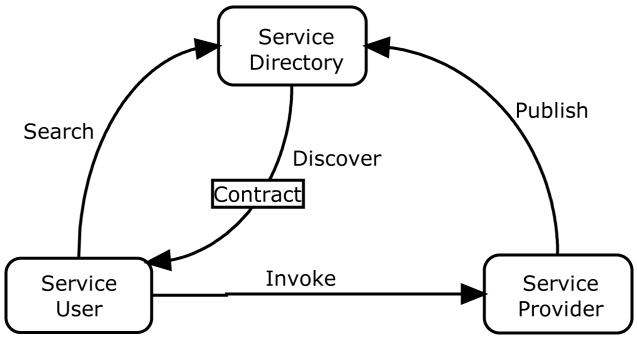

Figure 6.1: Functional model of an SOA architecture

Web services are actually the most important achievement of the SOA architecture. The reason is that, they can be easily composed to build new applications. Furthermore, a Web service can invoke other Web services as it can be invoked by other services and a service composition can be deployed as a Web service.

6.2.2 Web services

The World Wide Web Consortium (W3C) [1] defines a Web service as :" *a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards*".

In other words, Web services are application components deployed through the Internet which can communicate between each other without worrying about the platforms on which they are running neither about the programming languages that were used to build them. They rely on a set of standard Web technologies based on XML data structuring: SOAP protocol for message exchanges, WSDL for service description, UDDI for service discovering and BPEL for service orchestration. The Web services model is illustrated in figure 6.2. It is in fact an instantiation of the SOA architecture presented in figure 6.1, for Web services.

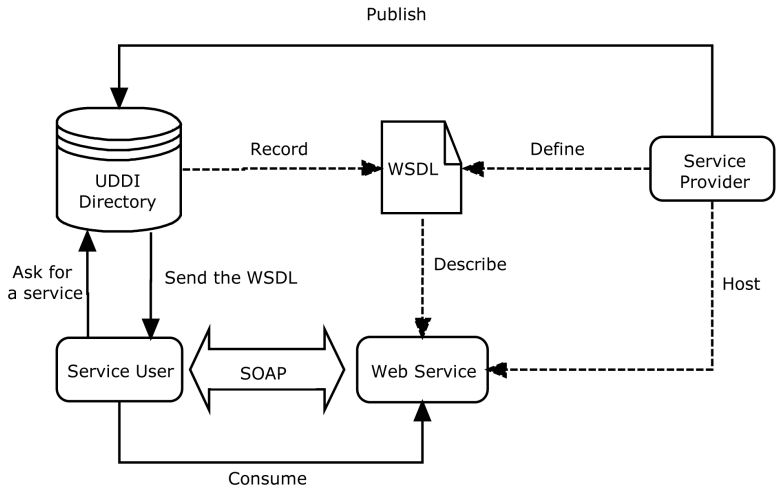

Figure 6.2: Web services model

In the following, we present the set of standard Web technologies used by Web services.

[1] www.w3.org

XML

The **Extensible Markup Language (XML)** [4] is a set of rules for encoding documents in a textual form. It has been defined by the W3C and can be used to format message exchanged between different kind of applications. For Web services, we rely mostly on XML schema [5] for describing data structure.

HTTP

The **Hypertext Transfer Protocol (HTTP)** [6] is a networking protocol for distributed information systems. It is the foundation of data communication for the Web. In the case of Web services, it is used to forward the exchanged messages.

WSDL

The **Web Services Description Language (WSDL)** [7] is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.

A WSDL document defines **services** as collections of network endpoints, or **ports**. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: **messages**, which are abstract descriptions of the data being exchanged, and **port types** which are abstract collections of **operations**. The concrete protocol and data format specifications for a particular port type constitutes a reusable **binding**. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Hence, a WSDL document uses the following elements in the definition of network services:

- **Types:** a container for data type definitions.
- **Message:** an abstract, typed definition of the data being communicated.
- **Operation:** an abstract description of an action supported by the service.
- **Port Type:** an abstract set of operations supported by one or more endpoints.

- **Binding:** a concrete protocol and data format specification for a particular port type.
- **Port:** a single endpoint defined as a combination of a binding and a network address.
- **Service:** a collection of related endpoints.

SOAP

The **Simple Object Access Protocol (SOAP)** [8] is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics. A SOAP messages is divided into two parts: the SOAP header which can be used to specify authentication and other session management data, and the SOAP body where operation names and parameters are specified (figure 6.3).

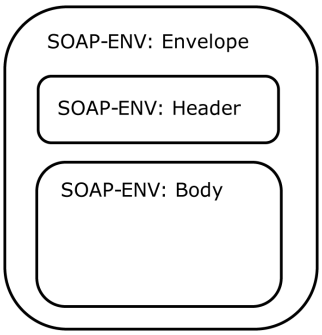

Figure 6.3: SOAP message structure

UDDI

The **Universal Description, Discovery and Integration (UDDI)** [9] is a platform-independent, XML-based registry. It has been designed to be interrogated by SOAP messages to provide access to WSDL documents describing the protocol

bindings and message formats required to interact with the set of web services listed in its directory.

6.2.3 Web services composition

Web services are considered as self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. In the case where no single Web service is able to satisfy the functionality required by the user, there is a possibility to combine existing services together in order to fulfill the request. The result of this combination is called a *service composition* and it can be deployed as a new Web service.

A Web service composition can be organized either as an orchestration or as a choreography. A Web service orchestration describes the way Web services can interact together. An orchestration defines particularly the message sequences and the system workflow of the composition and there is always a main process (the orchestrator) which is in charge of managing and controlling all interactions between the services of the composition (the service partners). The **Business Process Execution Language (BPEL)** [10] is the most known standard language for defining Web service orchestrations.

Web services choreography describes also a services collaboration. At the opposite of an orchestration, in a service choreography there is no main process. It is a decentralized coordination where each service partner is responsible of a part of the workflow.

BPEL

The BPEL language has become a standard language for implementing Web services orchestrations. It has been widely used for building service oriented architectures. The BPEL language allows one to describe both the behavioral interface as well as the services orchestration.

- The behavioral interface defines an abstract process describing the message exchanges between service partners.

- The orchestration defines an executable process (the BPEL process) which specifies the types and the order of the messages exchanged between service partners.

Compared to other existing orchestration languages, BPEL offers the following features:

- Exception handling (particularly, fault and event exceptions).
- Handling synchronous flows and parallel execution of activities.
- Possibility to describe stateful transactions.
- Handling message correlation.
- Compensation support. A compensation consists to undo some steps in the process that has been already completed successfully. BPEL offers a relatively easy way to perform this kind of operations.

A BPEL process is directly executable by a BPEL orchestration engine like activeBPEL [11] or Oracle BPEL Process Manager [12]. The deployment and the publication of a BPEL process is performed as for any other Web services, using WSDL. Thus, operations, data and bindings of the BPEL process are all described, as well as all the needed elements for interacting with its service partners like their addresses, the used communication protocol, the available operations, etc.

The BPEL language handles also other Web services standards as :

- **WS-Addressing** [13] which provides transport-neutral mechanisms for forwarding SOAP messages in both synchronous and asynchronous mode.
- **WS-Policy** [14] which is an extension of WSDL supporting description of some functional aspect of service partners.
- **WS-Security** [15] which is a SOAP extension for securing message exchanges.
- **WS-ReliableMessaging** [16] which describes a protocol that allows SOAP messages to be reliably delivered between service partners in the presence of software, component, system, or network failures.

- **WS-Transactions** [17] which defines interoperable mechanisms that allow transactions between different service domains.

Figure 6.4 depicts the Web services architecture stacks.

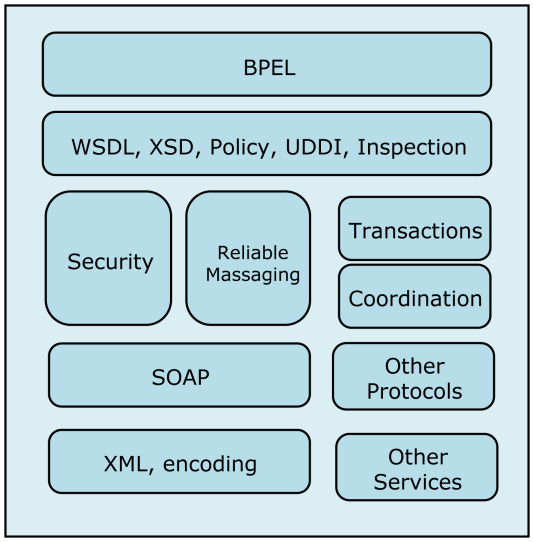

Figure 6.4: BPEL in the Web services architecture stack

The description of a BPEL process contains four main parts: (i) declaration of variables using types described or imported from the WSDL interface, (ii) description of service partners, (iii) specification of fault handlers and (iv) the main activity describing the process behavior.

6.3 Instantiation of the robustness approach for Web services

In this section we present an instantiation of the proposed robustness testing approach for Web services. Figure 6.5 illustrates the architecture of our robustness testing framework.

We can see in this figure the use of a Web service fault injector (WSInject [36]) that we have developed for our testing platform. A detailed description of this tool is presented in section 6.4. This tool is used to intercept and possibly modify all communication messages exchanged between a Web service and its client application

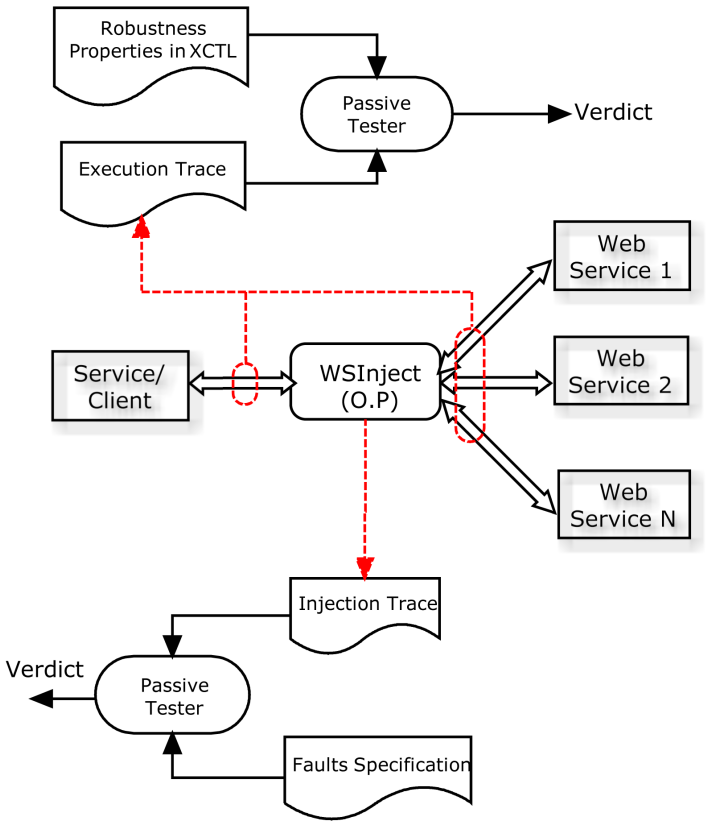

Figure 6.5: A framework for testing Web services robustness

or between a main service (a BPEL process for example) and its service partners. Therefore, this framework could be used either to test single or composed services.

A set of observation points is implemented between the client application (or the main service) and the fault injector as well as between the fault injector and the rest of services of the composition. The execution traces collected by these observation points are then aggregated following a strict sequential order (based on timestamps of event occurrences) to build a global trace. This later will be used to check the conformance of the robustness requirements specified as a set of XCTL formulas.

WSInject also provides an injection trace which contains information about all intercepted messages, the injection operations that were executed and the forwarded messages. This trace will be used to verify the injection process against the faults specification given as a set of temporal Hoare triples.

The testing framework follows the black-box testing approach. Therefore, it

relies essentially on SOAP messages exchanges between the components of the tested system as they are the only observable events. This means also that all robustness requirements as well as the injected faults must be specified at the SOAP level.

6.3.1 Specification of robustness requirements

We propose here to specify Web services robustness properties as XCTL formulas. As we are focusing on communication messages and because SOAP messages can carry both procedure calls (operations) and data, we specify each event from the trace as a SOAP operation with its expected parameter values according to the following syntax.

$$OperationName(BooleanExpression(Parameter_1), ..., BooleanExpression(Parameter_n))$$

For example, we can a specify a login request of a user *Bob* as follows:

$$Login(username = "Bob") \ [2]$$

Where *username* is a parameter name and "*Bob*" is a possible value. As a response, the invoked service may send a login notification which we specify as follows.

$$LoginResponse(username = "Bob", state = "CONNECTED")$$

We will consider that this kind of expression constitutes an atomic proposition. Therefore, in the implementation of Algorithm 3 for Web services, the satisfiability of $\phi_j \in e_i$ is validated by checking on the trace that the current event corresponds to the operation specified in ϕ_j with the appropriate parameter values.

For illustration, we will take an example of a Web services orchestration and specify some robustness requirements. The scenario is an example of a heater controlling system which deploys three Web services: the *HeaterCmd*, the *Thermocouple* and the *HeaterController*. These Web services can be seen as interfaces of real hardware devices used to control and monitor a Heater Coil as illustrated in figure 6.6.

[2] Here, we specify only important information for our test purposes. For example, if we do not need to know the used password, we do not specify it.

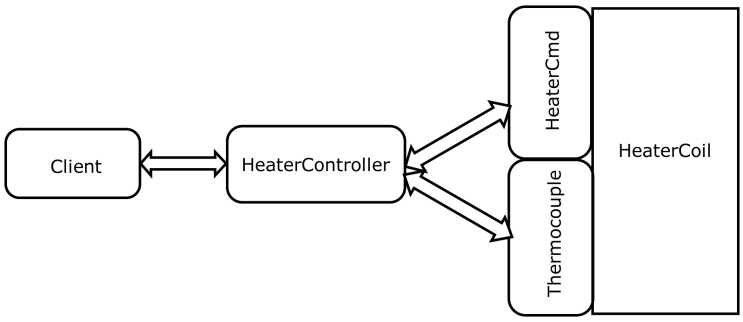

Figure 6.6: An example of a Web services orchestration scenario

The *HeaterCmd* service allows power to be applied in small increments via two operations: *incPower* and *decPower*. The *Thermocouple* allows current temperature to be read back via the *getHeaterTemp* operation. There is also a client application (*Client*) which interacts with the main service *HeaterController* via its two operations : *getTemp* and *setTemp*. The first operation (*getTemp*) returns the current temperature (by invoking operation *getHeaterTemp*) while the second one, *setTemp*, uses a time-based algorithm that invokes *incPower* and *decPower* operations provided by the *HeaterCmd* to set the correct power level. The current temperature is monitored by *HeaterController* to provide feedback into the algorithm.

We can summarize the general behavior of this system as follows:
The client application is deployed as a monitor which periodically asks for the current temperature (*getTemp*). The heater coil temperature value must always be between a minimum and a maximum threshold. Otherwise, the client invokes operation *setTemp* to readjust it to an average value (this value is specified as a parameter of operation *setTemp*). In that case, the *HeaterController*, uses its time-based algorithm to gradually regulate the temperature to its average value by invoking operations *incPower* and *decPower*.

Based on this scenario, we can define a set of robustness requirements to describe critical safety and liveness properties. In the following, we give examples of such properties specified as XCTL formulas.

Rule 1: The client application must ask for the current temperature each 10

seconds (Periodicity).

$$\Box((getTemp() \wedge T = x) \rightarrow \Diamond(getTemp() \wedge T = x + 10000))$$

We suppose here and in the following that time units are expressed in milliseconds.

Rule 2: The client must receive a response to its request within the following 5 seconds.

$$\Box((getTemp() \wedge T = x) \rightarrow \Diamond(getTempResponse() \wedge T <= x + 5000))$$

Rule 3: When the temperature exceeds 150°C, the client application must, within the following 5 seconds, ask the *HeaterController* to readjust it to 100°C.

$$\Box((getTempResponse(return >= 150) \wedge T = x) \rightarrow \Diamond(setTemp(Tmp = 100) \wedge T <= x + 5000))$$ [3]

6.3.2 Specification of the injection process

WSInject is a SOAP level fault injector. This means that all implemented injection operations concern only SOAP messages. We have already shown in chapter 3 how we can use a temporal extension of Hoare logic to specify formally fault operations. The same formalism can be instantiated for SOAP messages as follows.

A SOAP message can be considered as a set of XML elements.

$$SoapMsg = \{XML\_elt_1, XML\_elt_2, ..., XML\_elt_n\}$$

Therefore, we can specify each injection operation as a Hoare triple as follows:

$$\{P(SoapMsg)\}\ OperationName(Param_1, ..., Param_n)\ \{Q(SoapMsg)\}$$

Where $P(SoapMsg)$ is a precondition on the intercepted message and $Q(SoapMsg)$ is the postcondition.

[3] "*return*" specifies the returned value.

Each XML element from the SOAP message can be accessed using a path structure. For example $SoapMsg.LoginRequest.username$ denotes the parameter *username* of the operation *LoginRequest* carried by the captured SOAP message $SoapMsg$.

To verify the injection process, we built a passive tester prototype which implements an instantiation of Algorithm 1 for Web services i.e. it addresses only SOAP messages. We use this tester to check the conformance of the injection trace file (collected during the injection experiment) against the specified set of injection rules. These injection rules are specified following a script grammar inspired from the specification language proposed in section 3.2.4. For example, if we consider the Web service orchestration scenario presented in the previous subsection, we can specify the following injections:

Injection rule 1: Delay the forwarding of all temperature requests for 10 seconds.

```
{SoapMsg.has(getTemp) and $val==now}
delay(10000)
{new(SoapMsg).equals(SoapMsg) and $val+10000<=now<=$val+10050}
```
[4]

Injection rule 2: Each time the client invokes operation *setTemp()*, delete the message content and forward an empty message.

```
{SoapMsg.has(setTemp)} empty() {new(SoapMsg).isEmpty()}
```

Note:

In section 3.3, we presented a set of examples to illustrate the use of our fault injection specification formalism. Those examples were specified using a high level abstract language where the injection operation names were given just as matter of examples. In practice however, we will specify the injection operations following exactly the same syntax provided by the used fault injector. Thus, in the injection rules specified above, we used the syntax of injection operations defined by the Web service fault injector (WSInject) on which we will rely for our experimentations. The next section gives a detailed presentation of this tool.

[4] Words preeded by a $ define variables and time values are specified in milliseconds

6.4 WSInject

In this section we present WSInject [36]. A fault injection tool for Web services that we have developed and integrated in our testing framework.

WSInject is a script-driven fault injector able to inject both interface and communication faults. Unlike other existing Web services fault injectors, WSInject allows users to combine several types of fault in one injection statement and is able to handle either single or composed services.

6.4.1 Motivation

In the case of Web services, faults can be injected at both interface and communication levels. Interface faults affect operations input/output parameters and other SOAP message fields by corrupting data or assigning invalid parameter values. On the other hand, communication faults consider SOAP messages as black boxes. Instead of corrupting carried data, SOAP messages are replicated, deleted or delayed.

The existing fault injection tools for Web service can be categorized into two main classes. First, we find all network level fault injectors which were not originally developed for Web services but which could be very useful for injecting communication faults. Doctor(integrateD sOftware fault injeCTiOn enviRonment) [62], Orchestra [61] and DEFINE [48] are all good examples of such injectors which fit perfectly on Web services.

However, as communication faults are not enough for testing Web service dependability, other researches focused on providing injection tools able to decode SOAP messages so that they can inject significant interface faults. This constitutes the second fault injector class: Web services fault injectors.

Although there exist several Web service fault injectors able to decode and corrupt SOAP messages (WSBang [18], PUPPET [24], GENESIS [49],etc.), only a very small subset of them can inject both interface and communication faults. In fact, tools like WSBang, PUPPET and GENESIS are more like active testers or client-side injectors than real network level fault injection mechanisms. They all proceed like a client application which consumes the tested Web service (figure 6.7). They

parse the WSDL file provided by the tested service and generate a set of test suites. Each test suite is a set of sequential invocations of the Web service operations. The main difference compared to active testing tools is the fault injection step. Before invoking the tested service, faults are injected inside the SOAP messages to corrupt carried data.

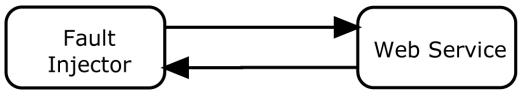

Figure 6.7: A client-side fault injection architecture

Actually, this kind of tools suffers from two main drawbacks. First, they can only inject interface faults by corrupting data and procedure parameters inside SOAP messages (communication faults such as message delaying or message deletion cannot be performed).

The second problem concerns the type of tests that can be conducted. As such tools proceed by simulating service clients, only simple Web services can be tested. The fault injector needs to consume the tested service. Therefore, it is impossible to use it for testing composed Web services (testing communication between service partners) or to test communication between a service and its original client application (as it will be substituted by the injector itself).

To address these problems, we need to rely on a fault injector mechanism which could intercept communication messages exchanged between service partners or between a service and its client application.

As far as we know, WS-FIT [50] is currently the only Web service fault injector which really fits to this architecture. However, WS-FIT needs to implement a set of *hooks* and *triggers* at the SOAP protocol layers of every machine hosting one or more tested services (figure 6.8). This approach is very useful when testing secure SOAP communications where all messages are signed and/or encrypted. In this case, the implemented hooks and triggers are used to intercept messages just before their encryption or signature, to be able to inject significant errors. However, there is absolutely no need to modify the protocol layers when testing unsecured

communication because this approach is very intrusive and can, unintentionally, disrupt the communication.

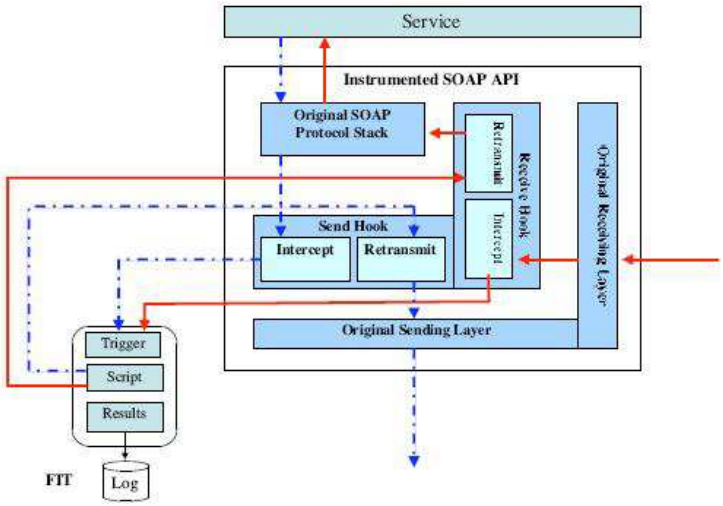

Figure 6.8: WS-FIT architecture

Moreover, WS-FIT can only be used to test Web services deployed in a completely controlled testing environment (because we need to modify the SOAP protocol layers). Thus, we cannot rely on this tool to test real world Web services i.e. Web services deployed by a third-party and running in their own environment.

For all these reasons, we propose WSInject: a Web service fault injector able to inject both communication and interface faults while being completely independent from the environments of the tested services. WSInject can test composed and simple Web services regardless whether they are running on real world or on a testing environment.

6.4.2 Tool presentation

Figure 6.9 depicts WSInject architecture, designed to be simple and loosely coupled.

Core WSInject components are **Proxy/Monitor** and **Fault Injection Executor**. **Proxy/Monitor** is the SOAP messages interception and failure monitoring point. **Fault Injection Executor** is the point where effective fault injection occurs. Other important components are **Controller**, **Script Compiler** and **Graphical**

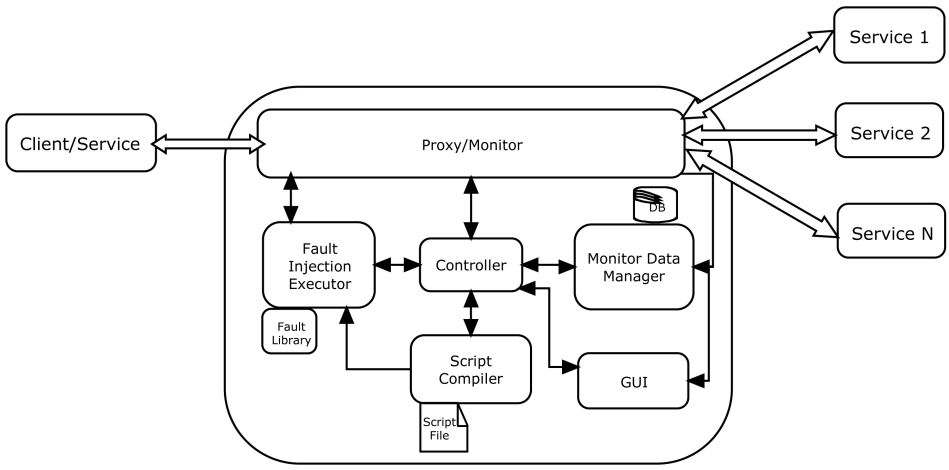

Figure 6.9: WSInject architecture

User Interface (GUI). WSInject was fully developed in Java.

Controller is the starting point of the tool; it activates and starts other components. **Script Compiler** is the component that reads a fault injection campaign script and converts it into a processable format and **GUI** is responsible for showing data collected by the **Proxy/Monitor**. All these components are explained below in more details.

Proxy/Monitor

Proxy/Monitor is a 2-in-1 component that intercepts SOAP messages and monitors system behavior. User is able to select the port on which the proxy should be bound to. Services of the composition should then be configured to connect through a proxy on the selected port and on the IP address of the machine where WSInject is running. If the tested system presents any kind of failure (like crashing for example), Proxy/Monitor will keep track of this behavior.

More specifically, Proxy is a socket-based HTTP proxy, implemented using the java.net.Socket and java.net.ServerSocket classes. It intercepts every HTTP message exchanged by Web service partners, parses it, sends it to the Fault Injection Executor, receives the (possibly) modified message and finally sends it to its original destination. Non-SOAP HTTP messages are also intercepted, but these suffer no modification before being redirected to their original destination.

Script Compiler

Fault injection campaigns are described by scripts. Script Compiler is the component responsible for compiling a script and transforming it into a **CampaignDescriptor**. A CampaignDescriptor is an Abstract Syntax Tree (AST) that is WSInject's internal representation of a script. It is part of the Fault Injection Executor component, more thoroughly explained later.

Scripts are simple text files containing one or more **FaultInjectionStatements**. FaultInjectionStatements are composed of a **ConditionSet** and a **FaultList**. A ConditionSet consists of one or more Conditions and a FaultList is composed of one or more Faults. FaultInjectionStatements work as condition-action statements: when a message arrives, if it matches a set of conditions, a list of faults is injected on it. Conditions are similar to boolean methods and faults are similar to void methods. Conditions have no defined order -hence being grouped in a set; faults do have a defined order -hence being grouped in a list. An abstract and simplified grammar of the script language is given in Figure 6.10.

```
CampaignDescriptor --> FaultInjectionStatement [CampaignDescriptor ]
FaultInjectionStatement --> ConditionSet : FaultList ; [FaultInjectionStatement ]
ConditionSet --> Condition [&& ConditionSet ]
Condition --> operation(String) | contains(String) | uri(String) | isRequest () |
isResponse ()
FaultList --> Fault [, FaultList ]
Fault --> delay(Integer) | multiply(String, Integer) | stringCorrupt (String, String) |
xPathCorrupt (String, String) | empty() || closeConnection ()
```

Figure 6.10: Script language grammar

Table 6.1 presents available conditions and Table 6.2 presents available faults to be injected (or "actions" to be taken). *Name/Class* is both the name of that condition or fault and its corresponding Java class on WSInject code. Syntax describes how that condition or fault is expressed on the script language.

Interface faults modify contents of SOAP messages, while communication faults affect the delivery of requests and/or responses. To emulate a message modification, user should simply choose the most appropriate interface fault for his/her needs.

Name	Syntax	Description
ContainsCondition	**contains**(String stringPart)	Matches SOAP messages containing the specified string.
URICondition	**uri**(String uriPart)	Matches request messages sent to a URI containing the specified string, and responses to those messages.
MessageDestinationCondition	**isRequest**()	Matches request messages, either from a client to a service, or from a service to another service.
	isResponse()	Matches response messages either from a service to a client, or from a service to a another service.
OperationCondition	**operation**(String operationName)	Matches request messages sent to a Web Service operation whose name is the specified string, and responses to those messages.

Table 6.1: Available conditions

To emulate an unresponsive Web Service (i.e., network packet loss), user has two options: (1) use DelayFault to delay a response message (possibly by a very large amount of time); (2) use ConnectionClosingFault to abruptly close the conection between proxy and client without returning any HTTP answer to the client. Note that a more accurate emulation of unresponsive services/packet loss is not possible working at the HTTP level like WSInject does. According to [59], this would require working at the network level.

Conditions can be combined by using the '&&' (AND) operator, meaning a ConditionSet will only be satisfied when all individual conditions are satisfied. Faults can be combined by the ',' (comma) operator, meaning all of them will be injected, on the specified order. The following injection rules show a sample script:

Name	Syntax	Description
INTERFACE FAULTS		
StringCorruptionFault	**stringCorrupt**(String fromString, String toString)	Replaces all occurences of fromString with toString. Works at String level. Ignores XML syntax (may be used to replace XML characters like '<' and '>').
XPathCorruptionFault	**xPathCorrupt**(String xPathExpression, String newValue)	Replaces all matches of an XPath [19] expression to the value specified. Can be used to modify either elements or attributes.
MultiplicationFault	**multiply**(String xPathExpression, int multiplicity)	Multiplies a part of a message by a number of times. For example, multiply("/", 2) duplicates the whole message contents, while multiply("/Envelope/MyNode",3) triplicates only the MyNode XML element.
EmptyingFault	**empty**()	Empties the SOAP message, delivering an HTTP message with no contents.
COMMUNICATION FAULTS		
DelayFault	**delay**(int delayInMilliseconds)	Delays a message delivery by the specified number of milliseconds.
ConnectionClosingFault	**closeConnection**()	Closes the connection between client and proxy.

Table 6.2: Available faults

```
uri("Hotel"):  stringCorrupt("Name", "Age"), multiply("/", 2);
uri("Airline"):  stringCorrupt("Flight", "Might");
contains("caught exception") && isResponse():  empty();
```

This example has three FaultInjectionStatements, one on each text line. The first one has a ConditionSet of a single condition: a **URICondition** with a "Hotel" argument. It also has a FaultList of two Faults: **StringCorruptionFault** with "Name" and "Age" arguments and a **MultiplicationFault** with "/" and '2' arguments.

The second FaultInjectionStatement has a ConditionSet with a URICondition and a FaultList with a StringCorruptionFault. The last FaultInjectionStatement has a ConditionSet with two conditions: a **ContainsCondition** and a **MessageDestinationCondition**; and a FaultList with an **EmptyingFault**. This script describes the following campaign:

- Whenever a URI of a Web service call contains the string "Hotel":
  1. Replace all text occurrences of "Name" by "Age".
  2. Duplicate the whole SOAP message.
- Whenever a URI of a Web service call contains the string "Airline":
  1. Replace all text occurrences of "Flight" by "Might".
- Whenever a message contains the string "caught exception" and is a response to a Web service caller:
  1. Empty the message.

Fault Injection Executor

Fault Injection Executor is the component in charge of effectively injecting faults. It processes the Abstract Syntax Tree (AST) produced by Script Compiler and injects faults where appropriate. For example, when a message should be corrupted, the Executor is the component which actually modifies the message; when the message should be delayed, the Executor is the component which actually inserts an emulated delay on the program execution. Fault Injection Executor code is called for all messages intercepted by the Proxy. For those that do satisfy the specified ConditionSet, Executor injects the appropriate faults. For those that do not, it takes no action.

Representing source code as ASTs is a common approach in the compilers field which facilitates the code processing. On WSInject, a CampaignDescriptor is an AST which is an exact representation of a fault injection script. Each element of

the script corresponds to an AST node, while each AST node corresponds to a Java class on WSInject code. Figure 6.11 shows the AST corresponding to the script example given in the previous paragraph.

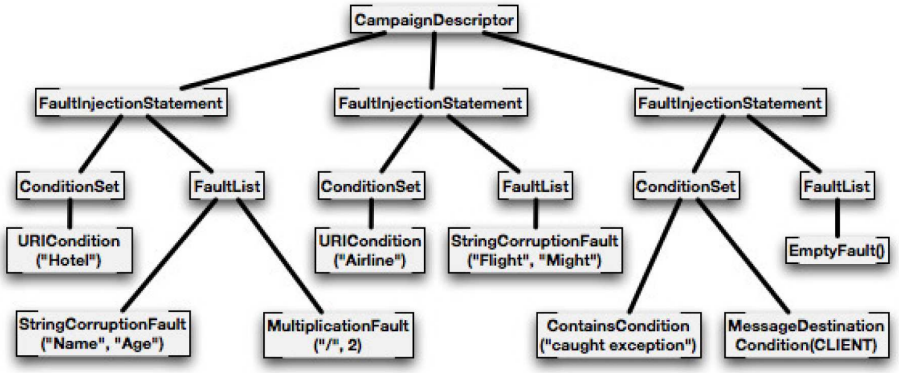

Figure 6.11: An example of an Abstract Syntax Tree

Controller

Controller is the central component of WSInject. It starts the tool and activates other components when required. WSInject can be started in two modes: graphical user interface (GUI) or command-line interface (CLI).

The initialization of WSInject with a fault injection campaign is described on the sequence diagram on figure 6.12. First, the Controller asks the Script Compiler to compile the script file into a CampaignDescriptor, which represents the entire fault injection campaign. Controller then creates and configures a Fault Injection Executor, and passes it to the Proxy/Monitor. After these steps, WSInject is ready to identify desired messages and inject faults described on the script file. Final steps are to start the Proxy/Monitor and to stop it after the experiment is completed.

Graphical User Interface (GUI)

The GUI component is responsible for receiving user inputs and for showing SOAP messages to the user. User inputs include setting the proxy port, turning the proxy on/off and loading/unloading scripts. Request and response messages can be seen

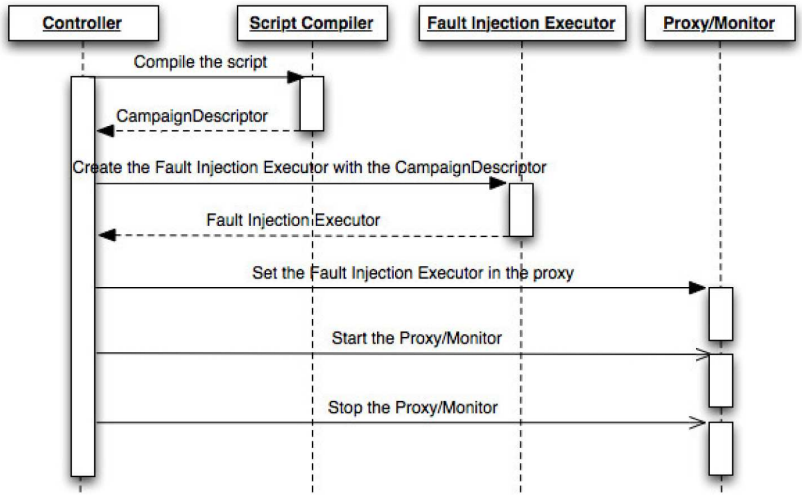

Figure 6.12: Initialization of WSInject's main components

by clicking their respective tabs. The left and right white panels respectively show messages contents before and after fault injection. Figure 6.13 depicts WSInject started on graphical user interface mode.

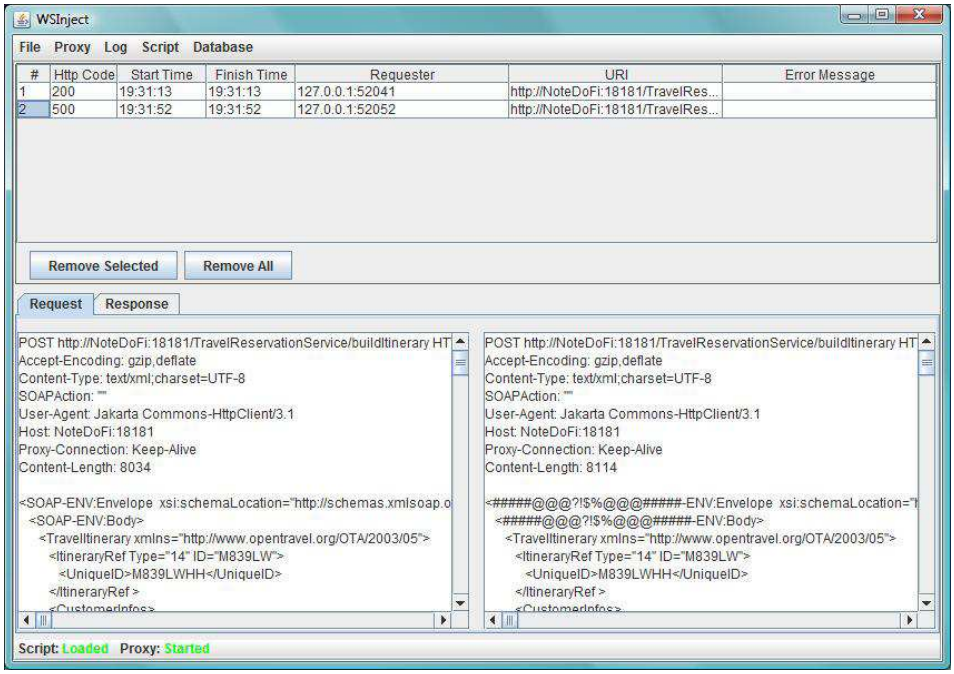

Figure 6.13: WSInject's GUI

Monitor Data Manager

The Monitor Data Manager is responsible for storing and retrieving data about messages intercepted by the Proxy/Monitor and also the log of WSInject.

6.5 Case study

In this section we carry out two case studies to illustrate our framework. First we apply our approach on the Heater Controlling System already introduced in section 6.3.1 and then, we will experiment our framework on a third-party system (the Travel Reservation Service) provided by Netbeans IDE 6.5.1 [20].

6.5.1 The Heater Controlling System (HCS)

The behavior of this system is illustrated by the sequence diagram presented in figure 6.14.

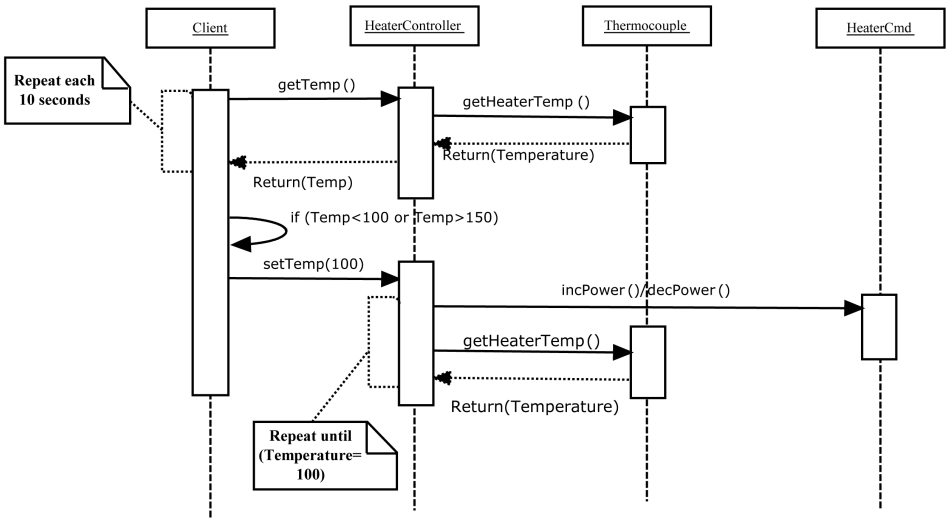

Figure 6.14: Sequence diagram of the Heater Controlling System

The *Client* periodically asks the *HeaterController* for the current temperature. The *HeaterController* forwards the request to the *Thermocouple* which returns the current temperature value. If the temperature value is outside a minimum and a maximum thresholds, the *Client* asks the *HeaterController* to readjust

it. The *Controller* will then use a time-based algorithm which invokes operations *incPower*() and *decPower*() of the service *HeaterCmd* until the heater temperature is adjusted to the right value. The heater coil is simulated by a simple shared database providing the current temperature. Each time the *Thermocouple* is invoked, it returns the current temperature and it updates its value randomly (either it increases or decreases the current value by five degrees each time). The *HeaterCmd* service also accesses this database each time operations *incPower*() or *decPower*() are invoked. According to the invoked operation, the *HeaterCmd* increases or decreases the current temperature value by five degrees each time.

The testbed architecture is illustrated in figure 6.15. It includes all service partners (the *HeaterController*, the *Thermocouple* and the *HeaterCmd*) and the client application which is in charge of monitoring the heater temperature and to adjust it when needed. The workload here, is implicitly generated and executed by the *Client*. For the faultload, we use WSInject for disturbing communication between the services of the composition. Observation points for collecting execution traces are implemented at communication interfaces of all services of the composition. This way we are able to keep information about all message exchanges (traces are sorted in a sequential order according to event occurrence times). In practice, the trace collection is easy because all services are configured to communicate through WSInject's proxy. WSInject provides also its own execution trace (the injection trace) telling about all executed injection operations and the involved messages. This trace will be used later for validating the injection process.

Robustness requirements

We specify five robustness requirements for this system.

Requirement 1: The client must ask for the current temperature each 10 seconds (Periodicity).

$$\Box((getTemp() \wedge T = x) \rightarrow \Diamond(getTemp() \wedge T = x + 10000))^{5}$$

[5] Time values are specified in milliseconds

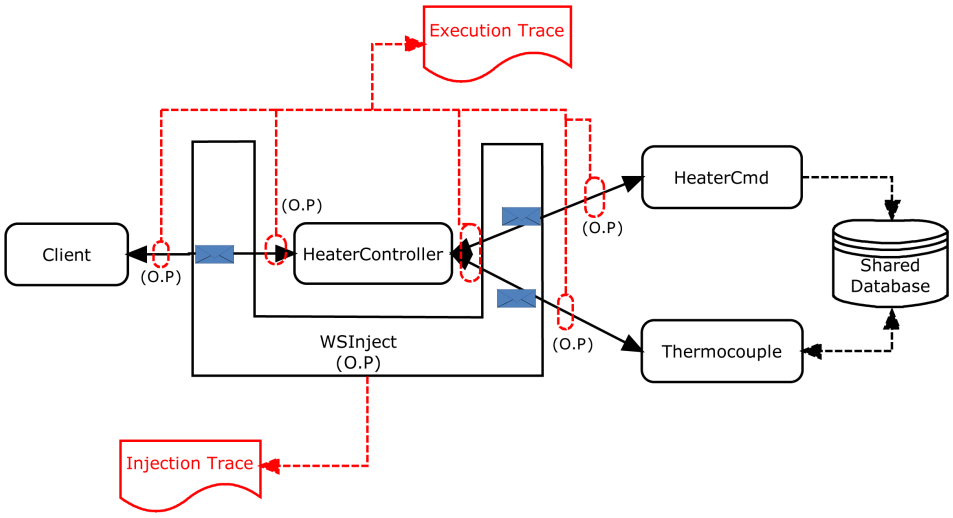

Figure 6.15: Testbed architecture of the heater controlling system

Requirement 2: The client must receive a response to its request within the following 5 seconds.

$$\Box((getTemp() \wedge T = x) \rightarrow \Diamond(getTempResponse() \wedge T <= x + 5000))$$

Requirement 3: The client must resend its request if it does not receive a response within the following 5 seconds. At worst it must resend its request 2 seconds after the timeout.

$$\Box(\neg((getTemp() \wedge T = x) \rightarrow \Diamond(getTempResponse() \wedge T <= x + 5000)) \rightarrow \Diamond(getTemp() \wedge T \leq x + 7000))$$

Requirement 4: The temperature value must always be between 100°C and 150°C. Outside this interval, the client application must, within the following 5 seconds, ask the *HeaterController* to readjust it to 100°C.

$$\Box(((getTempResponse(return > 150) \wedge T = x) \vee (getTempResponse(return < 100) \wedge T = x)) \rightarrow \Diamond(setTemp(Tmp = 100) \wedge T <= x + 5000))$$

Requirements 5: When the *HeaterController* is asked to readjust the temperature, it must regulate the Heater power until it is stabilized in the right value.

$$\Box(setTemp(Tmp = 100) \rightarrow \Diamond((incPower() \vee decPower())$$
$$\cup(getHeaterTemp() \rightarrow \Diamond getHeaterTempResponse(return = 100))))$$

Injection process

The Heater Controlling System deploys 5 operations: *getTemp()*, *setTemp(Integer Temp)*, *getHeaterTemp()*, *incPower()*, *decPower()*.

WSInject provides 6 kinds of simple faults:

- 4 interface faults:
  1. Structure and content message corruption using either the StringCorruptionFault or the XPathCorruptionFault;
  2. MultiplicationFault;
  3. EmptyingFault.
- 2 communication faults : the DelayFault and the ConnectionClosingFault.

We will address content corruption faults later as they will not affect all operations. Therefore, we have for now 5 simple faults (3 interface faults and 2 communication faults). WSInject can also combine indifferently between all these faults. If we choose to combine only one interface fault with one communication fault, we will have 6 possibilities. This increases the total number of the possible faults to inject to 11 (5 simple faults and 6 combinations). Now, if we want to inject all possible faults on each operation provided by the tested system **in both request and response sens**, we will have 110 injection configurations (as there are 5 operations).

Parameter values corruption (content corruption) can only be applied on the operation *setTemp(Integer Temp)* (as a request) and on responses of operations *getTemp()* and *getHeaterTemp()*. If we rely on the Ballista approach [52] for integer corruption, we will have 3 possibilities for each parameter (-MaxInt,+MaxInt and 0). Therefore, we have in all 9 possibilities; and if we combine each possibility with a communication faults, we will have 18 configurations. Therefore, the total number of all injection configurations is 128.

For structure corruption, MultiplicationFault and DelayFault, there are a infinite injection possibilities. The number of injection configurations found above was calculated while considering one possibility for each of these faults. For structure corruption, we inverse opening and closing XML tags; for MultiplicationFault we duplicate all the message body and for DelayFault, we delay the forwarding of messages by a sufficient amount of time for violating the specified timeout. For example, when the *Client* asks for the current temperature, the response is delayed for more than 5 seconds (as it should receive a response within the following 5 seconds).

Examples of injected faults

We give in the following some examples of the injected faults.

Eg.1: When the client asks for the current temperature, delay the response for 10 seconds.

```
operation("getTemp") && isResponse(): delay(10000);
```

Eg.2: Corrupt the parameter value of operation *setTemp()*.

```
operation("setTemp"): xPathCorrupt("//Temp/text()","0");[6]
```

Eg.3: Duplicate invocations of operation *getHeaterTemp()*.

```
operation("getHeaterTemp"): multiply("/",2);
```

Eg.4: Forward empty messages each time operations *incPower()* and *decPower()* are invoked.

```
operation("incPower"): empty();
operation("decPower"): empty();
```

To verify the injection process, we also specify the injected faults as Hoare triples following the proposed instantiation of this formalism for Web services. The specification of the above examples gives the following set of injection rules.

[6] When not specified, faults are injected on requests by default.

Injection rule 1:

```
{SoapMsg.has(getTempResponse) and $val==now}
delay(10000)
{new(SoapMsg).equals(SoapMsg) and $val+10000<=now<=$val+10050}
```

Injection rule 2:

```
{SoapMsg.has(setTemp)} xPathCorrupt("//Temp/text()","0")
{new(SoapMsg).Temp=="0" }
```

Injection rule 3:

```
{SoapMsg.has(getHeaterTemp)} multiply("/",2) { \forall $XML_elt;
SoapMsg.has($XML_elt) \implies new(SoapMsg).count($XML_elt) ==
2*SoapMsg.count($XML_elt) }
```

Injection rule 4:

```
{SoapMsg.has(incPower)} empty() { new(SoapMsg).isEmpty()}
{SoapMsg.has(decPower)} empty() { new(SoapMsg).isEmpty()}
```

Test execution and result analysis

We conducted 5 injection campaigns (one for each operation) and for each campaign, we executed the appropriate number of runs according to the considered operation. Therefore, we had 22 runs for operations: *incPower()* and *decPower()* as we considered both request and response senses based on 11 injection possibilities. For operations: *setTemp(integer Temp)*, *getTemp()* and *getHeaterTemp()*, we have 11 basic configurations for each one which gives 22 runs while considering both communication senses. In addition we have the content corruptions which produce 6 possibilities for each operation. Therefore, we will have at all, 28 runs for each one of these operations. The total number gives the previously calculated number of fault configurations i.e. 128 possibilities ($128 = (22 \times 2) + (22 \times 3) + (6 \times 3)$).

After experimentations, we first verified the good execution of the injection process using the instantiation of Algorithm 1 for Web services. The issued verdict was PASS which means that, according to the collected trace all injection operations were well performed. Then, we checked the collected execution trace with respect to the specified robustness properties based on the Web services instantiation of Algorithm 3. We summarize the obtained results in the following points:

- For the communication between the *Client* and the *HeaterController*, most robustness requirements were verified. For example, when the responses of the *HeaterController* were delayed for more than 5 seconds, the client resends its requests (satisfiability of robustness requirement 3) and when the returned parameter values were corrupted (with values outside the defined thresholds interval), the client always asks the *HeaterController* to readjust the temperature (satisfiability of robustness requirement 4).

- The different perturbations of *incPower* and *decPower* operations did not allow violation of robustness requirement 5. The *HeaterController* keeps invoking those operations until the current temperature value returned by the *Thermocouple* was conform to the defined minimum and maximum thresholds (satisfiability of robustness requirement 5).

- The CloseConnectionFault stopped completely the system execution. Each time we inject this fault on one system operation, the system stops its execution and all communications terminate. This is due probably to the fact that all service partners composing our system were deployed on the same Web application server (we used the server GlassFish v2.1 [21]). Therefore, when we close the connection between two services from the composition, it is actually the whole connection to the server which is closed.

6.5.2 The Travel Reservation Service (TRS)

To show the reliability of our approach, we applied it also on a second case study developed by a third party. It is the Travel Reservation Service (TRS) provided

by Netbeans IDE 6.5.1 [20]. TRS is a simulation of a real-life organization that manages airline, hotel and vehicle reservations using Web service partners. It is composed of three services - VehicleReservationService (VRS), AirlineReservation-Service (ARS) and HotelReservationService (HRS)- and one BPEL process (TRS), which orchestrates partner services to build a travel itinerary.

The TRS process assumes that an External Partner initiates the process by sending a message that contains a partial travel itinerary document. The client's travel itinerary may have: no pre-existing reservations, or a combination of pre-existing airline, vehicle and/or hotel reservations.

The TRS examines the incoming client itinerary and processes it for completion. If the client itinerary does not contain a pre-existing airline reservation, the TRS passes the itinerary to the ARS in order to add the airline reservation. The ARS passes back the modified itinerary to the TRS. The TRS conducts similar logic for both vehicle and hotel reservations. In each case it will delegate the actual provisioning of the reservation to the VRS and HRS. Finally, the TRS passes the completed itinerary back to the original client, completing the process.

The TRS implements also some temporal constraints to regulate the reservation process. In fact, each time the TRS passes the client itinerary to one of its service partners, it waits for a response within the following 20 seconds. In the case of no response, it must send a cancellation message to abort the reservation request. Figure 6.16 shows the sequence diagram of the TRS system.

Testbed architecture

The testbed architecture is presented in figure 6.17. SoapUI [22] is a well known test tool for Web services. We use it in our experiments for generating and running the workload. It plays the role of a TRS's client, sending requests with travel itineraries and activating the BPEL process, which in turn makes reservations with its partner services.

All services of the composition were deployed on the Glassfish server v2.1. Then, SoapUI and GlassFish were configured to make connections through WSInject's

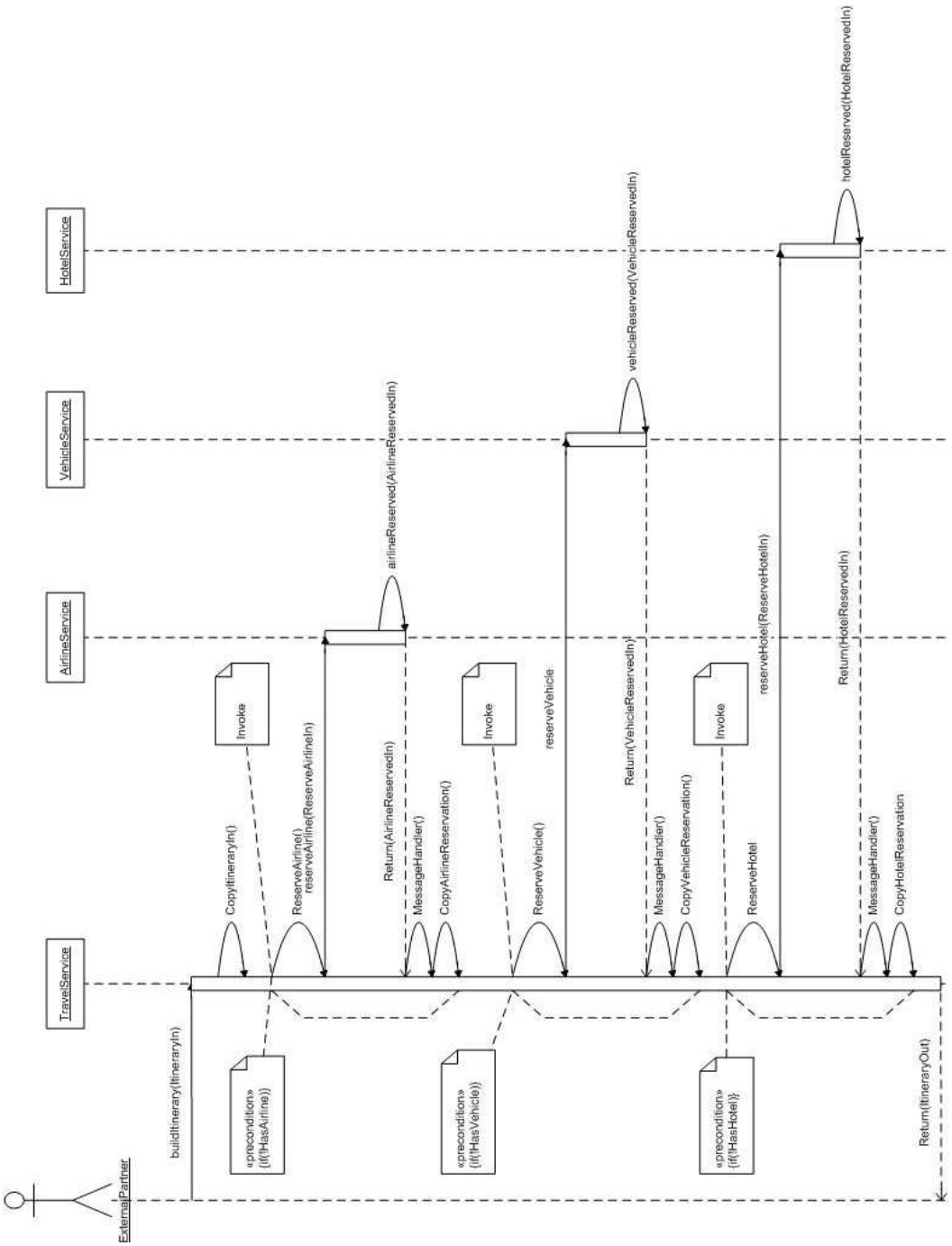

Figure 6.16: Sequence diagram of the TRS system

proxy component. Thus, all communications between the client, the BPEL process and the partner services were intercepted by WSInject, which was able to inject

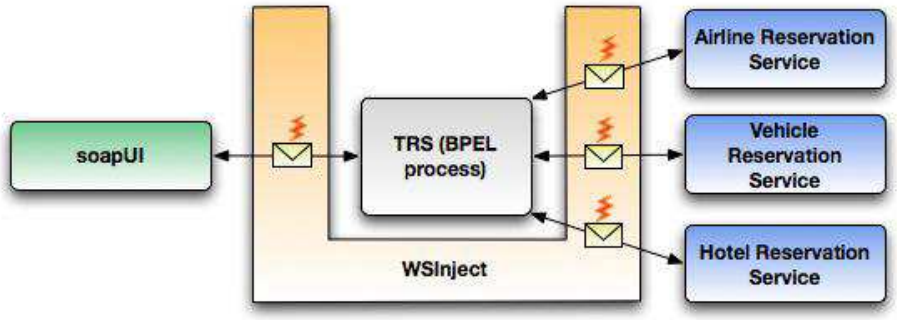

Figure 6.17: Testbed architecture of the TRS system

faults on all exchanged SOAP messages. Figure 6.18 shows the sequence diagram of the injection process.

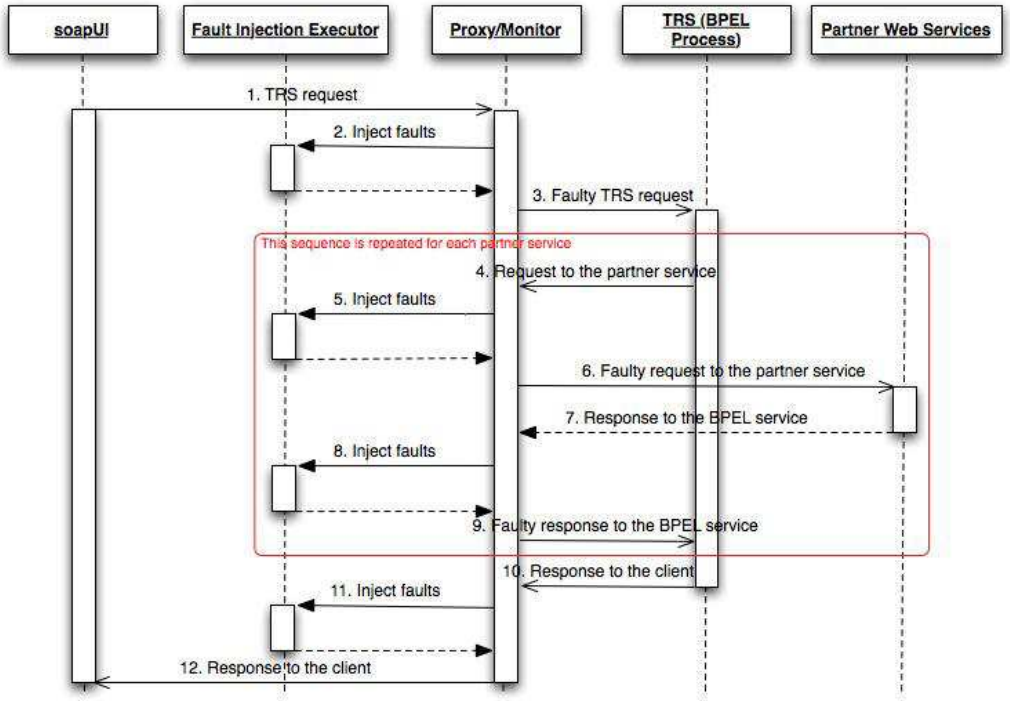

Figure 6.18: Sequence diagram of the injection process applied on TRS

Specification of robustness requirements of the TRS

TRS defines two main temporal constraints which can be specified as XCTL robustness requirements. The first is a simple response constraint specifying that each time the BPEL process sends a reservation requests to one of its service partners, it must receive a reservation confirmation within the following 20 seconds. Therefore, we have in all 3 response constraints (one for each service partners). For the ARS for example, we specify this requirement as follows:

Requirement 1:

$$\Box((reserveAirline() \wedge T = x) \rightarrow \Diamond(airlineReserved() \wedge T <= x + 20000))$$

The second robustness requirement is an illustration of the alternative pattern presented in section 4.5. It specifies that the BPEL process must send a cancellation message to its service partner (to cancel the reservation request it sent), if it does not receive the confirmation within 20 seconds. For the ARS, we specify this property as follows:

Requirement 2:

$$\Box(\neg((reserveAirline() \wedge T = x) \rightarrow \Diamond(airlineReserved() \wedge T <= x + 20000)) \rightarrow \Diamond cancelAirline())$$

This requirement concerns also the VRS and the HRS. Therefore, we will have in all 6 robustness requirements for the TRS system.

Test execution and results

Workload

The workload of our experiments consisted of sending itinerary requests from the SoapUI tool. The TRS system comes with pre-defined test cases on NetBeans - *hasAirline*, *hasHotel*, *hasVehicle* and *hasNoReservations* -, which are functional tests to verify the correct behavior of the system. *hasAirline* (resp. *hasHotel* or

hasVehicle) defines the case where the client has already an airline (resp. a Hotel or a vehicle) reservation. The *hasNoReservations* test case means that the client does not have any pre-existing reservation.

The SOAP requests from these test cases were used to activate the TRS during the fault injection campaign. SOAP messages sent by SoapUI were always the same: the "Input" message taken from the *hasNoReservations* test case from TRS (also named TestCase1 on some versions of NetBeans).

Faultload

The robustness requirements that we can specify for this case study do not involve data. Also, the XSD file defining the XML schema of the TRS is huge (more than 17000 lines) and defines a large set of parameters. We conducted preliminary experiments involving parameter value and structure corruptions and we noticed that the TRS does not implement any data validation procedure [36]. But this actually does not affect our robustness validation process as we are performing a black box testing and because our robustness requirements are independent from the parameter values that may be handled.

For these reasons, we do not consider in our faultload, the structure and the content corruption faults. Therefore, we will have 8 possible types of faults (4 simple faults and 4 combinations).

Communication between service partners involves the following set of messages:

- *buildItinerary()*: to activate the BPEL process asking for an itinerary reservation.
- *itineraryProblem()*: to inform about a possible itinerary fault.
- *reserveAirline(); reserveVehicle(); reserveHotel()*: to request an airline, a vehicle or a Hotel reservation.
- *cancelAirline(); cancelVehicle(); cancelHotel()*: to eventually cancel and airline, a vehicle or a Hotel reservation request.

- *airlineReserved(); vehicleReserved(); hotelReserved()*: to confirm an airline, a vehicle or a Hotel reservation.

If we consider injections on all these messages, we will have at all 88 fault configurations. As we do not consider content corruptions for this case study, we will have at all 11 injection campaigns (one for each message) and a uniform distribution of runs i.e. 8 runs for each operation.

Result analysis

After we verified the injection process to ensure the good execution of the injection campaign, we checked the robustness requirements on the collected execution trace. We had the following results:

- Probably, the most important result we got, is when injecting the DelayingFault for testing the robustness requirement 1 and 2. Each time we delay the forwarding of a request for more than 20 seconds (for example when delaying invocation of operation $reserveVehicle$ provided by the VRS), the TRS system hangs until the Glassfish server timeout is reached (2 minutes) and no cancellation message was sent. The automatic verification of the trace returns a FAIL verdict (requirement 1 and 2 were violated). Also, when we delayed the forwarding of the response message (reservation confirmation returned by the VRS for instance), the cancellation message was not sent and thus, the requirements 1 and 2 were also violated. In fact, when we examined manually the collected trace, we noticed that the sun-bpel-engine sent an error message indicating that there has been an instantiation error when sending the cancellation message. This shows a bug in the implementation of the cancellation process.

- The EmptyingFault caused an internal server error. Each time we injected this fault between two communication partners of the TRS, the system execution stops and all robustness requirements are violated. We examined the execution trace and we noticed that an HTTP 500 error code is sent by the GlassFish

server to the client application notifying that the connection was closed due to an internal server error.

- The ConnectionClosingFault had the same effect as for the previous case study. When applied on any TRS operation, the whole connection is lost and the system execution is stopped.

The application of our approach on this case study allowed us to reveal an important failure. We discovered that the cancellation process is actually never handled. This result demonstrates the efficiency of our approach as this failure could not be discovered using traditional conformance testing methods.

6.6 Conclusion

We presented in this chapter a testing framework for modeling and assessing Web services robustness. It is actually an instantiation of the robustness testing approach we proposed in the previous chapter for Web services. The framework includes a fault injection tool (WSInject) that we developed to inject interface and communication faults on both single and composed services. It also provides an implementation of Algorithm 1 and Algorithm 3 for Web services. These implementations are used to verify the injection process as well as the robustness requirements of the tested services.

The proposed framework can be used to test either simple or composed services. For illustration, we presented at the end of this chapter, an application on two case studies, where we detailed all the necessary testing steps starting from the different specifications (specification of the robustness requirements and the injection rules) till the test execution and the result analysis. The results we obtained are very promoter. We were able for example to discover some failures (for the second case study) that could not be revealed using traditional testing methods. This demonstrates the efficiency of our approach and motivates us to study the possibility to extend our framework to support other kinds of distributed systems.

Chapter 7

Conclusion

The main objective of this PhD thesis was to address the problems facing robustness testing and to propose a new and an innovative approach for assessing system robustness.

We first presented, in chapter two, the state of the art of the most relevant approaches for both conformance and robustness testing. For conformance testing, we focused mainly on passive testing techniques, because our proposed approach relies on this testing theory. Then, for robustness testing methods, we classified the existing approaches into two categories: those based on empirical fault injection techniques and those who rely on model-based testing.

The major issues with fault injection techniques applied on robustness testing are : (i) the absence of a formal test oracle for validating the test results and (ii) the lack of control on the injection process. The first problem could be resolved by relying on formal robustness testing approaches. For the second issue, we proposed a formal approach to specify and to verify the injection process. Our contribution consisted to define a fault injection formalism based on a timed extension of Hoare logic. We proposed to specify each injection operation by a Hoare triple describing the preconditions that must be satisfied before the execution of this operation and the postconditions that must be verified after its execution. This way, one can specify the set of injected faults for a given experiment and then, verify the good execution of the injection process using a proposed passive testing algorithm. This

algorithm checks the satisfiability of the specified injection rules (a set of Hoare triples) against injection traces. The injection traces are provided by the used fault injector. They log all injection operations executed within an injection experiment and the states of intercepted communication messages before and after the execution of those operations. This verification step must be performed after each injection process because we cannot guarantee that a fault injection mechanism used for a given experiment would work correctly when integrated in another testing framework. We presented this approach in chapter three and illustrated it with a set of examples of injection specifications.

Formal robustness testing approaches inspire from active testing techniques. As far as we know, they all create variants (mutants) of the behavioral model of the tested system, to generate and to execute their test. We believe that robustness requirements can be different from the functional ones. When facing abnormal environmental conditions, a software system may violate some of its functional requirements provided that the set of its robustness requirements are satisfied. For example, a functional property of a server application could be to response all the received requests within a relatively short period of time. However, when receiving a huge number of requests within a very short time interval (stressful conditions), the server application could be configured to close all its external connections to avoid the crash. This could be seen as a robustness property. Therefore, we proposed to formalize the robustness requirements as a set of real-time safety and liveness properties, using the explicit clock temporal language (XCTL). XCTL is an extension of the classical linear temporal logic to support real time specifications. The syntax of XCTL defines a dynamic state variable over the time domain (the clock variable) which can be used to refer to the value of the global time of the tested system. In chapter four, we discussed the expressiveness of XCTL compared to other existing real-time formalisms and we proposed a backward checking algorithm to check XCTL formulas on execution traces. This approach follows the passive testing architecture. Observation points are seeded in different system location to collect execution traces. This way, one can track all system components; which is

particularly interesting when testing distributed and/or composed applications.

In chapter five, we proposed a new robustness testing approach. The proposed technique relies on both fault injection and passive testing. The basic idea was to use fault injection as a perturbation mechanism and then, verify the robustness requirements against the collected execution traces. This way, the defined fault domain would be much larger, because the set of faults which is usually considered by existing formal robustness testing approaches is always limited by the original input domain. On the other hand, robustness requirements could be specified independently from the functional ones, as we are not constrained by the original behavioral model. Also, by combining fault injection and passive testing, one can study the behavior of all components of a distributed system. Faults are injected between different communication partners and traces are collected all over the composition. To control the injection campaigns, we specify the injected faults as a set of Hoare triples and we used this specification to verify the injection process based on the algorithm presented in chapter three. For robustness assessment, we specify the robustness requirements as a set of XCTL formulas and we use our passive testing algorithm, proposed in chapter four, to check their correctness on the collected execution traces.

Finally, for our last contribution, we proposed in the sixth chapter, a testing framework for modeling and testing Web services robustness. We chose Web services because they present interesting testing challenges. They are distributed and heterogeneous systems, widely used for building business applications and integration softwares. They also provide two kinds of compositions: the orchestration and the choreography. The proposed framework is an instantiation of our robustness testing approach for Web services. We implemented in this framework, the passive testing algorithms that we proposed for checking the injection process and the robustness requirements on execution traces. We also proposed and built an innovative fault injection tool for Web services: WSInject. This tool was integrated in our framework to simulate hostile environments. Its main features are: (i) its ability to inject both interface and communication faults and (ii) the way it can be

used to test single and composed services. We presented also, at the end of this chapter, two case studies on Web services compositions. The first one, is a simulation of a heater controlling system. It describes a critical system scenario which illustrates an example of a system that requires a high robustness level. For the second case study, we chose to test a third-party Web service composition provided by NetBeans (the Travel Reservation Service). For each case study, we presented the complete testing steps and we described for each step the specified properties (robustness requirements and examples of the injected faults). We also presented the used testing architecture and discussed the obtained results. Particularly, for the Travel Reservation Service, we were able to discover interesting failures that could not be revealed using classical testing methods.

7.1 Perspectives

Formal methods for robustness testing is a relatively recent direction in the testing literature. The work we presented in this manuscript, is a set of contributions which aim at addressing the new challenges facing this kind of testing. A possible extension of our work could be to study the possibility of upgrading the proposed passive testing algorithms to on-line monitoring. This way, one can check both the injection process and the robustness requirements during experimentations and raises exceptions as soon as some of the specified properties are violated. This avoids also to collect execution traces and hence, makes the test execution faster.

The fault injector we developed (WSInject), can also be improved by implementing new fault injection operations. It would be also interesting to study the possibility of deploying it as a Web service and thus, making it easily available for the testing community to be able to perform larger and deeper experimentations.

Another direction that could be considered for future work, is the possibility of instantiating the proposed robustness testing approach for other kind of systems. The Web services testing framework that we proposed, is an example to show how our robustness testing technique could be applied for testing real systems. This approach is based on abstract concepts. Therefore, it could be easily implemented

for various kind of communication protocols and other distributed applications.

Bibliography

[1] Glossary of Software Engineering Terminology- IEEE Std 610. 12-1990. 3, 19

[2] Resilience-Building Technologies: State of Knowledge. Delivrable D12. ReSIST: Resilience for Survivability in IST. http://www.resist-noe.org/. 3, 20

[3] ISO/IEC 9646. Information Technology- Open Systems Interconnection- Conformance testing methodology and framework- Part 1-5. 27, 28

[4] W3C. eXtensible Markup Language XML, 2008. http://www.w3.org/XML. 98

[5] W3C. XML Schema. http://www.w3.org/XML/Schema. 98

[6] W3C. HTTP - Hypertext Transfer Protocol. http://www.w3.org/Protocols/. 98

[7] W3C. WSDL- Web Services Description Language. http://www.w3.org/TR/wsdl. 98

[8] W3C. Simple Object Access Protocol SOAP (Version 1.1), May 2000. http://www.w3.org/TR/soap. 99

[9] UDDI. Universal Description, Discovery and Integration UDDI. http://www.uddi.org/. 99

[10] OASIS Standard. WSBPEL Ver. 2.0, April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. 100

[11] Active endpoints. activeBPEL. http://www.activevos.com/community-open-source.php. 101

[12] ORACLE. Oracle BPEL Process Manager. http://www.oracle.com/technology/products/ias/bpel/index.html. 101

[13] W3C. WS-Addressing. http://www.w3.org/Submission/ws-addressing/. 101

[14] W3C. WS-Policy. http://www.w3.org/TR/2007/REC-ws-policy-20070904/. 101

[15] W3C. WS-Security. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss. 101

[16] OASIS. OASISWeb Services Reliable Messaging. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm. 101

[17] IBM. WS-Transactions. http://www.ibm.com/developerworks/library/specification/ws-tx/. 102

[18] WSBang at https://www.isecpartners.com/wsbang.html. 108

[19] W3C. XPATH. http://www.w3.org/TR/xpath. 114

[20] NetbBeans IDE at http://netbeans.org/. 118, 125

[21] The GlassFish Web application server at https://glassfish.dev.java.net/. 124

[22] SoapUI at http://www.soapui.org/. 125

[23] R. Carbone A. Armando and L. Compagna. Ltl model checking for security protocols. *20th IEEE Computer Security Foundations Symposium (CSF 2007), Proceedings.*, 2007. 65

[24] G. D. Angelis A. Bertolino and A. Polini. A qos test-bed generator for web services,. *In ICWE, ser. Lecture Notes in Computer Science, L. Baresi, P. Fraternali, and G.-J. Houben, Eds.*, 4607:17–31, 2007. 108

[25] W. Mallouli K. Li A. Cavalli, A. Benameur. A passive testing approach for security checking and its practical usage for web services monitoring. *9ème Conférence Internationale sur les NOuvelles TEchnologies de la REpartition (NOTERE09)*, June 29 - July 3, 2009. 66

[26] D. Chen D. Khuu B. Alcalde, A. Cavalli and D. Lee. Protocol system passive testing for fault management : A backward checking approach. *In Formal Techniques for Networked and Distributed Systems (FORTE), LNCS 3235, Springer*, pages 150–166. 37

[27] E. Brinskma. A theory for the derivation of tests. *In S. Aggarwal and K. Sabnani editors , Protocol Specification, Testing and Verification*, 8:63–74, 1988. 30, 31

[28] T. Chow. Testing software design modelled by finite state machines. *In IEEE Transaction on Software Engineering*, 1989. 30

[29] R. Hao R.E. Miller J. Wu D. Lee, D. Chen and X. Yin. A formal approach for passive testing of protocol data portions. *In Proceedings of the IEEE International Conference on Network Protocols, ICNP02*, 2000. 35, 37

[30] P.S. Deepinder and T.K. Leung. Formal methods for protocol testing: A detailed study. *In IEEE Transaction on Software Engineering*, 15:413–426, 1989. 30

[31] M. Nunez E. Bayse, A. Cavalli and F. Zaidi. A passive testing approach based on invariants: Application to the wap. *Computer Networks*, 48:235–245, 2005. 38, 40

[32] O. Lichtenstein E. Harel and A. Pnueli. Explicit clock temporal logic. *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on*, 1990. 6, 8, 22, 23, 64, 69, 71

[33] Thomas Erl. Service-oriented architecture : A field guide to integrating xml and web services. *Prentice Hall*, 2004. 96

[34] Thomas Erl. Service-oriented architecture : Concepts, technology, and design. *Prentice Hall*, 2005. 96

[35] B. P. Miller et al. Fuzz revisited: A re-examination of the reliability of unix utilities and services. *Networked Computer Science Technical Reports Library CS-TR-95-1268*, April 1995. 50

[36] W. Maja E. Martins F. Bessayah, A. Cavalli and A.W. Valenti. A fault injection tool for testing web services composition,. *In Testing Practice and Research Techniques (TAIC PART'10), LNCS 6303*, pages 137–146, 2010. 7, 22, 102, 108, 129

[37] A. Rollet F. Saad-Khorchef and R. Castanet. A framework and a tool for robustness testing of communicating software. *In Proceedings of the 2007 ACM symposium on Applied computing (SAC07)*, pages 1461–1466, 2007. 21, 47, 91

[38] G. Gonenc. A method for the design of fault detection experiments. *In IEEE Transactions Computer*, pages 551–558, 1970. 30

[39] K. Havelund H. Barringer, A. Goldberg and K. Sen. Rule-based runtime verification. *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI04), Proceedings.*, 2004. 65

[40] A. Rollet H. Fouchal and A. Tarhini. Robustness of composed timed systems. *In 31st Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM05)*, pages 155–164, 2005. 21, 46, 91

[41] W. Hoarau and S. Tixeuil. A language-driven tool for fault injection in distributed applications,. *In Proceedings of the IEEE/ACMWorkshop GRID 2005*, 2005. 44

[42] C.A.R. Hoare. An axiomatic basis for computer programming,. *Communications of the ACM*, 12, October 1969. 5, 21, 51, 52, 54, 91

[43] J. Hooman. Specification and compositional verification of real-time systems. *PhD Thesis*, 1991. 54

[44] J. Hooman. Extending hoare logic to real-time. *Formal Aspects of Computing*, 6:6–801, 1994. 54

[45] A. Cavalli J.A. Arnedo and M. Nunez. Fast testing of critical properties through passive testing. *In Dieter and Hogerefe and Anthony Wiles editors, TESTCOM. LNCS 2644, Springer*, pages 295–310, 2003. 38

[46] C. Pachon J.C. Fernandez, L. Mounier. A model-based approach for robustness testing. *IFIP International Conference on Testing of Communication Systems (TESTCOM05)*, pages 333–348, 2005. 21, 46, 91

[47] W.L. Kao and R.K. Iyer. Fine: A fault injection and monitoring environment. *IEEE Transaction on Software Engineering*, pages 1105–1118, 1993. 43

[48] W.L. Kao and R.K. Iyer. Define: A distributed fault injection and monitoring environment. *In Proceedings of IEEE Fault-Tolerant Parallel and Distributed Systems (IEEE-FTPDS'94)*, pages 252–259, 1994. 43, 86, 108

[49] H.L. Truong L. Juszczyk and S. Dustdar. Genesis - a framework for automatic generation and steering of testbeds of complexweb services,. *In Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems(ICECCS'08)*, pages 131–140, 2008. 108

[50] J. Xu N. Looker, M. Munro. Ws-fit: A tool for dependability analysis of web services,. In *Proceedings of the 28th Annual International Computer Software and Applications Conference,*, 2004. 109

[51] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transitions tours. *In IEEE Fault Tolerant Computer Systems*, 2007. 30

[52] K. DeVale J. DeVale K. Fernsler D. Guttendorf N. Kropp J. Pan C. Shelton Y. Shi P. Koopman, D. Siewiorek. Ballista project:cots software robustness testing. 1998. 50, 121

[53] K. Sen P. Naldurg and P. Thati. A temporal logic based framework for intrusion detection. *Formal Techniques for Networked and Distributed Systems (FORTE 2004), Proceedings.*, 2004. 65

[54] M. Phalippou. Relation d'implantation et hypothèse de test sur des automates a entrées et sorties. *PhD Thesis, Universite de Bordeaux I*, 1994. 30, 31

[55] Amir Pnueli. The temporal logic of programs,. *In 18th Annual Symposium on Foundations of Computer Science*, 1977. 64, 66

[56] T.A. Henzinger R. Alur. An overview of existing tools for fault-injection and dependability benchmarking in grids,. *Real-Time: Theory in Practice. Lecture Notes in Computer Science 600, Springer-Verlag*, pages 74–106, 1992. 68, 69

[57] H. Waeselynk R. Castanet. Techniques avancées de test de systemes complexes: test de robustesse. *report CNRS-AS23*, 2003. 85

[58] M.-O. Rabin. Decidability of second order theories and automata on infinite trees. *Transactions of the AMS*, pages 1–35, 1969. 46, 92

[59] P. Reinecke and K. Wolter. Towards a multi-level fault-injection test-bed for service-oriented architectures: Requirements for parameterisation,. *In SRDS Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems*, 2008. 113

[60] G. Rosu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 2005. 67, 71, 72, 79

[61] F. Jahanian S. Dawson and T. Mitton. Orchestra: A probing and fault injection environment for testing protocol implementations. 41, 86, 108

[62] K. G. Shin S. Han and H. A. Rosenberg. Doctor: An integrated software fault injection environment for distributed realtime systems. *Presented at International computer performance and dependability symposium*, 1995. 41, 86, 108

[63] L. Silva S. Tixeuil, W. Hoarau. Logics and models of real time: A survey,. *CoreGRID Technical Report Number TR-0041*, 2006. 42, 43

[64] K. Sabnani and A. Dahbura. A protocol test generation procedure. *In Computer Networks and ISDN Systems*, 15:285–297, 1988. 30

[65] D.T. Stott and al. Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. 42, 86

[66] M. Tabourier and A. Cavalli. Passive testing and application to the gsm-map protocol. *Journal of Information and Software Technology*, 41:813–821, 1999. 34

[67] J. Tretmans. A formal approach to conformance testing. *PhD Thesis; Twente University*, 1992. 30

[68] J.L. Richier V. Darmaillacq and R. Groz. Test generation and execution for security rules in temporal logic. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:252–259, 2008. 65

[69] A.Cavalli W. Mallouli, F. Bessayah and A.Benameur. Security rules specification and analysis based on passive testing. *In Proceedings of The IEEE Global Communications Conference*, 2008. 65