Passive interoperability testing for communication protocols

Nanxing Chen

ANNÉE 2013

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : INFORMATIQUE

Ecole doctorale MATISSE

présentée par

Nanxing Chen

Préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Système Aléatoires
Composante universitaire ISTIC

Passive Interoperability Testing for Communication Protocols

Thèse soutenue à Rennes le 24.06.2013

devant le jury composé de :

Richard CASTANET
Professeur , Labri-ENSEIRB / *rapporteur*

Gérardo RUBINO
Directeur de recherche, INRIA / *examinateur*

Stephan MAAG
Maitre de conférences, Telecom Sud-Paris / *examinateur*

Xiaohong HUANG
Maitre de conférences, BUPT / *examinateur*

César VIHO
Professeur, Universtité de Rennes 1 / *directeur de thèse*

Yan MA
Professeur, BUPT / *rapporteurs avant soutenance*

Acknowledgement

I would like to express my gratitude to all those who helped me during the thesis.

My deepest gratitude goes first and foremost to my beloved family and my relatives by marriage for their loving considerations and great confidence in me all through these years. Without their constant support and encouragement, this thesis could not have succeeded.

Second, I would like to express my gratitude to Professor César Viho. Due to his guidance and support all along my PhD thesis.

I am also greatly indebted to Anthony Baire, who have instructed and helped me a lot in technical aspects.

Last my thanks would go to my colleagues, who are not only my work partners, but also trustworthy friends.

Contents

Résumé de la Thèse

Dans le domaine des réseaux, le test de protocoles de communication est une activité importante afin de valider les protocoles applications avant de les mettre en service. Généralement, les services qu'un protocole doit fournir sont décrits dans sa spécification. Cette spécification est une norme ou un standard défini par des organismes de normalisation tels que l'ISO (International Standards Organisation), l'IETF (Internet Engineering Task Force), l'ITU (International Telecommunication Union), etc. Le but du test est de vérifier que les implémentations du protocole fonctionnent correctement et rendent bien les services prévus.

Pour atteindre cet objectif, différentes méthodes de tests peuvent être utilisées. Parmi eux, le test de conformité vérifie qu'un produit est conforme à sa spécification. Le test de robustesse vérifie les comportements de l'implémentation de protocole face à des événements imprévus. Dans cette thèse, nous nous intéressons plus particulièrement au test d'interopérabilité, qui vise à vérifier que plusieurs composants réseaux interagissent correctement et fournissent les services prévus.

L'architecture générale de test d'interopérabilité fait intervenir un système sous test (SUT) composé de plusieurs implémentations sous test (IUT). Les objectifs du test d'interopérabilité sont à la fois de vérifier que plusieurs implémentations (basées sur des protocoles conçus pour fonctionner ensemble) sont capables d'interagir et que, lors de leur interaction, elles rendent les services prévus dans leurs spécifications respectives.

En général, les méthodes de test d'interopérabilité peuvent être classées en deux grandes approches: le test actif et le test passif. Le test actif est une technique de validation très populaire, dont l'objectif est essentiellement de tester les implémentations (IUT), en pratiquant une suite de contrôles et d'observations sur celles-ci. Cependant, une caractéristique fondamentale du test actif est que le testeur possède la capacité de contrôler les IUTs. Cela implique que le testeur perturbe le fonctionnement normal du système testé. De ce fait, le test actif n'est pas une technique appropriée pour le test d'interopérabilité, qui est souvent effectué dans les réseaux opérationnels, où il est difficile d'insérer des entrées

arbitraires sans affecter les services ou les fonctionnements normaux des réseaux.

A l'inverse, le test passif est une technique se basant uniquement sur les observations. Le testeur n'a pas besoin d'agir sur le SUT notamment en lui envoyant des stimuli. Cela permet au test d'être effectué sans perturber l'environnement normal du système sous test. Le test passif possède également d'autres avantages comme par exemple, pour les systèmes embarqués où le testeur n'a pas d'accès direct, de pourvoir effectuer le test en collectant des traces d'exécution du système, puis de détecter les éventuelles erreurs ou déviations de ces traces vis-à-vis du comportement du système décrit dans sa spécification. En outre, les tests peuvent être mises en place sur une longue période, et ainsi à tout moment traiter les anomalies contrairement au test actif qui est généralement limité dans le temps par une campagne de test. De ce fait, cette thèse vise à proposer les méthodes de test basées sur la technique de test passif pour effectuer le test d'interopérabilité.

L'organisation de thèse est suivante:

Le chapitre 1 est l'introduction. Le contexte, les problématiques, les motivations ainsi que le plan de thèse y sont présentés.

Le chapitre 2 présente l'état de l'art du domaine. Les notions générales, les architectures et les critères du test d'interopérabilité y sont décrits. Dans cette thèse, on considère la configuration le plus souvent utilisée: L'architecture dite "one-to-one" où le SUT contient deux IUTs. Chacune de ces IUTs est une boîte noire, dont le comportement n'est connu que par les interactions à travers leurs interfaces. Les différentes méthodes (actives et passives) pour effectuer le test d'interopérabilité sont présentées et comparées. Les motivations pour le choix de la méthode passive sont expliquées.

Les problématiques du test passif sont aussi exposées dans ce chapitre. En effet, malgré le fait que le test passif s'avère être une technique prometteuse, peu de travaux l'ont appliqué dans le domaine d'interopérabilité. Pourtant, il y a un certain nombre de problèmes à résoudre à cause de la non-contrôlabilité du test passif. Par exemple, la vérification de comportement des IUTs (trace), l'émission des verdicts, etc. Pour faire face à ces problématiques, dans les chapitres suivants, plusieurs contributions sont proposées.

Dans le chapitre 3, une approche pour effectuer le test d'interopérabilité en utilisant la technique du test passif est proposée. Cette méthode est basée sur les modèles formels. Elle commence par le choix d'un ensemble d'objectives de test, qui décrit les propriétés à vérifier lors du test. Puis, pour chaque objectif de test, un cas de test correspondant est généré, décrivant en détail le comportement des IUTs à observer. Le cas de test est généré de la manière automatique avec

un algorithme de recherche en profondeur, considérant l'interaction asynchrone partielle entre les spécifications par rapport à l'objectif de test. La manipulation des entités considérées lors de la génération de test peut conduire à stocker un nombre d'états trop important par rapport à la capacité des systèmes sur lesquels sont exécutés les algorithmes de génération. Pour éviter le problème classique d'explosion combinatoire des états lors du calcul, chaque objectif de test a des attributs spécifiques pour limiter le calcul de l'interaction asynchrone. Le modèle formel utilisé, les règles de calcul partiel, la dérivation de cas de test passifs sont décrits dans ce chapitre.

Par ailleurs, en test passif, une difficulté est la vérification de trace. Le testeur observe et collecte les messages échangés entre les entités protocolaires. Puis il analyse les informations collectées (traces d'exécution) pour vérifier que les traces sont conformes aux propriétés que l'on souhaite tester. Une problématique est que le système de test n'a pas de connaissance de l'état où le système sous test peut être par rapport au début de la trace. Pour surmonter le problème ci-dessus, un algorithme de vérification a été proposé pour analyser le comportement des IUTs ainsi qu'émettre les verdicts appropriés. L'algorithme vise à rechercher dans la trace les cas de test qui correspondent aux observations contenues dans la trace.

Une autre contribution dans ce chapitre est l'émission de verdicts. Respectivement, le verdict est Pass si l'objectif de test a été atteint et aucun erreur n'a été observée, Fail si au moins une erreur a été observée, ou Inconclusive si le comportement observé est correct par rapport à la spécification mais ne correspond pas à l'objectif visé par le test. A cause de la non-contrôlabilité du test passif, les règles d'attribution de verdicts sont différentes de ceux utilisés pour la méthode active et doivent être assouplis. Dans ce chapitre, les règles d'attribuer des verdicts Pass, Fail ou Inconclusive sont expliqués en détail.

Dans le chapitre 4, une autre approche du test d'interopérabilité passif a été proposée. Ce chapitre se focalise sur les protocoles largement utilisé de type requête/réponse, qui reste incontournable pour réaliser les services Web. Cette approche commence aussi par le choix des objectifs de test. Pour chacun des objectifs de test, un cas de test est généré (manuellement). Pour la vérification de trace, l'idée est d'utiliser le modèle de transactions client-serveur requête/réponse de ce type de protocoles. La stratégie consiste à décomposer la trace en un ensemble de conversations entre le client et le serveur. Une fois que la trace est découpée en une série de conversations, la prochaine étape est d'établir la correspondance entre chaque cas de test et ces conversations. Pour chaque cas de test présent dans la trace, l'algorithme proposé vérifie automatiquement si celui-ci est satisfait et émet un verdict.

Dans ce chapitre, un outil de test nommé ttproto (testing tool prototype) a

été proposé pour automatiser le test pour les protocoles de type requête/réponse. Plus précisément, après qu'une trace est découpée en conversations, l'outil de test commence à exécuter la procédure de vérification. Pour ce faire, l'outil prend en entrée deux fichiers: les cas de test et les conversations. L'étape suivante consiste à analyser la trace capturée à l'aide d'un algorithme de vérification de trace. Cet outil est inspiré de TTCN-3 (Testing and Test Control Notation Version 3), un outil très utilisé dans le domaine de test, initialement conçu pour la méthode active. Ttproto conserve certaines notions utiles de TTCN-3, mais intègre de nouvelles fonctionnalités pour s'adapter au contexte de test passif.

La méthode et l'outil ont étés appliqués sur un protocole réel CoAP (Constrained Application Protocol) dans les évènements CoAP Plugtest organisés par l'ETSI (European Telecommunications Standards Institute) dans le contexte d'Internet des objets. Ces évènements ont eu lieu à Paris et Sophia Antipolis en 2012. Les résultats lors de leurs applications sont présentés et montrent la validité, la pertience et l'efficacité de cette approche.

Le chapitre 5 propose une synthèse des travaux réalisés, ainsi que des perspectives pour des travaux futurs. Particulièrement, on s'intéresse à l'amélioration de la vérification de traces. En effet, les algorithmes de vérifications présentés auparavant fonctionnent de manière séquentielle. Cela implique une durée de vérification potentiellement longue s'il y a beaucoup de cas de test à vérifier, ou si la trace est très longue. Pour y faire face, dans ce chapitre, nous présentons les premiers résultats de travaux visant à regrouper les cas de test. L'objectif du groupement de cas de test est de permet qu'ils puissent être vérifiés en parallèle sur la trace. Nous montrons qu'un cas de test obtenu en utilisant cette approche rend la vérification plus efficace.

Dans ce chapitre, quelques pistes sont données concernant les autres travaux long termes. Les méthodes proposées devront être étendues dans le contexte plus général de l'interaction de plus de deux implémentations. A cause de la non-contrôlabilité du test passif, les verdicts sont souvent Inconclusive. Une solution doit être trouvée pour raffiner les verdicts. Dans le Chapitre 4, on s'intéresse uniquement aux protocoles de type requête/réponse. La méthode et l'outil proposés pour ce type de protocole devront être étendus aux autres types de protocoles plus complexes.

Chapter 1

Introduction

1.1 Background

The deregulation of the telecommunication industry has led to the rapid evolution of technologies and know-how in the field of communication systems. The computer systems attached to a network communicate with each other using a common set of conventions called *protocols.* In other words, protocols are the rules that govern the communication between the different components within a distributed computer system. In order to organize the complexity of these rules, they are usually partitioned into a hierarchical structure of protocol layers, as exemplified by the seven layers of the standardized Open Systems Interconnection (OSI) model [54]. The layering concept was used to divide the communication functions into sets which can be specified separately. This allows independent development and implementation of standards at each layer.

A protocol, in general, is quite complex and takes a considerable effort to implement on a system. The implementation of a protocol is usually derived from a specification standard, which defines the required behavior for a protocol entity, and can lead to several different applications. On one hand, the complexity of protocols necessitates the thorough study of the specification, as well as verification, development and testing. On the other hand, with the never-ending changes and improvement of computer technologies, today's network services and applications are required to be rich, on-time, with high quality. These aspects, consequently, call the needs of ensuring proper functionality of network-enabled products in order to satisfy customer expectations. In this context, testing has steadily become more and more important within the development of software and systems, aiming at solving both new challenges imposed by the advancement in various areas of computer science and long-standing problems.

To increase confidence in protocol products implemented according to inter-

national standards, various protocol testing methodologies have been developed and applied for different purposes. They can be generally classified into two categories, *Stress and Reliability Tests* and *Functional Tests* (c.f. Fig.1.1).

Stress and reliability tests addresse the ability of a system or component to perform its required functions under stated conditions for a specified period of time. Typical stress and reliability tests are for example:

- Performance testing, which is conducted to evaluate the compliance of a system or component with specified performance requirements.
- Stress testing, which is conducted to evaluate a system or component at or beyond the limits of its specified requirements.
- Robustness testing determines the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions. etc.

Different from stress and reliability tests, functional tests focus on verifying whether a system meets external requirements and achieve goals. Typical functional tests include *conformance testing* and *interoperability testing*, etc., among which *interoperability testing* is the topic of this thesis.

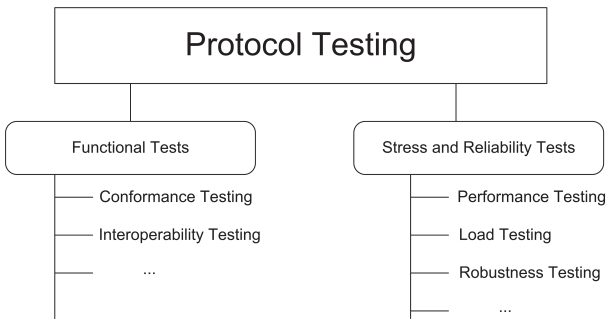


Figure 1.1: Protocol Testing

There is a close relationship between conformance testing and interoperability testing. Conformance testing, as one of the most fundamental one, checks whether an implementation is correct with respect to its relevant specifications. The importance of conformance testing lies in: equipments from different vendors conforming to the same standards have a higher likelihood of interoperability. In

other words, protocol implementations are tested against the protocol specification in order to assure compatibility with other implementations based on the same protocol. It is important for both equipment suppliers and buyers that, different vendors can independently implement standards with higher assurance of product interoperability. Also, equipment buyers can have confidence that the purchased equipments from different suppliers interoperate.

However, although the ISO 9646 standard [1] mentions that conformance testing can increase the probability of interoperability, it cannot guarantee the successful interaction between different implementations. It must be noted that conformance testing is a necessary step to insure interoperability but it is not enough. It is a known fact that, even following the same standards, two protocol implementations from different suppliers may still fail to interoperate due to various reasons: lack of clarity of standards, poorly specified protocol options, incompleteness of specifications, incoherence of protocol implementations, etc. Nevertheless, on one hand, customer needs are growing and manufacturers permanently develop new equipments with improved quality of service. On the other hand, these services must be very quickly validated so that the proposed solutions are reliable and ready to work. Therefore, in parallel, efficient techniques in communication systems engineering must be developed in order to reduce the time-to-market. In this context, *interoperability testing* is holding a strategic position in the design of new technologies. Its role is to determine whether several interconnected products from different product lines interoperate correctly and provide the expected services. Indeed, conformance testing improves the chances of interoperability while interoperability testing checks at a user level if interoperability has been achieved.

Despite its importance, contrary to conformance testing, much less attention has been paid to interoperability testing. Currently, interoperability testing has not yet standardized. Moreover, test automation and reusability is difficult. Therefore, this thesis focuses on interoperability testing, aiming at researching methods and solutions to improve interoperability testing.

Looking into the literature, throughout the past twenty years or so, various research works have been proposed to carry out interoperability testing ([19, 36, 6, 18], etc.) So far, the conventional method is to generate a sequence of tests from the specifications of the interconnected network equipments under test. Then these tests will be run against the network equipments to check if they work together properly. Generally, the tester has the ability to stimulate the equipments and verify whether the outputs obtained for each test message are as expected according to the specifications. This method is usually called *active testing*, a technique based on gathering information actively. By "actively"

we mean injecting test messages into the network. Although widely used, active testing has some drawbacks. Its biggest drawback is that the tester has to interact directly with the network components, which disturbs inevitably their normal operations. Therefore, active testing does not fit certain situations such as operational networks, where inserting arbitrary inputs disturbs the environment and could in the worst case even provoke the crash of the services. Also, this approach requires to deploy a specific testing environment to execute test cases and to observe implementations reactions. Moreover, active testing may not be always be applied due to the complexity of systems, for example, when the tester is not provided with a direct interface to interact with the equipments.

Different from active testing, *passive testing* ([2], [3], [4], etc.) represents another interesting alternative. It is a technique based only on observing the behaviors of the system to be tested. The tester collects the interactive messages of network equipments and analyzes them to draw a conclusion of interoperability. Compared with active testing, the main advantage of passive testing is its non-intrusive nature. It does not disturb the normal operation of the protocol implementations, thus is suitable for interoperability testing in operational environment. Also, passive testing has other advantages: by using passive testing, no extra testing traffic overhead is introduced into the networks; no direct access interface to control the system is required, etc. For these reasons, we feel that passive testing is worth investigating, and consider in this thesis to use it for interoperability testing.

1.2 Scope, Motivations and Objectives

The work described in this thesis pertains to the interoperability testing of communication protocols, and by using passive testing technique in particular. Two main reasons account for the scope of the thesis: First, contrary to conformance testing which has been formalized in [1], interoperability testing, despite its importance, remains informal and has many open issues to be solved. Second, we find that the non-intrusive nature of passive testing makes it a suitable technique to perform interoperability testing (as stated in Section1.1). As interoperability focuses on the interworking of several protocol entities, verifying their behavior without disturbing their interaction is necessary.

The notion of passive testing can trace its history to several decades. For example it can be found in the early 70's to test integrated circuit. But not until the recent decade, the needs of network management [2] changes its research into a hot topic. The authors of [38] proposed some basic principles and methods of testing finite state machines using passive testing and presented a fault detection

algorithm in [3]. [40] adopted the model of communication finite state machine (CFSM) for passive testing and did some research on the problem of fault location. Later, passive testing is further applied in other areas such as network security [39], communication protocol testing [4, 5], etc.

Though passive testing is sometimes mentioned as an alternative to active testing, looking into the literature of interoperability testing, only few works [6] consider using passive testing techniques. Currently, the field of interoperability testing is dominated by active testing ([18, 19, 36], etc.). However, with the rapid advancement of computer science and technology, there are needs of more efficient and accurate testing procedure to help telecommunications suppliers achieve faster time-to-market. This, consequently, requires that the interoperability of protocol implementations be tested in operational environment. Different from in isolated environment, equipments that are running in operational networks cannot be shutdown or interrupted for a long period of time, and therefore call for the needs of passive testing.

Moreover, most existing research works are not based on a rigorous definition of interoperability and leave many unsolved issues and challenges: First of all, as interoperability testing involves at least two protocol implementations, therefore the joint behavior of implementations under test should be taken into account. But the calculation of the joint behavior often encounters the *state explosion*[55], amounts to exploring in a systematic manner the complete state space of a system. As a result, the number of generated global states is usually "large", making it difficult for practical use. Also, the non-controllable nature of passive testing makes the verification of observed behavior complex. Indeed, the tester has no idea about the state at which the implementations are when a trace is produced. This factor may cause interoperability verdicts error-prone. Moreover, as passive testing is often carried out in operational networks, therefore the conventional testing assumption of reliable testing environment (where there is no packet loss), is more or less no longer suitable. Last but not least, currently most interoperability testing is still carried out manually (test case generation and verification), which is time consuming and error prone. Due to this reason, interoperability testing is known to be costly, which calls a strong requirement for test automation.

Our contributions, in response to the above issues, intend to give improvements for more rigorous and better passive interoperability testing methods.

1.3 Organization of the Thesis

The thesis is organized in five chapters. The remainder of this thesis is as follows: Chapter 2 presents the necessary prerequisites. It begins with the introduction of the basic concepts of protocol testing activities, and in particular interoperability testing. Then the state of the art of interoperability testing, including different testing methods, i.e., active and passive testing, as well as their advantages and drawbacks are elaborated. In this chapter, we draw a conclusion on the needs of using passive technique to test interoperability. Also, the issues and challenges of passive interoperability testing are stated.

Chapter 3 proposes a formal specification-based approach for passive interoperability testing. Given the formal specifications of network equipments to be tested and a test purpose, a passive test case is derived by partial asynchronous interaction calculation to avoid unnecessary reachability analysis. In addition, a trace analysis algorithm is proposed to check a recorded execution trace against the passive test case in order to determine their interoperability relationship. Moreover in this chapter, the verdict assignment issue is discussed. Different from active testing, due to the uncontrollable nature of passive testing, verdict attributed to each test case must be prudent.

In Chapter 4, a passive testing approach is proposed for request-response protocols, which are widely used in the context of transactional communications. According to the interaction pattern of request-response protocols, the observed interaction between network components (trace) can be considered as a set of conversations made between network components. Then, a procedure to map each test case into the conversations is carried out in order to verify its occurrence, as well as whether it is respected. This approach was successfully put into operation during two CoAP Plugtest events in the context of the Internet of Things [22]. The results of case studies are also presented.

Moreover, Chapter 4 also presents the automation of passive interoperability by using a new testing tool called ttproto. It is inspired from the widely used tool Testing and Test Control Notation Version 3 (TTCN-3) [56] in the field of testing. However, TTCN-3 is not adapted for passive testing for several reasons: strong controllability, tedious templates, etc. To solve these problems, in ttproto, we have developed new features that do not exist in TTCN-3 and make the automation of passive interoperability testing feasible.

In the last chapter we draw a conclusion of the whole thesis and give perspectives for future works. We classify the future work into short term and long term work. Short term future work concerns mainly improving passive verification. We will give the first results of improving trace verification by grouping test cases

that share some common prefix. In this way, passive verification (which often involves a large number of traces to verify) can become more efficient. At the end, long term future work will also be presented.

Chapter 2

Prerequisites

This chapter aims at presenting the necessary prerequisites concerning the work presented in this thesis.

The organization of the chapter is as follows: Section 2.1 presents the general concept of protocol testing, including different testing methodologies and testing techniques. The importance of interoperability is also underlined in this section. Section 2.2 introduces the overview of interoperability testing, including the important elements, different testing architectures and testing process. In section 2.3, the state of the art of interoperability testing is described. As interoperability testing can be performed in both active and passive way, in this section we compare these two testing methods and develop in detail their advantages and drawbacks. Finally, we draw the conclusion of the needs of passive testing, as well as the various challenges and issues that might be encountered and to be solved.

2.1 Introduction to Protocol Testing

Communication protocols are the rules that govern the communication between the different components within a distributed computer system. Generally, a protocol defines the syntax, semantics, synchronization of communication, as well as specified behavior, which is independent of how it is to be implemented. Based on protocols, network equipments exchange messages within a communication system. In other words, protocols provide a reference for cooperation among network components of a system.

In order to guarantee successful communication, first of all the design of protocols must be checked for logical correctness. Then, as a protocol can lead to a variety of equipments, these implementations must be checked for compli-

ance with the protocol standard. Moreover, the last decade in the history of networking witnesses increasing heterogeneity, as existing networks evolved from analog voice networks to digital voice networks, from circuit-switched to packed-switched networks, from wire-line to wireless networks, from electronic to optical networks. New technologies introduce heterogeneity and complexity. In consequence, modern networks contain both legacy equipments and equipments based on state-of-art technology. They typically come from dozens of vendors. In this context, correct coordination and communication between different devices using a multitude of protocols are often problematic. As a result, implementations are usually also tested for other properties such as interoperability, robustness, performance, etc.

2.1.1 The Needs of Interoperability Testing

As introduced in Chapter 1, protocol testing can be classified into functional tests and stress and reliability tests. In this thesis, we focus on functional tests, more specifically interoperability testing.

Functional tests include typically conformance testing and interoperability testing. The relation between conformance testing and interoperability testing is illustrated in Fig.2.1. Compared to conformance testing that checks whether an implementation conforms to its specification, interoperability testing is in checking correct operation of multiple interconnected equipments.

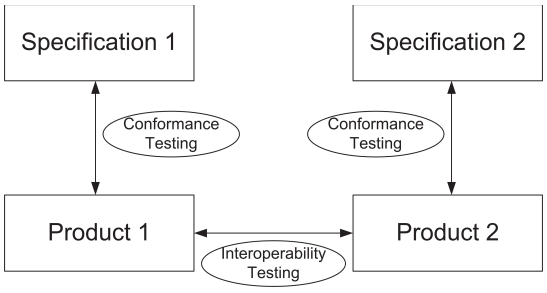


Figure 2.1: Conformance testing and interoperability testing

Conformance testing [1] for computer networking protocols is defined in ISO / IEC 9646-1:1994(E) as "*testing both the capabilities and behavior of an implementation, and checking what is observed against the conformance requirements in the relevant International standards.*" Its main objective is to verify whether a network component conforms to its specification. A specification is a detailed document describing the required behavior, as well as interfaces and mechanism of protocol entities. It is often defined by standards organizations such as ISO (In-

ternational Standards Organization)[1], IETF (Internet Engineering Task Force)[2], ITU (International Telecommunication Union)[3], etc. It is an important step to ensure compatibility of implementations. Nevertheless, it is widely agreed that conformance testing has limitation in ensuring interoperability [35, 36]. Even following the same standard, two different implementations might not be interoperable. In fact, since service provider networks may contain elements from different vendors, it is possible that errors are introduced by differences in implementations. Therefore, although much work in the standards arena has been done to ensure interoperability, non-interoperation can still be caused by a variety of reasons. For example, they may come from the following aspects (cf.Fig.2.2):

- Specifications. The specification of a protocol standard is in general, a detailed document describing the expected behavior of the protocol entities. If the description technique used is not formal, it is possible that the specification is ambiguous itself in some sense. The presence of ambiguities can lead to different interpretations and consequently different implementations. Such implementations may fail to interoperate. Also, the different options provided in a specification can be incompatible so that they defeat interoperation.

- Implementations. It involves for example human errors, e.g. programmer errors, different interpretations of the standard or different choice of options allowed by the standard. Also, sometimes a new service will enter the market before any standard exists for it, and the protocols involved may not necessarily be formally specified.

- Incompleteness of conformance testing. Conformance test suites are not complete so that they do not guarantee conformance to protocol standards. In fact, they just detect non-conformance that may appear during the test execution.

- Besides technical reasons, there are also business reasons that can account for non-interoperability. For instance, businesses sometimes view features that defeat interoperation as a competitive advantage, as it can lead to monopoly in the market.

[1]http://www.iso.org/iso
[2]http://www.ietf.org/
[3]http://www.itu.int/en/pages/default.aspx

To conclude, due to a variety of reasons, independently developed network protocol entities may be different enough to communicate with each other. Or sometimes they can communicate, but may not collaborate to provide the desired services.

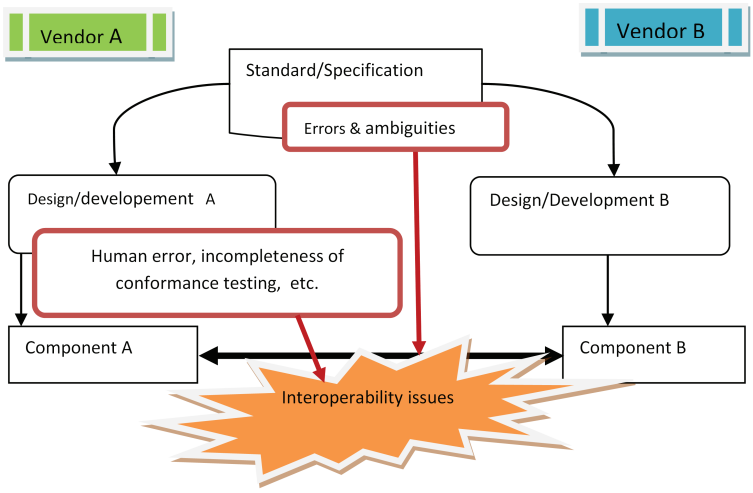


Figure 2.2: Needs of interoperability testing

However, poor interoperability can be expensive. Non-interoperability does not only affect the reputation of vendors, but also bring annoyance to the end customer, even cause the loss of investor confidence. Indeed, in a world of converging diverse technologies, complex network systems must communicate and inter-work on all levels in a multi-vendor, multi-network and multi-service environment. Interoperability ensures that users have a much greater choice of products and that manufacturers benefit from the economies of scale that a wider market brings. Interoperability testing is therefore a crucial factor in the success of modern technologies. This is one of our motivation in this thesis to suggest solutions that can improve interoperability testing.

A detailed description of interoperability testing can be found in Section 2.2.

2.1.2 Different Testing Techniques

In order to perform protocol testing, overtime the IT industry and the testing discipline have developed several techniques for analyzing and testing protocol applications. In this thesis, these techniques are classified according to the accessibility and controllability. Respectively, accessibility refers to whether the tester has access to the internal structure of the protocol applications, while controllability means whether the tester can stimulate the components to be tested or

not.

2.1.2.1 Testing Techniques According to Accessibility

Black-box Testing also known as input/output driven it is an important testing strategy. Black-box testing technique treats the system as a "*black-box*", i.e., in stead of explicitly using knowledge of the internal structure, it focuses on testing functional requirements by only accessing to the interfaces of the system. Applying this approach, test data are derived only from the specifications.

White-box Testing allows the tester to access the internal code of a system, and focuses specifically on using internal knowledge of the software to guide the selection of test data. It is a method that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs.

Gray-box Testing is a combination of white-box testing and black-box testing. It involves having knowledge of internal data structures and algorithms for purposes of designing tests, while executing those tests at the user, or black-box level. The tester is not required to have full access to the software's source code.

2.1.2.2 Testing Techniques According to Controllability

Active Testing is often done by applying a series of control and observation on the applications under test. Usually, the tester carries out the test in specific test environment, sends carefully designed inputs to the applications, receives what it responds and makes a conclusion.

Passive Testing consists in observing the input and output events of an implementation in run-time. It should not disturb the natural run-time of a protocol or service, that is why it is called passive testing. It is sometimes also referred to as monitoring. The record of the event observation is called an event trace. This event trace will be analyzed in order to determine the properties to be verified.

In this thesis, we consider the black-box testing context, which is generally used to perform functional tests. Moreover, we argue for the use of passive testing technique. The reasons about this choice are explained in Section 2.3.

2.2 Interoperability Testing Overview

2.2.1 Definition of Interoperability

Generally speaking, interoperability testing (*iop* for short in the sequel), assesses the end-to-end service provision across two or more products from different vendors. But when taking a closer look, interoperability testing can mean different things to different people. A number of statements can be found:

the IEEE Glossary defines interoperability as *the ability of two or more systems or components to exchange information and to use the information that has been exchanged* [47].

At ETSI, interoperability is defined as *the ability of two systems to interoperate using the same communication protocol, the ability of equipment from different manufacturers (or different systems) to communicate together* [48].

The definition of interoperability provided by Wikipedia is *a property of a product or system, whose interfaces are completely understood, to work with other products or systems, present or future, without any restricted access or implementation.*[4]

In [42], interoperability involves testing both, the capabilities and the behavior of an implementation in an interconnected environment and checking whether an implementation can communicate with another implementation of the same or of a different type.

In [41], different definitions of interoperability are also provided from different views of the product and the supplier network, and that of the user. The general definition of interoperability is "*the ability of two or more applications to communicate using a specific mechanism in a known environment, in order to achieve the objectives of the user.*"

Many other different definitions of interoperability may still be found in a number of research works. However, no matter what the definition is, the common points between them have given several objectives, which are:

1. The ability of systems to communicate with each other.

2. The ability to render the services requested by users.

To sum up, interoperability testing aims at ensuring that two or more network products communicate correctly while providing the required services.

[4]Wikipedia http://en.wikipedia.org/wiki/Interoperability

2.2.2 Interoperability Testing Architectures

Fig.2.3 illustrates the elements that compose the general architecture of interoperability testing. They are:

1. A Test System (TS) contains one or more test components that control or observe the behavior of the implementations under test.

2. A System Under Test (SUT) composed of at least two interconnected Protocol Implementations Under Test (IUT) from different manufacturers or product lines. Each IUT has interfaces name as Implementation Access Points (IAP) to communicate with the environment or with other IUTs.

3. The interfaces between an IUT and its adjacent layers within the SUT are referred to as points of control and observation (PCO). As the name suggests, at these points the behavior of the IUT in performing communication tasks can be controlled and observed. If an interface is only observable, it is then called Points of Observation (PO). Information gathered from these interfaces will be analyzed for verdict emission, which reveal the degree of interoperability of the IUTs.

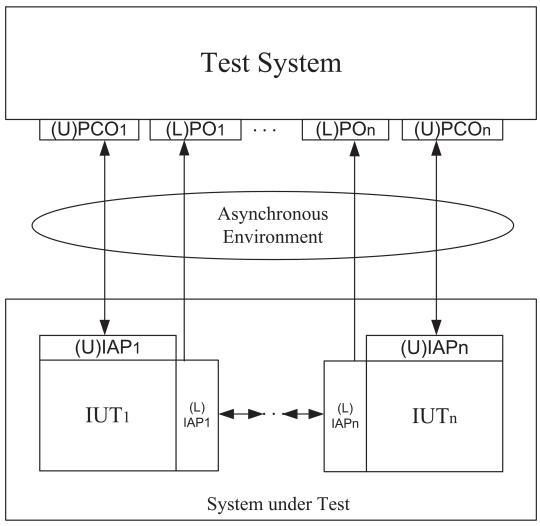


Figure 2.3: General interoperability testing architecture

According to the number of IUTs, interoperability testing can be done in either of the following context [19]:

- Multi-Component context where SUT has n ($n > 2$) IUTs. In this setting, an IUT is supposed to communicate simultaneously with several real open systems. A network of application relays, for instance, maintains communication links with several peers at the same time.

- One-to-One context where the test focuses particularly on the interoperability between exactly two IUTs. It is the most common context in practice, either by testing the interoperability of exactly two protocol implementations, or by testing the interoperability between one protocol implementation and a system already in proper operation composed of n implementations. For this reason, in this thesis we focus mainly on this context. In the sequel, the interoperability is by default one-to-one.

As IUTs communicate with other testing entities through interfaces, two kinds of interfaces can be further classified as follows. In Fig.2.3, the direction of an arrow represents the direction of messages transmission.

1. The lower interface $LIAP_i$ ($i = 1, ...n$, LI_i for short in the sequel) is only used for the interaction between two IUT. It is only observable but not controllable. A tester connected to such interface through interface LPO_i (LP_i for short in the sequel) can only observe the input/output events on that interface but not send any message to it. Indeed, any message sent to lower interfaces will disturb the normal operation of the IUTs.

2. The upper interface $UIAP_i$ (UI_i for short in the sequel) are the interfaces through which IUT communicates with its upper layer to provide the required services. In some testing architecture where upper interfaces are accessible, they can be used to both observe and control the corresponding IUT through interface $UPCO_i$ (UP_i for short in the sequel).

According to different situations, it is possible that the test system does not have the access to all interfaces of IUTs. In [19], different interoperability testing architectures with respect to the accessibility to different interfaces are classified. Respectively, an interoperability testing architecture is called:

- **Lower architecture** if TS has only access to the lower interfaces of the IUT(s). This architecture is possible in case where the upper interfaces of IUT(s) are embedded in its upper layer protocol. For example, in the test of IP protocol in operational TCP/IP protocol stack, the test system does

not have access to the upper interfaces of IP, which are indeed encapsulated in TCP. However, the lower IP interfaces can be observed by using some network analysis tools, for instance, one of the most common tools Wireshark[5].

- **Upper architecture** if TS has only access to the upper interfaces of the IUT(s). This situation is often encountered when the lower layer protocol prevents the display of messages transmitted to it.

- **Total architecture** if TS can access to both lower and upper interfaces. It is the most commonly used configuration in practice.

- **Unilateral architecture** if TS has only access to one of the IUTs. This situation is possible for example one of the IUTs is embedded in a system to which the tester does not have access. It can also be encountered for reasons of confidentiality: sometimes a manufacturer does not allow access to the interfaces of its implementation.

- **Global architecture** if TS has access to all the IUTs in a synchronous way.

- **Bilateral architecture** if TS has access to the IUTs but not in a synchronous way.

2.2.3 Interoperability Criteria

Before giving the definition of interoperability criteria we first underline the objectives of interoperability testing, which state that: n ($n \geq 2$) protocol implementations must communicate properly while rendering services specified in the corresponding specifications. Therefore, interoperability testing involves the following two aspects:

1. Verification of interaction: n implementations must communicate properly. i.e., the outputs sent by all the network components must be correct w.r.t the their specifications, and the messages sent by one network component to the corresponding receivers must be correctly received by the latter.

[5]http://www.wireshark.org/

2. Verification of service: the messages sent by the network components to their upper layer must match the services described in the corresponding specifications.

Based on the objectives, depending on different testing architectures, different *interoperability criteria* are used to give formal definitions of the interoperability. As we assume that interoperability testing is done in the context of black-box, the tester has no knowledge of the inner structure of the SUT. Therefore, interoperability can be only verified regarding the events produced by the IUT(s) from the interfaces. According to different architectures, different iop criteria can be defined [19]. Here we list the criteria of the most widely used testing architectures, lower iop criterion and global iop criterion. Respectively:

- The lower iop criterion consider the events produced from the lower interface(s) of the IUT(s) (according to the accessibility of the number of the IUTs). It says that: at any moment, the observed outputs produced from the lower interface(s) of the IUT(s) must be foreseen in the specification(s). Also, the IUT(s) must correctly receive the packets sent to it via the lower interface(s).

- The global iop criterion compares events executed by the IUTs during their interaction with the events described in the interaction of their specifications. It says that: two IUTs are considered interoperable iff, at any moment, all outputs observed during the interaction of the implementations are foreseen in the interaction of their specifications. Also, both IUTs must correctly receive the packets sent to them.

2.2.4 Interoperability Testing Process

2.2.4.1 Preliminary: Compatibility of Specifications

As they are developed, protocols must be described for many purposes. Early descriptions provide a reference for cooperation among designers of different parts of a protocol system. Moreover, as interoperability is black-box testing, this implies that a precise reference specification must be provided, which is the basis for the derivation of the test cases and the analysis of the test results.

Besides, one of the aims of communications standardization is to ensure that implementations of a single standard (or set of standards) will interoperate without the need for complex proprietary interfacing equipment. In order to achieve

this goal, it is necessary to consider interoperability right from the start. It requires that the specifications must be checked for logical correctness. Also, they must anticipate all of the likely scenarios which could arise from the interchange of commands and data between two implementations from different manufacturers.

In this sense, before any activity of interoperability testing, one preparatory work is to check the specifications in order to get rid of incompatibility problems. As described in [19], this property is called iop-compatibility. It aims to check that the options implemented by one IUT are compatible with the ones implemented by those IUTs interconnected to it in the used topology. For example, it is no use testing a functionality if the client implements an option while the server does not.

Another preparatory work requires the examination of both specifications and ICS (Implementation Conformance Statement): The ICS stipulates the features of the specification effectively implemented by the IUT. This step guarantees that an IUT is conform to its standard according to some specific context of conformance testing, which is considered a pre-requist for interoperability.

2.2.4.2 Interoperability Testing Activities

After the preparatory work, the test of the interoperability can be performed. In the sequel, we present the different stages of testing activities and corresponding objectives (c.f. Fig.2.4). They are:

1. The specification of the System Under Test SUT, which may be composed of one or more implementations. This phase includes the specification of one or more systems to be tested, the testing architecture, the description of the objectives of the test, etc. Respectively, the testing architecture represents the functional configuration in which the testing experiment will be undertaken. Note that it does not provide information for a concrete design of interfaces. This aspect is left to test laboratory and test execution concern.

   An objective of the test, also called *interoperability test purpose* (ITP) is derived from the requirements stated in one or more base specifications that define the implementation. It provides an essential abstraction of a test that specifies what is to be tested without going into the details of how a test is to be implemented. Test purposes are not test steps. They usually describe a sequence of actions (not necessarily consecutive) to be observed during the test run. Test purposes are written using the language and terminology of the base specification(s) and are independent of any particular

programming language, test system or platform on which corresponding tests might eventually be executed. They need to be developed, discussed and stabilized prior to any test case specification.

2. Generation of abstract *Interoperability Test Cases* (ITC). For each interoperability test purpose, an abstract test case is generated. A Test Case is the detailed set of instructions (or steps) that need to be taken in order to perform the test. In the case where the test driver is a human operator, these instructions will be in natural language. In the case where the tests are automated, they may be written in a programming or test language. The set of test cases is called a Test Suite (TS). A common test case includes a preamble, a testbody and a postamble. The test body is the set of events that are essential to achieve the test purpose. The preamble is the sequence of test events which drive the implementations into a state from which the test body can be performed. The postamble is the sequence of test events which drive the implementations back into their initial states after the test body is verified.

   A test case contains also associated verdicts with its possible executions. In detail, verdict *Pass* corresponds to the case where the expected behavior to be observed which can satisfy the test purpose. Verdict *Inconclusive* corresponds to the case where the behavior is allowed by the specifications, but can not satisfy the test purpose. Indeed, as it is generally impossible to completely control the SUT in a testing procedure, the observed behavior can be allowed in the specifications, however does not correspond to the test purpose. All other unexpected behavior is associated with *Fail* verdicts.

   Test cases can be either automatically computed if formal models of reference specifications are available, or manually obtained by studying carefully the specifications and specifying possible interactions that allow to satisfy the ITPs. Currently, most of interoperability test cases are generated manually.

3. Test deployment and execution. This stage involves of compilation and execution of test cases on the system under test. The results of the execution are then reflected by assigning a verdict *Pass*, *Fail* or *Inconclusive*.

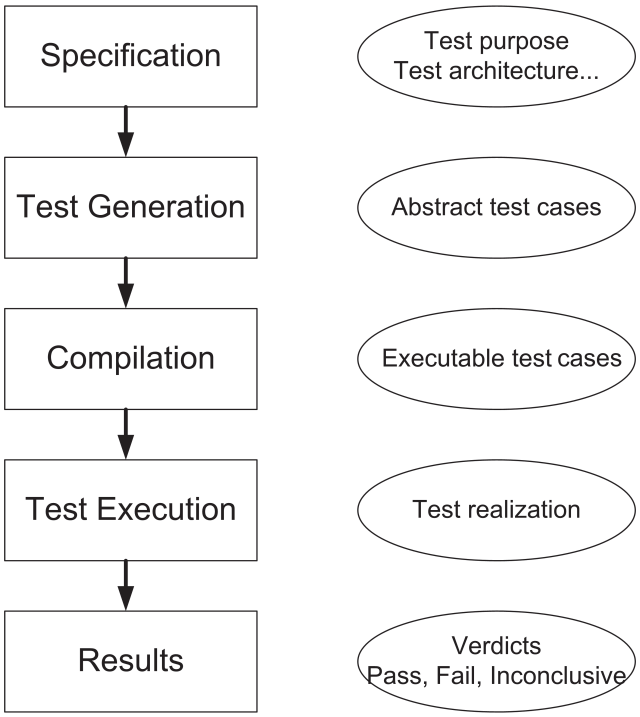


Figure 2.4: Interoperability Testing activities

2.3 State of the Art of Interoperability Testing

As systems used for most commercial service provisioning have evolved and involve today hundreds of hardware and software components, interoperability is one of the key factors to success when deploying new technologies for highly distributed systems. To ensure interoperability, efforts have been made by both industrial and research fields.

Interoperability testing in industry has always been practiced in the context of testing laboratories, field trials, and acceptance testing. Manual testing is today the most dominant way to perform interoperability testing. Typical examples using manual testing are interoperability testing events, which bring together products from different vendors purely for the purpose of executing tests for a period which may last from a couple of days to a couple of weeks. These events enable developers from different (and competing) companies to get together to test their companies' own implementations and ensure interoperability between products. Manual testing is however highly costly and leaves room for human error simply considering its repetitive nature and the number of entities and interfaces involved in the testing of these complex systems.

In research domain, many works address the problems of interoperability testing over the past decades, which put emphasize on test suite derivation.

No matter in industrial or research context, the two main approaches to interoperability testing can be broadly classified into two approaches: active testing and passive testing. The difference is that active testing allows controlled error generation and a more detailed observation of the communication, whereas passive testing on the other side involves observation the behavior only. In the sequel, we present the state of the art of interoperability testing, including elaborating both active and passive testing methods, as well as their advantages and drawbacks.

2.3.1 Active Interoperability Testing

Until now, the majority of research works on interoperability testing are based on active testing. Generally, the active testing paradigm considers this activity as consisting of three elements:

- Stimulating a SUT in order to place an IUT in one of the pre-defined conditions (states);
- Observing the behavior of SUT as it appears under the influence of applied stimuli.
- Analyzing the relation between the stimuli and observed behavior, in order to decide if this relation is as predicted by a pre-difined reference.

2.3.1.1 Active Interoperability Testing Overview

The research work on interoperability testing can be roughly classified into three categories: experimental results, general concepts and formal framework, and systematic generation of interoperability test suites.

Most of the early work belongs to the first category. Bonnes [43] presented their interoperability testing experiences at IBM. [44] describes their experiences with interoperability testing of FTAM protocol that uses a single tester between two IUTs that can observe and control the communication between two FTAM entities. There are also studies that focus on interoperability testing of some specific protocols or specific properties of the protocols tested. For example, [9] focuses on testing the interoperability of the specific protocols TCP/IP based system. Shin and Kang [59] proposed and applied a test derivation method

suitable for testing interoperability for the class of communication protocols like the ATM signaling protocol. [11] concentrates on time constraint in real-time systems. Most of these works however, are not based on a rigorous definition of interoperability.

Another testing method is based on formal framework and general concepts of interoperability: [37] gives a comprehensive discussion of various aspects of interoperability testing. [19] proposes a formal framework for iop testing and also discussed iop test generation. One methodology is based on the fault models [18] or the research of a particular error type [15]. Test suite design and derivation is based on these typical problematic to have a clear vision of whether IUTs work properly.

Most of the recent research work in the field is related to interoperability test suite derivation. One approach is to apply conformance test generation techniques on composed finite state machines, which are constructed from several components systems via a reachability analysis. A finite set of test cases is then selected to test the interoperability. [35] was one of the first papers in this field. It proposes the test generation procedure and also an interoperability test architecture. [17] reuses automatic generation of conformance test suites to define an approach for automatic generation of interoperability test suites. [61] have formalized the relation of interoperability by adapting the existing conformance relations. [18] proposed a test generation technique for protocol control portion of interoperability testing. Previous works can be classified as methods based on reachability analysis, that often suffer from the well-known state space explosion problem when applied to real case studies. To avoid state space explosion, some strategies have also been proposed: Griffeth et al. [60] presented a method for automatic generation of test cases to test a system interface. In order to avoid state space explosion, the method makes use of a single entity that interoperates with the rest of the integrated communication systems. [19] proved the equivalence between global and bilateral interoperability testing, and then used the feature of bilaterial interoperability architecture to generate one test case for each IUT. [62] proposed algorithms to generate a test suite that are based on stable states of their composition algorithm.

All the above work is based on an active testing approach, where the tester has the ability to stimulate the IUTs and verify whether the output obtained for each input is according to the specification.

2.3.1.2 Advantages and Drawbacks of Active Testing

As the most frequently used testing technique, active testing has several advantages: First, the strength of active testing method is its ability to focus on particular aspects of SUT. Test cases can be for example a specific error type or an important state of the specification. By sending well designed inputs to the SUT, the test system is able to check whether the response to each stimulus is consistent with the test cases.

Another advantage of active testing is the capability to control SUT. The SUT can be set to an initial state to execute the test cases, and reset or brought to a stable state afterward. Therefore, controllable test cases design is more preferable in practice to avoid possible non-determinism. A controllable test case means that in any state of the specification, if the state allows both a stimulus of the test system and a message sent by the IUT, test case should always choose to send the stimulus.

Although active testing is commonly used, it has some drawbacks:

First, the active testing technique does not suit protocol testing in operational networks. In fact, the arbitrary inputs disturb inevitably the normal operations of the SUT. Moreover, in the context where the network is in use, it is desired to keep testing traffic overhead to a minimum. Therefore, active testing is rather suitable for protocol testing in isolated environment. For example, in conformance testing, where an IUT is tested off-line to insure that it conforms to its specification. However, for interoperability testing, where at least two IUTs interact with each other, it is difficult to insert arbitrary inputs without affecting their normal functioning of the tested protocol applications as well as other services.

Second, active testing needs specific test environment and APIs so that the tester can interact directly with the SUT. However, this condition may not be necessarily satisfied. For example, for the testing between protocol layers, no direct access to the embedded layer is given. Therefore, active testing may not be feasible since the test system is not able to send stimuli to the SUT.

Moreover, with active testing, the test scope, the coverage and the duration are all limited by the test suite design. Indeed, the cost and complexity of putting together an entire network for testing suggests that pair-wise testing would be more cost-effective. But configuration of the network elements to obtain the required connectivity is a difficult, error-prone and time-consuming process. Moreover, to correctly draw the conclusion of interoperability, the model of the environment must be accurate, which requires both variety and depth of expertise.

Although the active testing method can be done to promote the quality of

network components, it can hardly cope with all these problems.

2.3.2 Passive Interoperability Testing

To cope with the limitations of active testing, passive testing has been studied with interest over the past few years. The generic passive interoperability testing architecture involves two interconnected IUTs. The communication between the IUTs is monitored. Contrary to active testing, in passive testing, the test system does not interact with the SUT, its role is limited to observing the behavior of the SUT (also name as *trace*) without controlling its inputs. Indeed, the passive interoperability testing architecture is based on an accurate level of observation in order to issue verdicts regarding the IUTs behavior with respect to the specifications.

We now breifly survey the main concerns, approaches, and results of research conducted on passive testing. In passive testing, the test system TS has two main roles:

1. Observe and collect the information exchanged between two protocol entities in the SUT.

2. Trace verification. The captured trace will be analyzed in order to determine whether the system operates correctly. Trace verification can be done either online [8] or offline. The idea of online testing is that the execution of a test pertains to behavior that the SUT genuinely exhibits during the test experiment, and the verdicts are issued during the test execution. For example, passive testing can be used to monitor the network for a long time and deal with abnormalities at any time. Else-wise it can be done offline. In offline testing, the behavior the the SUT is captured and then be assessed. i.e, the traces during the test execution are stored in a file and will be analyzed once the trace recording is finished [3], [6], [7].

2.3.2.1 Passive Testing Techniques

Currently, passive testing techniques can be broadly classified into two approaches: The first approach consists of recording the trace produced by the implementation under test and trying to find a fault by comparing this trace with the specification. The other approach explores relevant properties required for a correct implementation, which are named as invariants, and then check them on the traces produced by the system under test.

Trace matching approach is a passive testing technique which compares each event in the trace produced by the SUT strictly with that in its specification. The SUT is said to be faulty as soon as a deviation between the trace and the specification is detected. This method is widely used for the network fault management [2], [5], [7].

In [2], the trace matching technique was first proposed to realize the fault management of signaling system SS7. The formal model used is *Finite State Machine* (FSM) [49]. Trace matching technique was initially proposed for deterministic FSM, and later extended to non-deterministic FSM [50]. In [7], trace matching is used for fault detection in system modeled by communicating finite state machine (CFSM).

The approach is composed of two stages: *passive homing* and *fault detection* :

- Passive Homing

  In passive testing, no assumption is made about the moment when the recording of trace begins. Therefore, the state of the implementation at that moment it is not necessarily its initial state. Thus, the aim of passive homing is to find out the current state of the implementation. For doing this, the transitions of the observed trace are studied one after the other to narrow down the states the implementation can be in. At first, all states in the specifications are regarded as possible candidate states. Then, by checking each i/o pair in the trace. The states which accept the pair are replaced by the destination state of the corresponding transition. While the states that do not accept it are eliminated. After a number of iterations, there are two possibilities:

  (i) A single state is obtained: it is in fact the current state of the implementation machine.

  (ii) The set of possible states becomes empty, it means that the behavior of IUT does not correspond to the specification, a fault is detected.

- Fault Detection

  Once the current state is found, the other transitions in the trace can be followed from the current state in the specification. Once a state that does accept a transition in the trace is reached, there is an error. Otherwise no error is detected.

Invariant approach The foundations of property verification approach can be found in [4], [6], [12] etc. The idea is to study the specification so as to disclose particular properties. The properties to be verified are called *Invariants,* which means the selected properties should be true at any time. Once the invariants are chosen, they will be verified against the trace to see whether they are satisfied from the obtained trace. The verification approach is pattern matching such as string searching algorithm [13].

An invariant contains two parts:

1. Test part, which is an input or an output.
2. Preamble part, which is a sequence that must be found in the trace before verifying the test part.

The invariant is then used to process the trace: The correct behavior of the SUT requires that the trace exhibits the whole invariant. This approach has been used for conformance testing, network security, etc. [4], [12]. It is also used to perform interoperability testing [6] and applied on a case study of two WAP protocol (Wireless Application Protocol) [6] implementations.

Passive testing now proves to be a promising testing methods. However, in the field of interoperability testing, few work has been done to testing interoperability passively. In [6], the authors test the interoperability between two IUTs (a client and a gateway WAP based protocol implementations) in a passive way. A set of properties are chosen before the test. Two IUTs are judged to be interoperable if the expected invariants are verified. However, this paper is not based on a rigorous definition of interoperability. Moreover, no well described formal methodology (property choice, testing architecture, rules of verdict emission, etc.) is given.

2.3.2.2 Advantages and Challenges of Passive Testing

Compared with active testing, the major advantage of passive testing is that it detects the faults in the system without disrupting its normal operation. It is thus non-intrusive as the internal behavior of the implementations is not influenced. For this reason, passive testing is suitable to be deployed in operational networks, where an arbitrary insert may trouble or in a worse case cause the crash of the tested protocol services.

Passive testing also has another significant advantage: For embedded systems where the test system has no direct access, test can still be carried out by col-

[6] http://www.wapforum.org/what/technical.htm

lecting execution events of the SUT. Errors or deviations from these events will be detected by comparing them with the specification.

Besides, passive testing has other advantages such as it does not introduce extra testing overhead, which it makes a suitable technique to be deployed in resource constrained environment such as today's hot topic the Internet of Things.

Moreover, passive testing is less costly to deploy than active testing, as it does not need complicated test configuration as is required for active testing. Also, passive testing can be used to monitor operational networks for a very long period and report abnormality at any moment, etc.

Nevertheless, passive testing also has challenges: the biggest issue of passive testing is its the uncontrollable nature. A major difference between active testing and passive testing is that, in active testing, the test system is aware of the states of SUT during the test. Eventually, the test system has the ability to set the SUT to a certain state. On the contrary, in passive testing, the tester is not aware of the state in which the SUT was at the beginning of the trace. At the moment when the trace began, SUT could be at any state of its specification and not necessarily the initial state. This drawback makes it difficult for the test system to analyze the implementation validity.

Uncontrollable nature also brings another issue of passive testing is: verdict assignment. In fact, verdicts are based on the observed trace produced by the IUTs. However, the trace may not be long enough to encompass the property to be tested. In consequence, the tester does not know whether the properties to be verified are satisfied on the trace or not.

Last but not least, as passive testing is based on observation and implies a large quantity information to deal with, the needs of automation are necessary to make testing efficient.

2.3.2.3 Conclusions

In the above subsections, we have elaborated the active and passive testing methods, including various testing methodologies, as well as their positive and negative points. Active testing aims at detecting faults on SUT by applying a series of control and observation, while passive testing detects faults by only observing the behavior of the SUT. The following table shows the synthesis of different characteristics of these two testing categories.

From the table, we can see that these two approaches are almost complementary. Actually, they can be used in different situations. By analyzing the advantages and drawbacks of both methods, we realize that active testing approaches

are suitable for testing in specified environment, where the SUT is isolated, for example conformance testing. On the contrary, passive testing is more suitable to test interoperability, where the SUT contains at least two IUTs under interaction. Also it is suitable for testing in operational networks with strong requirements for proper cooperation of network devices.

	Active Testing	Passive Testing
Testing architecture	Controllable APIs are needed	Points of observation are needed
Impact on network environment	Interference	No interference
IUT controllability	Controllable	Non-controllable
Testing environment	Known, isolated	Operational
Deployment	Costly	Less expensive

Table 2.1: The comparison of active testing and passive testing

Indeed, current trends in protocol implementations deployment are to shorten time-to-market and test their interoperability in operational environment. In this sense, we argue for the application of passive approach on interoperability testing, and propose solutions to the related various issues.

2.4 Conclusions

Regardless of the application domain – telecommunication, transportation, health care, computation, or etc – end users today are mainly using services that are provided by distributed systems composed of products from different vendors. The overall system complexity is usually too high and costly for a single vendor to develop or maintain one product for the complete distributed system. Another challenge arises from the fact that multiple evolving technologies are continuously integrated and need to interoperate in such systems. Service providers use products from different vendors to reduce their cost to build the system needed to sell their service. In addition, service providers increasingly rely on working with other service providers to offer their service, e.g., telecoms operators. Finally, the end users expect to be able to use their services anytime from anywhere regardless of the composition of the system they are using. These facts require interoperability and without interoperability there is simply no chance to succeed in today's market place.

Currently in the field of interoperability testing, numerous work has been done to develop the solutions for active testing. Despite many efforts, passive interop-

erability testing contains a number of aspects that present difficult problems to the tester.

First, due to the large amount of products and standards involved in complex distributed systems, interoperability testing is a manual, extremely time consuming, cost intensive, and repetitive task. The high amount of required test executions mainly comes from the fact that interoperability testing is not transitive: If a product A interoperates with a product B and C, it does not necessarily mean that B interoperates with C. Another issue is that technically each new product release requires that all interoperability tests have to be re-executed again. This results in a clear market need for a approach to automate interoperability testing.

Then, the multi-IUTs nature of interoperability testing makes existing passive testing methodologies impractical. Typically, trace mapping approach does not suit interoperability testing: As the SUT concerned in interoperability testing involves several IUTs, to model the SUT by a single FSM encounters state explosion [35] - a drastic increase in the number of states and transitions in the resulting FSM.

Also, although the invariant approach has been used to perform interoperability testing, it is not based on a rigorous definition of interoperability.

Moreover, due to the uncontrollable nature of passive testing, it is not always easy to draw a correct verdict. This issue however, in most of the works, is neglected.

To conclude, there does not exist a well enough defined methodology for passive interoperability testing. To the best of our knowledge, there is no clearly specified architecture or rule to carry out interoperability testing by using passive testing.

Corresponding to the above challenges, in the following of the thesis, we propose:

- Methodologies for interoperability test case generation based on passive approach.
- Algorithms for trace verification.
- A testing tool to automate passive interoperability testing.

The methods, algorithms and tools have been applied to various experimentations on real protocols. The obtained results show the interests of our approaches.

Chapter 3

A Method for Passive Interoperability Testing

3.1 Introduction

Today's communication system, with the never-ending emergence of new protocols and services, lead to heterogeneous networks that need to interoperate while providing the required quality of service. In this context, interoperability testing (*iop* for short in the sequel) is of a crucial importance to guarantee successful integration of network components in communication networks.

To perform interoperability testing, nowadays the most widely used approaches rely on the active testing method. As stated in Chapter 2, the strength of active testing is its ability to control the IUTs and evaluate their behavior in particular circumstances. Nevertheless, it has some limitations too: Test can be difficult or even impossible to perform if the tester is not provided with a direct interface to stimulate the IUTs, or in operational environment where the normal operation of IUTs cannot be disturbed or interrupted for a long period of time.

To cope with the drawbacks of active testing, passive testing has been studied for the verification of network components by only observing their external behavior. In Chapter 2, we have compared active testing and passive testing and drawn a conclusion that passive testing is an appropriate technique to perform interoperability testing. However, currently in the field of interoperability testing, to our knowledge, not only few works consider the passive testing technique, but also there lacks a well described methodology.

This chapter, arguing for the passive testing technique, presents a specification-based methodology for passive interoperability testing. The approach first derives a passive iop test case by calculating partially the interaction of the protocol im-

plementations' specifications with respect to a given iop test purpose. Then, to evaluate the network components' behavior, a trace analysis algorithm is proposed. The suggested methodology has been successfully performed on a Session Initiation Protocol (SIP) case study, where some non-interoperability behaviors were detected.

The chapter is organized as follows. Section 3.2 presents the used passive testing architecture. In section 3.3, related preliminaries, the formal model and passive interoperability testing criteria are presented. Section 3.4 presents the methodology proposed to carry out passive interoperability testing . The application of the proposed approach and the experimental results are exhibited in section 3.5. Finally, section 3.6 gives the conclusions and the perspectives.

3.2 Testing Architecture

The passive interoperability architecture considered in this chapter (cf. Fig. 3.1) involves a *Test System (TS)* and a *System Under Test (SUT)* composed of 2 *Implementations Under Test (IUT)* from different product lines.

The testing activities rely on observing the external behavior of IUTs (black box testing). As required by passive testing, no test message is injected. *Points of observation* (PO) are installed at different interfaces, which allow the test system to capture the traffic carried by the communication medium:

Respectively, *Upper Points of Observation* (UPO) collect the messages sent by the IUTs to their upper layer through their *Upper Interface* (UI), while *Lower Points of Observation* (LPO) collect the messages exchanged between peer IUTs through their *Lower Interface* (LI). In this chapter, we consider the TS is able to reconstructs a global traces from the trace retrieved by different POs.

Moreover, in this thesis we consider that the interaction between two IUTs is asynchronous [10], as messages may be actually exchanged by traversing several protocol layers and networks. In consequence, the communication between two IUTs can be modeled as two unidirectional FIFO (first in first out) queues [10].

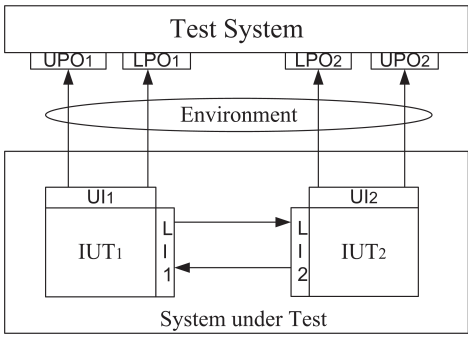


Figure 3.1: Passive interoperability testing architecture

3.3 Formal Model

Specification languages for reactive systems can often be given a semantics in terms of labeled transition systems. In this thesis, we use the IOLTS (Input-Output Labeled Transition System) model [10], which allows differentiating input, output and internal events while precisely indicating the interfaces specified for each event. In our research, IOLTS is used to model the specifications, the interaction among testing entities, etc.

Definition 3.1 An IOLTS is a tuple $M = (Q^M, \Sigma^M, \Delta^M, q_0^M)$ where:

- Q^M is the set of states of M with q_0^M its initial state.
- Σ^M is the set of observable events at the interfaces of M. In IOLTS model, input and output actions are differentiated: We note $p?a$ (resp. $p!a$) for an input (resp. output) a at interface p.
- $\Delta^M \subseteq Q^M \times (\Sigma^M \cup \tau) \times Q^M$ is the transition relation, where $\tau \notin \Sigma^M$ stands for an internal action.

Other notations Let us consider an IOLTS M, and let an observable event $\alpha \in \Sigma^M$ with $\alpha = p \cdot \{?, !\} \cdot m$, a succession of events $\sigma \in (\Sigma^M)^*$, and states q, $q' \in Q^M$. We define the following notations:

- Σ^M can be partitioned into: $\Sigma^M = \Sigma_U^M \cap \Sigma_L^M$, where Σ_U^M (resp. Σ_L^M) is the set of events at the upper (resp. lower) interfaces. Σ^M can also be partitioned to distinguish input (Σ^{M^I}) and output events (Σ^{M^O}).

- The transition relation is noted as $(q, \alpha, q') \in \Delta^M$.

- q after σ is the set of states which can be reached from q by the sequence of actions σ. By extension, all the states that can be reached from the initial state of the IOLTS M are (q_0^M after σ) and will be noted by (M after σ).

- $Out(q)$ is the set of possible outputs at state q. Similarly, $In(q)$ is the set of possible inputs and $\Gamma(q)$ is the set of all possible events at state q. By extension, $Out(M, \sigma)$ is the set of executable events by the system M after the trace σ.

- The projection of an IOLTS on a set of events is used to represent the behavior of the system reduced to specific events. For example, the projection of M on the set of executable events on its lower (resp.upper) interfaces Σ_L^M (resp.Σ_U^M) is noted M/Σ_L^M (resp. M/Σ_U^M). It is obtained by hiding events (replacing by τ-transitions) that do not belong to Σ_L^M (resp.Σ_U^M), followed by determinization. In the same way, Out_X (M, σ) corresponds to a projection of the set of outputs (M, σ) on the set of events X.

The testing theory we consider in this chapter is based on the notions of specification, implementations and interoperability criterion. Both specifications and IUTs are assumed to be IOLTS. Besides, we recall that our work is in the context of *black box* testing: The test system has no knowledge of the IUTs' inner structure. Only their external behavior can be evaluated.

Asynchronous Interaction In this thesis, we consider that in interoperability testing, the interaction between two IUTs is asynchronous [10], [19] as messages may be actually exchanged by traversing several protocol layers. According to [10], asynchronous interaction is usually modeled as two unidirectional FIFO queues. Formally, asynchronous interaction is noted $S_1 \|_A S_2$ for two communicating IOLTS S_1 and S_2. A state in $S_1 \|_A S_2$ is a four-tuple (q_1, q_2, f_1, f_2) where $q_1 \in Q^{S_1}$, $q_2 \in Q^{S_2}$, while f_1 (resp.f_2) stands for the strings of messages in the input channel of S_1 (resp.S_2). Informally, a state (q_1, q_2, f_1, f_2) means that the execution of S_1 and S_2 have reached state q_1 and q_2 respectively, while the input channels of S_1 and S_2 store the strings f_1 and f_2 respectively.

Passive one-to-one Interoperability Criterion says that: Two IUTs are considered interoperable iff after a trace of the interaction of the IUTs, all outputs observed must be foreseen in the corresponding interaction of their specifications.

The passive one-to-one interoperability criteria is stated in a more formal way in the following (where$=_{def}$ means by definition):

Definition 3.2 $I_1 iop I_2 =_{def} \forall \sigma \in Traces\ (S_1 \parallel_A S_2),\ Out\ (I_1 \parallel_A I_2,\ \sigma) \subseteq Out\ (S_1 \parallel_A S_2,\ \sigma)$

3.4 Passive Interoperability Testing Method

3.4.1 Passive Interoperability Testing Method Overview

As introduced in Section 2.3.2, the existing research on passive testing can be broadly classified into two approaches: trace mapping and invariant approach. However these approaches have some limitations to be used for interoperability testing:

Trace mapping approach aims at evaluating the behavior of the SUT by comparing the trace produced by the SUT strictly with the specification. The specification of the SUT is modeled as a *Finite State Machine* (FSM). With each observed message in the trace, the FSM is traversed to check if this message can be accepted by the FSM: The correct behavior of the SUT implies that the whole trace should be accepted by the specification. This approach has been applied to protocol conformance testing [5], network management [2], etc. But it does not suit interoperability testing. As the SUT concerned in interoperability testing involves several IUTs, to model the SUT by a single FSM encounters state explosion [35] - a drastic increase in the number of states and transitions in the resulting FSM.

The invariant approach involves extracting a set of important properties of the SUT (named as invariants) from the specification. Each invariant is composed of a preamble and a test part, which are cause-effect events respectively w.r.t a property. The invariant is then used to process the trace: The correct behavior of the SUT requires that the trace exhibit the whole invariant. Invariant approach has been used to perform interoperability testing [6] and applied on a case study of two WAP protocol implementations: A range of invariants were chosen from the specification of WAP. Two WAP IUTs are determined interoperable if the invariants are satisfied on the recorded trace. However, the work is not based on formal interoperability definitions. Moreover, in this work, no clearly defined general method for passive interoperability testing is provided.

To overcome the problems above, in the sequel we will introduce an approach to passive interoperability testing based on formal interoperability definitions. The approach contains two main stages:

1. Passive interoperability test case generation.

2. Trace analysis.

The approach involves choosing a set of interoperability test purposes (ITP). Then, for each ITP, a passive interoperability test case is derived which describes in detail the expected observable behavior of the IUTs regarding the ITP. At last, a trace analysis algorithm is proposed to evaluate the behavior of the IUTs w.r.t the ITP.

The outlines of the passive interoperability testing are illustrated in the following figure. The start point is the *interoperability test case* (ITC) generation algorithm. The inputs of the algorithm are the specifications (S_1 and S_2) on which the IUTs are based, and an *interoperability test purpose* (ITP). The ITC is obtained by carrying out partial asynchronous interaction calculation of S_1 and S_2 with respect to the ITP, so that different ways of specifications' interaction to reach the ITP are obtained.

Then, a *passive iop test case* (PITC) is derived by only keeping the relevant observable events w.r.t the ITP. Also, appropriate verdicts will be attributed to corresponding states.

During test execution, the trace produced by two IUTs are recorded by a sniffer. Once the trace recording finishes, the verification of the ITP on the trace is performed and an appropriate verdict *Pass, Fail or Inconclusive*, is emitted. Respectively, *Pass* means that the ITP is successfully verified on the trace. *Inconclusive* means the behavior of the SUT is allowed in its specification, however does not satisfies the ITP. *Fail* means that an non-interoperable behavior is detected.

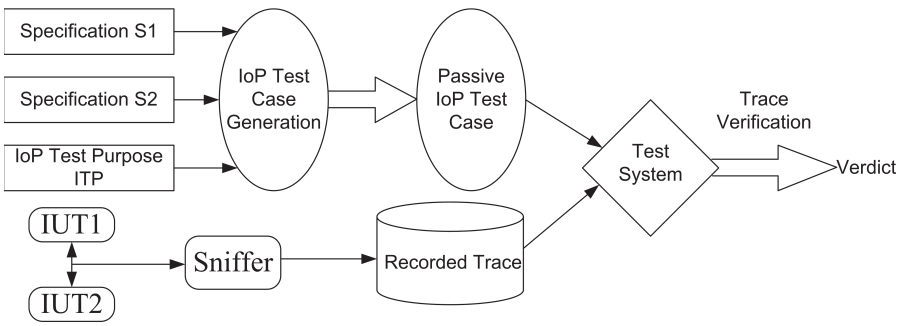


Figure 3.2: Passive Interoperability Testing Methodology

Detailed description of each step can be found in the following subsections.

3.4.1.1 Formalizing Interoperability Test Purpose

The goal of passive interoperability testing is to test the valid behaviour of a SUT [42]. However proving correctness is elusive as it is generally impossible to validate all possible behavior described in specifications. Normally, the behavior of a SUT can be obtained by computing the global behavior of the IUTs. But this technique often encounters the *state explosion* problem [15]. i.e., the number of states of the asynchronous specifications' interaction is in the order of $O((n.m^f)^2)$, where n is the state number of the specifications, f the size of the input FIFO queue on lower interfaces and m the number of messages in the alphabet of possible inputs on lower interfaces. This result can be infinite if the size of the input FIFO queues is unbounded.

To relief the state explosion problem, in this thesis, we argue for the use of iop test purposes (ITP) to generate interoperability test case. An ITP is in general informal, in the form of an incomplete sequence of actions representing a critical property to be verified. It can be designed by experts or provided by standards guidelines for test selection [24]. With an ITP, only the interesting parts of the global behavior have to be computed: The interoperability test case generation corresponds to partial asynchronous interaction calculation of S_1 and S_2 on the fly with ITP.

Moreover, Each ITP is assigned with attributes *Accept* or *Refuse* to indicate the important states to be explored as well as to inhibit unnecessary states exploration. The formal definition of an ITP is as follows:

Definition 3.3 an **interoperability test purpose** ITP is a *partial* (unnecessarily consecutive) sequence of actions representing a critical property to be verified. Formally, an ITP is a deterministic complete acyclic IOLTS. ITP = (Q^{ITP}, Σ^{ITP}, $\triangle^{ITP}$, q_0^{ITP}) where:

- $\Sigma^{ITP} \subseteq \Sigma^{S_1} \cup \Sigma^{S_2}$. where S_1 and S_2 are the specifications on which the IUTs are based.
- Q^{ITP} is the set of states. An ITP has two sets of trap states $Accept^{ITP}$ and $Refuse^{ITP}$. $Accept^{ITP}$ represents the state indicating that the iop test purpose has been satisfied and is associated with attribute *Accept*. $Refuse^{ITP}$ stand for possible options in specifications rather than the desired behavior to be verified, and are associated with attribute *Refuse*. $Accept^{ITP}$ and $Refuse^{ITP}$ are only directly reachable by the observation of outputs produced by the IUTs. This is because in black box testing, the test system can only know if a message has been sent to an IUT, but not whether this IUT has actually processed the message.

- ITP is complete, which means that each state allows all actions. This is done by inserting "$*$" label at each state q of the ITP, where "$*$ " is an abbreviation for the complement set of all other events leaving q. By using "$*$" label, ITP is able to describe a property without taking into account the complete sequence of detailed specifications interaction.

An example of ITP can be found in Fig.3.5-(a).

3.4.1.2 Interoperability Test Case Generation

The interoperability test case ITC corresponds to partial asynchronous interaction calculation $S_1 \|_A S_2$ w.r.t ITP: ITC calculate the different ways of specifications' interactions that allow reaching ITP. Guided by the ITP, only the necessary part of the global behavior of S_1 and S_2 is computed. Indeed, the attributes *Accept* and *Refuse* in each ITP inhibit the generation of unnecessary transitions. In fact, asynchronous interaction calculation is stopped as soon as an attribute is reached.

Definition 3.4 an **interoperability test case** is an IOLTS implementing the detailed procedure to observe and control the IUTs for satisfying a given ITP. It corresponds to partially calculating the asynchronous interaction of S_1 and S_2 w.r.t the ITP. Formally, $ITC = (Q^{ITC}, \Sigma^{ITC}, \triangle^{ITC}, q_0^{ITC})$ where:

- $\Sigma^{ITC} \subseteq \Sigma^{S_1} \cup \Sigma^{S_2}$.
- Q^{ITC} is the set of the states. Each state $(q^{S_1}, q^{S_2}, q^{ITP}, f_1, f_2) \in Q^{ITC}$ is a composite state such that $q^{S_1} \in Q^{S_1}$, $q^{S_2} \in Q^{S_2}$, $q^{ITP} \in Q^{ITP}$, where f_1 (resp.f_2) represents the input queue of S_1 (resp.S_2). $q_0^{ITC} = (q_0^{S_1}, q_0^{S_2}, q_0^{ITP}, \varepsilon, \varepsilon)$ is the initial state of ITC, where $q_0^{S_1}, q_0^{S_2}, q_0^{ITP}$ are the initial states of S_1, S_2 and ITP respectively. ε denotes an empty input queue. $Accept^{ITC}$ and $Refuse^{ITC}$ are two sets of trap states in *ITC*, where $Accept^{ITC} = Q^{ITC} \cap (Q^{S_1} \times Q^{S_2} \times Accept^{ITP}, f_1, f_2)$. $Refuse^{ITC} = Q^{ITC} \cap (Q^{S_1} \times Q^{S_2} \times Refuse^{ITP}, f_1, f_2)$.

The set of transitions $\triangle^{ITC}$ is obtained in the following way: Let $q^{ITC} = (q^{S_1}, q^{S_2}, q^{ITP}, f_1, f_2)$ be a state in ITC, and a be a possible event at state q^{S_1} or q^{S_2} or q^{ITP}. A new transition $(q^{ITC}, a, p^{ITC}) \in \triangle^{ITC}$ is created according to the following conditions:

1. If ITP is in neither *Accept* nor *Refuse* state, check the *asynchronous interaction operation rules* below in Definition 3.5. If one rule is executable, a new transition in ITC labeled by a is created according to the rule.

2. If ITP is in *Accept* or *Refuse* state, no new transition in created. i.e., *Accept* and *Refuse* inhibit unnecessary global behavior calculation.

Definition 3.5 Asynchronous interaction operation rules

Rule1 If a an upper interface event of S_i $(i = \{1, 2\})$, a new transition in ITC labeled by a is created. Upper interface event does not influence the communication channels of the IUTs. $q(q^{S_i}/p^{S_i})$ represents that in the global state q^{ITC}, the local state q^{S_i} is change to p^{S_i}, and other local states keep unchanged.

$$\frac{a\in\Sigma_U^{S_i},(q^{S_i},a,p^{S_i})\in\triangle^{S_i},(q^{ITP},a,p^{ITP})\in\triangle^{ITP}}{(q^{ITC},a,p^{ITC})\in\triangle^{ITC},p^{ITC}=(q(q^{S_i}/p^{S_i}),p^{ITP},f_1,f_2)}$$

Rule2 If a is a lower interface output of S_i , a new transition in ITC labeled by a is created. a is put into the tail of the input queue of S_j (i,j=$\{1,2\}$, i$\neq$j). $f(f_j.a))$ represents that a is put into the tail of the input queue of S_j.

$$\frac{a\in\Sigma_L^{S_i^O},(q^{S_i},a,p^{S_i})\in\triangle^{S_i},(q^{ITP},a,p^{ITP})\in\triangle^{ITP}}{(q^{ITC},a,p^{ITC})\in\triangle^{ITC},p^{ITC}=(q(q^{S_i}/p^{S_i}),p^{ITP},f(f_j.a))}$$

Rule3 If a is a lower interface input, and a is the first element in the S_i, a new transition in ITC labeled by a is created. $f(f_i \setminus a))$ represents that a is removed from the head of the input queue of S_i.

$$\frac{a\in\Sigma_L^{S_i^I},a\in head(f_i),(q^{S_i},a,p^{S_i})\in\triangle^{S_i},(q^{ITP},a,p^{ITP})\in\triangle^{ITP}}{(q^{ITC},a,p^{ITC})\in\triangle^{ITC},p^{ITC}=(q(q^{S_i}/p^{S_i}),p^{ITP},f(f_i\setminus a))}$$

The *asynchronous interaction operation rules* allow building recursively the *ITC* from its initial state q_0^{ITC}. At each state, a transition labeled by a can be created in ITC if it is firable:

(i) in both S_1 (or S_2) and ITP, or

(ii) in S_1 (or S_2) only.

It means that the state exploration in S_1, S_2 and ITP always progresses but the automaton ITC progresses only when one of the above rules is matched.

The ITC generation in fact, is based on partial reachability calculation. In this thesis, the ITC generation algorithm uses a depth-first traversal to construct a composition IOLTS of both specifications S_1, S_2 and the ITP to minimize memory requirement:

From the initial state q_0^{ITC}=$(q_0^{S_1}, q_0^{S_2}, q_0^{ITP}, \varepsilon, \varepsilon)$, at each step we check if a transition could be created according to the asynchronous interaction operation rules until no more state can be explored. Three cases are possible:

1. At state q, only one rule can be applied: The corresponding rule is applied to generate a new transition in ITC.

2. At state q, several rules can be applied. Such q is called *branching state* and stored in a stack. Then we choose one of the applicable rules to carry out ITC construction until no more rules can be applied. i.e., a state in $Accept^{ITC}$ or $Refuse^{ITC}$ is reached. Then we backtrack to state q and choose another possible rule to continue ITC construction. If all possible rules at state q have been applied, q is removed from the stack.

3. At state q, no rule can be applied. Then, we check if the stack that stores branching states is empty. If it is not empty, we move to the top of the stack and continue ITC construction. Otherwise no more state can be explored, the algorithm exits.

An example of ITC can be found in Fig. 3.6.

The *ITC Construction* algorithm is written below in a formal way:
Variables and Functions for depth-first ITC construction:

- *Current* : The current state under construction. Initially, Current:= q_0^{ITC}.

- *Rules(q)*: Function *Rules(q)* returns two values *q.succ* and *q. nb _ succ*: *q.succ* is the set of the successor states of state q that can be generated according to the rules in Definition 3.5. *q.nb_succ* is the number of the successor states of *q.*

- *q.visited* : The set of the transitions at state q that have already been generated. The operation of this set involves: *add (q.visited, (q→p)):* add a transition from q to p in *q.visited.*

- *Create_transition (ITC, (q→ p))*: Generate a transition from state q to state p in ITC according to the rules in Definition 3.5.

- *End_Exploration*: Boolean equal to *TRUE* iff no more state can be explored.

- *Branch*: A stack to store branching states. The operations of *Branch* involves: *Push* adds a new item to the top of *Branch; Pop* removes an item from the top of *Branch; Stacktop* returns the value of the item from the top most position of *Branch* without deleting it.

Algorithm 3.1 Depth-first *ITC* Construction

Input : S_1, S_2,ITP
Output : ITC
Initialization: $Current$=q_0^{STP}, $Branch$=$\emptyset$, Current.visited =$\emptyset$, End_Exploration=False
while *not End_Exploration* **do**
  Rules (Current)
  // If only one rule can be applied
  if *Current.nb_succ ==1* **then**
    Create_transition (ITC, (Current→ Next)) where Next ∈ Current.succ
    Current = Next
  end
  // If at least two rules can be applied
  if *Current.nb_succ>=2* **then**
    if $Current \notin Branch$ **then**
      Push(Branch,Current)
    end
    if $\exists$ *Next where Next ∈ Current.succ* $\wedge$ *(Current→ Next)* $\notin$ *Current.visited*
    then
      add (Current.visited, (Current→Next))
      Create_transition (ITC, (Current→Next))
      Current = Next
    end
    // All rules at current states have been checked
    else
      Pop(Branch, Current)
      Current.visited=$\emptyset$
      if $Branch \neq \emptyset$ **then**
        Current =Stacktop (Branch)
      end
      else
        End_Exploration = True
      end
    end
  end
  else
    if $Branch \neq \emptyset$ **then**
      Current =Stacktop (Branch)
    end
    else
      End_Exploration = True
    end
  end
end

3.4.1.3 Passive Interoperability Test Case Derivation

The Specifications - ITP asynchronous calculation product ITC is an acyclic graph that exhibits the different ways of $S_1 \|_A S_2$ to reach ITP. In order to derive a passive interoperability test case, the ITC should be processed.

Actually, the ITC obtained by Algorithm 3.1 can be divided into two parts: the preamble and the testbody. Note that generally a complete active test case includes a preamble, the testbody and a postamble. The preamble is used to bring the SUT into a specific state before the verification of the test body. The test body represents the detailed behavior related to the ITP. While the postamble is used to bring the SUT to a stable state after the verification of the testbody. However in passive testing it can not always be feasible as the test system has no control over the SUT, consequently the test system can not always observe the SUT to enter a desired stable state. Moreover, to obtain a postamble implies further asynchronous interaction calculation. Therefore, in this thesis, we donot calculate postamble. The ITC only contains a preamble and the testbody.

1. The preamble involves the states and transitions that begin from q_0^{ITC} to (not including) the state $q^{ITC} = \left\{q^{S_1}, q^{S_2}, q_0^{ITP}, f_1, f_2\right\}$ where (q^{ITC}, a, p^{ITC})$\in \triangle^{ITC}$ and (q_0^{ITP}, a, q_1^{ITP}) $\in \triangle^{ITP}$. (a is the common lable in both ITC and ITP). In other word, the preamble involves the states and transitions that traverse the initial state of ITC and (not include) the first transition concerned by the test purpose.

2. The rest is the testbody, representing the relevant events that should be performed by the IUTs, which are related to the chosen ITP.

Then, the derivation of a *Passive Iop Test Case* (PITC) consists of traversing the ITC graph, applying the following operations to extract a sub-graph of ITC:

- Only the testbody is preserved. ITC calculates the different ways to reach ITP from the initial state of S_1 and S_2. But in passive testing, the state where SUT can be in w.r.t ITC at the beginning of a recorded trace is not known. As a result, it is probable that the trace does not start from the initial state of the SUT. In fact, the preamble is usually used in active testing to put the SUT into a desired state to begin the verification of the testbody. However passive testing is only based on observation. Consequently, the preamble is not needed. i.e., PITC describes the detail events that are relevant to the property to be verified (ITP), which should be performed by the IUTs.

- Only observable events in ITC are preserved. The obtained ITC involves both *controllable* events and *observable* events. Controllable events represent the stimuli sent by the upper layer of an IUT and should be removed, as passive testing is only based on observation.

- Only output events sent by the IUTs are preserved. In black box testing, an input of IUT is in fact an internal event that cannot be directly observed.

All the events that need to be removed in ITC are replaced by an undistinguished action denoted by τ. The other events in ITC are extracted by applying τ - reduction, which hides internal actions τ from ITC followed by determinization: Hiding τ actions means automatically reducing the state space of the ITC by those states, i.e., a sequence of states. Then the ITC is determinized to get a deterministic PITC.

Moreover, verdicts should be assigned to the trap states of PITC. The state in $Accept^{ITC}$ is associated with Pass verdict, while states in $Refuse^{ITC}$ are associated with Inconclusive verdicts (traces that cannot be extended to $Accept^{ITP}$).

An example of PITC derivation can be found in Section 3.5.1.

3.4.2 Trace Verification

The derived passive iop test case PITC represents the expected behavior of the IUTs w.r.t the ITP. Contrary to active testing, where test case is run on IUTs, PITC will not be executed on the IUTs. In other words, PITC is only used to analyze the observed trace (messages produced by the IUTs). The correct behavior of IUTs implies that the trace produced by the IUTs should exhibit the events described in the set of PITCs.

However, in passive testing, the test system has no knowledge of the state where the SUT can be in w.r.t PITC at the beginning of the trace. In passive testing, no assumption is made about the moment when the recording of trace begins, and thus it is not necessarily the initial state of the PITC. Therefore passive testing trace analysis is not merely comparing each event in the trace with the PITC from its initial state. In order to realize the trace analysis, we propose an algorithm, which aims at reading all branches in the PITC graph in parallel and checking whether the recorded trace σ encompasses at least one branch in PITC which is assigned with the *Pass* attribute. i.e., to check whether the corresponding ITP is reached.

Let us consider the PITC graph, its initial state q_0^{PITC} which may contain several transitions to other states. The trap states of PITC are associated with

attributes *Pass* and *Inconclusive*. Note that there might be several Inconclusive attributes in a PITC tree. The idea is to find which of the branches in PITC are encompassed by the trace. Specifically, we call *States_under_reading* the list of states in PITC under reading. Initially, *States_under_reading* contains only the initial state q_0^{PITC}. Then, for each event a taken in order from σ, we check whether a can be accepted by the states in the set *States_under_reading*. If it is the case, these states are replaced by the destination states led by a. The states (except the initial state q_0^{PITC}) that can not be led to other states by a are deleted for coherence reason. In fact, these states can not be taken into account in the next step. Because it is possible, but not consistent, to have a transition from these states with lable a' (the next event to a in the trace). Moreover, we require that the initial state is always in the list *States_under_reading*. This is because a trace may contain several events that belong to $\Gamma(q_0^{PITC})$, however not all of them will lead to *Pass* attribute.

The algorithm is written formally below:

Variables for trace analysis:

- *States_under_reading:* the list of states in PITC under reading.
- *Pass_reached*: Boolean value. *Pass_reached == True* means that the trace encompasses a branch which is assigned with *Pass* attribute in the PITC graph.
- *Inc_reached*: Boolean value. *Inc_reached == True* means that the trace encompasses a branch which is associated with *Inconclusive* attribute in the PITC graph. Note that in PITC, there could be several branches that are associated with *Inconclusive* attribute.
- *pick* (σ)*:* Take the first element from trace σ. *i.e.* σ=$a.\sigma$', *pick*(σ) = a, $\sigma = \sigma$'.

The trace analysis algorithm contains two loops. In the worst case, the *while* loop will be executed M times where M is the size of the trace. The same, the *for* loop will be executed N times where N is the number of states in *States_under_reading*. Therefore, the complexity of the trace analysis algorithm is $O(M \times N)$.

After the trace analysis, according to different values that can be obtained, an appropriate verdict will be issued. Different from active testing, passive testing

verdict assignment should be more careful as the test system has no control over the SUT. Therefore the usually used active testing verdict rules can not be directly applied. Verdict assignment will be discussed immediately in the next section.

Algorithm 3.2 Trace verification algorithm

Input : Trace σ, $PITC$
Output : $Pass\_reached$, $Inc\_reached$
Initialization: $State\_under\_reading = q_0^{PITC}$, $Pass\_reached = False$, $Inc\_reached = False$
while $\sigma \neq \emptyset$ and *not* $Pass\_reached$ **do**
  pick (σ)
  forall the *state* q *in* $State\_under\_reading$ **do**
    if $a \in Out(q)$ **then**
      if $q == q_0^{PITC}$ **then**
        add (State_under_reading, p) where $(q_0^{PITC}, a, p) \in \triangle^{PITC}$
      end
      else
        $q = p$ where $(q, a, p) \in \triangle^{PITC}$
      end
    end
    if $a \notin Out(q)$ *and* $q \neq q_0^{PITC}$ **then**
      $remove(State\_under\_reading, q)$
    end
    if $q == Pass$ **then**
      $Pass\_reached = True$; *exit* //* exit from the for loop
    end
    if $q == Inconclusive$ **then**
      $Inc\_reached = True$
    end
  end
  Return $Pass\_reached$, $Inc\_reached$
end

3.4.3 Verdict Assignment

After the execution of the trace verification algorithm, according to the values $Pass\_reached$ and $Inc\_reached$ returned by the trace analysis algorithm, an appropriate verdict should be emitted. In this chapter, we have worked out the verdict assignment rules for passive interoperability testing. Three cases are possible:

1. If *Pass_reached=True, a Pass* verdict is emitted as the ITP is satisfied.

2. If *Pass_reached* $\neq$*True* $\wedge$ *Inc_reached=True,* an *Inconclusive* verdict is emitted. In fact, *Inconclusive* means the behavior of IUTs is correct, however does not allow reaching the ITP.

3. If *Pass_reached* $\neq$*True* and *Inc_reached* $\neq$ *True*, i.e., the behavior of IUTs does not match any branch in the PITC, an *Inconclusive* or *Fail* verdict is emitted after post analysis according to different situations. This is because in passive testing, *Fail* verdict is very severe and should be treated carefully.

The verdict assignment rules are different from the rules used in active testing. In fact, in active interoperability testing, verdict is decided according to the consistency between the behavior of IUT and the interaction of their specifications. Transposing this mode of verdict assignment on passive test is trivial in case that the behavior of the IUTs is as expected. On the contrary, in passive testing, the reason for which an ITP is not reached can be due to several reasons:

- The state of SUT is unknown: In active testing, the test system is able to bring the SUT to the desired states, as well as to restart and reconfigure the SUT. Thus the behavior of the IUTs can be judged accurately. The slightest deviation can be considered $Fail$. However, in the context of passive testing, it is impossible for the test system to control the IUTs. Therefore the observed behavior of the IUTs may be allowed by the specifications, however does not correspond to the test case.

  Moreover, the environment of SUT is not under control: Active testing is usually executed in an environment where the topology of the network is well known. Network can be isolated and dedicated exclusively to testing without external perturbation. These conditions may not be satisfied in operational environment where network implementations are in their normal function. Moreover, delay and packet loss may take place.

- Recorded traces may be incomplete: If the traffic capture begins after the start point of the PITC, then only the end of the PITC may be observable in the traces. Similarly, if traffic capture is interrupted before the end of the PITC, then only the beginning of the PITC maybe observable.

Fig.3.3 illustrates an example explaining why Fail verdict cannot be easily assigned in passive testing.

Example 3.1 Let's consider a passive interoperability "Ping" test. In this example, the test purpose is "Ping" functionality, which operates by sending Internet Control Message Protocol (ICMP) [1] *EchoRequest* packets to the target host and waiting for an *EchoReply*. However in reality, destination host or intermediate router will send back an ICMP error message, i.e. "host unreachable" or "TTL exceeded in transit". In these cases, different verdicts can be assigned as illustrated in Fig.3.3.

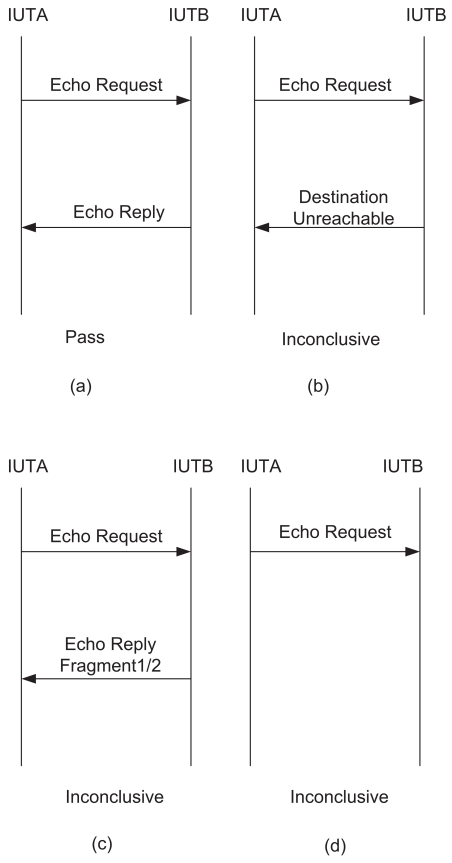


Figure 3.3: Different verdicts in passive interoperability "Ping" test execution

- Fig.3.3-(a) is the expected behavior of two IUTs where *EchoRequest* sent by IUT_A is followed by the *EchoReply* sent by IUT_B.

- In Fig.3.3-(b), IUT_B responds with a *Destination Unreachable*. This may

[1] http://www.ietf.org/rfc/rfc792.txt

be due to the non-configuration of destination address in the *EchoRequest* on the IUT_B.

- In Fig.3.3-(c), the *EchoRequest* sent by the IUT_B is fragmented but only one fragment is received. The second fragment of the packet may have followed another path which was not under observation.

- In Fig.3.3-(d), no response is sent from the IUT_B, (which can be detected by a timer). *EchoRequest* may have followed a different non-observable path. Or, trace capture was stopped before the observation of the *EchoReply*.

Nevertheless, none of the last three cases allows concluding an abnormality in IUT. Therefore, all these three cases can lead to *Fail* in active testing while they can only be assigned *Inconclusive* in passive testing.

To sum up, *Fail* verdict assignment in passive testing must be treat prudentially. The non-satisfaction of an ITP on a trace does not always allow to conclude an abnormality in the IUTs. thus this cannot be easily regarded as non-interoperable behavior. Therefore, if the trace does not match any branch in the PITC, an *Inconclusive* will be given. On the contrary, in case of obvious erroneous message, an unspecified message or time-out, a badly-formatted packet will be considered abnormal. Specific abnormal behavior can also be defined according to different individual protocols. In these situations, a *Fail* verdict may emitted. Indeed, we require that the PITC be sound, i.e., interoperable IUTs cannot be rejected.

3.5 Application on SIP Protocol

3.5.1 SIP Protocol Overview

In order to test the usefulness of our approach we have chosen a real case study: the Session Initiation Protocol (SIP).

The Session Initiation Protocol (SIP), as defined in [45], is an application-layer control (signaling) protocol for creating, modifying and terminating sessions with one or more participants. These sessions include Internet multimedia conferences, Internet telephone calls, multimedia distribution and similar applications.

Under the SIP model, IUTs communicate with each other through asynchronous messaging. Because SIP is a transport-independent signaling protocol,

SIP messages can be transferred via UDP, TCP, or other transport protocols. There are three types of messages: request, response and acknowledgment.

Fig.3.4 illustrates an example of a simplified Session Initiation Protocol (SIP):

S_1 and S_2 represent respectively a SIP Call Management Client (CMC) and a SIP Call Management Server (CMS). They describe the basic behavior of CMC and CMS: Initially, if the CMC user decides to initiate a connection, the CMC will transmit an *Invite* request message ($L1!Invite$). After the reception of the *Invite* request, CMS can either accept ($U2?Accept$) or refuse ($U2?Decline$) the connection. CMC user can hang up at any time ($U1?hangup$). If the user decides to hang up before the connection is established, the CMC will cancel the *Invite* request ($L1!Cancel$). If the user decides to hang up after the connection is established ($L1!Ack$ is sent), the CMC needs to issue a *Bye* request towards the CMS ($L1!Bye$).

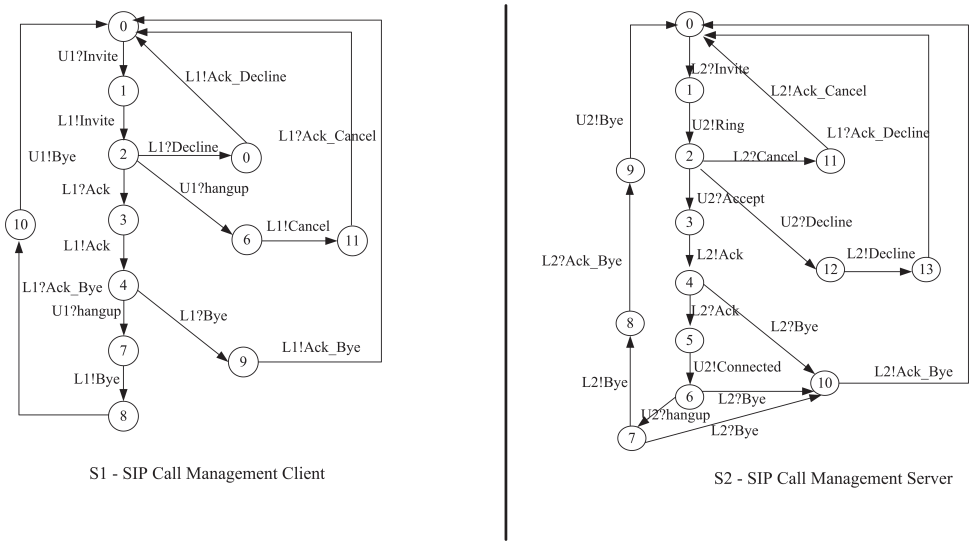
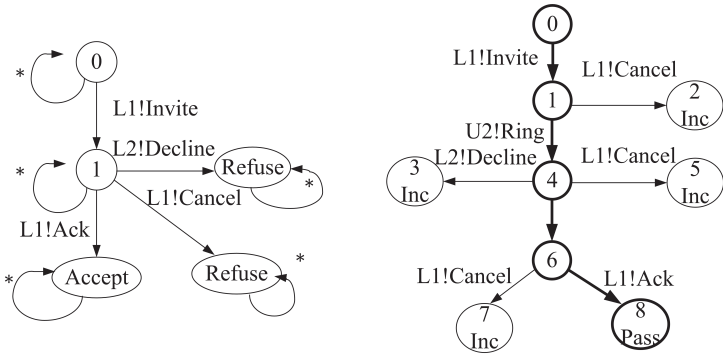


Figure 3.4: Simplified SIP CMC and CMS specifications

Fig. 3.5-(a) illustrates the ITP, which aims at testing the hand-shake property of SIP connection establishment: After that CMC sends out an *Invite* request, after a number of interactions, CMC sends ACK to indicate that the handshake is done and a call is going to be setup. Meanwhile, the possibility that CMC cancels the Invite request ($L1!Cancel$) or CMS refuses the connection ($L2!Decline$) may exist but they do not belong to the test objective. Thus they are associated with *Refuse* attribute. Fig.3.6 illustrates the ITC built by Algorithm 3.1: The interaction of S_1 and S_2 to reach the ITP is calculated. Then, by reducing the

state-space of ITC and determinization, the PITC is obtained and illustrated in Fig. 3.5-(b).



(a) ITP (b) PITC

Figure 3.5: Example of ITP and PITC

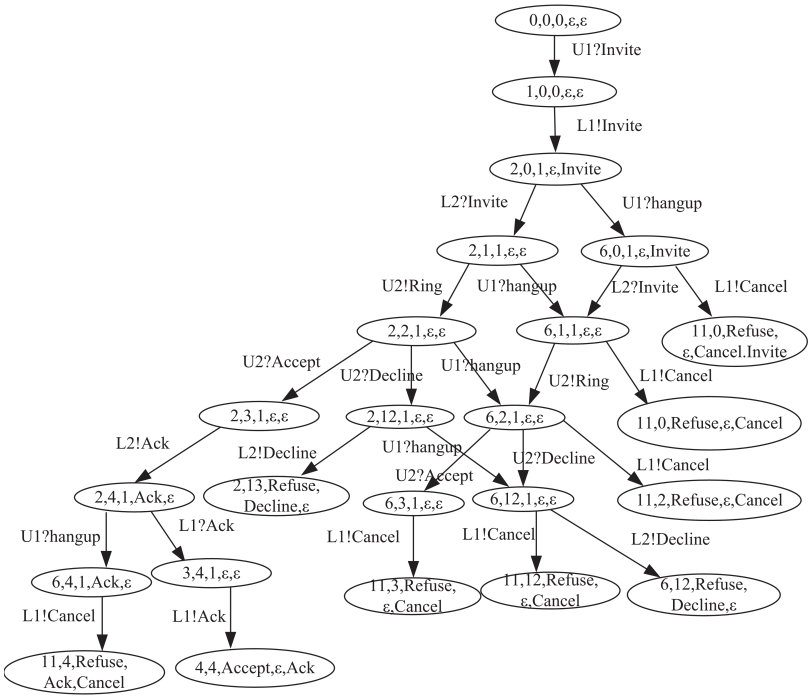


Figure 3.6: Example of an ITC

3.5.2 Test Execution

To carry out passive SIP interoperability testing, two IUTs (SIP phones) are interconnected. A sniffer is connected to the same link to capture all traffic exchanged between the two IUTs. Sniffer must not emit any message which can disturb the IUTs' normal operations. Captured traces are formatted and stored in a file, which will be read and analyzed by the test system.

To make the experiments, 4 SIP phones: Blink [2], Ekiga [3], Jitsi [4] and Linphone [5], have been installed.

The algorithms of ITC construction, passive iop test case derivation and trace analysis were implemented in a testing tool prototype programmed in language Python2.

To observe the messages exchanged by two SIP phones, a *Wireshark*[6] sniffer is used. 100 traces produced by the SIP phones during their interaction were collected, filtered by Wireshark sniffer and stored in pcap format[7]. The traces involve different duration (from 5 seconds to 5 minutes) and different pair wise combinations of SIP phones.

We have chosen different ITPs concerning the basic functionalities of SIP: The establishment of the media session, the call cancellation and the call termination.

(i) ITP_1 - The call setup requested by one IUT should finish with the acknowledgment sent by the peer IUT.

(ii) ITP_2 - The cancellation requested by one IUT should finish with the acknowledgment sent by the peer IUT.

(iii) ITP_3 - Disconnection requested by one IUT should finish with the acknowledgment sent by the peer IUT.

In the case of two IUTs from the same companies, almost all traces have obtained the *Pass* verdicts. We obtained few *Inconclusive* verdict due to the fact that of the length of the traces that were too short. For example, the trace was cut too early so that the verification of call termination was not succeeded. During the communication of two IUTs from different companies , respectively, ITP_1 got 98% Pass, ITP_2 got 86% Pass, while ITP_3 got 90% Pass. The Inconclusive

[2]http://icanblink.com/
[3]https://www.ekiga.net/
[4]https://jitsi.org/
[5]http://www.linphone.org/
[6]http://www.wireshark.org/
[7]http://www.tcpdump.org/

verdicts are due to the following reasons:

- Short trace length: due to the fact that trace registration is cut before the end of PITC Inconclusive attributes reached.

- Inconclusive verdicts are reached: due to the fact that the observed behavior does not correspond to the passive test case, however allowed in the specifications.

Moreover, a Fail verdict was obtained:

- Unexpected event: During the communication between Jtisi and Ekiga Siphones. After Jtisi sends Bye request, instead of an Ack_Bye, Ekiga Siphone replies 481 Call/Leg Transition. After post analysis, this behavior is determined to be abnormal.

3.6 Conclusions

This chapter proposes an approach for passive interoperability testing. Given the formal specifications of the IUTs and an iop test purpose ITP, a PITC can be derived by partial asynchronous interaction calculation. In addition, a trace analysis algorithm was proposed to check the recorded trace. The proposed algorithms have been implemented and applied to a real protocol SIP. Moreover, the verdict assignment issue, which is often problematic in passive testing, is discussed.

The proposed passive testing method has the following features:

1. Testing architecture is simple, no upper tester is needed. Active test cases need stimuli to set the SUT to a specific configuration. Stimuli must be well-designed, however they can be also error prone and introduce extra overhead in networks. Passive test case is only based on observation, which is able to avoid these issues, perform interoperability testing without interfering with the normal operations of the SUT.

2. Partial asynchronous interaction calculation alleviates states explosion. By setting an iop test purpose ITP and using attributes, only the necessary parts of joint behavior of IUTs need to be computed.

3. PITC describes in detail the expected interaction of IUTs that allow to reach the given ITP, which can be used to evaluate the interoperability degree of the SUT.

However, this method also have some drawbacks. In fact, this method requires that the specification of a protocol must be formalized, which is difficult in practice, especially for complicated protocols or protocols designed in an evolutive way.

Therefore, in the following, we will try to find another solution to carry out passive interoperability testing. Future work will also concentrate on reducing passive testing execution time: Executing test cases in parallel, i.e. each test case is run independently on the same trace will be considered. Besides, aggregating test cases with a similar preamble will be useful to reducing the time of test derivation procedure to some extent. This part will be discussed in detail in Chapter 5.

Chapter 4

A Passive Interoperability Testing Method for Request-Response Protocols

4.1 Introduction

With the development and increasing use of distributed systems, computer communication mode has changed. There is increasing use of clusters of workstations connected by a high-speed local area network to one or more network servers. In this environment, resource access leads to the communications that are strongly transaction oriented. This tendency resulted in a large amount of new protocols designed for request-response communications. Typical examples are Hypertext Transfer Protocol (HTTP)[27], Session Initiation Protocol (SIP), and very recently the Constrained Application Protocol (CoAP) [20], etc. Due to the heterogeneous nature of distributed systems, the interoperability of these protocol applications is becoming a crucial issue. In this context, interoperability testing is required before the commercialization of the protocol applications to ensure their correct collaboration and guarantee the expected quality of services.

This chapter proposes a methodology for the interoperability testing of request-response protocols by using the technique of passive testing. The methodology consists of the following main steps:

- Interoperability test purposes extraction from the protocol specifications.
- For each test purpose, an interoperability test case is generated, specifying the detailed set of events that need to be observed.

- Behavior analysis. In order to verify whether the test purposes are satisfied, traces produced by protocol implementations are filtered and divided into a set of *conversations* with respect to the special *request/response* interaction model of request-response protocols. These conversations will help further identifying the occurrence of test cases as well as emitting an appropriate verdict.

Moreover, in this chapter, we will present a testing tool developed for automating passive interoperability testing tool. The proposed passive interoperability testing method has been implemented and automated in this tool, which was successfully put into operation during the CoAP Plugtests - the first formal CoAP interoperability testing event held in Paris, Mars 2012 in the context of the Internet of Things, and the other held in Sophia Antipolis in November, 2012.

This chapter is organized as follows: Section 4.2 introduces the background and motivation. Section 4.3 proposes the methodology for passive interoperability testing of request-response protocols. Section 4.4 presents a testing tool to automate passive interoperability testing. Section 4.5 describes the application of this method on CoAP Plugtest as well as the obtained experimental results. Finally, we conclude the chapter and suggest further research directions in Section 4.6.

4.2 Background and Motivation

The transaction oriented communication, also called request-response communication, is used in conjecture with the client-server paradigm to move the data and to distribute the computations in the system by requesting services from remote servers. The typical sequence of events in requesting a service from a remote server is: a client entity, a process, task or thread of control, sends a request to a server entity on a remote host, then a computation is performed by the server entity, and, finally a response is sent back to the client.

Request-response communications are now common in the fields of networks. Request/response exchange is typical for database or directory queries and operations, as well as for many signaling protocols, remote procedure calls or middleware infrastructures. A typical example is REST (Representational State Transfer) [30], an architecture for creating Web service. In REST, clients initiate request to servers to manipulate resources identified by standardized Uniform Resource Identifier (URI). The HTTP methods GET, POST, PUT and DELETE are used to read, create, update and delete the resources. On the other hand, servers process requests and return appropriate responses. REST is nowadays

popular, which is applied in almost all of the major Web services on the Internet, and considered to be used in the *Internet of Things* [22], aiming at extending the Web to even the most constrained nodes and networks. This goes along the lines of recent developments, such Constrained RESTful Environments (CoRE)[1] and CoAP, where smart things are increasingly becoming part of the Internet and the Web, confirming the importance of request-response communication.

Promoted by the rapid development of computer technology, protocols using the client-server request-response are increasing. Normally, protocol specifications are defined in a way that the clients and servers interoperate correctly to provide Web services. On the one hand, customer needs are growing and manufacturers permanently develop new equipments with improved quality of service. On the other hand, with the rapid widespread commercial adoption of complex and diverse technologies, interoperability is essential to provide cooperative services. To ensure that they collaborate properly and consequently satisfy customer expectations, interoperability testing is an important step to validate request-response protocol implementations before their commercialization.

In this context, In this chapter, we will provide a passive interoperability testing methodology for request-response protocols.

4.3 Passive Interoperability Testing Method for Request-response Protocols

4.3.1 Testing Method Overview

As studied previously, currently used passive testing methods are trace mapping and invariants approaches. But both of them have limitations: Trace mapping suffers from state space explosion. Invariants on the other side, focus only on expected properties, which limit the capacity of non-interoperability detection. In Chapter 3, we proposed a solution based on partial formal specification calculation on the fly with a test purpose. This method however, requires that the specifications must be formalized. This rends it difficult to apply in practice, especially when a protocol is complex or still in its early stage.

To solve the above problems, in this chapter, we propose another solution, which allows performing passive interoperability testing while avoiding formalizing the whole specifications.

The testing procedure consists of the following main steps: interoperability

[1] http://datatracker.ietf.org/wg/core/charter/

test purposes selecion, interoperability test case derivation and trace verification. They are illustrated in Fig.4.1.

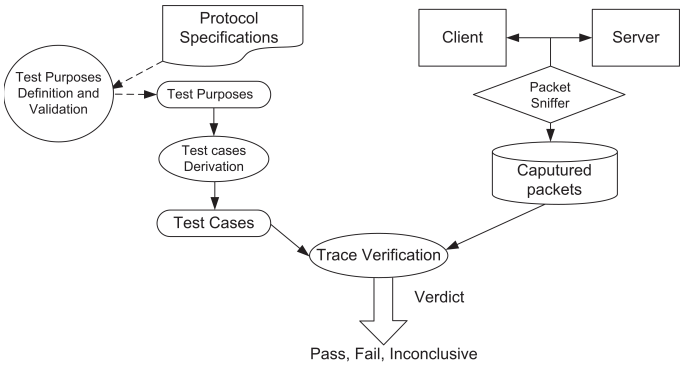


Figure 4.1: Passive interoperability testing procedure

1. Interoperability test purposes (ITP) selection from protocol specifications. An ITP is in general informal, in the form of an incomplete sequence of actions representing a critical property to be verified. Generally it can be designed by experts or provided by standards guidelines for test selection [24]. Test purpose is a commonly used method in the field of testing to focus on the most important properties of a protocol, as it is generally impossible to validate all possible behavior described in specifications. Nonetheless, it should be emphasized that an ITP itself must be correct w.r.t the specification to assure its validity.

   Formally, an ITP can be represented by a deterministic and complete IOLTS equipped with trap states used to select targeted behavior (c.f. Definition 3.3 in Chapter 3). Note that in this chapter we only focus on the desired property to be verified. i.e. only *Accept* attribute is used. However other events that can lead to *Inconclusive* or *Fail* verdicts will be specified in the corresponding test case derived from the ITP.

2. Once the ITPs chosen, an iop test case (ITC) is generated for each ITP. An ITC is the detailed set of instructions that need to be taken in order to perform the test. In active testing, ITCs are usually controllable. i.e, ITC contains stimuli that allow controlling the IUTs. On the contrary, in passive testing, ITCs are only used to analyze the observed trace produced by the IUTs. The correct behavior of IUTs implies that the trace produced by the IUTs should exhibit the events described in the test cases. The generation of iop test case can be either manually, as usually done in most of the interoperability events. These interoperability testing events bring

together products from different vendors purely for the purpose of executing tests for a period which may last from a couple of days to a couple of weeks.

Also, manual ITC generation is usually applied for new protocols on draft whose specifications are not yet stable. ITCs can also be generated automatically by using various formal description techniques existing in the literature such as [19, 18]. However, due to the multi-implementations nature of interoperability testing, the problem of state-space explosion is inherent and can be only relived other than avoided. Moreover, formalizing the whole specification of a protocol is not an easy task, when it involves a huge number of events, options, etc. Especially nowadays, a protocol is often designed in an extensive way. Thus the specification can change with the evolution of new functions. Therefore in this chapter, we choose to use manual ITC generation. It is done by carefully reading the specifications of a protocol, and derive the detailed behavior to be verified, which is related to the corresponding ITP.

Formally, an iop test case ITC is represented by: ITC = (Q^{ITC}, Σ^{ITC}, Δ^{ITC}, q_0^{ITC}) where q_0^{ITC} is the initial state. $\{Pass, Fail, Inconclusive\} \in Q^{ITC}$ are the trap states representing interoperability verdicts. Σ^{ITC} denotes the observation of the messages from the interfaces. Δ^{ITC} is the transition relation. In this work, we underline the importance of specifying all the behavior that can lead to *Pass, Fail, Inconclusive* verdicts to help draw a conclusion of interoperability.

An example of ITP and ITC can be found in Section 4.5.2.2.

3. Analyze the observed behavior of the IUTs against the test cases ITC and issuing a verdict *Pass, Fail* or *Inconclusive*.

   In passive interoperability testing for request-response protocols, the test system has two main roles:

   (a) Observe and collect the information exchanged (trace) between the protocol applications.

   (b) Analyze the collected trace to check interoperability. Generally, trace verification can be done *online* or *offline*. In this chapter we choose to use offline testing method, where the test cases are pre-computed before they are executed on the trace.

   As passive testing does not apply any stimulus, testing activity is only based on an accurate level of observation, relying on the set up of

sniffer at *point(s) of observation* (PO) to observe the messages passing through the system under test (composed of two IUTs, namely a *client* and a *server*).

4.3.2 Trace Verification

As said above, in this thesis we choose the offline testing method. In passive interoperability testing for request-response protocols, the packets exchanged between the client and server are captured by a packet sniffer. The collected traces are stored in a file. They are key to conclude whether the protocol implementations interoperate.

In passive testing, one issue is that the test system has no knowledge of the global state where the system under test SUT can be in w.r.t a test case at the beginning of the trace. In this chapter, we propose a solution. The idea is to make use of the special interaction model of *request/response* protocols. As the interoperability testing of this kind of protocol essentially involves verifying the correct transactions between the client and the server, therefore each test case consists of request and responses, and generally starts with a request from the client. A strategy is as follows:

- The recorded trace is filtered to keep only the messages that belong to the tested request/response protocol.

- Each event in the filtered trace will be checked one after another according to the following rules, which correspond to the algorithm of trace verification (c.f. Algorithm 4.1).

1. If the currently checked message is a request sent by the client, we verify whether it corresponds to the first message of (at least one of) the test cases (noted TC_i) in the test suite TS. If it is the case, we keep track of these test cases TC_i, as the matching of messages implies that TC_i might be exhibited on the trace. We call these TC_i *candidate test cases*. The set of candidate test cases is noted TC. Specifically, the currently checked state in each candidate test case is kept in memory (noted $Current_i$).

2. If the currently checked message is a response sent by the server, we check if this response corresponds to an event of each candidate test cases TC_i at its currently checked state (memorized by $Current_i$). If it is the case, we

further check if this response leads to a verdict *Pass, Fail* or *Inconclusive.* If it is the case, the corresponding verdict is emitted to the related test case. Otherwise we move to the next state of the currently checked state of TC_i, which can be reached by the transition label - the currently checked message. On the contrary, if the response does not correspond to the event at the currently checked state in a candidate test case TC_i, we remove this TC_i from the set of the candidate test cases TC.

3. Besides, we need a counter for each test case. This is because in passive testing, a test case can be met several times during the interactions between the client and the server due to the non-controllable nature of passive testing. The counter $Counter_i$ for each test case TC_i is initially set to zero. Each time a verdict is emitted for TC_i, the counter increments by 1. Also, a verdict emitted for a candidate test case TC_i each time when it is met is recorded, noted $verdict.TC_i.Counter_i$. For example, $verdict.TC_1.1{=}Pass$ represents a sub-verdict attributed to test case TC_1 when it is encountered the first time in the trace. All the obtained sub-verdicts are recorded in a set $verdict.TC_i$. It helps further assigning a global verdict for this test case.

4. The global verdict for each test case is emitted by taking into account all its sub-verdicts recorded in $verdict.TC_i$.

The complexity of the algorithm is O(M×N), where M is the size of the trace, N the number of candidate test cases. The trace verification procedure in fact, involves looking for the possible test cases that might be exhibited in the trace by checking each event taken in order from the trace. By taking advantage of the transaction mode of request-response protocols, each filtered traces are in fact composed of a set of *conversations.* The objective of the algorithm is intended to map the test cases into the conversations, so that the occurrence of the test cases in the trace is identified. Moreover, by comparing each message of the test case with that of its corresponding conversation, we can determine whether the behavior of IUTs interactions are as expected as they are described by the test cases.

Moreover, the possibility that a test case can appear several times in the trace is also taken into account. Therefore the global verdict for a given test case is based on the set of subverdicts, increasing the reliability of interoperability testing. Not only we can verify whether the test purposes are reached, but also non-interoperable behavior can be detected due to the difference between obtained subverdicts.

Algorithm 4.1 Trace Verification Algorithm

Input : filtered trace σ, test suite TS
Output : $verdict.TC_i$
Initialization: $TC = \emptyset$, $Counter_i = 0$, $Current_i = q_0^{TC_i}$, $verdict.TC_i = \emptyset$
while $\sigma \neq \emptyset$ **do**
  σ=$\alpha.\sigma'$
  if α *is a request* **then**
    for $TC_i \in TS$ **do**
      if $\alpha \in \Gamma(Current_i)$ **then**
        $TC = TC \cup TC_i$
        /*Candidate test cases are added into the candidate test case set*/
        $Current_i$=$Next_i$ where $(Current_i, \alpha, Next_i) \in \Delta^{TC_i}$
      end
    end
  end
  else
    for $TC_i \in TC$ **do**
      if $\alpha \in \Gamma(Current_i)$ **then**
        $Current_i$=$Next_i$ where $(Current_i, \alpha, Next_i) \in \Delta^{TC_i}$
        if $Next_i \in \{Pass, Fail, Inconclusive\}$ **then**
          $Counter_i$=$Counter_i$+1
          $verdict.TC_i.Counter_i$=$Next_i$ /* Emit the corresponding verdict to the test case*/
          $verdict.TC_i$= $verdict.TC_i \cup verdict.TC_i.Counter_i$
        end
      end
      else
        TC=$TC \setminus TC_i$
      end
    end
  end
end
return $verdict.TC_i$

The rules to draw a global verdict is the following:

1. *Fail* if at least one of the subverdict is *Fail*.
2. *Inconclusive* if all the subverdicts are *Inconclusive*.
3. *Pass* if at least one of the subverdict is *Pass*, while no subverdict is *Fail*.

This is coherent with the verdict assignment rules defined in Chapter 3, except that in this work, we choose to manually define all the behavior that can lead to *Pass*, *Fail* or *Inconclusive* verdicts, therefore it is more easily to conclude a *Fail* verdict to indicate none-interoperability.

Also we underline that in active testing, a *Pass* verdict need that all sub verdicts must be *Pass*. However, due to the uncontrollable nature of passive testing and non-deterministic behavior of protocol applications, the condition is not always favorable (even often difficult) for the test system to observe all the expected behavior. Therefore an *Inconclusive* verdict is not sufficient to draw a conclusion of non-interoperability. On the contrary, a *Pass* verdict means the ITP has been satisfied. In fact, transposing the verdict combination rule of active testing is not suitable, as it will result in a huge amount of *Inconclusive* verdicts, therefore decrease the meaning of passive testing.

4.4 A Passive Interoperability Testing Tool

4.4.1 Motivation

Interoperability testing is today one of the key activities in the development of network systems. However, majority of interoperability testing and validation is performed manually, which is labor intensive and error prone. Fig.4.2 shows the time proportion of each activity spent on testing, where test execution and verification occupy more than 50% of the time[58]. Among these four aspects, test design and test specification is the innovative phase that cannot be automated. The test runs and verifying the results and looking at problems of unsuccessful test are the phases occupying by far the most time and should be automated.

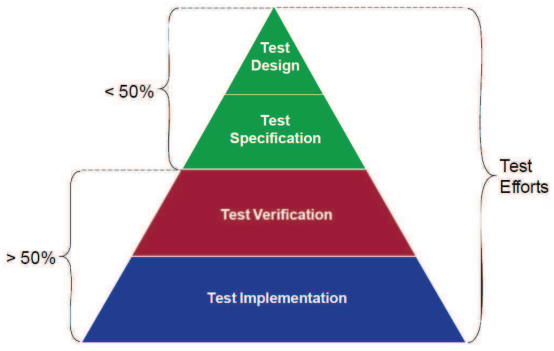


Figure 4.2: The current test effort problem

In order to automate testing, In the community of testing, TTCN (Testing and Test Control Notation)[2] is widely used. For example: ETSI, ITU for the testing of telecommunication protocols. Conformance test cases of ETSI standards like ISDN (integrated services data digital network), DECT (digital enhanced cordless telecommunications), GSM (global system for mobile communications), EDGE (enhanced data rate for GSM evolution), DSRC (dedicated short-range communications) have also been written in TTCN. Recently it has also been used for testing various protocol standards e.g. Bluetooth. TTCN was created by leading experts from ETSI. The standards address not only the language for specifying tests but also the interfaces that control and adapt a test to any given environment. Two complementary parts of the standardized TTCN testing technology: a test specification language and a framework for building test systems, jointly allow for the automatic execution of tests. The currently used version is TTCN-3. Initially designed for conformance testing, TTCN-3 has nowadays been expanded from purely functional conformance testing into load, performance and interoperability testing.

However, TTCN applies the paradigm of active testing: A test case runs from start to end as a single unit of test activity and expected to be executed as quickly as possible. Active tests are performed in order to be able to obtain meaningful information during a finite test campaign. In this thesis, after careful study, we claim that, in addition to the traditional concept of active testing, also passive testing and monitoring can benefit from the mature TTCN technology. We thus propose a tool, called ttproto (testing tool prototype), which internally reuses some features of TTCN technology, but which can also be used for empirical investigation of system behavior, where passive observation is desirable or needed.

4.4.1.1 TTCN-3 Overview

TTCN-3 (Testing and Test Control Notation Language Version 3) is a test specification language used to define test procedures for reactive black-box testing of distributed systems. The standardized specification of the language and its environment is contained in the ETSI standard [57].

A TTCN-3 program is referred to as an Abstract Test Suite (ATS). It expresses the configuration and behavior of an abstract test system, which is composed of a set of concurrently executing test components (TC). TTCN-3 allows the specification of dynamic and concurrent test systems. In fact, it offers a test configuration system made of two kinds of test components: Main Test Component (MTC) and Parallel Test Component (PTC). For each test case, an MTC is created. PTCs can be created dynamically at any time during the execution

[2]http://www.ttcn-3.org/

of test case. Thus, test system can use any number of test components to realize test procedures in parallel. Communications between the test system and the SUT are established through ports. These ports, referred to as PCOs (Points of Control and Observation) are the means of communication with a SUT/IUT.

The main notational unit of TTCN-3 is a module, which is composed of a definition part and a control part. The definition part contains the definitions of types, constants, templates, ports, components, and component behaviors: functions and test cases. Syntactically, a test case defines the behavior of a MTC. Dynamically, a MTC may create and start multiple PTCs, with their behavior defined as functions. The control part governs the execution of individual test cases, by means of language constructs that, in general, use the verdicts of previous test cases as conditions for the execution of further tests. Despite the shallow similarity to programming languages, TTCN-3 is specialized in its ability to express a particular kind of programs: tests. To this end it includes:

- the send and receive operations for asynchronous message passing on a given port.

- template: a particular kind of data structure that combines the features of a variable. A template is used to explicitly define a concrete data value for send operations, and to implicitly define a set of values (by allowing wildcards), any of which will be accepted by a receive operation.

- timer: an expiration of a timer is recognized by means of an operation similar to a receive. The use of timers includes for example, assuring a limited execution time, and inferring a "no response" event.

- alternative behavior (alt): a list of alternatives, composed of a (possibly guarded) operation and a following instruction block. Within an alt block, each alternative is "tried" in the order of its syntactic appearance. If none of them can be executed successfully (e.g., for a receive operation, if there is no matching message awaiting reception), then the alt block is entered again, with a new "snapshot" of the state of environment. According to the pragmatics of active testing, within an alt block there is usually a list of branches in which the awaited response is received and dealt with (i.e., the receive operations for alternative SUT responses and the timeout operations indicating the lack of response).

- verdicts: values {pass, fail, inconc, none, error} of a special verdict type; by assigning a verdict, the outcome of a test case is made known to the control

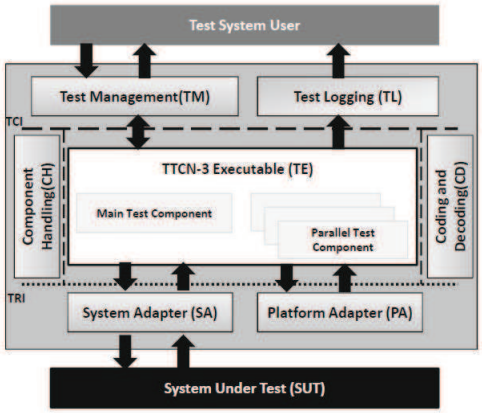


Figure 4.3: TTCN-3

part. Respectively, none is implicitly assigned in the beginning of every test case by default and is reported as a final verdict in the absence of any other verdict assignment during the test case execution. Pass means that everything is OK. Inconc means that neither pass nor fail can be reliably assigned, for example due to a network connection failure; fail means that something definitely went wrong. Error means that there was an error in the test harness, this verdict cannot be assigned by the end user directly.

One of the most important aspects of TTCN-3 standardization is that it also covers the abstract architecture of a test system and its implementation-oriented mappings. For each implementation, the ports must be properly connected to a SUT. The implementation of such connection is hidden from a TTCN program and delegated to a SUT Adapter (SA) module. Similarly, the functionality and the interface of a Coder-Decoder (CD) module has been standardized. The joint functionality of a SA and a CD is to deliver and handle the events expressed as values of a TTCN-3 type system. This is the general scope of an event language. The TTCN-3 technology allows a way in distributing this functionality among a SA, a CD, and a test program expressed in TTCN. In particular, it is possible to deliver a raw bitstring to a test program, if the templates for receive operations are thus defined. However, in practice the message definitions in TTCN-3 are structural, and a decoder will have to deliver data in chunks that reflect this structure.

4.4.1.2 Main Issues

Although TTCN-3 is widely used nowadays in testing communities, it has some limitations in the context of passive testing including the follwowing:

1. Initially, TTCN was developed strictly within the framework of conformance testing of black-box implementations of communication protocols in a layered (e.g., OSI) system. This framework only covers the logical behavior and explicitly excludes other, mostly non-functional properties, such as performance, robustness, and scalability. It was natural for a TTCN community to try and extend the applicability of the language, e.g., by proposing its experimental modifications for performance testing (PerfTTCN[51], TimedTTCN-3[52]), real-time properties (Real-Time TTCN), and interoperability testing. However, all these modifications respected the main paradigm of active testing. As a result, it does not allow easily perform passive testing as concerned by our work.

   Therefore, in order to realize passive testing, some TTCN features especially controllability must be modified.

2. Moreover, TTCN-3 also has other drawbacks, especially concerning template proliferation, that make it "inconvenient" in practical use: For example, it needs two different sets of templates for sending and receiving messages; parametrization and re-use are not easy so that it results in tedious code duplication; etc. Therefore, we tend to find a solution to facilitate templates definition and reusing, so that to improve readability and extensibility.

However, we find that, passive testing can benefit from the TTCN technology by utilizing some of its concepts. Therefore, we have developed a tool called *ttproto*, which is partly inspired by TTCN-3, however contains new features designed with modularity and flexibility in mind, that make it more suitable for passive testing.

4.4.2 Ttproto: A Testing Tool for Passive Interoperability Testing

4.4.2.1 Ttproto Overview

In order to overcome the drawbacks of TTCN-3 and find a solution to automate passive interoperability testing, in this work we introduce an experimental tool

ttproto, which allows implementing new features and explore new concepts for the TTCN-3 standard.

Ttproto is partly inspired from the concepts of TTCN-3: abstract model, templates, snapshots, behavior trees, communication ports, logging. Therefore, these concepts are implemented in the tool. However, ttproto is not a subset or a replacement of the TTCN language. Ttproto is written in a high level language – python3[3] . Python is chosen for the following reasons: simple, easy to learn, widespread and which allows rapid prototyping new features. By using python, implementing new features does not require any skills in language grammars and parsers and to understand the sources of a real TTCN-3 compiler. The design strategy is to maximize modularity and readability rather than performances or realtime constraints. In this way, test developers can experiment and demonstrate features they would like to integrate in TTCN-3, without having to modify the language or dig into the sources of a TTCN-3 compiler.

Below is a short list of features of ttproto:

- Contrary to TTCN-3 in which a "test script" is available in a finite tangible form, organizing the execution of tests in a finite test suite. A test script runs from start to end as a single unit of test activity and expected to be executed as quickly as reasonably possible, and perform active tests, in order to be able to obtain meaningful information during a finite test campaign. These preconceptions and patterns strongly influence the perception of the needs of testing, which however can be challenged in the context of passive testing.

  In order to solve this problem, ttproto uses another philosophy, which is proposed to decompose the methodology of empirical investigations of system behavior into separate conceptual and technological elements *modules*. In this way, each module is responsible for separate tasks.

  In case of passive testing, a module may express a passive testing algorithm, together with a data structure that encodes expected properties. Passive test cases are then implicitly contained in such module. In this way, ttproto is actually a device for the empirical investigation of the behavior of systems that is not bound by preconceptions inherent in existing TTCN systems, and thus can be used for passive testing.

  Moreover, ttproto also provides some advantages over TTCN-3, and consequently better help realizing passive testing.

[3]http://www.python.org/getit/releases/3.0/

```
template IPv6 IPv6HopLimit_is_255 :=
{
        Version                 := ?,
        TrafficClass            := ?,
        FlowLabel               := ?,
        PayloadLength           := ?,
        NextHeader              := ?,
        HopLimit                := 255,
        SourceAddress           := ?,
        DestinationAddress      := ?,
        Payload                 := ?,
}
```

(a)

```
template IPv6 send_IPv6HopLimit_is_255 :=
{
        Version                 := 6,
        TrafficClass            := 0,
        FlowLabel               := 0,
        PayloadLength           := 0,
        NextHeader              := 0,
        HopLimit                := 255,
        SourceAddress           := '00000000000000000000000000000000'O,
        DestinationAddress      := '00000000000000000000000000000000'O,
        Payload                 := ''O,
}
```

(b)

Figure 4.4: TTCN send and receive templates

- Default values and template matching

  In TTCN-3, templates are used to explicitly define a concrete data value for send operations, and to implicitly define (or generate) a set of values, any of which will be matched by a receive operation. Templates allow wildcards to be used instead of an actual value. However, in TTCN-3, two separate templates have to be used for the reason that wildcard can not be used in templates for generating messages to be sent. As a result, the whole number of templates is quite big. Code lines are so tedious that rend them less readable. For example, to match an IPv6 message whose hop limit is 255, we could write in TTCN-3 in Fig.4.4-(a). The wildcard "?" indicates that we can match any value. This is useful because in many cases one only cares about a part of a message. But, Wildcard can not be used in templates for generating messages to be sent. In that case we must define a second template with all fields set to a value (cf. Fig.4.4-(b)).

  In order to solve the issues, default values *undef* are introduced in ttproto when defining a structured message type. Respectively, undef means "any value or none" when receiving, while "use the default value" when sending. Therefore the two previous templates written in TTCN-3 can be instantiated in ttproto as a single template: IPv6 (hl = 255).

```
>>> Message (IPv6 (hl=255)).display()
###[ IPv6 ]###
  Version=                6
  TrafficClass=           0x00
  FlowLabel=              0x00000
  PayloadLength=          0
  NextHeader=             0
  HopLimit=               255
  SourceAddress=          ::
  DestinationAddress=     ::
  Payload=
###[ BytesValue ]###
     Value=               b''
```

Figure 4.5: Ttproto default values

The use of default values is useful in passive testing as it avoids creating a large number of templates, thus decreases drastically the size of the code and increases the readability of the test case.

- Message presentation formats

  In ttproto, both a raw byte string or binary string representing the address can be used. In sequence, an IP address can be presented in more human-readable byte format instead of the tedious binary string. Moreover, byte and binary type and can be converted at any time. This feature is important for passive testing, as a large number of traces should be dealt, a more readable format will help analysis.

- Template inheritance

  In ttproto, messages types are represented using object-oriented model (python3). Therefore, it allows template inheritance through the parametrization notion in stead of modifying keywords. Actually, in TTCN-3, modifying keywords means changing the keywords. While in ttproto, more constraints can be added by template inheritance. This is important firstly, by using object-oriented model, user-defined type (like TTCN-3's subtypes and structured types) can be inherited from a base type provided by ttproto. Moreover, template inheritance is important especially for post analysis. In passive interoperability testing for instance, if a template fails to be matched, by checking the keywords we can quickly find the source of non-interoperability.

- Encoding/Decoding framework

  Encoding and decoding messages (i.e. converting between their abstract representation in the test case and the actual binary message sent to the

system under test) is not addressed in the TTCN-3 language. The standard defines instead an interface to implement this feature in an external module (there are bindings in Java, C, C++ and C#). On the one hand the developer is free to implement his codecs and in the other hand the standards provides no framework at all.

In ttproto we decided to integrate encoding and decoding mechanisms within the tool, but with some latitude to allow implementing families of codecs. In fact encoding rules are often very similar within a family of protocols. For example in the Internet area, message-based protocols are mostly encoded from left-to-right, using big endian integers, 32-bit padding and ubiquitous patterns like TLV (Type-Length-Value). With encoding/decoding framework, users are able to reuse the existing codecs and quickly develop new codecs if needed.

4.4.2.2 Description

A description of ttproto is given in Fig.4.6.

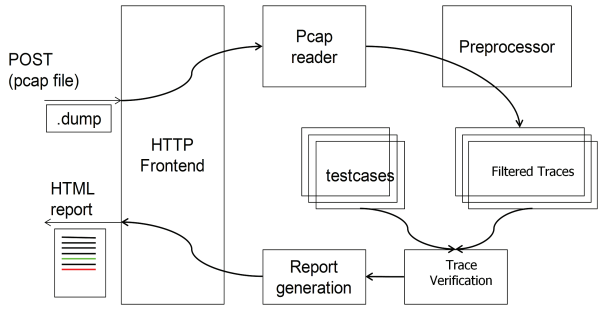


Figure 4.6: Passive interoperability testing tool

As illustrated in Fig.4.6, a web interface (HTTP frontend) was developed. Traces produced by a client and a server implementation of a request-response protocol, captured by the packet sniffer are submitted via the interface. Specifically in our work, the traces should be submitted in pcap format [4] by using tool *Wireshark*[5]. Each time a trace is submitted, it is then dealt by a preprossesor to filter only the messages relevant to the tested request-response protocol. In this way, the trace contains only the conversations made between the client and server.

[4]http://www.tcpdump.org/
[5]http://www.wireshark.org/

The next step is trace verification, which is carried out by taking into two files as input: the set of test cases and the filtered trace. The trace is analyzed according to Algorithm 4.1, where test cases are verified on the trace to check their occurrence and validity. Finally, unrelated test cases are filtered out, while other test cases are associated with a verdict *Pass*, *Fail* or *Inconclusive*.

The results are then reported from the HTTP frontend: Not only the verdict is reported, also the reasons in case of *Fail* or *Inconclusive* verdicts are explicitly given, so that users can understand the blocking issues of interoperability (c.f. a use case in Section 4.5.3.1).

The testing tool was put into operation in two interoperability testing events for CoAP protocol. The application of this tool can be found in Section 4.5.

4.5 Application to the CoAP Protocol

4.5.1 CoAP Interoperability Testing Event

In the field of interoperability testing, interoperability testing events, often called by ETSI plugtests are regularly organized especially in industrial context to guarantee the interoperability of products based on a protocol. Famous plugtests events are for instance:

1. ETSI's Plugtests[6]. ETSI hosts about 15 interoperability events per year. Each focuses on different information and communication technologies.

2. IHE Connect-a-thons[7]: This week-long interoperability event is organized each year by the International Healthcare Enterprise (IHE), an initiative by healthcare professionals and industry to improve the way of sharing information by computer systems in healthcare. This event provides an opportunity to vendors to test their products. These events enable developers from different (and competing) companies to get together to test their companies' own implementations and ensure interoperability between products. In addition, these events are immensely valuable in validating a draft standard, and identifying and removing ambiguities and misinterpretations that may exist.

Recently, the Internet of Things (IoT) has become a hot topic. IoT an integrated part of future Internet and could be defined as a dynamic global network

[6]http://www.etsi.org/Website/OurServices/Plugtests/home.aspx
[7]http://www.ihe.net/participation/

infrastructure with self configuring capabilities based on standard and interoperable communication protocols where physical and virtual things use intelligent interfaces, and are seamlessly integrated into the information networks. One of the objectives of the IoT is using the captured information by smart objects (e.g. automation systems, mobile personal gadgets, building-automation devices, cellular terminals, the smart grid, etc.) to improve peoples life in a large range of fields: healthcare, environment monitoring, smart energy control, industrial automation and manufacturing, logistics, etc. Promoted by IoT, more and more devices are becoming connected and benefit from interacting with each other to achieve cooperative services. Over the next decade, this could grow to trillions of embedded devices and will greatly increase the Internet's size and scope. However, the evolution of technologies also brings challenges: devices behind Machine-to-Machine (M2M) applications are generally resource limited. Typically, they are battery-powered and frequently asleep, limiting them to an average consumption on the order of micro-watts. Power limitations also lead to constraints on available networking. Most devices connect wirelessly as stringing wires are prohibitively expensive. In consequence, many packet losses might occur during data transfer.

To deal with these challenging issues, the IETF Constrained RESTful Environments (CoRE) working group[8] has worked out the Constrained Application Protocol (CoAP) [20], an application-layer protocol to provide resource constrained devices with low overhead and low power consumption Web service functionalities. Different from traditional Web services protocol, CoAP protocol involves new performance engineering methods, tools, and benchmarking needs. Especially, the ubiquitous nature of CoAP requires interoperability to ensure that smart objects using CoAP work well together in low-power and lossy environment without human intervention, while guaranteeing the services described in the specifications. As one of the most important protocol for the future Internet of Things, the number of smart objects using CoAP is expected to grow quickly. In this context, the Probe-IT[9] (Pursuing roadmaps and benchmarks for the Internet of Things, an European project in the context of Internet of things), the IPSO Alliance (Internet Protocol for Smart Object communications)[10] and ETSI[11] (the European Telecommunication Standard Institute) co-organized the CoAP interoperability event – CoAP Plugtest. The objective of this event is to enable CoAP implementation vendors to test end to end interoperability with each other.

[8] http://datatracker.ietf.org/wg/core/charter/
[9] http://www.probe-it.eu/
[10] http://www.ipso-alliance.org/
[11] www.etsi.org/

4.5.2 CoAP Protocol

4.5.2.1 CoAP Protocol Overview

Most Internet applications today depend on the Web architecture, using HTTP [27] to access information and perform updates. HTTP is based on Representational State Transfer (REST) [30], an architectural style that makes information available as resources are identified by URIs (Uniform Resource Identifier): applications communicate by exchanging representations of these resources by using a limited set of methods. This paradigm is quickly becoming popular, even spreading to Internet of Things applications, aiming at extending the Web to constrained nodes and networks. In this context, the IETF Constrained Application Protocol (CoAP) has been designed, which is an application-layer protocol on keeping in mind the various issues of constrained environment to realize interoperations with constrained networks and nodes. CoAP adopts some HTTP patterns such as resource abstraction, URIs, RESTful interaction and extensible header options, but with a lower cost in terms of bandwidth and implementation complexity. CoAP has the following features:

Unlike HTTP over TCP, CoAP operates over UDP, with reliable unicast and multicast support (c.f. Fig.4.7).

1. CoAP transaction layer is used to deal with UDP and the asynchronous nature of the interactions. Within UDP packets, CoAP uses a four-byte binary header, followed by a sequence of options. Four types of messages are defined, which provide CoAP with a reliability mechanism: Confirmable (CON, messages require acknowledgment), Non-Confirmable (NON, messages do not require acknowledgment), Acknowledgment (ACK, an acknowledgment to a CON message), and Reset (RST, messages indicate that a Confirmable message was received, but some context is missing to properly process it. For example, the node has rebooted).

2. On top of CoAP's transaction layer, CoAP Request/Response layer is responsible for the transmission of requests and responses for resource manipulation and interoperation. The familiar HTTP request methods are supported: *GET* retrieves the resource identified by the request URI. *POST* requests the server to update/create a new resource under the requested URI. *PUT* requests that the resource identified by the request URI to be updated with the enclosed message body. *DELETE* requests that the resource identified by the request URI to be deleted.

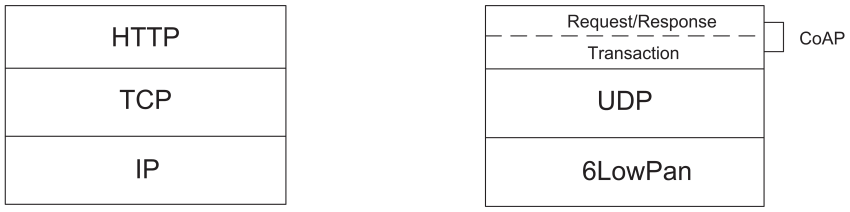


Figure 4.7: Protocol stacks of HTTP and CoAP

CoAP supports built-in resource discovery, which allows discovering and advertising the resources offered by a device. A subscription option is provided for client to request a notification whenever a resource changes. This is then accomplished by the device with the resource of interest by sending the response messages with the latest change to the subscribers.

CoAP supports block wise transfer. Basic CoAP messages work well for the small payloads such as data from temperature sensors, light switches, etc. Occasionally, applications need to transfer larger payloads – for instance, for firmware updates. Instead of relying on IP fragmentation, CoAP is equipped with *Block* options to support the transmission of large data by splitting the data into blocks.

Besides, CoAP also have other features like best-effort multicast, cachability, HTTP mapping, etc. These characteristics of CoAP provide a flexible and versatile application framework. Although CoAP is still a work in progress, many famous embedded operating systems, e.g. Tiny OS[12] and Contiki[13], have already released their CoAP implementations. It is slated to become one of the most important ubiquitous application protocols for the future Internet of Things.

4.5.2.2 Test Purposes Selection and Test Cases Generation

Generally in an interoperability testing event, the organizers will study and discuss together to choose the important properties (ITP) to test. Then for each ITP, an iop test case is derived and must be valided by all the members. Regarding the specifications of CoAP [20, 23, 25, 26], the organizers of the plugtest event were agree on focusing on a set of 27 interoperability test purposes concerning the 4 aspects listed below. They represent the most important properties of the protocol. To ensure that the ITPs are correct w.r.t the specifications, the ITPs were chosen and cross-validated by experts from ETSI[14], IRISA[15] and BUPT[16],

[12]http://www.tinyos.net/
[13]http://www.sics.se/contiki/
[14]http://www.etsi.org/WebSite/homepage.aspx
[15]http://www.irisa.fr/
[16]http://www.bupt.edu.cn/

and reviewed by IPSO Alliance. The test purposes involve:

- Basic CoAP methods: Similar to HTTP, CoAP uses the Representational State Transfer (REST) architectural style. Applications communicate by exchanging resources using respectively RESTful methods GET, POST, PUT, and DELETE. RESTful methods testing involves verifying that both CoAP client and server react correctly to the received messages according to [20]. Specifically, it requires to verify that each time the client sends a request, it contains the correct method code and correct message type code (CON or NON). Upon the reception of the request, the server sends piggybacked reply to the client accordingly: *(i)* if the request is a confirmable message, the server must send an acknowledgment ACK. *(ii)* If the request is non-confirmable, the server also sends a nonconfirmable reply. An example can be found in Fig.4.8-(a).

  Sometimes however, a server cannot obtain immediately the resource requested by the client. In this case, the server will first send an acknowledgment with empty payload, which effectively is a promise that the request will be acted. When the server finally has obtained the resource representation, it sends the response in a confirmable mode to ensure that this message not be lost. (c.f. an example in Fig. 4.8-(b))

  The interoperability testing of CoAP RESTful methods involves verifying that both CoAP client and server interoperate correctly w.r.t different methods as specified in [20]. Even in lossy context as often encountered by M2M communication. Moreover, as CoAP protocol is designed for constrained networks, where many packet losses will occur, therefore an important aspect is to show that CoAP application should still interoperate correctly even in lossy context. Especially, they must correctly retransmit the request and response if they are lost.

- Resource discovery. As CoAP applications are considered to be M2M, they must be able to discover each other and their resources. Thus, [26] standardizes a resource discovery format to discover the list of resources offered by a device, or for a device to advertise or post its resources to a directory service. In [26], path prefix for CoRE discovery is defined as */.well-known/core*. This description is then accessed with a GET request on that URI. The interoperability of resource discovery involves verifying that: when the client requests */.well-known/core* resource, the server sends a response containing the payload indicating all the available links. For example, *GET /.well-*

known/core?rt=Temperature[17] would request only resources with the name *Temperature*. The interoperability of resource discovery involves verifying that: when the client requests */.well-known/core* resource, the server sends a response containing the payload indicating all the available links.

- Block-wise transfer: CoAP is based on datagram transports such as UDP, which limits the maximum size of resource representations that can be transferred. In order to handle large payloads, [25] defines an option *Block*, in order that large sized resource representation can be divided in several blocks and transferred in multiple request-response pairs. It supports the transmission of larger amounts of data by splitting the data into blocks for sending and manages the reassembly on the application layer upon receipt in order to avoid fragmentation on the lower layers.

  Block-wise transfer is in form of stop-and-wait mechanism. If the client knows the large resource that it requires, it sends a GET request containing Block option, indicating block number and desired block size. In return, the server sends a response containing the requested block number and size. The transaction repeats until the client obtains the whole resource. If a response generated by a resource handler exceeds the client's requested block size, the server automatically divides the response and transfer it in a block-wise manner. Then the client sends further requests until completely receiving the resource and displaying it. Similarly, block options also make it possible for the client to update or create a large size resource on the server, by using PUT and POST request respectively.

  The interoperability testing of this property therefore involves in verifying that: when the client requests or creates large payload on the server, the server should react correctly to the requests.An example can be found in Fig.4.8-(c).

- Resource observation: The conventional communication model of REST is that a client always initiate requiring resource representations from the server. However, this model does not work well when a client is interested in having a current representation of a resource over a period of time. Therefore, [23] extends the CoAP core protocol with a mechanism CoAP Observe. It is an asynchronous approach to support pushing information from servers to clients.

[17] *rt*: Resource type attribute. It is a noun describing the resource.

The interoperability testing of Observe consists of the following aspects: If a client is interested in the current state of specific resource, it can register its interest in this resource by issuing a GET request with an empty Observe option to the resource. If the server accepts this option, it keeps track of the client and sends a response whenever the observed resource changes. If the client rejects a notification with a RST message or when it performs a GET request without an Observe option for a currently observed resource, the server will remove the client from the list of observers for this resource. And the client will no longer receive any updated information about the resource. If a client wants to receive notifications later, it needs to register again. An example can be found in Fig.4.8-(d).

Example 4.1 The following figure demonstrates some typical examples of CoAP transactions:

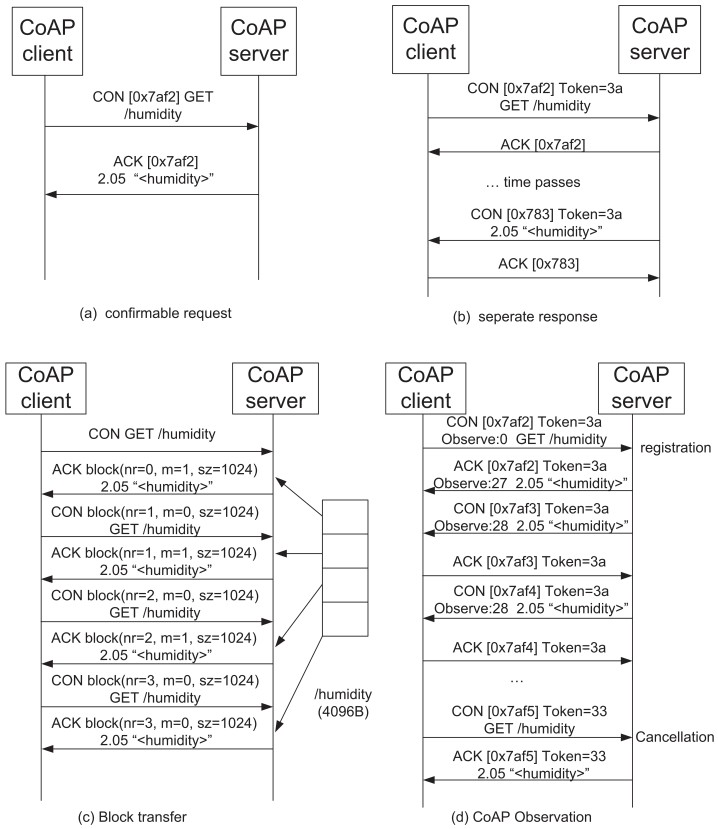


Figure 4.8: CoAP transaction examples

Fig.4.8-(a) illustrates a confirmable request sent by the client, asking for the

resource of humidity. Upon the reception of the request, the server acknowledges the message, transferring the payload while echoing the Message ID generated by the client.

Fig.4.8-(b) is an example of a separate response: When a CoAP server receives a request which it is not able to handle immediately, it first acknowledges the reception of the message by an acknowledgment with empty payload, and later sends back the response in a separate manner.

Fig.4.8-(c) illustrates a block-wise transfer of a large payload (humidity) requested by the client. Upon the reception of the request, the server divides the resource into 4 blocks and transfer them separately to the client. Each response indicates the block number and size, as well as whether there are further blocks (indicated by value m [25]).

Fig.4.8-(d) illustrates an example of resource observation, including registration and cancellation. At first, the client registers its interest in humidity resource by indicating Observe option. After a period, it cancels its intention by sending another GET request on the resource without Observe option.

Once the set of test purposes are defined, a test case is derived for each test purpose which describes in detail the expected behavior of the CoAP implementations to be observed. In this work, test cases derivation is done manually by carefully studying the protocol specification document. They are also cross-validated by experts from ETSI, IRSIA, BUPT and IPSO Alliance.

Example 4.2 Fig.4.9 shows an example of an iop test purpose and the corresponding iop test case. The test purpose (Fig.4.9-(a)) focuses on the GET method in confirmable transaction mode. i.e., when the client sends a GET request (with parameters: a Message ID, Type=0 for confirmable transaction mode, Code=1 for GET method), the server's response contains an acknowledgment, echoing the same Message ID, as well as the resource presentation (Code=69(2.05 Content)).

The corresponding test case is illustrated in Fig.4.9-(b). The bold part of the test case represents the expected behavior that leads to *Pass* verdict. Behavior that is not forbidden by the specifications leads to *Inconclusive* verdict (for example, response contains code other than 69. These events are noted by m in the figure). However other behavior leads to *Fail* verdict (for example non-match of Message ID. These events are labeled by *otherwise*). The test cases are derived w.r.t the specifications of CoAP and implemented in ttproto.

Moreover, during the test, expected behavior to be observed is provided in text form to the users as test specification document (Fig.4.9-(c)).

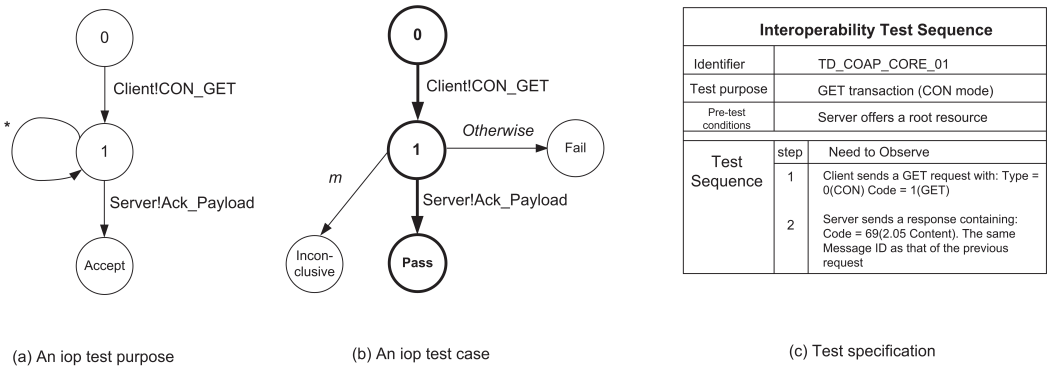


Figure 4.9: CoAP request examples

Example 4.3 This example shows a test case implemented in the test tool ttproto.

Interoperability Test Description			
Identifier:	TD_COAP_CORE_12		
Test Purpose:	Handle request containing several URI-Path options		
References:	CoAP Specification, clause 5.4.5, 5.10.2,6.5		
Pre-test conditions:	• Server offers a **/seg1/seg2/seg3** resource		
Test Case:	**Step**	**Type**	**Description**
	1	Check	Client sends a confirmable GET request to server's resource. Request must contain: • Type = 0 (CON) • Code = 1 (GET) • Option type = URI-Path (one for each path segment)
	2	Check	Server sends response containing: • Code = 69 (2.05 content) • Payload = Content of the requested resource • Content type option

```
class TD_COAP_CORE_12 (CoAP Testcase)
        def run (self):
                self.match_coap ("client", CoAP (code = "get",
                                                 opt = Opt(CoAPOptionUriPath(), superset=True)))
                opts = list (filter ((lambda o: isinstance (o, CoAPOptionUriPath)), self.frame.coap["opt"]))
                        if len (opts) > 1:
                                self.setverdict ("pass", "multiple UriPath options")
                        else:
                                self.setverdict ("inconc", "only one UriPath option")

                        for o in opts:
                                if "/" in str (o["val"]):
                                        self.setverdict ("fail", "option %s contains a '/'" % repr (o))
                        self.next_skip_ack()
                        self.match_coap ("server", CoAP (       code = 2.05,
                                                                pl = Not (b""),
                                                                opt = Opt (CoAPOptionContentType()), ))
```

Figure 4.10: A CoAP test case example

The test case aims to test a request containing several URI-Path options [20]. CoAP uses the URI schemes for identifying CoAP resources and providing a means of locating the resource. The syntax of URI schemes is as follows: coap-URI = "coap:" "//" host [":" port] path-abempty ["?" query]. In this scheme, Uri-Host specifies the Internet host of a resource; Uri-Port specifies the port of the host; Uri-Path specifies the resource of the host; Uri-Query indicates additional

options for the request. In this work we focus on the test of Uri-Path and Uri-Query. Particularly in case of a request containing several URI-Path options or URI-Query options, the test system verifies that the server send a response message with the correct message type and code in corresponding to the previous request, as well as the requested resource. The test case implemented in ttproto specifies in detail the events that can lead to *Pass*, *Fail* or *Inconclusive* verdicts.

4.5.3 Application in Industrial Context

4.5.3.1 CoAP Plugtest Overview and Testing Architecture

The interoperability approach that we proposed for request-response protocol was applied in the CoAP Plugtest event held in Paris, Mars 2012 [18]. It was the first formal two-day's event held for CoAP protocol in the scope of Internet of Things. 15 developers and vendors of CoAP implementations, such as Sensinode[19], Watteco[20], Actility[21], etc. participated in the event. Test sessions are scheduled by ETSI so that each participant can test their products with all the other partners. During the test event, CoAP implementations from different manufactures are interconnected in pair-wise combinations. Test sessions are scheduled by ETSI so that each participant can test their products with all the other partners (1 hour per session). Fig.4.11 shows the test bed architecture provided by ETSI for this event. Each company was given with a switch to connect their implementations in the test bed. Communication were routed using layer 2 and layer 3 routers.

The test suite composed of 27 test cases (c.f. Section 4.5.2.2), concerning the basic RESTful methods, Link format, Observation and Blockwise transfer of CoAP were served as test reference.

Two test architectures have been defined for different purposes. The basic test architecture is illustrated in Fig.4.12-(a). It involves a *Test System (TS)* and a *system under test (SUT)* composed of 2 CoAP applications, namely a CoAP client and a CoAP server. Since we apply the technique of passive testing, a packet sniffer is used to capture the packets (traces) exchanged between the IUTs. Moreover, as CoAP is designed for constrained networks, which imply many packet loss, we also need to consider testing the interoperability of CoAP applications in lossy environment. The corresponding architecture is as Fig.4.12-(b): A UDP gateway is used in-between the client and server to emulate a lossy

[18] http://www.etsi.org/plugtests/coap/coap.htm
[19] http://www.sensinode.com/
[20] http://www.watteco.com/
[21] http://www.actility.com/

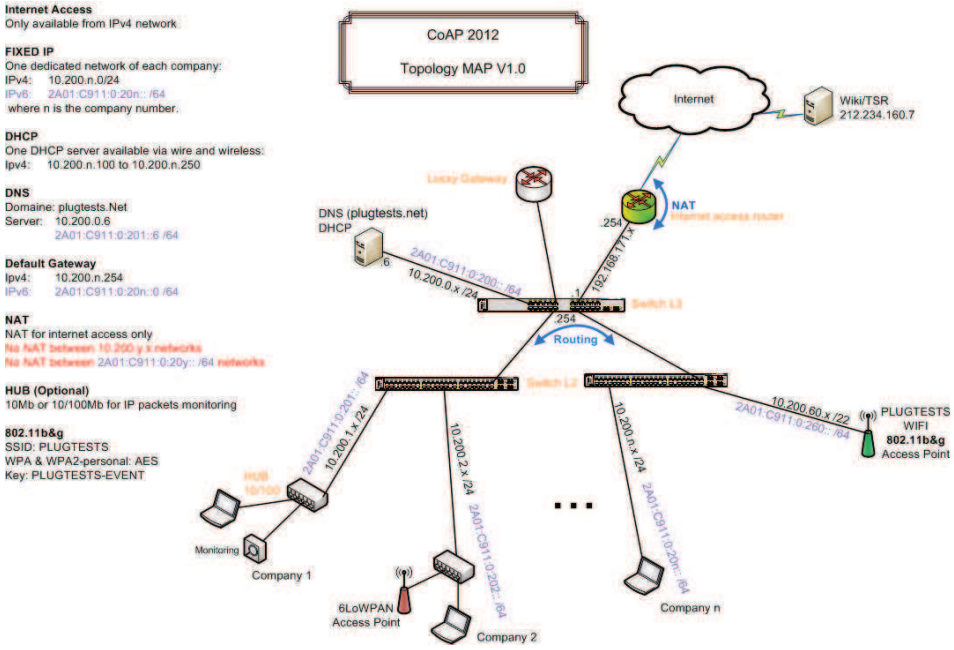


Figure 4.11: CoAP Plugtest Test Bed

medium. The gateway does not implement the CoAP protocol itself (It is not a CoAP proxy). It plays the following roles:

- It performs NAT-style UDP port redirection towards the server (thus the client contacts the gateway and is transparently redirected towards the server).
- It randomly drops packets that are forwarded between the client and the server. In Plugtest, the gateway drops the packet randomly between Client and the server which goes more than 50% packet loss, which corresponds to the unreliable environment of the Internet of Things.

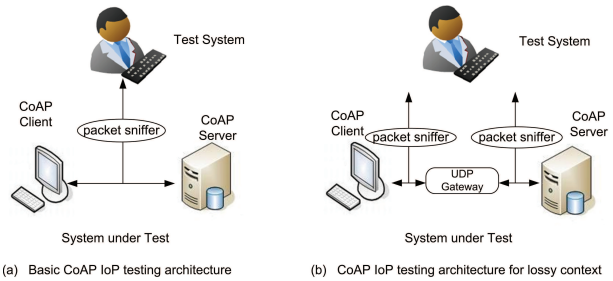


Figure 4.12: CoAP interoperability testing architectures

4.5.3.2 Test Execution with Ttproto

The testing method is based on the technique of passive testing as described in Section 4.3. During the test, the tool *Wireshark* was used to capture the packets changed by the CoAP applications. It produces pcap files which contain the traces. Participants then submit the traces to the trace validation tool ttproto. Once a pcap file is submitted, a CoAP filtering is made using source IP address and destination IP address to filter only the conversations made between the client and the server. When the conversations are isolated, then trace verification is executed by using the passive testing tool presented in Section 4.4.2. For CoAP Plugtest, the tool was developed to support the message formats of the CoAP drafts. CoAP test cases are implemented.

During the plugtest, 410 traces produced by the CoAP devices were captured and then processed by the passive validation tool. Received traces are filtered, parsed and analyzed against the test cases. And an appropriate verdict *Pass, Fail,* or *Inconclusive* is issued for each test purpose. A snapshot of the tool is as follows:

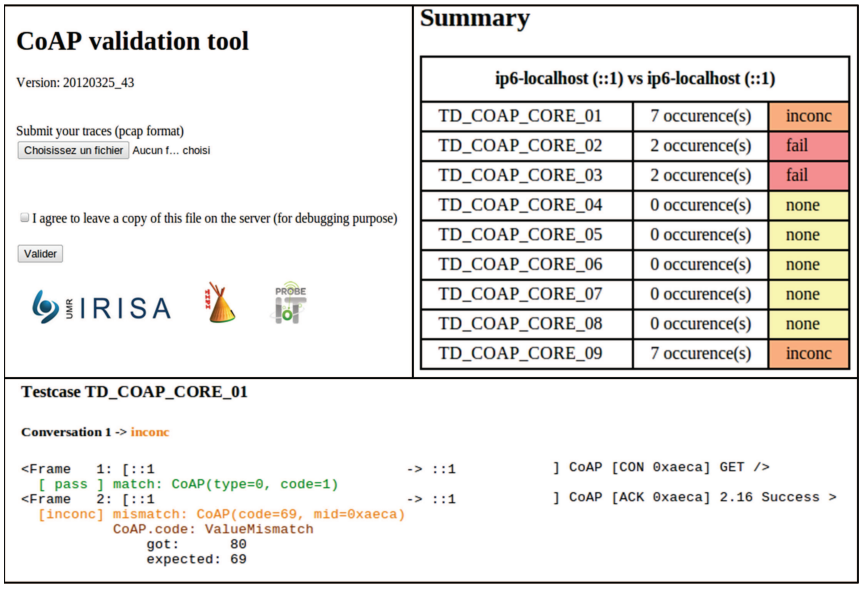


Figure 4.13: Passive testing trace verification tool

The top left image is the user interface of the tool. Users can submit their traces in pcap format. Then, the tool will execute the trace verification algorithm and return back the results as shown at the top right corner in the summary table. In this table, the number of occurrence of each test case in the trace is counted, as

```
1   -> GET
2   -> POST
3   -> PUT
4   -> DELETE
:      :
69  -> 2.05 Content
:      :
131 -> 4.03 Forbidden
132 -> 4.04 Not Found
```

Figure 4.14: Ttproto field description

well as a verdict *Pass*, *Fail* or *Inconc(lusive)* is given (For a test case which does not appear in the trace, it is marked as "*none*" and will not be verified on the trace). Moreover, users can view the details about the verdict for each test case. In this example, test case TD_COAP_CORE_1 (GET method in CON mode) is met 7 times in the trace. The verdict is *Inconclusive*, as explained by the tool: *CoAP.code ValueMismatch* (cf. the bottom of Fig.4.13). This precision is thanks to the value description function provided in ttproto. In ttproto, for values in a field, descriptions are attached (c.f. Fig.4.14). It helps when analyzing test logs, for example each Code field defined in CoAP protocol contains a different meaning. the value description makes the result analysis easier to understand. In this example, according to the test case, after that the client sends a request (with Type value 0 and Code value 1 for a confirmable GET message), the server should send a response containing Code value 69(2.05 Content). However in the obtained trace, the server's response contains Code value 80, indicating that the request is successfully received without further information. This response is not forbidden in the specification, however does not allow to satisfy the test case. In fact, the same situation exists in all the other conversations that correspond to this test case. Therefore, the verdict is *Inconclusive*.

4.5.3.3 Results of the CoAP Plugtest

During the CoAP Plugtest, a total of 3081 tests were executed during this two days event within 234 test sessions. The feedback from participants on the testing method and passive validation tool is positive, due to the following results:

- To our knowledge, it is the first time that an interoperability event is conducted by passive testing. Conventional interoperability events rely on *active testing*, which is done by actively stimulating the implementations and verifying the corresponding outputs. However, most of stimulation of these IUTs is manual, which need the intervention of experts for installation, syn-

chronization, etc., Besides, according to our experience [28], active testing cause many false negative verdicts: most of *Fail* verdicts are in fact due to the inappropriate network configuration, synchronization and inappropriate IUTs configuration. Also, the non-intrusive property of passive testing allows to discover interoperability issues that may appear in operational environment, where the normal operations of the IUTs are not disturbed, as it was the case during the plugtest event.

- The automation of trace verification increases the efficiency. According to ETSI, most of the time (about 60%) of interoperability testing is spent on trace validation, including verifying the results and looking at the problems of unsuccessful tests with the help of experts. Passive automated trace analysis allows to considerably reduce the time. In consequence, within the same time interval, the number of executed tests are drastically increased. During the CoAP plugtest, 3081 tests were executed within two days. Compared with past conventional plugtest event, e.g. IMS InterOp Plugtest[22], 900 tests in 3 days, the number of test execution and validation benefited a drastic increase. The re-usability of the test cases implemented by the tool also, will contribute to increasing efficiency for future CoAP interoperability tests. In fact, it has been the case for the second CoAP Plugtest held in November 2012 in Sophia Antipolis, France. [23]

- The passive testing tool is easy to use. In fact, the participants only need to submit their traces via a web interface. Complicated test configuration is avoided. The test reports provided by the validation tool makes the reason of non-interoperable behavior be clear. Besides, another advantage of the validation tool is that it can be used anytime and anywhere, not only during an interoperability event. (It is hosted at http://senslab2.irisa.fr/coap/). In fact, the participants started trying the tool one week before the event by submitting more than 200 traces via internet. This allows the participants to prepare in advance the test event and to revise, if necessary, their implementations.

- Moreover, the passive testing tool shows its capability of non - interoperability detection (c.f. the second column of Table 4.1): Among all the tests, 5.9% show non-interoperability w.r.t basic RESTful methods; 7.8% for Link Format, 13.4% for Block transfer and 4.3% for Observe. The results help the vendors uncover the blocking issues and to achieve higher

[22]http://www.etsi.org/plugtests/ims2/About_IMS2.htm
[23]http://www.ipso-alliance.org/nov-29-30-etsiipso-coap-plugtest-sophia-antipolis-france

quality implementations.

	Executed Tests	Non-interoperable
RESTful Methods	2798	166 (5.9%)
Link Format	77	6 (7.8%)
Block transfer	112	15 (13.4%)
Observe	94	4 (4.3%)
Total	3081	191 (6.2%)

Table 4.1: CoAP Plugtest Results

4.6 Conclusions

In this chapter, we have proposed a passive testing methodology for request-response protocols, including passive interoperability test case generation, methods of trace verification, a test tool and the application of the proposed solution on a case study.

In this chapter, passive interoperability test cases are generally manually, while more emphasize is put on test execution. According to the special interaction mode of request-response protocols, the traces collected during the test were filtered and analyzed to verify the occurrence of the test cases and their validity. The trace verification procedure has been automated in a testing tool ttproto, which was successfully put into operation in the interoperability testing events of CoAP protocol.

The contributions and originality of the work presented in this chapter are the following:

1. It provides a solution to carry out passive interoperability testing of request-response protocols. The non-intrusive nature of passive testing makes it appropriate for interoperability testing, especially in the context of IoT.

   This approach, although targets for transactional protocols, contains other important meanings: First of all, it realizes interoperability testing without formal specification model. Moreover, based on property verification, the approach improves the method of *invariants* in terms of accuracy, while invariants only focus on the expected behavior to be verified.

2. Contrary to manual verification used in conventional interoperability testing events, the verification procedure has been automatized by a test validation tool, which increases considerably the efficiency. The tool is implemented in language Python3 mainly for its advantages: easy to understand, rapid prototyping and extensive library. The tool is influenced by TTCN-3, it implements basic TTCN-3 snapshots, behavior trees, ports, timers, messages types, templates, etc. However it provides several improvements, for example object-oriented message types definitions, automatic computation of message values, interfaces for supporting multiple input and presentation format, implementing generic codecs to support a wide range of protocols, etc. These features makes the tool flexible, allowing to realize passive testing.

3. As IoT implies providing services in lossy networks, we also take into account fundamental CoAP implementations interoperability testing in lossy context. Lossy context is frequently met in operational networks, however neglected in most of the current works.

Future work will consider improving the test method. In fact, due to the uncontrollable nature of passive testing, Inconclusive verdicts are frequently emitted, which needs rerunning the test or post-analysis of experts. Therefore, a solution to reduce Inconclusive verdicts is being worked on. Also, in this work, we have chosen to use offline trace verification, i.e., traces are at first recorded and then processed. Future work will consider online trace verification, in order to monitor the network for a long time and report abnormalities at any time. Moreover, we tend to improve the testing tool so that it can be adapted to a wider range of protocols.

Chapter 5

Conclusions

5.1 Summary of Contributions

In this thesis, we focus on interoperability testing, especially using the passive testing technique. Currently, the field of interoperability testing is dominated by active testing. After study and comparison, we find that active and passive testing are complementary and they fit different situations. Respectively, active testing aims at detecting faults on SUT by applying a series of control and observation on it, while passive testing is based on observing the behavior of the SUT. After studying the state of the art of interoperability testing, we feel that applying passive approach on interoperability testing worth further work and research. Based on the subject of the thesis, several contributions have been made:

In Chapter 2, the various testing methodologies of active and passive testing were elaborated. The advantages and drawbacks of these two techniques were compared. By analyzing both methods,we draw a conclusion that passive testing is more suitable to test interoperability due to several reasons: The non-intrusive nature of passive testing does not influence the normal operation of the IUTs. In consequence it is suitable for testing in operational networks as it is often the case for interoperability testing. Also, passive testing requires less complexity in terms of network configuration, which however is not only time-consuming, but also may be the source of error that may influence the verdicts. Besides, passive testing is also appropriate for some specific context for example the Internet of Things, where the environment is often resource limited. The fact the passive testing does not introduce extra overhead is favorable for this kind of environment. Last but not least, passive testing allows monitoring the behavior of a SUT, as the analysis is only based on observation.

Despite of the advantages of passive testing, in interoperability testing, well defined methodologies is missing. To the best of our knowledge, there is no clearly

specified architecture or rule to carry on interoperability testing in passive manner. Neither the issue of verdict is well discussed. Moreover, as interoperability testing and validation are a tedious task, there is strong requirement of tool support and automation. In order to solve these problems, in the rest of the thesis, we have proposed several contributions.

In Chapter 3, a formal specification-based approach to perform passive interoperability testing is proposed. It includes two parts:

1. Automatic interoperability test cases generation. The method involves in carrying out a partial asynchronous interaction calculation of the specifications of each implementation under test with respect to a preselected test objective. To relieve state exploration, several techniques are applied: The partial asynchronous calculation is carried out in a depth-first manner to minimize the memory requirements. Each test objective is assigned with attributes to indicate the important states to be explored as well as to inhibit unnecessary states exploration. We also apply rules to avoid generating redundant transitions. The obtained graph intends to keep only the events which are relevant to the test objective. We further apply verdict assignment and minimization rules to build an executable minimized passive interoperability test case.

2. Trace verification. A trace analysis algorithm is proposed to check a recorded execution trace against the passive test case. Also the verdict assignment issue, which is often neglected, however important, is discussed. The proposed approach has been successfully performed on a Session Initiation Protocol (SIP) case study.

Also in this chapter, verdict assignement was discussed, which must be treated carefully in passive interoperability testing.

In Chapter 4, another method for passive interoperability testing is proposed, especially for request-response protocols in the context of client-server communications. According to the interaction pattern of request-response protocols, the observed interactions (trace) between the network components under test can be considered as a set of conversations between client and server. Then, a procedure to map each test case into these conversations is carried out, which intends to verify the occurrence of the generated test cases as well as to determine whether interoperability is achieved. The trace verification procedure has been automated in a passive testing tool ttproto, which analyzes the collected traces and deduces appropriate verdicts.

Although the solution aims at request response protocol, it has some important meanings: First, it allows carrying out test with out formalizing specifications, which is generally a time-consuming task. Moreover, it allows to perform test when a protocol is in its early stage (for example, when the specification is still under draft version). Also it is appropriate for complex protocol designed in a way that allows future evolution. Second, request response protocols represent a popular protocol family, which is widely used today and certainly in the future. The testing procedure is automated in a testing tool and was put into operation during the test events of CoAP protocol plugtests. Last but not least, actually in field of interoperability, most of the work considers the assumption of reliable environment where there is no loss or duplication. However in reality this assumption is often challenged (for example in the context of the Internet of Things). To solve this issue we take into account the lossy factor during the interoperability testing event.

Also in this chapter, a new passive testing prototype ttproto to automate the tests is introduced. It allows easy try-out of TTCN-3 and experimenting with different language constructs. It is inspired from TTCN, but explores some new features. Especially it proposes a concept of feasibility and modularity, which jumps out the limitation of the active testing paradigm of TTCN. This makes the tool feasible in the scope of this thesis, for passive testing. Its successful application during the CoAP Plugtests showed its efficiency, flexibility and capacity of property validation.

5.2 Future Work

Besides the contributions presented in Section 5.1, a list of future work has also been identified and will be presented in the sections 5.2.1 and 5.2.2. The contributions concern the following aspects: improve trace verification, extending passive interoperability to multi-iut context, refine verdict assignment, etc. Among these aspects, we have began to work on a solution to make trace verification more efficient by test case grouping. Section 5.2.1 will present the first result of improving trace verification. Other long term future work will be presented in Section 5.2.2.

5.2.1 Improve Trace Verification by Test Case Grouping

5.2.1.1 Motivation

During the thesis, besides the proposed testing methods and verification algorithms, we also feel some aspects that worth further study. One of them is trace verification. Actually, the algorithms proposed previously in this thesis allow

verifying each time the trace against only one passive interoperability test case. If there are n test cases to verify, trace verification will be executed n times. If the trace is long, the time spent on verifying sequentially the n test cases will be considerable.

To deal with this problem, this section suggests a solution to improve passive interoperability trace verification. In fact, the main factors that influence trace verification time are the length of the trace and the number of passive iop test cases (PITC). As the length of the trace plays an important role in passive testing, we try to reduce the number of PITCs to reduce the time of trace verification. Therefore, the method aims at grouping adequately the individual passive iop test cases into a global test case, so that the traversal of the trace is executed only once. We also propose an associated trace verification algorithm, which allows verifying the trace against the global test case. This method in fact, allows analyzing the trace against all the passive test cases in parallel. In consequence, the time used to carry out trace verification time is reduced. Nevertheless, using this method does not reduce the performance of property verification: All the individual passive iop test cases are adequately integrated into a global one without being removed.

5.2.1.2 State of the Art

Test suite reduction is a subject on which researchers work as testing and retesting occur continuously during the software development life cycle. As software grows and evolves, the accompanying test suites have to be generated and used to verify the expected properties. Over time, the test suite may become huge and needs a long time to verify each test case. Therefore, conventional test reduction focuses mainly choosing a minimal subset of test cases from a large test suite, to avoid executing all the test cases without sacrificing the fault coverage.

In many of the research papers (e.g. [31], [33], [34]), test suite reduction aims to remove test cases from a test suite in such a way that redundant test cases are eliminated. For example, a reduced test suite *TS* may provide the same structural coverage as a test suite *TS-reduced* with significantly fewer test cases. Particularly, test case minimization algorithm picks randomly a test case from a test suite, and mark the corresponding coverage that it covers. Any test case that improves the coverage is kept, while other test cases are considered as redundant and are removed. In this way, the size of the test suite is often considerably reduced, so as the time of test case execution.

Since test suite minimization removes test cases, minimized test suites may be weaker at detecting faults than their unminimized counterparts. Over the past years, studies show conflict results: in [31], it was shown that minimizing test

suites could result in little loss in fault detection effectiveness. However, the empirical study in [32] suggests that minimized test suites can severely compromise the fault detection capability. As a result, a common way to solve this problem is thus taking into account of several coverage requirements and criteria, keeping some redundant test cases in stead of removing all of them.

5.2.1.3 Grouping Passive Interoperability Test Cases

The test suite reduction techniques introduced in Section 5.2.1.2 however, do not fit for passive interoperability testing. This is due to the fact that interoperability testing involves a SUT that contains at least two IUTs. Normally, the behavior of the SUT can be obtained by computing the global behavior of the IUTs. But due to state explosion, coverage based test suite reduction by global behavior calculation is difficult to realize.

In this chapter, we choose to use another strategy. We study the following issue: We have a large set of test cases and we want to reduce the time of trace verification in passive interoperability testing without sacrificing the property verification capability. In interoperability testing in fact, it happens frequently that several test cases concern the similar functionality. So they often share the same preamble. Therefore, the idea is that: In stead of removing test cases from a test suite, the method aims at integrating adequately all the individual passive iop test cases $PITC_i$ into a global one – $PITC_G$.

Suppose we have a passive iop test suite TS, which contains n PITCs. Each $PITC_i$ $(1 \leqslant i \leqslant n)$ is in the form of a deterministic acyclic tree, associated with attributes $Pass_i$, $Inconclusive_i$ and/or $Fail_i$ on their trap states. PITCs grouping procedure is in fact constructing a global $PITC_G$ tree, which are composed of all the transitions of all the PITCs. i.e., it contains the all the information in each individual $PITC_i$ derived from the corresponding iop test purpose. Moreover, the point is that in this way, the common prefix shared by several test cases will appear only once in the $PITC_G$. Trace verification is thus turning to check whether the recorded trace σ contains one or more branches of the $PITC_G$ tree which are associated with *Pass* attribute. In other words, trace verification is to traversing the trace only once against the global test case, i,e, all the test cases in parallel. Moreover, as no test case is removed, the global test case has the same non-interoperability detection power as the original unminimized test suite.

The detailed PITC integration procedure is as follows: Initially, the $PITC_G$ is empty. We start by creating an initial root node $PITC_0^G$. Then, we pick up one $PITC_i$ from the test suite TS. Then, according to certain order (breath-first or depth-first for example, in this work more precisely in depth-first order), we

check all the transitions of $PITC_i$ from its initial state to see how they can be integrated into the $PITC_G$. We require that the global passive iop test case $PITC_G$ must be able to:

1. Identify and associate with appropriate state any attribute information. i.e, all the attributes *Pass/Inconclusive/Fail* in each individual PITC must be kept for verdict assignment reason.
2. Associate all the states and transitions in each individual PITC with appropriate position in the global passive iop test case.
3. Take into account the common preamble shared by several individual PITCs only once.

In order to respect the above obligations and generate correctly the PITC, we use a tuple ($PITC_i^{Current}$, $PITC_G^{Current}$) to indicate the current state in $PITC_i$ under checking, and its corresponding position (state) in the global PITC where a transition at $PITC_i^{Current}$ could be integrated. The way of integrating a transition is done according to the following rules:

Rule 1

For a tuple ($PITC_i^{Current}$, $PITC_G^{Current}$), if ($PITC_i^{Current}$, α, $PITC_i^{Next}$) $\in \triangle^{PITC_i}$ and ($PITC_G^{Current}$, α, $PITC_G^{Next}$) $\in \triangle^{PITC_G}$. i.e., a transition labeled by α already exists in the global PITC (It must have been generated in the global PITC some time before). In this case there is no need to generate another transition labeled by α in the global PITC (Common prefix appears only once in the global PITC). However, if $PITC_i^{Next}$ is associated with an attribute $Pass_i$, $Inconclusive_i$, or $Fail_i$, the attribute must be kept and associated to its appropriate state (state $PITC_G^{Next}$, which is led by the transition label α) in the global passive iop test case.

The rule is expressed below, where $PITC_G^{Next}.Attribute(PITC_i^{Next})$ means that according to the attribute $Pass_i$, $Inconclusive_i$, or $Fail_i$ associated to $PITC_i^{Next}$, the same attribute will be associated to $PITC_G^{Next}$ as well.

$$\frac{\begin{array}{c}(PITC_i^{Current}, \alpha, PITC_i^{Next}) \in \triangle^{PITC_i} \\ (PITC_G^{Current}, \alpha, PITC_G^{Next}) \in \triangle^{PITC_G} \\ Attribute(PITC_i^{Next}) \in \{Pass_i, Inconclusive_i, Fail_i\}\end{array}}{PITC_G^{Next}.Attribute(PITC_i^{Next})}$$

Rule 2

At ($PITC_G^{Current}$, $PITC_i^{Current}$), if ($PITC_i^{Current}$, α, $PITC_i^{Next}$) $\in \triangle^{PITC_i}$, however $(PITC_G^{Current}, \alpha, PITC_G^{Next}) \notin \triangle^{PITC_G}$. It means that a transition labeled

by α has not been generated in the global PITC yet. Then a new transition is created in $PITC_G$ at its appropriate position. The same, if $PITC_i^{Next}$ is associated with an attribute *Accept* or *Refuse*, the attribute must be kept and associated to its corresponding state in the global passive iop test case.

$$\frac{\begin{array}{c}(PITC_i^{Current}, \alpha, PITC_i^{Next}) \in \triangle^{PITC_i} \\ (PITC_G^{Current}, \alpha, PITC_G^{Next}) \notin \triangle^{PITC_G} \\ Attribute(PITC_i^{Next}) \in \{Pass_i, Inconclusive_i, Fail_i\}\end{array}}{(PITC_G^{Current}, \alpha, PITC_G^{Next}) \in \triangle^{PITC_G}, PITC_G^{Next}.Attribute(PITC_i^{Next})}$$

Therefore, the traversal of each test case is in fact a tree traversal, that integrates an individual PITC into the global tree. The procedure continues until all the $PITCs$ have been examined. After all the test cases have been processed, we can obtain a global passive iop test case $PITC_G$.

The PITC integration algorithm is written formally below:

In this thesis, we use the depth-first method to traverse the states and the transitions of each PITC.

- **Variables and functions for PITC grouping**
  - End_i: Boolean value. $End_i == True$ means that all the transitions in $PITC_i$ have been checked.
  - $PITC_i^{Current}$.trans: Return the transitions at state $PITC_i^{Current}$.
  - $|PITC_i^{Current}$.trans|: Returns the number of transitions at the state $PITC_i^{Current}$.
  - $PITC_i^{Current}$*.visited:* A set that stores all the transitions that have been checked.
  - *Branch*: A stack to store branching states (states which contain more than one transition). Each item in *Branch* is composed of *(*$PITC_i^{Current}$, $PITC_G^{Current}$*)*. i.e., it stores a branching state of $PITC_i$ and its corresponding state in $PITC_G$. The operations of *Branch* involves: *Push* adds a new item to the top of *Branch; Pop* removes an item from the top of *Branch; Stacktop* returns the value of the item from the top position of *Branch* without deleting it.
  - *transitiongeneration* ($PITC_i^{Current}$, $PITC_G^{Current}$, α) is a fonction that implements the two rules *Rule 1* and *Rule 2* introduced above to add a new transition in $PITC_G$ labeled by α. It takes three inputs: $PITC_G^{Current}$, $PITC_i^{Current}$, as well as the currently checked transition label α, and adds the transition according to Rule 1 or Rule 2.

Algorithm 5.1 Global Passive Iop Test Case $PITC_G$ Construction

Input : Test suite TS, which contains n $PITC_i$ $(1 \leq i \leq n)$
Output : The global passive iop test case $PITC_G$
Initialization: $PITC_G = \emptyset$;
create_node($PITC_G$, $PITC_G^0$) //*Create the initial node in $PITC_G$
forall the $i = 1, ...n$ **do**
  $PITC_i^{Current} \leftarrow PITC_i^0$, $PITC_G^{Current} \leftarrow PITC_G^0$
  //*Global PITC constructrion begins from the initial node
  $End_i = False$, $PITC_i^{Current}.visited = \emptyset$
  while *not* End_i **do**
    //*Each transition in $PITC_i$ must be checked
    $PITC_i^{Current}.trans$
    if $\mid PITC_i^{Current}.trans \mid == 1$ **then**
      $a \leftarrow \Gamma(PITC_i^{Current})$
      transitiongeneration ($PITC_i^{Current}$, $PITC_G^{Current}$, a)
    end
    if $\mid PITC_i^{Current}.trans \mid > 1$ **then**
      if $PITC_i^{Current} \notin Branch$ **then**
        Push(Branch,($PITC_i^{Current}$, $PITC_G^{Current}$*))*
      end
      if $\exists a$ *where* $a \in PITC_i^{Current}.trans$ *and* $a \notin PITC_i^{Current}.visited$ **then**
        add ($PITC_i^{Current}.visited$, a)
        transitiongeneration ($PITC_i^{Current}$, $PITC_G^{Current}$, a)
      end
      else
        Pop(Branch,($PITC_i^{Current}$,$PITC_G^{Current}$*))*, $PITC_i^{Current}.visited \leftarrow \emptyset$
        if $Branch \neq \emptyset$ **then**
          Stacktop(Branch)
        end
        else
          $End_i = True$
        end
      end
    end
    else
      if $Branch \neq \emptyset$ **then**
        Stacktop(Branch)
      end
      else
        $End_i = True$
      end
    end
  end
end

The function *transitiongeneration* is written below:

Algorithm 5.2 Transition Generation Function in $PITC_G$

Input : $PITC_i^{Current}$, $PITC_G^{Current}$, a
Output : $PITC_i^{Current}$, $PITC_G^{Current}$, $PITC_G$
if $(PITC_i^{Current}, a, PITC_i^{Next}) \in \triangle^{PITC_i}$ *and* $(PITC_G^{Current}, a, PITC_G^{Next}) \notin \triangle^{PITC_G}$ **then**
  createtransition $(PITC_G^{Current}, a, PITC_G^{Next})$
end
if $PITC_i^{Next} \in \{Pass_i, Inconclusive_i\, Fail_i\}$ *where* $(PITC_i^{Current}, a, PITC_i^{Next}) \in \triangle^{PITC_i}$ **then**
  $PITC_G^{Next}.Attribute(PITC_G^{Next})$ where $(PITC_G^{Current}, a, PITC_G^{Next}) \in \triangle^{PITC_G}$
  //* attributes information must be kept
end
$PITC_i^{Current} \leftarrow PITC_i^{Next}$ where $PITC_i^{Current}, a, PITC_i^{Next}) \in \triangle^{PITC_i}$
$PITC_G^{Current} \leftarrow PITC_G^{Next}$ where $PITC_G^{Current}, a, PITC_G^{Next}) \in \triangle^{PITC_G}$
Return $PITC_i^{Current}$, $PITC_G^{Current}$, $PITC_G$

This PITCs integration algorithm needs to check each transition of each PITC once. Therefore it takes time proportional to the length of each PITC. i.e., the algorithm takes time *O(L)* where L is the total number of transitions of all the PITCs.

From a constructive point of view, the composition tree $PITC_G$ allows to integrate graphically all the transitions in each individual PITC: We construct an integrated iop test purposes composition tree incrementally by adequate integration of the transitions in each individual PITC. At last we obtain a global tree produced by arranging all the transitions into a structured hierarchy, where all the information of each PITC are kept, as well as the associated attributes. The grouped passive iop test case allows parallel trace verification, which is explained in the next subsection.

5.2.1.4 Trace Verification

The constructed global passive iop test case $PITC_G$ completely represents the transitions of each PITC in a structural hierarchy. The common prefix shared by several PITCs appear only once in the global iop test case, while all the attribute information is kept to help emitting a verdict for each individual ITP. Another objective of constructing the global passive iop test case $PITC_G$ is to support parallel trace verification. i.e., the recorded trace σ needs to be traversed only once against the $PITC_G$ to check the number of satisfied given iop test purposes.

As explained before, in passive testing, no assumption is made about the moment when the recording of trace begins, and thus it is not necessarily the initial state of the $PITC_G$. To deal with this issue, we propose an algorithm that aims at realizing trace verification against the global $PITC_G$ tree. Let us consider the $PITC_G$ tree, which is in the form of a directional acyclic deterministic tree. It has one initial state $q_0^{PITC_G}$ which may contain several transitions to other states. The trap states of $PITC_G$ are associated with attributes $Pass_i$, $Inconclusive_i$, $Fail_i$ of each $PITC_i$. The idea is to find which of the branches in $PITC_G$ are encompassed by the trace. To realize the trace verification, we call *Currently_Checked_States* the set of states in PITC under current checking. Initially, *Currently_Checked_States* contains only the initial state $q_0^{PITC_G}$. Then, for each event a taken in order from the trace σ, we check whether a can be accepted by the states in the set *Currently_Checked_States*. If it is the case, these states are replaced by the destination states led by a. The states that can not be led by a are deleted (except for the initial state $q_0^{PITC_G}$, which will always be kept in the set). The algorithm stops if all the events in the trace are checked, or all the *Pass* attributes are reached (All the ITPs are satisfied). Each time an attribute $Pass_i$ (resp. $Inconclusive_i$, $Fail_i$) is reached, it will be recorded in a set *Pass_set* (resp. *Inc_set*, *Fail_set*) to help verdict assignment.

The trace verification algorithm is written formally below:

- **Variables for trace analysis:**

  - $Pass_i$*_reached, (resp.* $Inconclusive_i$*_reached,* $Fail_i$*_reached)*: Boolean. $Pass_i$*_reached (resp.* $Inconclusive_i$*_reached,* $Fail_i$*_reached)* = True means that the trace encompasses a branch in $PITC_G$ which is associated with $Pass_i$ (resp. $Inconclusive_i$,$Fail_i$) attribute in the $PITC_G$ graph.

  - *pick* (σ)*:* Take the first element from a trace σ.

  - *Pass_set* (resp.*Inc_set*,*Fail_set*): The set which stores value of the attributes $Pass_i$ (resp. $Inconclusive_i$,$Fail_i$) when $Pass_i$*_reached* (resp. $Inconclusive_i$_reached,$Fail_i$_reached)= $True$.

  - n: the number of PITCs

After the execution of the trace verification algorithm, the sets of Boolean values *Pass_set*, *Inc_set* and *Fail_set* will be returned. According to the values stored in the sets, an appropriate verdict should be emitted for each corresponding ITP_i. The rule for verdict assignment is the same as introduced in Chapter 3.

Algorithm 5.3 Trace verification

Input : Trace σ, $PITC_G$
Output : $Pass\_set, Inc\_set, Fail\_set$
Initialization: $Currently\_Checked\_States = q_0^{PITC_G}$, $Pass_i\_reached = False$, $Inconclusive_i\_reached = False$, $Pass\_set = \emptyset$, $Inc\_set = \emptyset$,$Fail\_set = \emptyset$
while $\sigma \neq \emptyset$ and *not* $\mid Pass\_set \mid == n$ **do**
  pick (σ)
  forall the *state q in Currently_Checked_States* **do**
    if $a \in Out(q)$ **then**
      if $q == q_0^{PITC}$ **then**
        add (Currently_Checked_States, p) where $(q_0^{PITC_G}, a, p) \in \triangle^{PITC_G}$
      end
      else
        $q = p$ where $(q, a, p) \in \triangle^{PITC_G}$
      end
    end
    if $a \notin Out(q)$ *and* $q \neq q_0^{PITC}$ **then**
      remove$(Currently\_Checked\_States, q)$
    end
    if $q == Pass_i$ **then**
      $Pass_i\_reached = True$, *add(Pass_set, Pass*$_i$*_reached)*
      if $\mid Pass\_set \mid == n$ **then**
        exist //* exit from the *for* loop
      end
    end
    if $q == Inconclusive_i$ **then**
      $Inconclusive_i\_reached = True$, *add(Inc_set, Inconclusive*$_i$*_reached)*
    end
    if $q == Fail_i$ **then**
      $Fail_i\_reached = True$, *add(Fail_set, Fail*$_i$*_reached)*
    end
  end
end
Return $Pass\_set$, Inc_set, $Fail\_set$

5.2.1.5 Results

The proposed passive iop test cases grouping algorithm aims at constructing a global passive iop test case incrementally by appropriate integration of the transitions of each individual test case. It contains all the behavior to be observed, which is calculated from different given test requirements. In interoperability testing, common prefix appear very often when similar functionalities are to be tested. One of the objectives of algorithm is therefore to merge the common prefix shared by several PITCs. Moreover, the procedure of trace verification is then executed only once to verify the number of the satisfied ITPs

By integrating individual passive iop test cases, the time for trace verification is reduced.

If we analyze in a general way, the time complexity for verifying a trace σ against one PITC is *O(M×N)*, where M is the length of the trace, N the number of states in the corresponding PITC which are currently under checking. Therefore, if there are n PITCs to verify, we need time $n \times O(M \times N)$. By integrating adequately all the PITCs into a global one, in stead of executing n times trace verification for n PITCs, trace verification is executed only once to verify if it satisfies one or more ITPs. We need time *O(L)* for PITCs integration where L is the sum of all the transitions contained in all the PITCs. The time complexity for trace verification is still $O(M \times N)$. Therefore totally we need $O(L) + O(M \times N)$ time by using this method, which is less than $n \times O(M \times N)$ (i.e., the time needed for verifying the trace against the PITCs one by one.)

If we analyze in a more detailed way. Suppose there are n PITCs to verify. If the trace verification is carried out in a sequential way, n PITCs needs n traversal of the trace. Let$\mid T_i \mid$be the number of the states that are actually traversed in $PITC_i$ after it is executed on the trace σ. After n times trace verification, the traversed states in all the PITCs are therefore $\sum_{i=1}^{n}(\mid T_i \mid)$. Then, as the initial state of each PITC is checked at the beginning of trace verification, we need n time units to read n initial states. At last, let $\mid \sigma_i \mid$be the time needed for traversing the trace for verifying $PITC_i$ (the number of the events in the trace that are traversed). The time used for n times trace traversing is thus $\sum_{i=1}^{n}(\mid \sigma_i \mid)$. Therefore, the total time used for trace verification is: T_sequential_trace_verification $= \sum_{i=1}^{n}(\mid T_i \mid) + n + \sum_{i=1}^{n}(\mid \sigma_i \mid)$. Note that it is often that the whole trace has to be traversed. If the trace is long, $\sum_{i=1}^{n}(\mid \sigma_i \mid)$ will be quite huge.

If we integrate the PITCs into a global $PITC_G$. We first need time L for PITCs integration where L is the sum of all the transitions contained in all the PITCs. Then, let $\mid T_G \mid$ be the number of the transitions that are traversed in

$PITC_G$ after it is executed on the trace σ. Indeed,$\mid T_G \mid$is in the worst case equal to $\sum_{i=1}^{n}(\mid T_i \mid)$ if no PITCs share the common prefix. On the contrary, if several PITCs share some common prefix, the common prefix will be merged and appears only once in the $PITC_G$, therefore $\mid T_G \mid$ is necessarily smaller than $\sum_{i=1}^{n}(\mid T_i \mid)$. Moreover, as the global PITC has only initial state, at the beginning we only need to read it once. At last, the time need for traversing the trace is $\mid pref(\sigma) \mid \leq \mid \sigma \mid$ and in the worst case the length of the trace $\mid \sigma \mid$. The total time used for trace verification is therefore T_global $= L + \mid T_G \mid + 1 + \mid pref(\sigma) \mid$, which is generally much smaller than T_sequential_trace_verification because: *(i)*$\mid pref(\sigma) \mid << \sum_{i=1}^{n}(\mid \sigma_i \mid)$. *(ii)* In passive iop testing, a trace is generally very long, sometimes even lasts for days. L is normally smaller than the length of the trace.

Moreover, the global passive iop test case $PITC_G$ has the same capability of non interoperability detection as sequential PITC verification: In fact, the global passive iop test case contains all the information (transitions, necessary observations, etc) and attributes of each individual PITC. In other words, all the behavioral information is integrated in a structural hierarchy in order to support parallel verification. No test case is removed.

Example 5.1

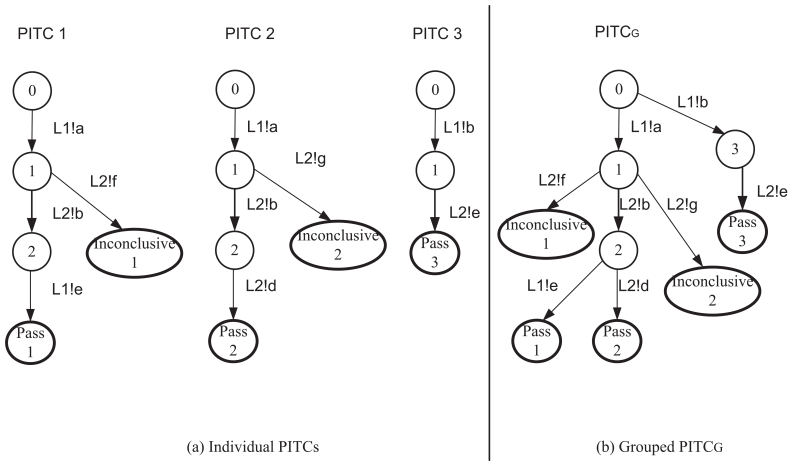


Figure 5.1: An example of PITC grouping

To clarify the idea, Fig.5.1 shows an example: Fig.5.1-(a)illustrates three individual PITCs. The global $PITC_G$ aims at integrating all the states and transitions in individual PITCs. More specifically, after the PITC grouping algorithm, the three individual PITCs should be grouped into one global $PITC_G$ (cf. Fig.5.1-

(b)) in such a way that common prefix $L1!a$ $L2!b$ shared by $PITC_1$ and $PITC_2$ have been merged and appear only once in $PITC_G$. Meanwhile, all the transitions in each PITC and their attributes are preserved in the $PITC_G$. In other words, each PITC is a sub tree of $PITC_G$. Their behavioral information is arranged compositionally that supports parallel trace verification. i.e., recorded trace needs to be traversed only once to check how many iop test purposes are satisfied.

In this example, suppose the recorded trace is: $L1!a$, $L2!b$, $L1!e$, $L1!b$, $L1!a$, $L2!g$, $L1!b$, $L2!e$. The comparison of sequential trace verification (by using the algorithm proposed in Chapter 3 for general protocols) against parallel trace verification is shown in Table 5.1.

	Currently checked states in $PITC_1$	Currently checked states in $PITC_2$	Currently checked states in $PITC_3$	Currently checked states in $PITC_G$
	{0}	{0}	{0}	{0}
$L1!a$	{0,1}	{0,1}	{0}	{0,1}
$L2!b$	{0,2}	{0,2}	{0}	{0,2}
$L1!e$	{0,$Pass_1$}	{0}	{0}	{0,$Pass_1$}
$L1!b$		{0}	{0,1}	{0,3}
$L1!a$		{0,1}	{0}	{0,1}
$L2!g$		{0,Inc_2}	{0}	{0,Inc_2}
$L1!b$		{0}	{0,1}	{0,3}
$L2!e$		{0}	{0,$Pass_3$}	{0,$Pass_3$}

Table 5.1: Trace verification comparison

If we verify the three PITCs on the trace one by one, and suppose the time to read each state is constant, and is equals to 1 time unit. we need 7 time units for $PITC_1$, 13 for $PITC_2$, and 12 for $PITC_3$. Therefore the total time needed is 32 time units. But if we group the three PITCs and verify the trace on the global $PITC_G$, we need 27 time units (10 to group PITCs, 17 for trace verification). In fact, the common prefix *L1!a, L2!b* shared by $PITC_1$ and $PITC_2$ are integrated and appear only once in $PITC_G$. But as all the attributes are kept, an appropriate verdict can be given to each PITC. In this example for instance, after verifying the trace against the $PITC_G$, $Pass_1 = Pass_3 = True$. *i.e.,* ITP_1 and ITP_3 are satisfied. Therefore the verdict *Pass* is emitted for both ITP_1 and ITP_3. For ITP_2 however, as $Pass_2 \neq True$ while $Inconclusive_2 = True$, an *Inconclusive* verdict is given. The results are the same if individual PITCs are executed on the trace in a sequential way.

5.2.2 Perspectives

Future work concerns the following aspects:

- In the previous chapters of this thesis, the trace verification is executed in a sequential trace way, which may sometimes need a long time to realize. We feel that test case grouping will improve the efficiency of trace verification. This theory, although introduced in Section 5.2.1, has not yet been put into real use. Future work will consider putting the test case grouping theory into practice.

- In this thesis, the context of passive interoperability testing is the mostly widely used one-to-one architecture – where there are two network components in interaction. Although one-to-one is the most common context, it is only a special case of interoperability testing. In reality, one-to-one architecture is not always sufficient to meet the needs of test. For instance, currently many protocols (such as SIP, HTTP, CoAP, etc...) involve the use of proxies. To assure protocol implementations work well with proxy, at least three IUTs are needed. Another example is in mobile IPv6 protocol, there exist three different kinds of nodes (correspondent node, home agent and mobile node) that need correct interaction. Therefore, in practice the interoperability of more than two IUTs is needed.

  However, more the number of IUTs in a SUT is, more complex the SUT topology can be. In this case, to have a control of the whole SUT might be difficult. Consequently, passive testing may be an appropriate strategy to observe the behavior of each IUT. Currently, the notion of interoperability in multi-iut context is yet to our knowledge still not formally defined. Future work will consider extending passive interoperability testing solutions into multi-iut domain.

- Due to the uncontrollable nature of passive testing, passive testing verdicts often encounter a large number of *Inconclusive* verdicts. This issue was discussed in Section 3.4.3 in Chapter 3. In fact, this thesis requires that a test case should be sound, in the sense that interoperable IUTs cannot be rejected. However in practice, a large number of *Inconclusive* verdicts may sometimes be time-consuming to the testers, as post analysis is often needed to further identify abnormal behavior. Therefore, a solution to refine test verdicts should be worked out to make passive interoperability testing more accurate.

- The passive interoperability testing method introduced in Chapter 4 targets mainly request-response protocols. It will be considered to be extended to other kinds of protocols, where the exchanged message patterns are more complicated than request response. As for the testing tool ttproto, we are thinking of implementing more test suites concerning other request-response protocols than CoAP. And its application in future plugtests will be considered.

Glossary

- **API** application programming interface
- **ATS** abstract test suite
- **BUPT** Beijing university of post and telecomunications
- **CD** coder-decoder
- **CoAP** constrained application protocol
- **CoRE** IETF constrained RESTful environments working group
- **DECT** digital enhanced cordless telecommunications
- **DSRC** dedicated short-range communications
- **EDGE** enhanced data rate for GSM evolution
- **ETSI** European telecommunications standards institute
- **FIFO** first in first out
- **FSM** finite state machine
- **GSM** global system for mobile communications
- **HTTP** hypertext transfer protocol
- **IAP** implementation Access Points
- **ICMP** Internet control message protocol
- **ICS** implementation conformance statement
- **IETF** Internet engineering task force
- **IOLTS** input output labled transition system

- **iop** interoperability
- **IoT** Internet of things
- **IPSO** Internet protocol for smart object communications
- **IRISA** institut de recherche en informatique et systèmes aléatoires
- **ISDN** integrated services data digital network
- **ISO** international standards organization
- **IT** information technology
- **ITC** interoperability test case
- **ITP** interoperability test purpose
- **ITS** interoperability test suite
- **ITU** international telecommunication union
- **IUT** implementation under test
- **LI** lower interface
- **MTC** main test component
- **NAT** network address translation
- **OSI** open systems interconnection
- **PCO** point of control and observation
- **PO** point of observation
- **PITC** passive interoperability test case
- **Probe-IT** pursuing roadmaps and benchmarks for the Internet of things
- **REST** representational state transfer
- **RFC** request for comments
- **PTC** parallel test component
- **SA** SUT adapter
- **SIP** session initiation protocol

- **SS7** signaling system number 7
- **SUT** system under test
- **TCP** transmission control protocol
- **TLV** type length value
- **TTCN-3** testing and test control notation version 3
- **TTL** time to live
- **ttproto** testing tool prototype
- **TS** test system
- **UDP** user datagram protocol
- **UI** upper interface
- **URI** uniform resource identifier
- **WAP** wireless application protocol

List of Publications

Conference

- Nanxing Chen, César Viho: Passive Interoperability Testing for Request-Response Protocols: Method, Tool and Application on CoAP Protocol. In 24th IFIP Int. Conference on Testing Software and Systems (ICTSS'12): 87-102, Aalborg, 2012.
- Nanxing Chen, César Viho: A Passive Interoperability Testing Approach Applied to Constrained Application Protocol. 15th CFIP (Colloque francophone sur l'ingénierie des protocoles) and 11th NOTERE (Nouvelles Technologies de la Répartition) NOTERE/CFIP, Anglet, 2012.
- Nanxing Chen, César Viho: A Methodology for Passive Interoperability Testing: Application to SIP protocol. 11th African Conference on Research in Computer Science and Applied Mathematics, Algiers, 2012.
- Nanxing Chen, César Viho: IoT Interoperability Testing: A Successful Experience on CoAP Protocol Testing. 3rd International Conference on the Internet of Things (IoT2012), Wuxi, 2012.
- Anthony Baire, César Viho, Nanxing Chen: Long-Term Challenges in TTCN-3: a Prototype to Explore New Features & Concepts. In TTCN-3 User Conference, Bangalore, 2012.
- Nanxing Chen, César Viho: An Approach to Passive Interoperability Testing. Short paper in 23th IFIP Int. Conference on Testing Software and Systems (ICTSS'11), Paris, 2011.

Journal

- Nanxing Chen, César Viho: A Passive Interoperability Testing Approach Applied to the Constrained Application Protocol (CoAP). (Extended ver-

sion of CFIP 2012 paper) To appear in RNTI journal (Revue Nouvelles Technologies de l'Information), 2013.

- Nanxing Chen, César Viho, Anthony Baire, Xiaohong Huang, Jiexi Zha: Interoperability Testing for Internet of Things: Application on CoAP Protocol. (Extended version of IOT 2012 paper) To appear in Automatika journal (Journal for Control, Measurement, Electronics, Computing and Communications), 2013.

Bibliography

[1] ISO. Information Technology-open system Interconnection Conformance Testing methodology and framework-Parts 1-7. International Standard ISO/IEC 9646/1-7,1994.

[2] D.Lee, A.N.Netravali, K.K.Sabnani, B.Sugla and A.John. Passive testing and applications to network management. In International Conference on Network Protocols, ICNP'97, pages 113-122. IEEE Computer Society Press, 1997.

[3] R.E.Miller and K.A.Arisha. Fault Identification in Networks by Passive Testing. In 34th Simulation Symposium, SS'01, pages 277-284. IEEE Computer Society Press, 2001.

[4] E.Bayse, A.Cavalli, M. Núñez and F.Zaidi. A passive testing approach based on invariants: application to the WAP. In Computer networks, vol. 48, no.2, pages 247-266, 2005.

[5] M. Tabourier and A.Cavalli. Passive testing and application to the GSM-MAP protocol. In Journal of Information and Software Technology 41(11), pages 813-821, Elsevier, 1999.

[6] F.Zaidi, A.Cavalli and E.Bayse. Network Protocol Interoperability Testing based on Contextual Signatures. The 24th Annual ACM Symposium on Applied Computing SAC'09, Hawaii, USA, March 9-12, 2009.

[7] R.E.Miller. Passive Testing of Networks Using a CFSM Specification. Proceedings of the IEEE International Performance, Computing and Communications Conferences, pages 111-116, February 1998.

[8] Y.Zhao, J.Wu and X.Yin. Online Test System, an Application of Passive Testing in Routing Protocols Test. Proceedings of the Ninth IEEE International Conference on Networks. ICON'01, pages 190-195 , 2001.

[9] T.Kato, T. Ogish, H. Shinbo, Y.Miyake, A.Idoue and K. Suzuki. Interoperability testing system of TCP/IP based system in operational environment. In Hasan Ural, Robert L. Probert, and Gregor von Bochmann, editors, TestCom, volume 176 of IFIP Conference Proceedings, page 143. Kluwer, 2000.

[10] L. Verhaard, J. Tretmans and P. Kars, Ed. Brinksma. On asynchronous testing. In Gregor von Bockmann, Rachida Dssouli, and Anindya Das, editors, Protocol Test Systems, volume C-11 of IFIP Transactions, pages 55-66. North-Holland, 1992.

[11] Z.Wang, J.Wu and X.Yin. Towards interoperability test generation of time dependant protocols: a case study. In Global Telecommunications Conference, GLOBECOM'04, Dallas, Texas, Etats-Unis, volume 2, pages 589-594, 2004.

[12] J.A. Arnedo, A. Cavalli and M.Núñez. Fast Testing of Critical Properties through Passive Testing. Lecture Notes on Computer Science, volume. 2644/2003, pages 295-310, Springer, 2003.

[13] R.S.Boyer and J.S.Moore. A fast string searching algorithm. In Communications of ACM 20, pages 762-772, 1977.

[14] E.Brinksma. A Theory for the Derivation of Tests. In S. Aggarwal and K.Sabnani, editors, Proceedings of the eighth international conference on protocol Protocol Specification, Testing and Verification, pages 63-74, North Holland, 1988.

[15] K.El-Fakih, V.Trenkaev, N.Spitsyna and N.Yevtushenko. FSM based interoperability testing methods for multi stimuli model. In roland Groz and Robert M. Hierons, editors, TestCom, volume 2978 of Lecture Notes in Computer Science, pages 60-65. Springer, 2004.

[16] O. Koné and R.Castanet. Test generation for interworking systems. In Computer Communications, Volume 23, Issue 7, pages 642-652, 2000.

[17] J.C.Fernandez, C.Jard, T.Jeron and C.Viho. Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur and Thomas A.Henzinger, editors, Proceedings of the Eighth International Conference on Computer Aided Verification CAV, volume 1102, pages 348-359, New Brunswick, NJ, USA/1996. Springer Verlag.

[18] S.Seol, M.Kim, S.Kang and S.T.Chanson. Interoperability test generation and minimization for communication protocols based on the multiple stimuli principle. IEEE Journal on selected areas in Communications, 22 (10), pages 2062-2074, december 2004.

[19] A.Desmoulin and C.Viho. Automatic Interoperability Test Case Generation Based on Formal Definitions. Lecture Notes in Computer Science, Volume 4916/2008, pages 234-250, 2008.

[20] Z. Shelby, K. Hartke and B. Frank. Constrained application protocol (CoAP), draft-ietf-core-coap-08, 2011.

[21] W. Colitti, K. Steenhaut, and N. De Caro. Integrating Wireless Sensor Networks with the Web, in Extending the Internet to Low power and Lossy Networks (IP+SN 2011), 2011.

[22] L. Atzori, A. Iera and G. Morabito. The Internet of Things: A survey, Comput. Netw., vol. 54, pages 2787–2805, 2010.

[23] K. Hartke. Observing Resources in CoAP, draft-ietf-core-observe-04, 2012.

[24] S.Schulz, A.Wiles and S.Randall. TPLan-A notation for expressing test purposes. ETSI, TestCom/FATES, LNCS 4581, pages.292-304, 2007.

[25] C. Bormann and Z. Shelby. Blockwise transfers in CoAP. draft-ietfcore- block-05, 2012.

[26] Z. Shelby. CoRE Link Format. draft-ietf-core-linkformat-09, 2011.

[27] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999.

[28] A.Sabiguero, A.Baire, A.Boutet and C.Viho. Virtualized Interoperability Testing: Application to IPv6 Network Mobility. 18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2007, Proceedings 01/2007, 2007.

[29] R.M.Fuhrer. Sequential Optimization of Asynchronous and Synchronous Finite-State Machines, Ph.D. thesis, Department of Computer Science, Columbia University, 1999.

[30] R.T.Fielding. Architectural Styles and the Design of Network-based Software Architectures, Doctoral dissertation, University of California, Irvine, 2000.

[31] W.Wong, J.Horgan, A.Mathur and A.Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. Proceedings of the 21th Annual International Computer Software and Application Conference, pages 522-528, 1997.

[32] G.Rothermel, M.J.Harrold, J.Ostrin and C.Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. International Conference on Software Maintenance, pages 34-43, 1998.

[33] D.Jeffrey and N. Gupta. Test suite reduction with selective redundancy. ICSM'05. Proceedings of the 21st IEEE International Conference on Software Maintenance, pages 549-558, 2005.

[34] M.P.E.Heimdahl and D.George. Test-suite reduction for model based tests: effects on test quality and implications for testing. 19th International Conference on Automated Software Engineering, pages 176-185, 2004

[35] O. Rafiq and R. Castanet. From conformance testing to interoperability testing, in: Proceedings of the 3rd International Workshop on Protocol Test System, 1990.

[36] N.Arakawa, M.Phalippou, N.Risser and T.Soneoka. Combination of conformance and interoperability testing, V (C-10), in: M. Diaz, R. Groz (Eds.), Formal Description Techniques, Elsevier, Amsterdam, 1993.

[37] J.Gadre, C.Rohrer, C.Summers and S.Symington, A COS Study of OIS Interoperability. Computer Standards & Interfaces, 9, 217-237, 1990.

[38] D.Lee and M.Yannakakis. Principles and methods of testing finite state machines - a survey. Proc. of the IEEE, vol.84, 1996.

[39] A.R. Cavalli, A.Benameur, W.Mallouli and K.Li, A Passive Testing Approach for Security Checking and its Practical Usage for Web Services Monitoring, invited paper, NOTERE 2009, 29-June 3-July, 2009, Montréal, Canada, 2009.

[40] R.E.Miller and K.A.Arisha. On fault location in networks by passive testing. Proc of IEEE IPCCC 2000, pages 281-287, 2000.

[41] J.P.Baconnet, C.Betteridge, G.Bonnes F.Van den Berghe and T.Hopkinson. Scoping further EWOS activity for interoperability testing. Technical report EGCT/96/130 R1, EWOS, 1996.

[42] T. Walter, I. Schieferdecker, and J. Grabowski. Test Architectures for Distributed Systems - State of the Art and Beyond. Testing of Communicating Systems, IFIP TC6 11th International Workshop on Testing Communicating Systems (IWTCS). Vol 131 of IFIP Conference Proceedings, pages 149-174, Kluwer, 1998.

[43] G.Bonnes. IBM OSI interoperability verification services. In IFIP TC6 WG6.1 The 3rd International Workshop on Protocol Test System, 1990.

[44] G.S.Vermeer and H.Blik. Interoperability testing: Basis for the acceptance of communicating systems. In Protocol Test Systems, VI(C-19). Elsevier Science PUblisher B.V, 1994.

[45] M.Handley, H.Shulzrinne, E.Schooler and J.Rosenberg: SIP: Session Initiation Protocol. Request For Comments (Proposed Standard) 2543, Internet Engineering Task Force, 1999.

[46] S.T.Eckmann, G.Vigna and R.A.Kemmerer: An Attack Language for State-based Intrusion Detection. In: JCS'02, 2002.

[47] U.Gasser and J.Palfrey. Breaking down digital barriers: when and how ICT interoperability drives innovation. 2007-2008.

[48] H.Van der Veer and A.Wiles. Achieving Technical Interoperability–the ETSI Approach. ETSI White Paper No. 3, Third edition, 2008.

[49] D.Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In Proceedings of the IEEE, volume 84, pages 1090 - 1126, 1996.

[50] A.Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan. Modeling and Verification of parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000, volume 2067 of Lecture Notes in Computer Science, pages 196-205. Springer, 2000.

[51] I. Schieferdecker, B. Stepien and A. Rennoch. PerfTTCN, a TTCN language extension for performance testing. Testing of Communicating Systems IFIP — The International Federation for Information Processing 1997, pages 21-36, 1997.

[52] ZR.Dai, J.Grabowski and H.Neukirchen. TimedTTCN-3–A Real-Time Extension for TTCN-3. Proceedings of the IFIP TC6/WG6 1 (14), pages. 407-424, 2002.

[53] A.Baire, C.Viho and N.Chen. Long-Term Challenges in TTCN-3: a Prototype to Explore New Features & Concepts. In T3UC Conference, 2012.

[54] JD.Day and H.Zimmermann. The OSI reference model. Proceedings of the IEEE, 1983.

[55] FJ.Lin, PM.Chu and MT.Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. In SIGCOMM '87 Proceedings of the ACM workshop on Frontiers in computer communications technology. pages 126-135, 1987.

[56] J.Grabowski, D.Hogrefe and G.Réthy, I.Schieferdecker. An introduction to the testing and test control notation (TTCN-3). Computer Networks, Vol. 42, Issue 3, pages 375–403, 2003.

[57] ETSI. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language

[58] ETSI whitepaper. After 14 Years of Manual Interoperability Testing, Finally the Process has been Automated. Teraquant Corporation. 2009.

[59] J.Shin and S.Kang. Interoperability test suite derivation for the ATM/B-ISDN signaling protocol. Testing of Communicating Systems, vol.11, Kluwer Academic Publishers, Dordrecht, pages 313-330, 1998.

[60] N.Griffeth, R.Hao, D.Lee and R.K.Sinha. Integrated system interoperability testing with applications to VoIP, IFIP TC6 WG6.1 Joint international conference on formal description techniques for distributed systems and communication protocols and protocol specification, testing and verification, Pisa, Italy, 2000.

[61] C.Viho, S.Barbin and L.Tanguy. Towards a Formal Framework for Interoperability Testing. Formal Techniques for Networked and Distributed Systems IFIP International Federation for Information Processing, vol. 69, pages 53-68, 2002.

[62] S. Kang and M. Kim. Interoperability test suite derivation for symmetric communication protocols. In FORTE/PSTV'97 , 1997.

Summary

This thesis focuses on interoperability testing, which aims at verifying that different network components cooperate correctly and provide expected services. Specifically, we argue for the use of the passive testing approaches, which is based only on observation. The non-intrusive nature of passive testing is suitable for interoperability, as it does not disturb the normal operation of the implementations under test (IUT). This feature makes it an appropriate solution for testing in operational environments. This thesis proposes a passive interoperability testing methodology has been proposed based on formal models. This method aims at checking a set of pre-selected test purposes against observed behaviors of interacting IUTs. An automatic test case generation algorithm and a trace validation algorithm were proposed. It also addresses the issue of verdict assignment. Furthermore, focusing on request-response protocols, which are a widely used category of protocols, another method has been worked out to carry out interoperability testing by using passive technique. An associated testing tool has also been proposed to automate the tests. The testing method and tool have been successfully applied in the ETSI Plugtests events on Constrained Application Protocol (CoAP) in the context of the Internet of Things. Results showing the interest and efficiency of the proposed solutions and associated tools are presented.
Keywords: Automatic test generation, CoAP, interoperability testing, IOLTS, passive testing,

Résumé

Cette thèse traite du test d'interopérabilité, qui vise à vérifier que les différents composants d'un réseau interagissent correctement et fournissent des services attendus. Plus précisément, on s'intéresse aux approches dites de test passif, qui sont basées uniquement sur l'observation. Le caractère non intrusif du test passif rend cette technique appropriée pour le test d'interopérabilité, car il ne perturbe pas le fonctionnement normal des implémentations sous test (IUT). Cela permet au test d'être effectué dans les environnements opérationnels. Dans cette thèse, nous proposons une méthode pour effectuer le test d'interopérabilité en utilisant la technique du test passif et basée sur les modèles formels. Elle vise à vérifier un ensemble d'objectifs de test présélectionnés sur la trace réelle observée des interactions des IUT. Un algorithme de génération de cas de test automatique ainsi qu'un algorithme de validation de trace sont également proposés. Nous traitons également le problème que pose de l'émission des verdicts. Pour les protocoles de requête-réponse, très largement utilisés actuellement, une autre méthode a été proposée pour réaliser des tests d'interopérabilité en utilisant la technique passive. Un outil de test associé a également été développé afin d'automatiser les tests. La méthode de test et l'outil ont été utilisés avec succès lors les événements plugtests de l'ETSI sur le protocole CoAP (Constrained Application Protocol) dans le contexte de l'Internet des objets. Les résultats montrant l'intérêt et l'efficacité des solutions proposées et des outils correspondants sont présentés.
Mots-clés: CoAP, test d'interopérabilité, test passif, IOLTS, génération automatique