Cross-fertilizing formal approaches for protocol conformance and performance testing

Xiaoping Che

INSTITUT MINES-TÉLÉCOM/TÉLÉCOM SUDPARIS

ÉCOLE DOCTORALE SCIENCES ET INGENIERIE
EN CO-ACCREDITATION AVEC L'UNIVERSITÉ ÉVRY VAL D'ESSONNE

THÈSE

Pour obtenir le grade de

DOCTEUR DE TÉLÉCOM SUDPARIS

Spécialité : Informatique

Présentée et soutenue par

Xiaoping CHE

Cross-Fertilizing Formal Approaches for Protocol Conformance and Performance Testing

Soutenue le 26 Juin 2014
Devant le jury composé de :

Directeur de thèse	Prof. Stéphane MAAG	--- Télécom SudParis
Rapporteurs	Prof. Mercedes MERAYO	--- Universidad Complutense de Madrid
	Prof. Joanna TOMASIK	--- Supélec
Examinateurs	Prof. Ana CAVALLI	--- Télécom SudParis
	Prof. Sylvain CONCHON	--- Université Paris Sud 11
	Ing. Emmanuel ALIBERT	--- IBM
	Dr. Frédéric DADEAU	--- Université de Besançon

Thèse n° <2014TELE0012>

INSTITUT TELECOM / TELECOM SUDPARIS

Abstract

Département Logiciels-Réseaux

Doctoral Degree

Cross-fertilizing formal approaches for Protocol Conformance and Performance Testing

by Xiaoping CHE

While today's communications are essential and a huge set of services is available online, computer networks continue to grow and novel communication protocols are continuously being defined and developed. De facto, protocol standards are required to allow different systems to interwork. Though these standards can be formally verified, the developers may produce some errors leading to faulty implementations. That is the reason why their implementations must be strictly tested.

However, most current testing approaches require a stimulation of the implementation under tests (IUT). If the system cannot be accessed or interrupted, the IUT will not be able to be tested. Besides, most of the existing works are based on formal models and quite few works study formalizing performance requirements. To solve these issues, we proposed a novel logic-based testing approach to test the protocol conformance and performance passively.

In our approach, conformance and performance requirements can be accurately formalized using the Horn-Logic based syntax and semantics. These formalized requirements are also tested through millions of messages collected from real communicating environments. The satisfying results returned from the experiments proved the functionality and efficiency of our approach. Also for satisfying the increasing needs in real-time distributed testing, we also proposed a distributed testing framework and an online testing framework, and performed the frameworks in a real small scale environment. The preliminary results are obtained with success. And also, applying our approach under billions of messages and optimizing the algorithm will be our future works.

Acknowledgements

I would like to thank my supervisor Professor Stephane Maag, for granting me the opportunity to work with him and the team, and all his support with the finalization of my studies. I also want to thank Professor Ana Cavalli, the director of our department, for all her support and her help during this period, for supporting me to training schools and listening to my ideas.

A big thank you also to the reviewers of my thesis Professor Mercedes Merayo and Professor Joanna Tomasik, for their comments and kind remarks, as well as the members of the jury, Professor Sylvain Conchon, Ing. Emmanuel Alibert and Dr. Frederic Dadeau, for taking the time of their schedules for reviewing my work.

On a more personal note, I would like to thank my wife, Yue WANG, for accompanying me in this adventure, with all the risks it implicated and for her support during all this time, particularly in the last year of my work, in all those late nights of writing.

I would also like to thank my family for all their support at the distance, the help they provide and the confidence they have always put in me. Also I have to thank my Chinese friends for their kindly help and support: Dingqi, Chao Chen, Xiao Han, Mingyue, Leye, Pei Li, Haixiang, Longbiao, Haoyi and others who accompanied me in these three years.

Finally, I would also like to thank the members of the LOR team, present and those that are already looking for new adventures: Felipe, Anderson, Pramila, Khalifa, Natalia, Denisa, Jeevan, Joao, Jorge, Olga, Raul, Kun, Diego and other colleagues worked with me.

Contents

List of Figures

List of Tables

Abbreviations

AS	**A**pplication **S**erver
AST	**A**bstract **S**yntax **T**ree
BNF	**B**ackus **N**aur **F**orm
CSCF	**C**all **S**ession **C**ontrol **F**unctions
EEFSM	**E**vent-based **E**xtended **F**inite **S**tate **M**achine
ESG	**E**vent **S**equence **G**raphs
HSS	**H**ome **S**ubscriber **S**ervice
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
IMS	**I**P **M**ultimedia **S**ystem
IoT	**I**nternet **o**f **T**hings
IUT	**I**mplementation **U**nder **T**est
JID	**J**abber **ID**entification
LAN	**L**ocal **A**rea **N**etwork
LDAP	**L**ightweight **D**irectory **A**ccess **P**rotocol
LTL	**L**ineartime **T**emporal **L**ogic
NTP	**N**etwork **T**ime **P**rotocol
P.O	**P**oint of **O**bservations
PAN	**P**ersonal **A**rea **N**etwork
POC	**P**ush to-talk **O**ver **C**ellular
PCO	**P**oint of **C**ontrol and **O**bservation
RFC	**R**equest **F**or **C**omments
RFID	**R**adio **F**requency **ID**entification
SEFSM	**S**implified **E**xtended **F**inite **S**tate **M**achine
SIP	**S**ession **I**nitiation **P**rotocol
SLD	**S**elective **L**inear **D**efinite-clause

SOAP	**S**imple **O**bject **A**ccess **P**rotocol
STGA	**S**ymbolic **T**ransition **G**raph with **A**ssignment
SVM	**S**upport **V**ector **M**achine
TLTL	**T**imed **L**ineartime **T**emporal **L**ogic
UAC	**U**ser **A**gent **C**lient
UAS	**U**ser **A**gent **S**erver
URI	**U**niform **R**esource **I**dentifier
WS	**W**eb **S**ervices
WSN	**W**ireless **S**ensor **N**etwork
XML	e**X**tensible **M**arkup **L**anguage
XMPP	e**X**tensible **M**essage **P**resence **P**rotocol

I would like to dedicate this thesis to the ones I love.

Chapter 1

Introduction

"Wonder is the beginning of wisdom."
– Socrates (469 BC – 399 BC)

1.1 General Context

While today's communications are essential and a huge set of services is available online, computer networks continue to grow and novel communication protocols are continuously being defined and developed. De facto, protocol standards are required to allow different systems to interwork. Though these standards can be formally verified, the developers may produce some errors leading to faulty implementations. That is the reason why their implementations must be strictly *tested* using appropriate testing approaches.

Testing is mainly known as the process of operating a system or component under specified conditions to observe the results and provide an evaluation of such system or component [1]. In the industry, many non-formal testing tools are still used for testing protocols. However, with the growing significance of protocols within new internet architectures, techniques that assist in the production of reliable protocol are becoming increasingly important. The use of formal testing approaches can eliminate ambiguity and thus reduce the chance of errors being introduced during protocol development. Two main types of formal approaches can be applied to test the communicating protocols: *Active* and *Passive* testing. While active testing techniques are based on the analysis of the protocol answers when it is stimulated, the passive ones focus on the observation of input and output events of the implementation under test (IUT) in run-time.

Active testing is based on the execution of specific test sequences against the implementation under test. As shown in Figure 1.1, the test sequences can be obtained from the

formal model according to different test coverage criteria. These criteria can be applied on the specification, e.g. coverage of all logical conditions, coverage of all paths. This allows us to set if we have covered the specification as well as the code in testing. The tests may be generated automatically or semi-automatically from test criteria, hypothesis, and test goals. When generating the tests, we are faced to the feasibility problem, the problem of deciding the feasibility of a path is undecidable. The format of these sequences which is commonly used by the testing community is TTCN3 [2], from which their execution are performed through Points of Control and Observation (PCOs) execution interfaces. These PCOs are installed in the context of a testing architecture, which means the way to put the testers (e.g. upper and lower testers to test a specific stack layer, the different interfaces, and the oracle in order to provide a verdict on the executed tests).

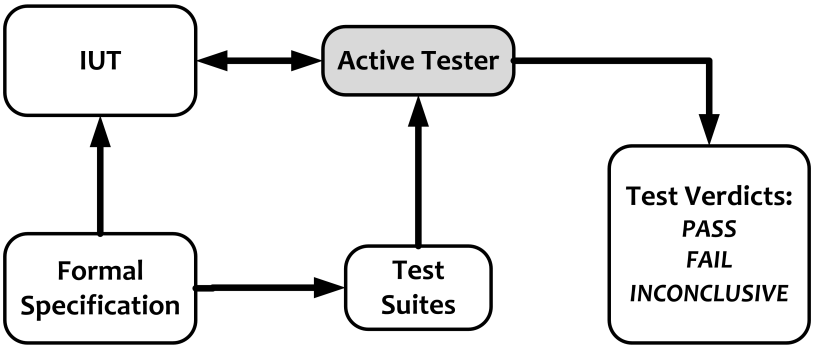


FIGURE 1.1: Active Testing

On the other hand, passive testing consists in observing the input and output events of an implementation under test in run-time. The term "passive" means that the tests do not disturb the natural run-time of a protocol, it is not intrusive as the implementation under test is not stimulated. This concept is sometimes also refereed to as monitoring in the literature. The record of the event observation is called an event, execution or log trace. In order to check the IUT, this trace will be compared to its expected behavior through the formal model or/and expected properties.

The passive testing techniques are applied especially because the active ones require important testing architectures, whose the testers need to control the system at some specific points. This is sometimes not feasible or even undesired. Nevertheless, while test sequences in active testing may give concrete verdicts (except for "inconclusive" ones), an event trace that satisfies the model does not mean that the whole implementation satisfies the specification. On the other hand, if a trace does not satisfy, then neither does the implementation. Passive testing may also successfully be used when the implementation cannot be shutdown or stopped for a long period of time.

Passive testing is often confused with run-time verification, they are both aiming to observe or monitor a run of the system and attempt to determine the satisfaction of a given correctness property [3]. Nevertheless, while passive testing has the specific purpose of delivering a verdict about the conformance of a black-box implementation, run-time verification deals with the more general aspects of property evaluation and monitor generation, without necessarily attempting to provide a verdict about the system.

Passive and active testing have their own advantages and drawbacks, the results that may be obtained depend on the system under test and essentially on the testing goals, the testing type. The testing type considers the whole testing process of the protocol, which consists of different steps: unit, conformance, interoperability, performance testing, and so on. Most of these testing types are normalized. For instance the active conformance testing is standardized by the ISO [4] in which common testing architectures, interfaces or points of control and observation are mentioned and specified. Nevertheless, these standards are mainly designed for wired systems and most of the time, the new inherent constraints of protocols in complex networks are omitted from these documents. In our work, we concentrate on the non-normalized passive conformance testing and performance testing.

1.2 Motivations and Contributions

Although passive testing does lack some of the advantages of active techniques, such as test coverage, it provides an effective tool for fault detection when the access to the interfaces of the system is unavailable, or in already deployed systems, where the system cannot be interrupted. In order to check conformance of the IUT, the record of the observation during runtime (called *trace*) is compared with the expected behavior, defined by either a formal model (when available) or as a set of formally specified properties [5] obtained from the requirements of the protocol.

In the context of black-box testing of communicating protocols, executions of the system are limited to communication traces, i.e. inputs and outputs to and from the IUT. Since passive testing approaches derive from model-based methodologies [6], such input/output events are usually modeled as: a *control* part, an identifier for the event belonging to a finite set, and a *data* part, a set of parameters accompanying the control part. In these disciplines, properties are generally described as relations between control parts, where a direct causality between inputs and outputs is expected (as in finite state-based methodologies) or a temporal relation is required. In modern message-based protocols (e.g. Session Initiation Protocol [7]), while the control part still plays an important role, data is essential for the execution flow. Input/output causality cannot be assured since

many outputs may be expected for a single input. Moreover when traces are captured on centralized services, many equivalent messages can be observed due to interactions with multiple clients. That is why temporal relations cannot be established solely through control parts. Furthermore, although the traces are finite, the number of related packets may become huge and the properties to be verified complex. In some conformance testing works, the researchers try to tackle these problems [8–11]. However, they still have some problems, such as based on models, language is still propositional in nature, data relation between multiple packets is not allowed.

For solving these issues, inspired from these previous works, we firstly present a passive conformance testing approach for communicating protocols based on the formalized functional requirements. We also add flexibility to the definition of formulas by considering data as the central part of communications, and our contributions are listed below.

- A Horn based logic is defined to specify the properties to be verified by taking into consideration the data values.
- The syntax and a three-valued semantics are provided, to define satisfaction within the truth values {*true, false, inconclusive*}, respectively indicating that the property is satisfied on the trace, not satisfied and no conclusion can be provided.

Moreover, in the literature, many performance related properties (e.g. package latency, loss rate, etc.) cannot be formalized, which raises our interests on accommodating our formalism for testing the non-functional requirements. In most of the protocol testing processes, performance testing is applied separately to the conformance testing. It is mainly applied to validate or verify the scalability and reliability of the system. Many benefits can be brought to the test process if both conformance and performance testing inherit from the same approach. Our main objective is then to adapt our conformance approach to performance testing. Also note that our work concentrates on performance testing, not on performance evaluation. Performance evaluation of network protocols focuses on the evaluation of its performance, while performance testing approaches aim at testing performance requirements that are expected in the protocol standard.

Generally, performance testing characteristics are: volume, throughput and latency [12], where volume represents "total number of transactions being tested", throughput represents "transactions per second the application can handle" and latency represents "remote response time". But for comprehensively testing the performance of a protocol, more performance requirements need to be formalized and tested. In this work, we aim at formalizing these time related requirements. Meanwhile, we introduce a four-valued

semantics {'*Pass*', '*Con-Fail*', '*Per-Fail*', '*Inconclusive*'} in our formalism, in order to solve the indeterminacy problems existed in non-positive verdicts. Based on these above mentioned challenges, our main contributions are:

- The definition of an extended language syntax and semantics, which provide the possibility to formalize and test performance requirements by taking into consideration the data values.
- A common ground for both conformance testing and performance testing, and a four-valued semantics for accurately determining non-positive verdicts.
- A proposal of detailed customized benchmark system for testing the performance of Session Initiation Protocol.
- A distributed framework is designed for testing Session Initiation Protocol and Extensible Messaging and Presence Protocol.

Among the well known and commonly applied approaches, the *passive* testing techniques are divided in two main groups: *online* and *offline* testing approaches. Offline testing computes test scenarios before their execution on the IUT and gives verdicts afterwards, while online testing provides continuously testing during the operation phase of the IUT.

With online testing approaches, the collection of traces is avoided and the traces are eventually not finite. Indeed, testing a protocol at run-time may be performed during a normal use of the system without disturbing the process. Several online testing techniques have been studied by the community in order to test systems or protocol implementations [10, 13, 14]. These methods provide interesting studies and have their own advantages, but they also have several drawbacks such as the presence of false negatives, space and time consumption and often related to a needed complete formal model. Although they bring solutions, new results and perspectives to the protocol and system testers, they also raise new challenges and issues. The main ones are the non-collection of traces and their on-the-fly analysis. The traces are observed (through an interface and an eventual sniffer) and analyzed on-the-fly to provide test verdicts and no trace sets should be studied a posteriori to the testing process. And the processes of some approaches are still offline with finite traces that are considered as very long

Due to these issues, we herein extend our previous proposed methodology and we develop our approach to test conformance and performance of protocols in an online way in considering the above mentioned inherent constraints and challenges. Furthermore, our framework is designed to test them at runtime, with new required verdicts definitions of '*Time-Fail*', '*Data-Inc*' and '*Inconclusive*' representing unobserved message

within timeout, untested data portion and uncertain status respectively. In order to demonstrate the efficiency of our online approach, we apply it on a real communicating environment for assessing its preciseness and efficiency.

- We provide a formal online passive testing approach to avoid stopping the execution of the testing process when monitoring a tested protocol. The analyzed traces are never cut which improves the accuracy of the test verdicts.
- Our approach allows the testing process to be executed in a transparent way without overloading, overcharging the CPU and memory of the used equipment on which the tester will be run.
- Data portion of the messages is taken into account in our online testing approach, and new definitions of online testing verdicts are introduced.

1.3 Thesis plan

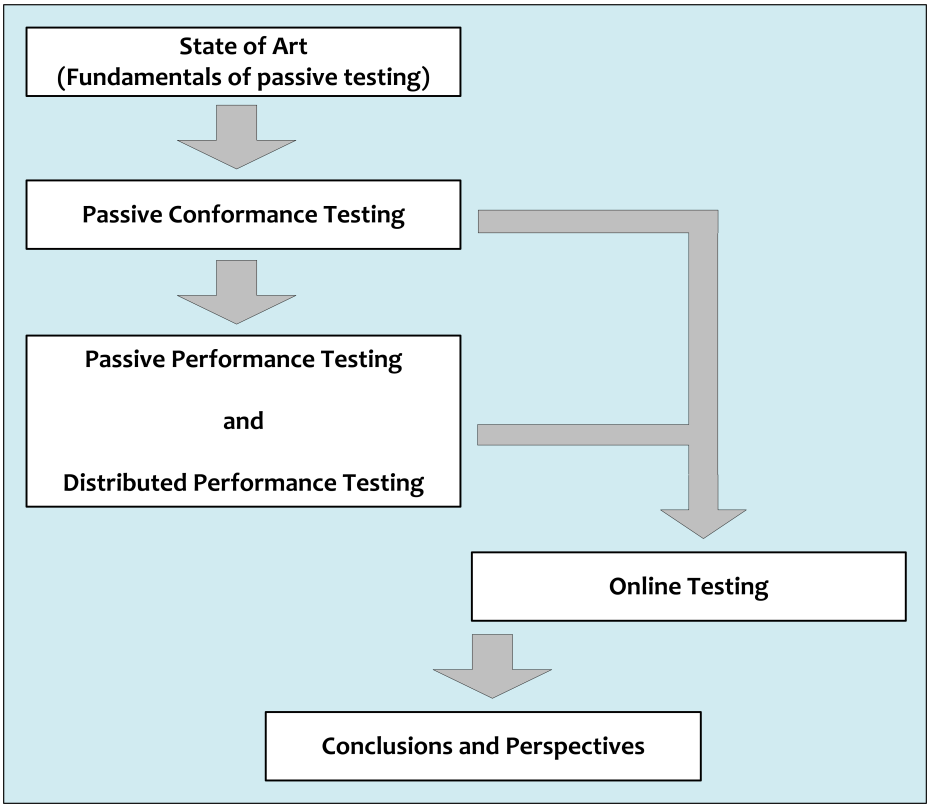


FIGURE 1.2: Chapters overview

As shown in Figure 1.2, the manuscript is organized as follows:

- In the second chapter, we present a state of the art of the passive conformance, performance testing techniques. We go from the general concepts of conformance and performance, to a brief overview of existing techniques for testing conformance and performance. We also provide some relevant works to online testing area.

- In the third chapter, we present our approach for conformance testing, through a real communicating environment. We also provide an overview of SIP, its entities and some of their behavior, along with the message syntax and relevant data carried by SIP messages. The experiment serves to describe the premier results and limitations of our approach, and it provides some motivation and inspiration for the further work.

- The fourth chapter contains our main contribution. We first detail on the modified syntax and semantics of formulas for satisfying the needs in performance testing. Then we describe the algorithm for evaluation of formulas in traces and we provide the relevant experiments results tested in a complex network environment. After, we design a distributed testing framework and verify our approach on another protocol, XMPP in IoT environment. Also, a brief description of XMPP with message syntax is provided. Finally, we provide the relevant experiments results and the motivations of further work.

- In the fifth chapter, we present an ongoing work on online testing. We describe our online testing architecture with new verdicts we defined. Then, we present a premier experimental result in a real case study.

- Finally, in the final chapter, we conclude the presentation of our work, provide a summary of our contributions and make some perspectives for future works.

Chapter 2

State of the Art

"We can be knowledgeable with other men's knowledge,
but we cannot be wise with other men's wisdom."
– Michel de Montaigne (1533 – 1592)

Testing is the process of operating a system or component under specified conditions to observe the results and provide an evaluation of such system or component [1]. Many types of testing exist, depending on the property being evaluated, for instance, it is possible to test for conformance, performance, usability, scalability, etc. Testing for conformance and performance is the main concern of our current work. Two main types of formal approaches can be applied to test the conformance and performance of communicating protocols: *Active* and *Passive* testing. While active testing techniques are based on the analysis of the protocol answers when it is stimulated, the passive ones focus on the observation of input and output events of the implementation under test (IUT) in run-time. In our work, we focus on passive testing techniques, these different types of passive testing approaches will be detailed in the following sections.

2.1 Overview of Passive Testing

Passive Testing consists in observing the input and output events of an implementation under test in run-time. The term "passive" means that the tests do not disturb the natural run-time of a protocol, it is not intrusive as the implementation under test is not stimulated. This concept is sometimes also refereed to as monitoring in the literature. The record of the event observation is called an event, execution or log trace. In order to check the conformance of the IUT, this trace will be compared to its expected behavior

through the formal model or/and expected functional properties, as shown in Figure 2.1.

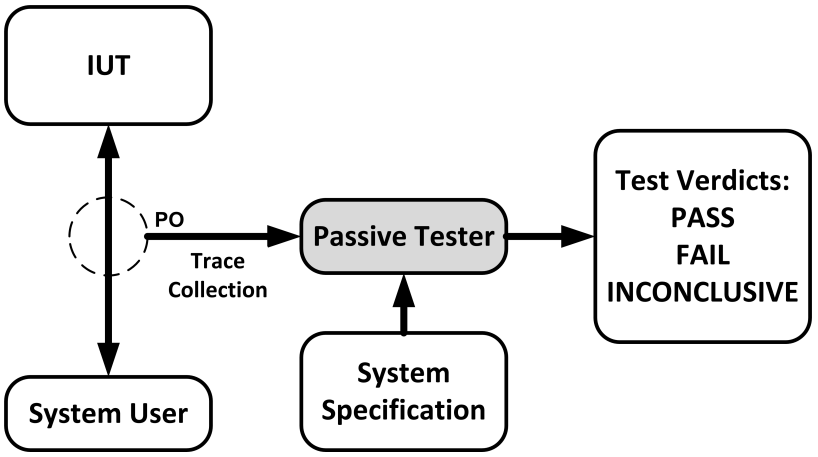


FIGURE 2.1: Passive Testing

The passive testing techniques are applied especially because the active ones require important testing architectures, whose the testers need to control the system at some specific points. This is sometimes not feasible or even undesired. Nevertheless, while test sequences in active testing may give concrete verdicts (except for "inconclusive" ones), an event trace that satisfies the model does not mean that the whole implementation satisfies the specification. On the other hand, if a trace does not satisfy, then neither does the implementation. Passive testing may also successfully be used when the implementation cannot be shutdown or stopped for a long period of time.

2.1.1 Runtime Verification

A number of different approaches for verifying formulas in traces exist in the literature for passive testing. However, there is another similar researching domain we need to mention: runtime verification. Although the verification and testing communities have usually dealt with different issues and through different methodologies, in the last couple of decades there has been increased work in using verification techniques for testing [15].

Runtime verification is a discipline, derived from model checking, that deals with the study, development and application of verification techniques that allow checking whether a run of a system under evaluation satisfies or violates a given correctness property [16]. It is portrayed in the literature as a lightweight verification technique, dealing only with the aspects of the system that can be evaluated during runtime, in opposition to traditional verification, which deals with all possible runs of a system.

In general terms, the methodology for runtime verification is the following: the system or implementation is assumed to behave as some model, M, some part of which is available during runtime. As with model checking, satisfiability of a given correctness property ϕ, must be determined on the runtime observations of the visible part of the model.

Determining the satisfaction of a correctness property, involves the creation of a *monitor*. The *monitor* incrementally reads the trace of the system and yields a *verdict*, usually in the form of a truth value in some range (e.g. $\{true, false\}$ or a probability [0,1]). A big part of works in this area deals with the generation of monitors for different types of properties and systems. An overview of different works in this area is provided by the authors of [17].

Depending on the context, runtime verification could be a revival of passive testing [18]. Runtime monitoring can achieve what passive testing can do, but the theoretical framework would be unnecessarily involved and it would be more difficult for classical testers. In the following, we describe works in both categories in relation with their ability to express data relations for defining properties.

2.1.2 Passive testing works

Formal testing methods have been used for years to prove correctness of implementations by combining test cases evaluation with proofs of critical properties. In [6, 19] the authors present a description of the state of the art and theory behind these techniques. Within this domain, and in particular for network protocols, passive testing techniques have to be used to test already deployed platforms or when direct access to the interfaces is not available. Some examples of these techniques using Finite State Machine derivations are described in [20, 21]. Most of these techniques consider only control portions, in [10, 11], data portion testing is approached by evaluation of traces in EEFSM (Event-based Extended Finite State Machine) and SEFSM (Simplified Extended Finite State Machine) models, testing correctness in the specification states and internal variable values. Our approach, although inspired by it, is different in the sense that we test critical properties directly on the trace without any models of the tested protocol. A study of the application of invariant to an IMS service was also presented in [19, 22].

In recent work, the authors of [9] defined a methodology for the definition and testing of time extended invariants, where data is also a fundamental principle in the definition of formulas and a *packet* (similar to a *message* in our work) is the base container data. In this approach, the satisfaction of the packets to certain *events* is evaluated, and properties are expressed as $e_1 \xrightarrow{When,n,t} e_2$, where e_1 and e_2 are events defined as a set of constraints on the data fields of packets, n is the number of packets where the event

e_2 should be expected to occur after finding e_1 in the trace, and t is the amount of time where event e_2 should be found on the trace after (or before) event e_1. This work served as an inspiration for the approaches described in the thesis.

Although closer to runtime monitoring, the authors of [23] propose a framework for defining and testing security properties on Web Services using the Nomad [24] language, based on previous works by the authors of [25, 26]. As a work on web services, data passed to the operations of the service is taken into account for the definition of properties, and multiple events in the trace can be compared, allowing to define, for instance, properties such as "Operation *op* can only be called between operations *login* and *logout*". Nevertheless, in web services, operations are atomic, that is, the invocation of each operation can be clearly followed in the trace, which is not the case with network protocols where operations depend on many messages and most of the time on the data associated with the messages.

2.1.3 Runtime monitoring works

Runtime monitoring and runtime verification techniques have gained momentum in the latest years, particularly using model checking techniques for testing properties on the trace. The authors of [16] provide a good survey and introduction of methodologies in this area. The usual approach, consists on the definition of some logic (LTL is commonly used), which is used to create properties from which a *monitor* is defined to test on the trace. The authors of [3] describe the definition of monitors as finite state machines for LTL formulas, they introduce a 3-valued semantics (true, false, inconclusive) in order to test formulas for finite segments of the trace[1], in [27] they expand their analysis on inconclusive results, by proposing a 4-value semantics to distinguish cases where the property is most likely to become true or become false on the continuation of the trace.

Regarding the inclusion of data, the concept of *parameterized propositions* is introduced by the authors of [8]. Propositions can contain data variables and quantifiers can be defined for the data variables by the introduction of a $\rightarrow$ operator, formulas of type $Q_1 x_1 \cdots Q_m x_m : p(x_1, \ldots, x_n) \rightarrow \psi$, where $Q_1, \ldots, Q_m$ are quantifiers and $x_1, \ldots, x_m, \ldots, x_n$ are variables. In this approach, valid data values in formulas are fixed, so if $p(x)$ is used on the left side, the set $\{p(1), p(2), \ldots\}$ with valid values must have been defined previously.

Another work, defined to test message based work-flows, is provided in [28] by the definition of the logic LTL-FO$^+$. Here, data is a more central part of the definition of

[1] In their work, a trace segment is considered a finite word with an infinite continuation, so formulas that deal with the future of the trace have to take into account that the property can become true (or false) on the continuation of the trace.

formulas and LTL temporal operators are used to indicate temporal relations between messages in the trace. Messages are defined as a set of pairs $(label, value)$, similarly to our work, and formulas are defined with quantifiers specific to the labels. As an example, the formula $\mathbf{G}(\exists_{method} x_1 : x_1 = \texttt{INVITE} \rightarrow \exists_{callId} x_2 : \mathbf{F}(\exists_{status} y_1 : y_1 = 200 \wedge \exists_{callId} y_2 : y_2 = x_2))$ indicates that generally, if a message with method `INVITE` is found, then there exists a field `Call-ID` in that message, such that a future message with status 200 exists with the same `Call-ID`. Although the syntax of the logic is flexible, it can quickly lose clarity as the number of variables required increases. The authors improved their work in [29] by separating the extraction of event data from the monitored system and from the property. Though this approach claims to be efficient, the current works presented in this thesis are not constrained to any extractions while the constraints are grouped with clause definitions.

In [30], the authors present a Java-based tool-supported software development and analysis framework: Monitoring-oriented programming (MOP), where monitoring is a foundational principle. MOP users can add their favorite or domain-specific requirements specification formalism into the framework by means of logic plug-ins, which essentially comprise monitor synthesis algorithms for properties expressed as formulas. The properties are specified together with declarations stating where and how to automatically integrate the corresponding monitor into the system, as well as what to do if the property is violated or validated. Based upon a carefully designed specification schema and upon several logic plug-ins, Java-MOP allows users to specify and monitor properties which can refer not only to the current program state, but also to the entire execution trace of a program, including past and future behaviors. However, from their paper, it is unclear that concrete verdicts will be given to each property after the evaluation. And the expressiveness of their approach is still based on the logic plug-ins they choose.

In [31], the authors propose a logic for runtime monitoring of programs, called EAGLE, that uses the recursive relation from LTL $\mathbf{F}\phi \equiv \phi \vee \mathbf{X}\phi$ (and its analogous for the past), to define a logic based only on the operators *next* (represented by $\bigcirc$) and *previous* (represented by $\odot$). Formulas are defined recursively and can be used to define other formulas. Constraint on the data variables and time constraints can also be tested by their framework. However, their logic is propositional in nature and their representation of data is aimed at characterizing variables and variable expressions in programs, which makes it less than ideal for testing message exchanges in a network protocol as required in our work. We have however to mention that this approach has been studied by the community and several rule-based systems have been implemented. We have thus to mention [32], a recent publication denoting that an efficient runtime verification implementation is now used to process telemetry from the Mars Curiosity rover at NASA's Jet

Propulsion Laboratory. But no results demonstrating these assessments are currently available.

Finally, some differences and similarities can be pointed between the concepts of runtime verification and the objectives of passive testing described before. In general terms, the application of runtime verification techniques can be considered as a form of testing, in particular, since the behavior of the system is being evaluated against some correctness property. While testing techniques have as objective to provide an evaluation of the system with respect to its requirements, runtime verification in general deals with the technical aspects of evaluation of properties on particular executions and generation of monitors, without necessarily attempting to provide a specific verdict on the system.

2.2 Passive Conformance Testing

The obtained passive testing results may depend on the system under test and essentially on the testing types. The testing type considers the whole testing process of the protocol, which consists in different steps: unit, conformance, interoperability, performance testing, and so on [33]. In our work, we mainly focus on passive conformance testing and performance testing.

Conformance testing of communicating protocols is a functional test which verifies whether the behaviors of the protocol satisfy the defined requirements. In the passive testing techniques, passive conformance testing is one of the crucial way to verify the IUT's functionality. A general definition of conformance is provided in [34], and we briefly introduce it here.

Conformance relates to specifications and implementations. The universe of specifications is defined by the set $SPECS$ and the universe of all IUTs is denoted by $IMPS$. Considering this, conformance is defined as the relation:

$$\textbf{conforms-to} \subseteq IMPS \times SPECS$$

where given an IUT $IUT \in IMPS$ and a specification $S \in SPECS$, IUT **conforms-to** S expresses that IUT is a correct implementation of the specification S.

In order to relate real implementation with specifications, the assumption is made that every IUT ($IUT \in IMPS$) can be modeled by a formal object $I_{IUT} \in MODS$, where $MODS$ is the universe of formal objects. This assumption is known as a *test hypothesis* [35]. Under this assumption, an implementation relation is defined between models and specifications as $imp \subseteq MODS \times SPECS$. An implementation

$IUT \in IMPS$ is said to conform to a specification $S \in SPECS$ if and only if the model of the implementation $I_{IUT} \in MODS$ is implementation related with S

$$IUT \textbf{ conforms-to } S \Leftrightarrow I_{IUT} \textbf{ imp } S$$

Since in passive conformance testing, the implementation is tested as a black box, the strongest conformance relation that can be tested is trace equivalence: two traces are equivalent if they cannot be distinguished by any sequence of inputs. In other words, both implementation and specification will generate the same outputs ("*trace*") for all specified input sequences. To prove trace equivalence it suffices to show that there is a set of implementation states $\{p_1, p_2, ..., p_n\}$ respectively isomorphic to specification states $\{s_1, s_2, ..., s_n\}$, and every transition in the specification has a corresponding isomorphic transition in the implementation. Formal methods for conformance testing have been used for years to prove correctness of implementations by combining test cases evaluation with proofs of critical properties. In [6] the authors present a description of the state of the art and theory behind these techniques. Passive conformance testing techniques are used to test already deployed platforms or when direct access to the interfaces is not available.

In [5], an invariant approach taking into account control parts has been presented. They introduced a new methodology and a relevant tool TESTINV for passive testing. This methodology includes the definition of a novel concept of invariant as well as a corresponding test architecture to deal with them. Two types of invariant have been defined: simple and obligation invariants. They can be used to express a wide range of properties. Another interesting work is in [36]. Since in passive testing, the tester does not interact with the implementation under test, and execution traces are observed without interfering with the behavior of the system. Invariants are used to represent the most relevant expected properties of the implementation under test. The authors of [36] give two algorithms to decide the correctness of proposed invariants with respect to a given specification and algorithms to check the correctness of a log, recorded from the implementation under test, with respect to an invariant. Based on the algorithms, they develop a tool called PASTE, which take advantage of mutation testing techniques in order to evaluate the goodness of an invariant according to its capability to detect errors in logs generated from mutants. These researchers did excellent works and embedded innovative passive testing algorithms in their approaches. However, in their approaches, the causality between the data portions in a trace is not considered.

Also, most of the existed techniques only consider control portions [10] [11], data portion testing is approached by evaluation of traces in state based models, testing correctness in the specification states and internal variable values. In [9], the authors have defined

a methodology for the definition and testing of time extended invariants, where data is also a fundamental principle in the definition of formulas and a packet (similar to a message in our work) is the base container data. In this approach, the packet satisfaction to certain events is evaluated. However, data relation between multiple packets is not allowed.

Our approach, although inspired by all the mentioned works, is different in the sense that we test critical properties directly on the trace, and only consider a model (if available) for potential verification of the properties. Our research is also inspired from the run-time monitoring domain. Though run-time monitoring techniques are mainly based on model checking while we do not manipulate any models, some proposed languages to describe properties are relevant for our purpose. The authors of [16] provide a good survey in this area.

There are also some interesting related works. The authors of [23] propose a framework for defining and testing security properties on Web Services using the Nomad [24] language. As a work on Web services, data passed to the operations of the service is taken into account for the definition of properties, and multiple events in the trace can be compared, allowing to define, for instance, properties such as "Operation op can only be called between operations login and logout". Nevertheless, in Web services operations are atomic, that is, the invocation of each operation can be clearly followed in the trace, which is not the case with network protocols, where operations depend on many messages and sometimes on the data associated with the messages. And in [31], the authors propose a logic for run-time monitoring of programs, called **EAGLE**, that uses the recursive relation from LTL, to define a logic based only on the operators *next* and *previous*. Since we already described in above, we will not repeat their works here.

2.3 Passive Performance Testing

Performance testing of communicating protocols is a qualitative and quantitative test, aiming at checking whether the performance requirements of the protocol have been satisfied under certain conditions. As shown in Figure 2.2, the test process in testing defined by [37] illustrates that the performance testing normally comes after the black and white box testing. In the testing process, developers and test engineers use black-box test methods to check incorrect and incomplete functions and behaviors based on the given specifications, and they use white-box test methods to uncover the internal errors in program logic and structure, data objects, and data structure. When these steps are ended, test engineers will exercise various component usage patterns through component interfaces to confirm that the correct functions and behaviors are delivered

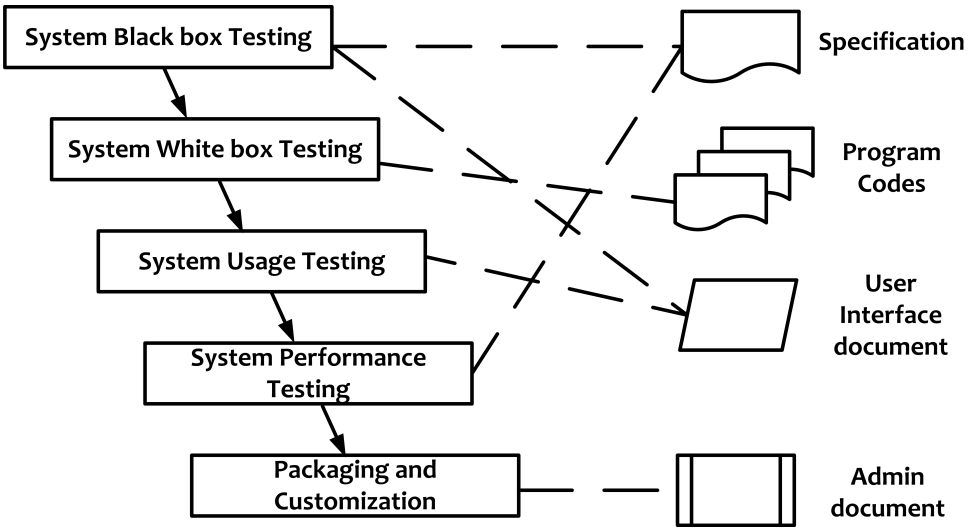


FIGURE 2.2: Test Process in Testing

through the given contract-based interfaces. After this step, test engineers and quality assurance staff will validate and test the performance of the system.

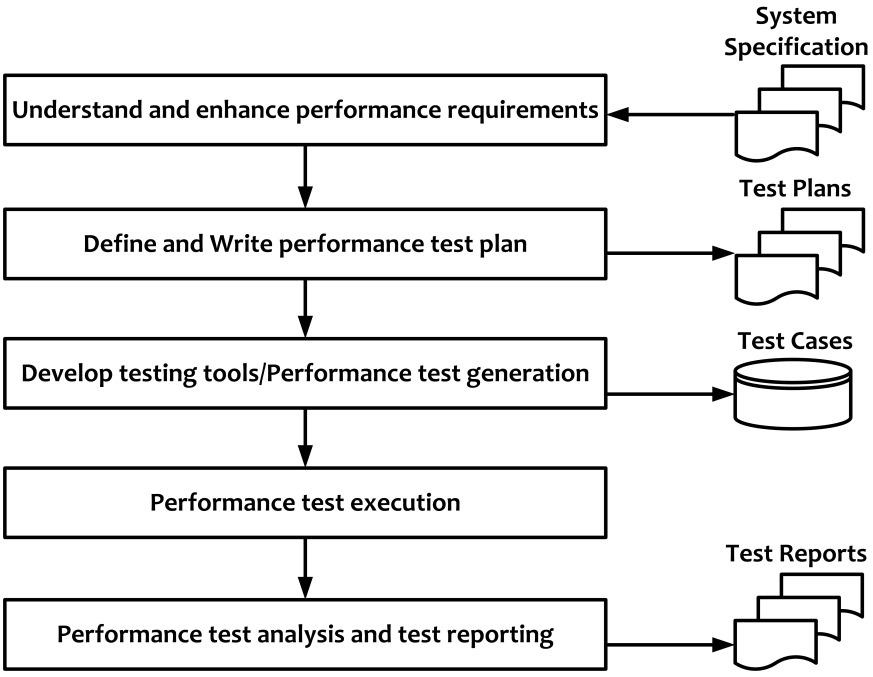


FIGURE 2.3: Performance Testing Process

The process of testing the performance of the system is shown in Figure 2.3. As shown in the figure, before carrying out any performance testing and evaluation activities, test engineers must firstly understand the system performance requirements. This is a crucial research point in this thesis, we try to provide an efficient approach for testers to formally define and test performance requirements. In many cases, the performance requirements are not clearly specified. Hence, performance engineers and testers need

to review system specification documents to identify, check, define, and enhance system performance requirements. And then, according to the requirements, they will define and write test plans. Based on the test plans, test cases and testing tools are developed and generated. Finally, performance tests are executed and performance analysis will be returned.

In the literature, many studies have investigated the performance of systems. However, few works on formally modeling requirements for performance testing have been studied, we can nevertheless cite the following ones.

A method for testing the functional behavior and the performance of programs in distributed systems is presented in [38]. In the paper, the authors discuss event-driven monitoring and event-based modeling. They use hybrid monitoring, a technique which combines advantages of both software monitoring and hardware monitoring. As an application of our monitoring and evaluation system, they described the analysis of a parallel ray tracing program running on the SUPRENUM multiprocessor. It is shown that monitoring and modeling both rely on a common abstraction of a system's dynamic behavior and therefore can be integrated to one comprehensive methodology. However, no evaluation of the methodology has been performed, but they provide a pioneer work on performance testing.

Later, in [39], performance testing is defined with performance requirements. In this book, performance tests are designed to validate performance requirements which are expressed either as time intervals in which the SUT must accomplish a given task, as performance throughput, volume or resource utilization. All the basic performance issues are well explained. Further, in [37], the authors provide a more accurate definition of performance testing. They define it as the activity to validate the system performance and measure the system capacity. They also define three major goals: validate the system ability to satisfy the performance requirements, find information about the capacity and boundary limits and assist the system designers and developers in finding performance issues, bottlenecks and improve the performance of the system. Their works provide prospective definitions of performance testing. Also in [40], the author presents a framework to perform passive testing for systems where time aspects affect their behavior. She raises the point that temporal aspects can be associated with both performance of actions and delays/timeouts. Inspired and based on their works, we define a formal passive formalism for formalizing the performance requirements and to provide a novel performance testing approach.

Since the formalism used in performance testing is the crucial part of our work, before introducing other related works on performance testing, we will briefly introduce some important similar related works on Temporal Logic and explain the difference between

our formalism. The authors of [16] introduce the methodologies in this area. The usual approach consists on the definition of some logic (such as LTL, TLTL, etc.), which is used to create properties from which a monitor is defined to test on the trace. The authors of [3] describe the definition of monitors as finite state machines for LTL formulas. They introduce a 3-valued semantics (true, false, inconclusive) in order to test formulas for finite segments of the trace. Moreover, in [27], they expand the analysis on inconclusive results by proposing a 4-value semantics. Although there are interesting approaches to data testing, they are still propositional in nature.

Furthermore, another similar work is provided by the authors of [41]. They present an algorithm for the run-time monitoring of data-aware workflow constraints. Sample properties taken from run-time monitoring scenarios in existing literature are expressed using LTL-FO^+, an extension of Linear Temporal Logic that includes first-order quantification over message contents. Similarly to our work, data are a more central part of the definition of formulas, and formulas are defined with quantifiers specific to the labels. Although the syntax of the logic they used is flexible, it can quickly lose clarity as the number of variables required increases.

After reviewing of similar works in temporal logic area, we return to the related works in performance testing. In [42], the authors present a performance monitoring tool for clusters of PCs which is based on the simple concept of accounting for resource usage and on the simple idea of mapping all performance related states. They identify several interesting implementations related to the collection of performance data on clusters of PCs and show how a performance monitoring tool can efficiently deal with all incurring problems. Besides, in [43], the authors present a distributed performance testing framework, which aimed at simplifying and automating service performance testing. They applied Diperf to two GT3.2 job submission services, and several metrics are tested, such as Service response time, Service throughput, Offered load, Service utilization and Service fairness. Similarly, in [10], the authors study network protocol system monitoring for fault detection using extended finite state machines, and in paper [44], the authors describe a CONCEPTUAL language which provides primitives for a wide variety of idioms needed for performance testing and emphasizes a readable syntax and a CONCEPTUAL compiler's novel code-generation framework. Although these techniques are interesting, they require the complete specification of the tested system.

Recently, there are also some interesting related works in complex event processing domain. In [45], the authors present the design, implementation, and evaluation of a system that executes complex event queries over real-time streams of RFID readings encoded as events. These complex event queries filter and correlate events to match specific patterns, and transform the relevant events into new composite events for the use of

external monitoring applications. They propose a complex event language that extends existing event languages to meet the needs of a range of RFID-enabled monitoring applications. Then they describe a query plan-based approach to efficiently implementing this language. Their approach uses native operators to efficiently handle query-defined sequences and pipeline such sequences to subsequent operators that are built by leveraging relational techniques. Some other researchers also have the same interest. The authors of [46] present a framework for complex event processing systems. It has five relevant characteristics: flexible, independent of particular workloads, neutral, correctness check and scalable. Their framework can help identify good design decisions and assist in improving engines. Likewise, in [47], the authors take an event-oriented approach to process RFID data, by devising RFID application logic into complex events. Then they formalize the specification and semantics of RFID events and rules. They discover that RFID events are highly temporal constrained, and include non-spontaneous events, and develop an RFID event detection engine that can effectively process complex RFID events.

Another work presents an adaptive performance testing method for stress testing web software systems by modeling the system with a two layers Queuing Network Model in [48]. In addition, a new measurement domain-specific language with specialized constructs concerning the automation of measurement procedures are proposed in [49]. The authors present a monitoring algorithm SMon in [50], which continuously reduces network diameter in real time in a distributed manner. Through simulations and experimental measurements, SMon achieves low monitoring delay, network tree, and protocol overhead for distributed applications. However, most of these approaches do not provide a formalism to test a specific performance requirement.

Although some works have been done in the related area. Inspired from and based on all these works, our work is different from focusing on using model-driven techniques, evaluating the performance of the system. We concentrate on how to formally and passively test the conformance and performance requirements written in the standard.

2.4 Online Testing

As we mentioned and introduced in previous sections, the passive testing techniques are today gaining efficiency and reliability. These techniques are also divided in two main groups: *online* and *offline* testing approaches. In offline testing the evaluation of the system is done in recorded traces, while in online testing, the tester attempts to detect faults during the execution of the system. In other words, online testing provides continuously testing during the operation phase of the IUT. With online testing

approaches, the collection of traces is avoided and the traces are eventually not finite. Indeed, testing a protocol at run-time may be performed during a normal use of the system without disturbing the process.

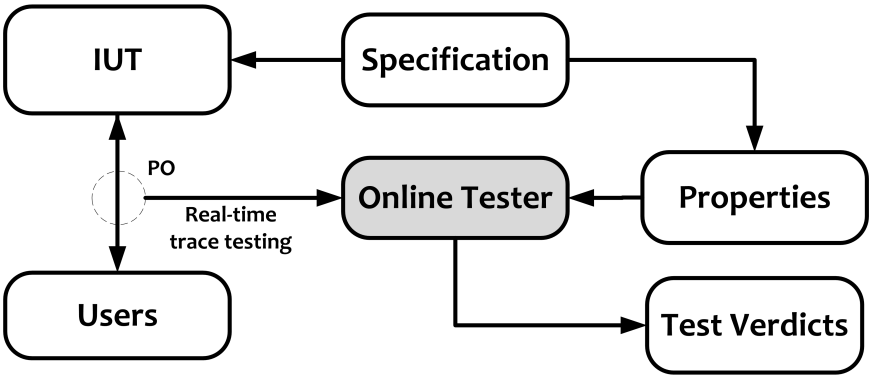


FIGURE 2.4: Online Passive Testing

As Figure 2.4 illustrates, the online passive testing process is quite similar to the offline one. However, the crucial difference is the real-time trace testing. It is a complex and challenging work. It requires the ability to handle numerous messages in a short time, and also requires the same offline testing preciseness. Meanwhile, since online testing is a long term continuously process, the tester has to undergo severe conditions when dealing with large amount of nonstop traces.

Several online testing techniques have been studied by the community in order to test systems or protocol implementations [10, 13, 14]. These methods provide interesting studies and have their own advantages, but they also have several drawbacks such as the presence of false negatives, space and time consumption, often related to a needed complete formal model, etc. Although they bring solutions, new results and perspectives to the protocol and system testers, they also raise new challenges and issues. The main ones are the non-collection of traces and their on-the-fly analysis. The traces are observed (through an interface and an eventual sniffer) and analyzed on-the-fly to provide test verdicts and no trace sets should be studied a posteriori to the testing process. In our work, we also present a novel formal online passive testing approach applied at run-time to test the conformance and performance of the IUT.

Some researchers presented a tool for exploring online communication and analyzing clarification of requirements over the time in [51]. It supports managers and developers to identify risky requirements. In [52], the authors defined a formal model based on Symbolic Transition Graph with Assignment (STGA) for both peers and choreography with supporting complex data types. The local and global conformance properties are formalized by the Chor language in their works. We should also cite the works [9, 53] from which an industrial testing tool has been developed. These works are based on

formal timed extended invariant to analyze run-time traces with deep packet inspection techniques. However, while most of the functional properties can be easily designed, complex ones with data causality can not. Moreover, although their approach is efficient with an important data flow, the process is still offline with finite traces that are considered as very long.

We may also cite some online active testing approaches from which we got inspired. In [14], the authors presented a framework that automatically generates and executes tests for conformance testing of a composite of Web services described in BPEL. The proposed framework considers unit testing and it is based on a timed modeling of BPEL specification, and an online testing algorithm that assigns verdicts to every generated state. In [54], they presented an event-based approach for modeling and testing the functional behavior of Web Services (WS). Functions of WS are modeled by event sequence graphs (ESG) and they raised the holistic testing concept that integrates positive and negative testing.

Inspired from all these above cited works, we propose an online formal passive testing approach by defining functional properties of IUT, without modeling the complete system, and by considering eventual false negatives. For this latter, we introduce a new verdict '*Time-Fail*' for distinguishing the real faults and the faults caused by timeouts. In addition, since online protocol testing is a long-term continuously testing process, we provide a temporary storage for remaining the integrity of incoming traces. Furthermore, for the lacking attention to test data portions of messages in current researches, our approach provides the ability to test both the data portion and control portion, accompanying with another new verdict '*Data-Inc*' which will be detailed in the Chapter 5.

Chapter 3

Formal Approach for Conformance Testing

"A great success is made up of an aggregation of little ones."
– Elbert Hubbard (1856 – 1915)

In the previous chapter, we shortly elaborates the related works in passive testing domain. However, in modern message-based protocols, while the control part still plays an important role, data is essential for the execution flow. Input/output causality cannot be assured since many outputs may be expected for a single input. Moreover, when traces are captured on centralized services, many equivalent messages can be observed due to interactions with multiple entities on clients. Furthermore, although the traces are finite, the number of related packets may become huge and the properties to be verified may become complex. For solving these issues, inspired and based on those prospective works, we present a passive testing approach for communicating protocols based on the formal specification of functional requirements and their analysis on collected (through Point of Observations) run-time execution traces. We will detail the approach in this chapter.

3.1 Basics and Syntax

A communication protocol message is a collection of data fields of multiple domains. Data domains are defined either as *atomic* or *compound* [55]. An *atomic* domain is defined as a set of numeric or string values. A *compound* domain is defined as follows.

Definition 1. A *compound* value v of length $n > 0$, is defined by the set of pairs $\{(l_i, v_i) \mid l_i \in L \wedge v_i \in D_i \cup \{\epsilon\}, i = 1...n\}$, where $L = \{l_1, ..., l_n\}$ is a predefined set of labels and D_i are sets of values, meaningful from the application viewpoint, and called data domains. Let D be a Cartesian product of data domains, $D = D_1 \times D_2 \times ... \times D_n$. A *compound* domain is the set of pairs (L, d), where d belongs to D.

Once given a network protocol P, a *compound* domain M_p can generally be defined by the set of labels and data domains derived from the message format defined in the protocol specification/requirements. A *message* m of a protocol P is any element $m \in M_p$.

Example 1. A possible message for the SIP protocol, specified using the previous definition could be

$$m = \{(method, \textbf{`INVITE'}), (time, \text{`644.294133000'}), (status, \epsilon), (from, \text{`alice@a.org'}), (to, \text{`bob@b.org'}), (cseq, \{(num, 7), (method, \textbf{`INVITE'})\})\}$$

representing an INVITE request from *alice@a.org* to *bob@b.org*.

A *trace* ρ is a sequence of messages of the same domain containing the interactions of a monitored entity in a network, through an interface (the P.O), with one or more peers during an arbitrary period of time.

In our work, we define a syntax based on Horn clauses [56] to express properties that are checked on extracted traces [55]. We choose Horn logic as the formalizing language since it has the benefit of allowing the re-usability of clauses. Besides, compared with other LTL based logic, it provides better expressibility and flexibility when analyzing protocols. It is more suitable for our work on testing protocols. Formulas in this logic can be defined with the introduction of terms and atoms, as it follows.

Definition 2. A *term* is defined in BNF as $term ::= c \mid x \mid x.l.l...l$ where c is a constant in some domain, x is a variable, l represents a label, and $x.l.l...l$ is called a *selector variable*.

Example 2. Let us consider the following message:

$$m = \{(method, \textbf{`INVITE'}), (time, \text{`523.231855000ms'}), (status, \epsilon), (from, \text{`alice@a.org'}), (to, \text{`bob@b.org'}), (cseq, \{(num, 10), (method, \textbf{`INVITE'})\})\}$$

In this message, the value of *method* inside *cseq* (a way to identify and order transactions, consists of a sequence number and a method) can be represented by **m.cseq.method** by using the *selector variable*.

Definition 3. An *atom* is defined as

$$\begin{aligned} A ::= & \overbrace{p(term, ..., term)}^{k} \\ & \mid term = term \\ & \mid term \neq term \\ & \mid term < term \\ & \mid term + term = term \end{aligned}$$

where $p(term, ..., term)$ is a predicate of label p and arity k.

The relations between *terms* and *atoms* are stated by the definition of clauses. A *clause* C is an expression of the form

$$A_0 \leftarrow A_1 \wedge ... \wedge A_n$$

where A_0 is the head of the clause and $A_1 \wedge ... \wedge A_n$ its body, A_i being *atoms*. Let K be the set of clauses $K = \{C_1, ..., C_p\}$.

A *formula* ϕ is defined by the following BNF:

$$\begin{aligned} \phi ::= & A_1 \wedge ... \wedge A_n \mid \phi \rightarrow \phi \mid \forall_x \phi \mid \forall_{y>x} \phi \\ & \mid \forall_{y<x} \phi \mid \exists_x \phi \mid \exists_{y>x} \phi \mid \exists_{y<x} \phi \end{aligned}$$

where $A_1, ..., A_n (n \geq 1)$ are *atoms*, x, y represent for different messages in a trace ρ and $\{<, >\}$ indicate the order relations of messages.

In our approach, while the variables x and y will be used to formally specify messages of a trace, the quantifiers commonly define "it exists" ($\exists$) and "for all" ($\forall$). The formula $\forall_x \phi$ is then equivalent to the expression "for all messages x in the trace, ϕ holds".

3.2 Semantics

The semantics used in our work is related to the traditional Apt–Van Emdem–Kowalsky semantics for logic programs [57], from which an extended version has been designed in order to deal with messages and trace temporal quantifiers [55]. Based on the above

described operators and quantifiers, we provide an interpretation of the formulas to evaluate them to '$\top$' ('*Pass*'), '$\bot$' ('*Fail*') or '?' ('*Inconclusive*').

Definition 4. A *substitution* θ is a finite set of bindings $\theta = \{x_1/term_1, ..., x_k/term_k\}$ where each $term_i$ is a *term* and x_i is a variable such that $x_i \neq term_i$ and $x_i \neq x_j$ if $i \neq j$.

The *application* $x\theta$ of a substitution θ to a variable x is defined as follows.

$$x\theta = \begin{cases} t & \text{if } x/t \in \theta \\ x & \text{otherwise} \end{cases}$$

The application of a particular binding x/t to an expression E (atom, clause, formula) is the replacement of each occurrence of x by t in the expression. The application of a substitution θ on an expression E, denoted by $E\theta$, is the application of all bindings in θ to all terms appearing in E.

Definition 5. Given $K = \{C_1, ..., C_p\}$ a set of clauses and $\rho = m_1, ..., m_n$ a trace. An *interpretation*[1] *in logic programming* is any function I mapping an expression E that can be formed with elements (clauses, atoms, terms) of K and terms from ρ to one element of $\{\top, \bot\}$. It is said that E is true in I if $I(E) = \top$.

The **semantics of formulas** under a particular interpretation I, is given by the following rules.

- The expression $t_1 = t_2$ is true, iff t_1 equals t_2 (they are the same term).
- The expression $t_1 \neq t_2$ is true, iff t_1 is not equal to t_2 (they are not the same term).
- The expression $t_1 < t_2$ is true, iff t_1 is less than t_2 ($term_1$ is smaller than the $term_2$).
- A ground *atom*[2] $A = p(c_1, ..., c_k)$ is true, iff $A \in I$.
- An atom A is true, iff every ground instance of A is true in I.
- The expression $A_1 \wedge ... \wedge A_n$, where A_i are atoms, is true, iff every A_i is true in I.
- A clause $C : A_0 \leftarrow B$ is true, iff every ground instance of C is true in I.

[1] Called an *Herbrand Interpretation*
[2] An atom where no unbound variables appear.

- A set of clauses $K = \{C_1, ..., C_p\}$ is true, iff every clause C_i is true in I.

An interpretation is called a *model* for a clause set $K = \{C_1, ..., C_p\}$ and a trace ρ if every $C_i \in K$ is true in I. A formula ϕ is true for a set K and a trace ρ (true in K,ρ, for short), if it is true in *every* model of K, ρ. It is a known result [57] that if M is a *minimal* model for K, ρ, then if $M(\phi) = \top$, then ϕ is true for K, ρ.

The general semantics of formulas is then defined as follows. Let K be a clause set, ρ a trace for a protocol and M a minimal model, the operator M defines the semantics of formulas.

$$\hat{M}(A_1 \wedge ... \wedge A_n) = \begin{cases} \top & \text{if } M(A_1 \wedge ... \wedge A_n) = \top \\ \bot & \text{otherwise} \end{cases}$$

The semantics for trace quantifiers requires first the introduction of a new truth value '?' (inconclusive) indicating that no definite response can be provided. The semantics of quantifiers $\forall$ and $\exists$ is defined as follows:

$$\hat{M}(\forall_x \phi) = \begin{cases} \bot & \text{if } \exists\theta \text{ with } x/m \in \theta \text{ and } m \in \rho, \\ & \text{where } \hat{M}(\phi\theta) = \bot \\ ? & \text{otherwise} \end{cases}$$

$$\hat{M}(\exists_x \phi) = \begin{cases} \top & \text{if } \exists\theta \text{ with } x/m \in \theta \text{ and } m \in \rho, \\ & \text{where } \hat{M}(\phi\theta) = \top \\ ? & \text{otherwise} \end{cases}$$

Since ρ is a finite segment of an infinite execution, it is not possible to declare a '$\top$' result for $\forall_x \phi$, since we do not know if ϕ may become '$\bot$' after the end of ρ. Similarly, for $\exists_x \phi$, it is unknown whether ϕ becomes true in the future. Similar issues occur in the literature of passive testing [5] and run-time monitoring [3], for evaluations on finite traces. The rest of the quantifiers are detailed in the following, where x is assumed to be found as a message previously obtained by $\forall_x$ or $\exists_x$

$$\hat{M}(\forall_{y>x} \phi) = \begin{cases} \bot & \text{if } \exists\theta \text{ with } y/m \in \theta, \\ & \text{where } \hat{M}(\phi\theta) = \bot \text{ and } m > x \\ ? & \text{otherwise} \end{cases}$$

$$\hat{M}(\exists_{y>x}\phi) = \begin{cases} \top & \text{if } \exists\theta \text{ with } y/m \in \theta, \\ & \text{where } \hat{M}(\phi\theta) = \top \text{ and } m > x \\ ? & \text{otherwise} \end{cases}$$

The semantics for $\forall_{y<x}$ and $\exists_{y<x}$ is equivalent to the last two formulas, exchanging $>$ by $<$. Finally, the truth value for $\hat{M}(\phi \to \psi) \equiv \hat{M}(\phi) \to \hat{M}(\psi)$.

3.3 Algorithm and complexity

The algorithm for evaluation of formulas uses a recursive procedure to evaluate formulas, coupled with a modification of SLD (Selective Linear Definite-clause) resolution algorithm [58] for evaluation of Horn clauses. The SLD resolution algorithm is provided in the following.

Algorithm 1: SLD resolution algorithm

Input: Set of clause K. Stack S containing the atoms remaining for evaluation. Substitution θ with the initial bindings
Output: $\top$ if the formula has a solution

```
1  if S is not empty then
2      A ← pop(S);
3      solved ←⊥;
4      for (B_0 ← B_1 ∧ ... ∧ B_q) ∈ K where B_0 matches with A do
5          renameVars(B_0, B_1, ..., B_q);
6          a ← θ;
7          if unify(A_0, B_0, a) then
8              if q > 0 then
9                  push({B_1, ..., B_q}, S);
10                 solved ← sldSolve(S, a);
11                 pop({B_1, ..., B_q}, S);
12             end
13             else
14                 solved ← sldSolve(S, a);
15             end
16         end
17         push(A, S);
18         return solved;
19     end
20 end
21 useSolution(θ);
22 return⊤;
```

As shown in algorithm 1, the resolution starts with a formula $A_1 \wedge ... \wedge A_p$ in the form of a stack (A_1 at the top of the stack). For each atom on the stack it looks for a matching clause (a clause with the same predicate label and arity) and adds the body of the clause to the stack to recursively call *solve*. When the stack is empty, a solution has been found and it notifies using the procedure *useSolution*(). An alternative to line

4 should also check whether A matches '=', '$\neq$', or '$<$', and respectively evaluate the equality, inequality or comparison.

We use $T_{eval}(\varphi)$ to represent the worst-case evaluating time of evaluation of a formula ϕ, and $T_{sld}(\phi)$ for the SLD evaluation of a formula $\phi = A_1 \wedge ... \wedge A_p$. Given a formula with k quantifiers $Q^1_{x_1} \cdots Q^k_{x_k}(A_1 \wedge ... \wedge A_p)$, where each $Q_j \in \{\forall, \exists\}$ and a trace $\rho = m_1, ..., m_n$, then the relation between T_{eval} and T_{sld} is described by:

$$T_{eval}(Q^1_{x_1} \cdots Q^k_{x_k}(A_1 \wedge ... \wedge A_p)) = \sum_{i_1=1}^{n} \cdots \sum_{i_k=1}^{n} T_{sld}((A_1 \wedge ... \wedge A_p)\theta_1 \cdots \theta_k)$$

where $\theta_j = \{x_j/m_{i_j}\}$ is the substitution obtained by the evaluation of the quantifier $Q^j_{x_j}$. For a simple formula, the resolution time is small compared with the evaluating time on the trace, therefore an upper bound for the SLD resolution time can be used inside the summation.

$$T_{sld}((A_1 \wedge ... \wedge A_p)\theta_1 \cdots \theta_k) \leq T, \theta_j = \{x_j/m_{i_j}\}, \forall i_1, ..., i_k$$

Then, applying this inequality

$$\sum_{i_1=1}^{n} \cdots \sum_{i_k=1}^{n} T_{sld}((A_1 \wedge ... \wedge A_p)\theta_1 \cdots \theta_k) \leq \sum_{i_1=1}^{n} \cdots \sum_{i_k=1}^{n} T = n^k T$$

which shows that the worst case complexity for this type of formula is $\mathcal{O}(n^k)$, where n is the length of the trace and k represents the number of quantifiers in the formula.

The complexity of the algorithm corresponds to the time to analyze the complete trace, and not for obtaining individual solutions, which depends on the type of quantifiers used. For instance for a property $\forall_x p(x)$, individual results are obtained in $\mathcal{O}(1)$, and for a property $\forall_x \exists_y q(x, y)$, results are obtained in the worst case in $\mathcal{O}(n)$. Finally, it can also be shown that a formula with a '$\rightarrow$' operator, where Q are quantifiers.

$$\underbrace{Q \ldots Q}_{k}(\underbrace{Q \ldots Q}_{l}(A_1 \wedge \ldots \wedge A_p) \rightarrow \underbrace{Q \ldots Q}_{m}(A'_1 \wedge \ldots \wedge A'_q))$$

This formula has a worst-case time complexity of $\mathcal{O}(n^{k+max(l,m)})$, which has advantages with respect to using formulas without the '$\rightarrow$' operator. Although this is an important point, through the experiments the complexity is evaluated successfully.

3.4 Comparison with other approaches

For better illustrating our approach, we make a short comparison with other similar approaches. However, in many solutions, the researchers assume that the current states of the observed trace are known. In our case, we do not require such assumptions. Our technique has no effect on the running behavior of the system being tested. Moreover, as above mentioned, our points of observation are set in a black-box framework which does not allow any homing phase [5]. Since no specification of the implementation under test is provided, the extracted traces are not related to any known states.

Because of these concerns, a comparison of the approaches according to their expressiveness, efficiency, complexity and capabilities are not easy to settle [59]. Nevertheless, we try in the following to compare some key aspects of these approaches. Our method can be compared to the techniques used in PASTE [36] and EAGLE [31]. These two tools are representative of how to test passively and efficiently a protocol. EAGLE provides an interesting formalism to express complex properties. However, they assume knowing the variables' values of each state in a trace. Furthermore, even if its expressiveness is close to ours, the design of such properties is difficult given their complex scheme, making it hard to implement them efficiently. PASTE embeds innovative passive testing algorithms but does not consider the causality between the data portions in a trace. With our approach, we argue first the need of checking all the packets in a trace since the states are unknown and second the analysis of data constraints through all the packets of the trace.

Tool	Datamon	EAGLE	PASTE	MOP
Time Complexity	$n^{k+max(l,m)}$	$n.p^4 2^{2p} log^2 p$	$k.n^2 + n.(p-k)$	*
Memory Complexity	$nlog(n)$	$n.p^2 2^p log(p)$	n	*
States unneeded	✓	✗	✓	✗
Temporal logic	✗	✓	✗	✓
Invariant	✓	✗	✓	✓
Condition	✓	✗	✗	✓
Actions to IUT	✗	✗	✗	✓
Data constraints	✓	✓	✗	✗

TABLE 3.1: Some comparative aspects of passive testing tools

When comparing memory and time complexities of the three algorithms (see Table 3.1), our tool Datamon presents a high time complexity in comparison to the others. However, the Datamon memory complexity is much more interesting. The reasons are obvious. Indeed, in our work we manage some data in the formula and we do not assume any

knowledge about the implementation states which increases the time complexity. However, in compensation, our top-down resolution tree leads to a linear memory complexity. The complexity is not the only key points allowing to compare our approach to others. Table 3.1 details the comparative aspects of the three above mentioned approaches plus the software MOP commonly used in benchmarks [60] (unfortunately, we cannot find the memory and time complexity of MOP in their publications).

Table 3.1 illustrates the advantages and limitations of our approach, where p is the number of operators in a formula and n, k, l represent the length of the trace, the number of quantifiers and respectively. Although the time complexity of our algorithm is higher, the space complexity is quite better.

3.5 Experiments

In this section, our approach has been implemented into an IMS framework. We provide some experiments results evaluated on large traces, in order to verify the functionality of our approach.

3.5.1 IP Multimedia Subsystem

The IMS (IP Multimedia Subsystem) is a standardized framework for delivering IP multimedia services to users in mobility. It was originally intended to deliver Internet services over GPRS connectivity. This vision was extended by 3GPP, 3GPP2 and TISPAN standardization bodies to support more access networks, such as Wireless LAN, CDMA2000 and fixed access network. The IMS aims at facilitating the access to voice or multimedia services in an access independent way, in order to develop the fixed-mobile convergence. To ease the integration with the Internet world, the IMS heavily makes use of IETF standards.

The core of the IMS network consists of the Call Session Control Functions (CSCF), that redirect requests depending on the type of service, the Home Subscriber Server (HSS), a database for the provisioning of users, and the Application Server (AS), where the different services run and interoperate. Most communication with the core network and between the services is done using the Session Initiation Protocol [7]. Figure 3.1 shows the core functions of the IMS framework and the protocols used for communication between the different entities.

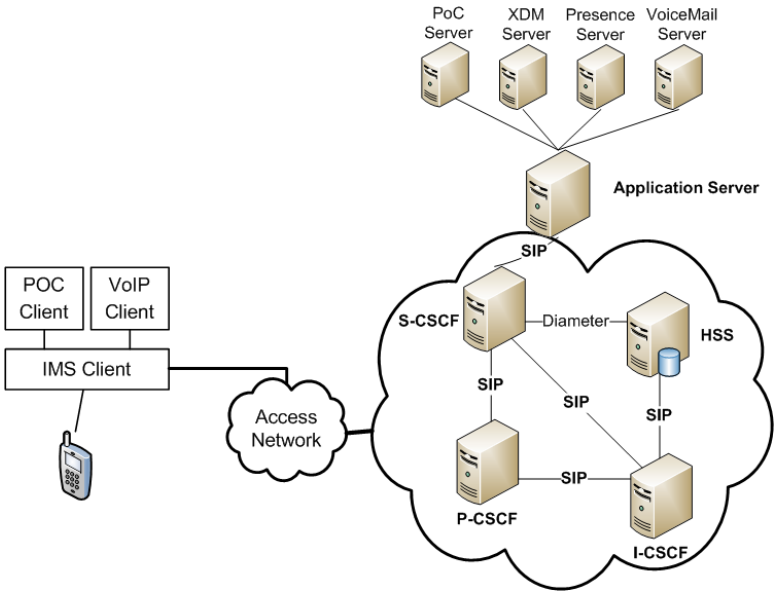


FIGURE 3.1: Core functions of IMS framework

3.5.2 Session Initiation Protocol

The Session Initiation Protocol (SIP) is an application-layer protocol that relies on request and response messages for communication, and it is an essential part for communication within the IMS (IP Multimedia Subsystem) framework. Messages contain a header which provides session, service and routing information, as well as an (optional) body part to complement or extend the header information. Several RFCs have been defined to extend the protocol with to allow messaging, event publishing and notification. These extensions are used by services of the IMS such as the Presence service [61] and the Push to-talk Over Cellular (PoC) service [62].

3.5.2.1 Overview

The Session Initiation Protocol is an application-layer control protocol specified by the IETF [7] for creating, modifying and terminating multimedia sessions with one or more participants, independently of the underlying transport. A typical SIP session is established as follows, where a user Alice calls another user Bob. A diagram of the entities in the communication and the message exchange is provided in Figure 3.2. We will briefly introduce the communication process in the following:

- Alice uses a SIP client software as a User Agent Client (UAC) to send a request, while Bob acts as a User Agent Server (UAS) to receive a request. Alice calls Bob using his SIP identity, a type of Uniform Resource Identifier (URI).

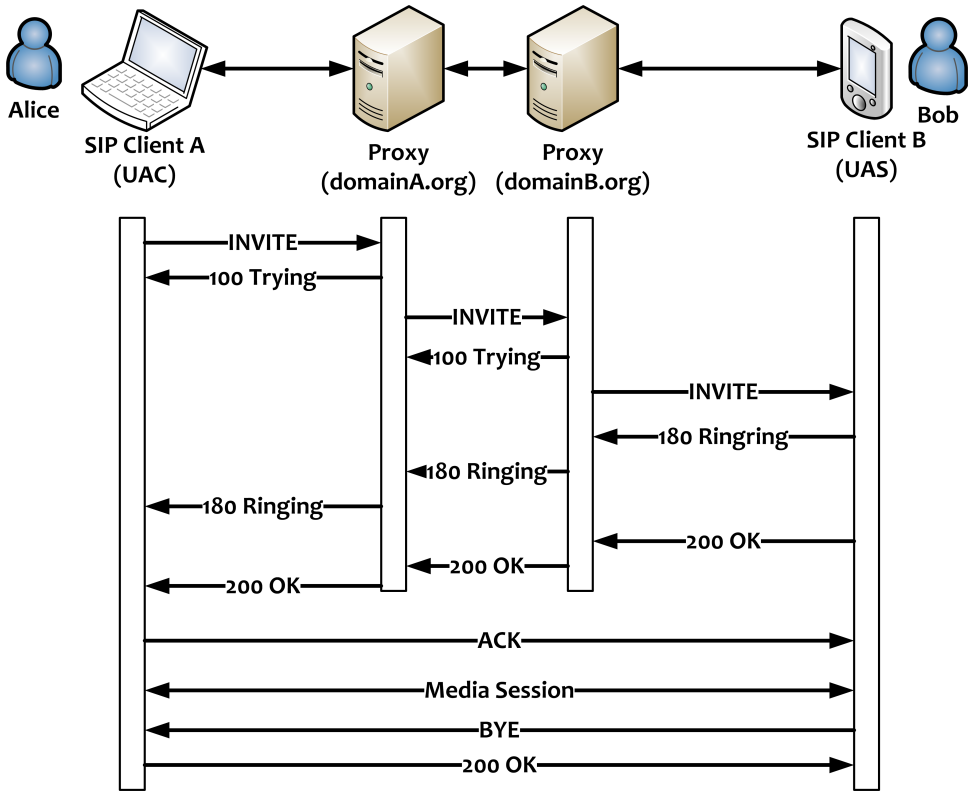


FIGURE 3.2: SIP entities and message exchange

- The UAC generates a SIP **INVITE** message, containing a request line indicating the method **INVITE**, the UAS's identifier (sip:bob@domainB.org) and version, followed by a number of headers.
- If the software client does not know the IP address of Bob, it locates a proxy server inside own domain (domainA.org).
- The proxy form the calling domain, sends a 100 Trying response to the UAC to let it know that the proxy is processing the request.
- The proxy locates another proxy in the reception domain (domainB.org), where it sends the message, adding its own address to the Via header.
- The proxy in the reception domain, sends a 100 **Trying** response to the proxy in the calling domain to let it know that the request is being processed.
- The proxy locates the address of Bob by consulting from a location server, and transmits the message with its own address in the Via header to that address.
- The client on Bob's receives the message and returns a 180 **Ringing** response, indicating that it is waiting for Bob to answer the call.
- When Bob answers the call, the client software sends a 200 **OK** to indicate that the call has been answered.

- Once the client on Alice's side receives the **OK** response, it immediately sends an **ACK** request to acknowledge the reception of the message, and starts the media session.
- When the media session is over, the terminating client sends a **BYE** message, which will be replied with a 200 **OK** response.

3.5.2.2 Entities and Network Elements

Some of the entities and elements that take part in a SIP session are described as follows:

- **User Agent**: UA is the endpoint in the SIP communication in charge of generating requests and responses. A UA can take the role of either an UAC (creating and sending requests) or an UAS (receiving requests and generating responses).
- **Proxy Server**: An intermediary entity that acts as both a server and client for the purpose of making requests on behalf of other clients.
- **Registrar**: A SIP server that receives SIP **REGISTER** requests and stores the information in those requests.
- **Redirect Server**: A user agent server that generates 3xx responses to requests it receives, directing the client to contact an alternate set of URIs.

3.5.2.3 Message Syntax

Each SIP message begins by a start line, called the request line (if it is a request) or a status line (if it is a response). The start line is followed by a number of headers and the message body. The request line is composed by the method of the request, indicating the type of operation requested, the request URI and the version of SIP used in the message. A short description of the methods defined in the RFC is provided as follows:

- **REGISTER**: Used by the UA to indicate its current SIP address and the SIP URI being used as identifier.
- **INVITE**: Used to initiate a media session between UAs.
- **ACK**: Used to acknowledge the reception of a message, usually a 2xx response.
- **CANCEL**: Used to terminate a previous request.
- **BYE**: Used to terminate an ongoing media session.

- **OPTIONS**: Used to query a server of its capabilities.

A response's status line is composed by a status code, a 3-digit integer indicating the outcome of a request, and a reason code, providing a short textual description of the status code intended for a human user. The different classes of status codes are defined below.

- **1xx. Provisional**: Indicating that the request has been received and the process is being continued.
- **2xx. Success**: Indicating that the action was successfully received, understood and accepted.
- **3xx. Redirection**: Further action needs to be taken in order to complete the request.
- **4xx. Client error**: The request contains bad syntax or cannot be fulfilled.
- **5xx. Server error**: The server failed to fulfill an apparently valid request.
- **6xx. Global failure**: The request cannot be fulfilled at any server.

SIP headers starts by the header name, followed by a colon and the header value, ending in a carriage-return line-feed sequence. The following six header fields are the mandatory minimum for any request formulated by a UAC according to the RFC.

- **To**: Specifies the desired logical recipient for the request in the form of a SIP URI or another URI scheme. It is usually composed of the identifier of the target, display name, as well as other optional parameters.
- **From**: Indicates the logical identity of the user initiating the request. It also contains an URI as the identity and an optional display name.
- **CSeq**: Serves as a way to identify and order transactions. It consists of a sequence number and a method, where the method matches the method from the request line.
- **Call-ID**: Acts as a unique identifier to group together a series of messages. It identifies uniquely a particular invitation or all registrations of a particular client.
- **Max-Forwards**: Serves to limit the number of hops a request can transit on its way to a destination.
- **Via**: Indicates the transport and addresses of each location where the message has gone through in order to arrive at its destination. Each time a request goes through a hop, the local UAC inserts new address in the Via header of the request.

3.5.3 Testing Framework

We have implemented this testing framework using Java. The implemented system is composed of three main modules:

- Filtering and conversion of collected traces
- Evaluation of tests
- Evaluation of formulas

Figure 3.3 shows the way the modules interact and the inputs and outputs from each one.

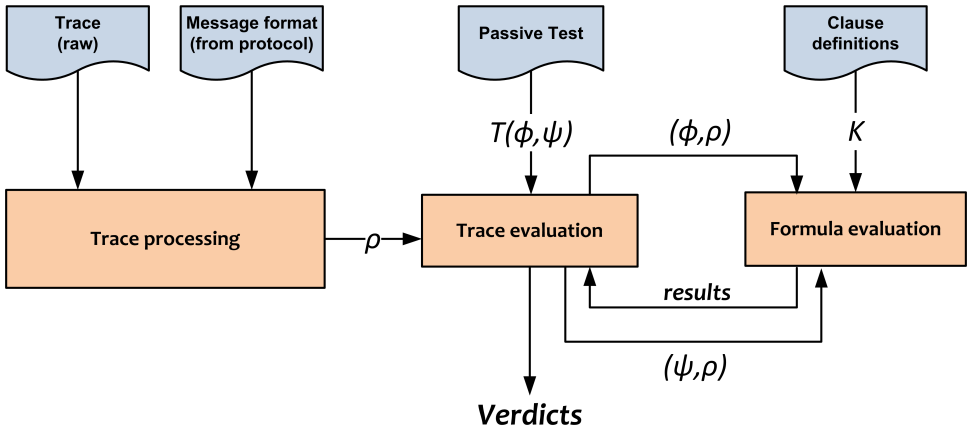


FIGURE 3.3: Architecture for the conformance testing framework

The trace processing module takes the raw traces collected from the network exchange, and it converts the messages from the input format. In our particular implementation, the input trace format is PDML, an XML format that can be obtained from Wireshark[3] traces. In the XML, data values are identified by a `field` tag, representing an individual data element in the message (a header, a parameter). Each sub-element in the target message is related to a field in the XML by its name, for instance, the '`status.line`' message element with the XML field '`sip.Status-Line`'. In the XML, fields are grouped by protocol, which also allows the tool to filter messages not relevant to the properties being tested.

The purpose of the module is to convert each packet in the raw trace into a data structure (a compound value) conforming to the definition of a message. The format for the message is defined in a different input file to the module. There, each sub-element for the target message (e.g. '.method', '.cseq.seq') is associated with the respective element

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

in the trace source format. This module also performs filtering of the trace, in order to only take into account messages of the studied protocol. Since this is a separate module of the implementation, alternative trace formats can be changed or expanded by modifying this module.

The test evaluation module receives as input a passive test as defined in Section 3.2, as well as a trace from the trace processing module and produces a verdict from the satisfaction results of the test and conditional formulas. The formula evaluation module is implemented as described in Section 3.3. It receives a trace and a formula, along with the clause definitions and returns a set of satisfaction results for the query in the trace, as well as the messages and variable bindings obtained in the process. The implementation and the files used for the experiments can be found at `http://www-public.it-sudparis.eu/~maag/Datamon/web/Datamon.html`. The results from the experiments are presented in the following.

3.5.4 Environments

For the experiments, traces were obtained from SIPp [63]. SIPp is an Open Source test tool and traffic generator for the SIP protocol, provided by the Hewlett-Packard company. It includes a few basic SipStone user agent scenarios (User Agent Client (UAC) and User Agent Server (UAS)) and establishes and releases multiple calls with the INVITE and BYE methods. It can also read custom XML scenario files describing from very simple to complex call flows. It features the dynamic display of statistics about running tests, TCP and UDP over multiple sockets or multiplexed with retransmission management and dynamically adjustable call rates. It also supports IPv6, TLS, SIP authentication, conditional scenarios, UDP retransmissions, error robustness, call specific variable, etc. SIPp can be used to test many real SIP equipments like SIP proxies, B2BUAs and SIP media servers [63]. The traces obtained from SIPp contain all communications between the client and the SIP core. Tests were performed using a prototype implementation of the formal approach mentioned above, using an algorithm developed by us and described in the Section 3.3.

In the experiment, we designed a real Local Area Network (LAN) architecture for testing. For ensuring the accuracy and authenticity of the results, we construct the environment by using real laptops. As shown in Figure 3.4, the LAN architecture is an environment containing several UACs, which can be used to test the correctness, robustness and reliability under tremendous number of calls. The observation points being are the UAS.

- Hardware configuration of UAS:

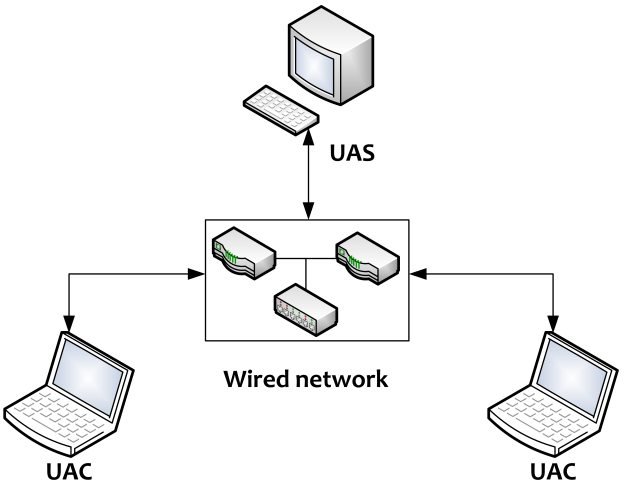


FIGURE 3.4: Our LAN Architecture

CPU- Intel Core i5-2520M 2.50 GHz, 4GB DDR3

- Hardware configuration of UACs:

  CPU- AMD Atholon 64 X2 5200+, 2GB DDR2

  CPU- Intel Core2 Duo T6500 2.10 GHz, 2GB DDR2

3.5.5 Properties and Results

In order to formally design the properties to be passively tested, we studied deeply the TTCN-3 test suite of SIP [2] and the RFC 3261 of SIP [7]. We designed 7 properties for the experiments, for the evaluation of each property we used a set of traces {500, 1000, 2000, ... , 512000} in order to get exhaustive results. We provide in the following the definition of our chosen properties as well as the obtained verdicts on the tested finite traces.

Property 1: For every request there must be a response

This property can be used for a monitoring purpose, in order to draw further conclusions from the results. Due to the issues relative to testing on finite traces for finite executions, a $fail$ results can never be given for this context. However *inconclusive* results can be provided and conclusions may be drawn from further analysis of the results (for instance if the same type of message is always without a response). The property evaluated is as follows:

$$\forall_x(request(x) \wedge x.method! = \textbf{`ACK'} \\ \rightarrow \exists_{y>x}(nonProvisional(y) \wedge responds(y,x)))$$

where $nonProvisional(x)$ accepts all non provisional responses (non-final responses, with $status \geq 200$), to requests with method different than **ACK**, which does not require

a response. The results from the evaluation on the traces are shown on Table 3.2. As

Trace	No.of messages	Pass	Fail	Inconclusive	Time(s)
1	500	150	0	0	0.941
2	1000	318	0	1	1.582
3	2000	676	0	0	2.931
4	4000	1301	0	1	5.185
5	8000	2567	0	1	10.049
6	16000	5443	0	0	20.192
7	32000	10906	0	1	39.016
8	64000	21800	0	0	84.015
9	128000	43664	0	0	155.903
10	256000	87315	0	1	382.020
11	450000	153466	0	0	1972.720
12	512000	?	?	?	?

TABLE 3.2: "For every request there must be a response"

expected, most of the traces show only true results for the property evaluation, however traces 2,4,5,7 and 10 show an unusual number of inconclusive results. Taking a closer look at trace 10, the inconclusive verdict corresponds to the **REGISTER** message, with an Event header corresponding to a **conference** event [64], this message is at the end of the trace, which could indicate that the client closed the connection before receiving the **REGISTER** message. The same phenomenon can be observed on the other traces (2,4,5 and 7). The last trace with question mark is too huge to be executed due to the limitation of the computer memory, the program crashed after 4 hours execution which raises a first limitation of our approach.

Property 2: No session can be initiated without a previous registration

This property can be used to test that only users successfully registered with the SIP Core can initiate a PoC session (or a SIP call, depending on the service). It is defined using our syntax as follows

$$\forall_x(\exists_{y>x} sessionEstablished(x,y)$$
$$\rightarrow \exists_{u<x}(\exists_{v>u} registration(u,v)))$$

where *sessionEstablished* and *registration* are defined as

$$sessionEstablished(x, y) \leftarrow x.method = \textbf{'INVITE'}$$
$$\wedge y.statusCode = 200$$
$$\wedge responds(y, x)$$

$$registration(x, y) \leftarrow request(x) \wedge responds(y, x)$$
$$\wedge x.method = \textbf{'REGISTER'}$$
$$\wedge y.statusCode = 200$$

The analysis of the results, however depends on the following condition: did the trace collection begin from a point in the execution of the communication before the user(s) registration took place? If the answer is positive, then inconclusive results can be treated as a possible fault in the implementation, otherwise, only inconclusive verdicts can be given. Unfortunately, in the collected traces such condition does not hold, therefore a definitive verdict cannot be provided. However it can be shown that the property and the framework allow to detect when the tested property holds on the trace, as shown in Table 3.3.

Trace	No.of messages	Pass	Fail	Inconclusive	Time
1	500	60	0	0	13.690s
2	1000	109	0	0	57.117s
3	2000	182	0	1	207.841s
4	4000	405	0	0	869.322s
5	8000	785	0	0	1.122h
6	16000	1459	0	0	5.660h
7	32000	2905	0	0	27.282h
8	64000	5863	0	1	136.818h
9	128000	?	?	?	?

TABLE 3.3: "No session can be initiated without a previous registration"

From the results on Table 3.3, it can also be seen that the evaluation of this property is much more time consuming than the one on Table 3.2. Based on previous results and the algorithm complexity, we predict the trace 9 will take approximately 23 days for the evaluation where the same trace took only 155s in property 1. Although this is expected given the complexity of evaluation (n^2 form the first property vs. n^4 in the current one), the current definition of the property is also quite inefficient, and shows a limitation of the syntax. During evaluation, all combinations of x and y are tested until $sessionEstablished(x, y)$ becomes true, and then all combinations of u and v are evaluated until $registration(u, v)$ becomes true. It would be much more efficient to look first for a message with method **INVITE**, then look whether the invitation was

validated by the server as a response with status 200 to then attempt to look for a registration. This could be achieved, for instance, by allowing quantifiers on the clause definitions, unfortunately, the syntax as currently specified does not allow that type of definition. This limitation is also raised in the following.

Property 3: Subscription to events and notifications

As described in the Section 3.4.2, in the presence service, a user (the watcher) can subscribe to another user's (the presentity) presence information. This works by using the SIP messages **SUBSCRIBE**, **PUBLISH** and **NOTIFY** for subscription, update and notification respectively. These messages also allow the subscription to other types of events other than presence, which is indicated in the header **Event** on the SIP message. It is desirable then to test, that whenever there is a subscription, a notification MUST occur upon an update event. This can be tested with the following formula:

$$\forall_x(\exists_{y>x}(subscribe(x, watcher, user, event) \wedge update(y, user, event)) \\ \rightarrow \exists_{z>y} notify(z, watcher, user, event)))$$

where *subscribe*, *update* and $notify$ hold on **SUBSCRIBE**, **PUBLISH** and **NOTIFY** events respectively. Notice that the values of the variables *watcher*, *user* and *event* may not have a value at the beginning of the evaluation, in that case their value is set by the evaluation of the *subscribe* clause, shown in the following

$$subscribe(x, watcher, user, event) \\ \leftarrow x.method = \textbf{`SUBSCRIBE'} \\ \wedge watcher = x.from \\ \wedge user = x.to \\ \wedge event = x.event$$

Here, the '=' operator compares the two terms, however if one of the term is an unassigned variable, then the operator works as an assignment. In the formula, the values assigned on the evaluation of *subscribe* will be then used for comparison in the evaluation of *update*. This is another way of defining formulas, different from using only message attributes.

The results of evaluating the formula are shown on Table 3.4. The results show no inconclusive results, although they also show that the full notification sequence is quite few in most traces. Notice that we are explicitly looking for a sequence *subscribe* $\rightarrow$ *update* $\rightarrow$ $notify$, however the sequence *subscribe* $\rightarrow$ $notify$ can also be present for subscription to server events, therefore **SUBSCRIBE** and **NOTIFY** events might also appear on the trace. To test the capabilities of detection, some **SUBSCRIBE**

messages were manually introduced on a trace, matching existing **PUBLISH** messages. The lack of notification for the update was correctly detected by the evaluation of the property.

Trace	No.of messages	Pass	Fail	Inconclusive	Time
1	500	3	0	0	10.412s
2	1000	7	0	0	42.138s
3	2000	10	0	0	160.537s
4	4000	19	0	0	632.192s
5	8000	30	0	0	2520.674s
6	16000	52	0	0	2.808h
7	32000	74	0	0	11.250h
8	64000	122	0	0	45.290h
9	128000	?	?	?	?

TABLE 3.4: "Whenever an update event happens, subscribed users must be notified on the set of traces"

Similarly to property 2, this property is quite inefficient in its evaluation, due to the same nesting of quantifiers. The evaluation time can be improved by rewriting the property as:

$$\forall_x(update(x, user, event)$$
$$\rightarrow (\exists_{y<x} subscribe(y, watcher, user, event)$$
$$\rightarrow \exists_{z>x} notify(z, watcher, user, event)))$$

which can be understood as: "if an update event is found, then if a previous subscription exists to such event, then a notification must be provided at some point after the update event". The results of evaluating this property are shown on Table 3.5. Notice that

Trace	No.of messages	Pass	Fail	Inconclusive	Time(s)
1	500	4	0	0	0.560
2	1000	7	0	0	1.158
3	2000	11	0	0	3.089
4	4000	19	0	0	6.164
5	8000	30	0	0	12.684
6	16000	52	0	0	25.416
7	32000	75	0	0	50.130
8	64000	122	0	0	99.372
9	128000	198	0	0	202.492
10	256000	342	0	0	394.756
11	512000	?	?	?	?

TABLE 3.5: "If an update event is found, then if a previous subscription exists, then a notification must be provided"

for trace 1,3 and 7, a different number of true results are returned. This is due to the order of search given by the property, in the previous property it sufficed with one pair

SUBSCRIBE - PUBLISH, in order to return a result. In the current property, for each **PUBLISH** it will look for a matching **SUBSCRIBE**. Since for every subscription there can exist multiple updates, the number of true results differs.

Also from the Table 3.4 and Table 3.5, we can observe that the evaluation time are sharply reduced if we follow an efficient way to write properties. The testers should avoid to write a property which recursively read the trace, and they should try to add as many detailed restrictions of the variables as they can. For instance, we just added a small restriction 'the message x should satisfy an update event' for the property 3, but it saved a lot of evaluation time as you can observe from the Table 3.4 and Table 3.5.

Property 4: Every 2xx response for INVITE request must be responded with an ACK

This property can be used to ensure that when the IUT (UAC) has initiated an **INVITE** client transaction, either it is in the Calling or Proceeding state, on receipt of a Success (200 OK) response, the IUT MUST generate an **ACK** request. The **ACK** request MUST contain values for the Call-ID, From and Request-URI that are equal to the values of those header fields in the **INVITE** request passed to the transport by the client transaction. The To header field in the **ACK** MUST equal the To header field in the 2xx response being acknowledged, and therefore will usually differ from the To header field in the original **INVITE** request by the addition of the tag parameter. The **ACK** MUST contain a single Via header field, and this MUST be equal to the top Via header field (the field without the branch parameter) of the original **INVITE** request. The CSeq header field in the **ACK** MUST contain the same value for the sequence number in the original **INVITE** request, but the value of Method parameter MUST be equal to '**ACK**'. This property evaluated is as follows:

$$\begin{aligned}&\forall_x(request(x) \wedge x.method = \textbf{INVITE}\\ &\rightarrow \exists_{y>x}(responds(y,x) \wedge success(y))\\ &\rightarrow \exists_{z>y}(ackResponse(z,x,y)))\end{aligned}$$

where *success* is defined as

$$success(y) \leftarrow y.statusCode >= 200 \wedge y.statusCode < 300$$

and $ackResponse$ is defined as

$$
\begin{aligned}
&ackResponse(x, y, z)\\
&\leftarrow x.method = \textbf{ACK}\\
&\wedge x.Call - id = y.Call - id\\
&\wedge x.CSeq = y.CSeq\\
&\wedge x.CSeq.method = \textbf{ACK}\\
&\wedge x.to = z.to\\
&\wedge x.From = y.From\\
&\wedge x.Request - URI = y.Request - URI\\
&\wedge x.TopVia = y.TopVia
\end{aligned}
$$

Trace	No.of messages	Pass	Fail	Inconclusive	Time
1	500	60	0	0	1.901s
2	1000	109	0	0	3.665s
3	2000	183	0	0	11.805s
4	4000	405	0	0	40.104s
5	8000	784	0	1	130.611s
6	16000	1459	0	0	522.050s
7	32000	2904	0	1	2237.442s
8	64000	5864	0	0	2.093h
9	128000	11555	0	1	8.630h
10	256000	23154	0	0	37.406h
11	450000	43205	0	0	142.568h
12	512000	?	?	?	?

TABLE 3.6: "Every 2xx response for **INVITE** request must be responded with an **ACK**"

The inconclusive messages observed in traces 5,7,9 of Table 3.6 are caused by the same phenomenon described in property 1. Besides, we observe a regular pattern in the results of this property: As the Table 3.3 and 3.6 illustrated, with the evaluation of same traces, the sum of Pass and Inconclusive verdicts of each trace in property 4 equal to the sums in property 2. This can be interpreted as the continuity of the transactions.

We are looking for a sequence:

$$sequence_a : \textbf{REGISTER} \rightarrow 200 \rightarrow \textbf{INVITE}$$

in property 2, on the other side, in property 4 we are searching a sequence:

$$sequence_b : \textbf{INVITE} \rightarrow 200 \rightarrow \textbf{ACK}$$

As described in property 2, each **INVITE** must have a previous **REGISTER** message. We can infer a new sequence:

$$\textbf{REGISTER} \rightarrow 200 \rightarrow \textbf{INVITE} \rightarrow 200 \rightarrow \textbf{ACK}$$

which means each **ACK** message in the transaction must be corresponded to one **REGISTER** request. Under the ordinary condition, the verdict numbers of $sequence_a$ should be equal to the ones of $sequence_b$.

Property 5: Every 300-699 response for INVITE request must be responded with an ACK

Similar to the previous one, this property can be used to ensure that when the IUT (UAC) has initiated an **INVITE** client transaction, either it is in the Calling state or Proceeding state, on receipt of a response with status code 300-699, the client transaction MUST be transited to "Completed", and the IUT MUST generate an **ACK** request. The **ACK** MUST be sent to the same address and port which the original **INVITE** request was sent to, and it MUST contain values for the Call-ID, From and Request-URI that are equal to the values in the **INVITE** request. The To header field in the **ACK** MUST equal the To header field in the response being acknowledged. The **ACK** MUST contain a single Via header field, and this MUST be equal to the Via header field of the original **INVITE** request which includes the branch parameter. The CSeq header field in the **ACK** MUST contain the same value for the sequence number in the original **INVITE** request, but the value of Method parameter MUST be equal to '**ACK**'.

Similarly to the property above, this property can be applied as:

$$\begin{aligned}&\forall_x(request(x) \wedge x.method = \textbf{INVITE}\\ &\rightarrow \exists_{y>x}(responds(y,x) \wedge fail(y))\\ &\rightarrow \exists_{z>y}(ackResponse(z,x,y)))\end{aligned}$$

where $fail$ is defined as

$$fail(y) \leftarrow y.statusCode >= 300 \wedge y.statusCode < 700$$

and *ackResponse* is defined as

$$
\begin{aligned}
&ackResponse(x, y, z)\\
&\leftarrow x.method = \textbf{ACK}\\
&\wedge x.Call - ID = y.Call - ID\\
&\wedge x.CSeq = y.CSeq\\
&\wedge x.CSeq.method = \textbf{ACK}\\
&\wedge x.to = z.to\\
&\wedge x.From = y.From\\
&\wedge x.Request - URI = y.Request - URI\\
&\wedge x.TopVia = y.TopVia
\end{aligned}
$$

Trace	No.of messages	Pass	Fail	Inconclusive	Time
1	500	10	0	0	3.445s
2	1000	18	0	0	10.798s
3	2000	49	0	0	34.331s
4	4000	91	0	0	137.083s
5	8000	165	0	0	557.803s
6	16000	367	0	1	1950.656s
7	32000	736	0	0	2.103h
8	64000	1403	0	0	8.498h
9	128000	2796	0	0	36.159h
10	256000	5513	0	0	145.088h
11	512000	?	?	?	?

TABLE 3.7: "Every 300-699 response for INVITE request must be responded with an ACK"

As shown in Table 3.7, the only one inconclusive verdict in trace 6 is due to the same phenomenon described in property 1. This property has the same time complexity as the previous one ($O(n^3)$), which means the evaluation times in property 5 should equal or close to the ones in the property 4 on the same traces. However, the actual evaluation time does not respect it. From the Table 3.6 and 3.7, we can observe that the evaluation times of property 5 are always one level higher than the times of property 4.

From the experiment results, we observe that the *evaluation time is proportional to the the number of fails*. Conversely in property 5, *the evaluation time is proportional to the the number of successes*. Considering the success responses are 10 times more than the fail ones, the phenomenon that property 5 consumes more time than the property 4 can thus be explained.

Property 6: A CANCEL request SHOULD NOT be sent to cancel a request other than INVITE

Since requests other than **INVITE** are responded to UAC immediately, sending a **CANCEL** for a non-INVITE request would always create a race condition. Once the **CANCEL** is constructed, the client should check whether it has received any response for the request being canceled. If no provisional response has been received, the **CANCEL** request must not be sent. Rather, the client must wait for the arrival of a provisional response (1xx) before sending the request. If the original request has generated a final response, the **CANCEL** should not be sent. This property can be used to ensure when the IUT having received a 1xx response to its **INVITE** request, to give up the call, it can send a **CANCEL** request with the same Request-URI, Call-ID, From, To headers, Via headers, numeric part of CSeq as in the original **INVITE** message, with a method field in the CSeq header set to "CANCEL".

This property can be defined by using our syntax as follows:

$$\begin{aligned}&\forall_x(request(x) \wedge x.method = \textbf{CANCEL}\\&\rightarrow \exists_{y<x}(continues(y,x) \wedge y.statusCode = 1xx)\\&\rightarrow \exists_{z<y}(responds(y,z) \wedge invite(z,x)))\end{aligned}$$

where *continues* is defined as

$$\begin{aligned}&continues(y,x) \leftarrow y.to = x.to\\&\wedge y.Call-ID = x.Call-ID\\&\wedge y.From = x.From\\&\wedge y.Request-URI = x.Request-URI\\&\wedge y.TopVia = x.TopVia\end{aligned}$$

and *invite* is defined as

$$\begin{aligned}&invite(z,x) \leftarrow z.method = \textbf{INVITE}\\&\wedge x.to = z.to\\&\wedge x.Call-ID = z.Call-ID\\&\wedge x.From = z.From\\&\wedge x.Request-URI = z.Request-URI\\&\wedge x.CSeq = z.CSeq\\&\wedge x.TopVia = z.TopVia\end{aligned}$$

As Table 3.8 illustrates, there is no inconclusive verdict. The evaluation time of each trace almost respect the linear increment of $y = 2x$ (x being the evaluation time of the

Trace	No.of messages	True	False	Inconclusive	Time(s)
1	500	5	0	0	0.780
2	1000	11	0	0	1.232
3	2000	21	0	0	2.309
4	4000	43	0	0	4.212
5	8000	87	0	0	8.284
6	16000	172	0	0	16.395
7	32000	344	0	0	32.870
8	64000	689	0	0	65.080
9	128000	1377	0	0	133.380
10	256000	2753	0	0	266.372
11	512000	?	?	?	?

TABLE 3.8: "A CANCEL request SHOULD NOT be sent to cancel a request other than INVITE"

current trace, y being the evaluation time of next trace), which means the complexity of evaluation is $O(n)$.

3.5.6 Discussions

In this section, we will elaborate some interesting phenomenons observed in the experiments, and also we will introduce some possible improvements for future works on performance testing. We will start this section with an interesting property.

Property: The session MUST be terminated after a BYE request
In this property, the **BYE** request is used to terminate a specific session or attempted session. When a **BYE** is received on a dialog, any session associated with that dialog SHOULD terminate. A UAC MUST NOT send a **BYE** outside of a dialog. Once the **BYE** is constructed, the UAC core creates a new non-INVITE client transaction and passes it to the **BYE** request. The UAC MUST consider the session terminated as soon as the **BYE** request is passed to the client transaction. If the response for the **BYE** is a 481 or a 408 or no response at all is received for the **BYE**, the UAC MUST consider the session and the dialog terminated. This property can be used to ensure that the IUT, once a dialog has been established, after sending a BYE request, the session MUST be terminated. As the 'terminated' is not clearly defined in the RFC, we define the 'terminated' as follows:

- The IUT stops sending messages.
- The IUT stops listening messages except the response for **BYE** request.

- The IUT transaction transmits to Completed state.

The **BYE** request must be constructed with a To header set to the same value as in the last received final response, the same Call-ID, From headers as in the original **INVITE** message, an incremented of one CSeq value and a method field in the CSeq header set to "BYE".

Differently as the properties before, this one is complicated to be formalized, due to the difficulty of detecting the 'terminated' state. Indeed, in our case we do not have any complete formal specification available and we can not stimulate the IUT. Moreover, we should ensure that no more messages will be exchanged after the 'terminated' state, which indicates that we need to keep monitoring the transaction even after it terminates. It is time consuming and unpredictable.

Time complexity

In the experiment, we observe a phenomenon which occurred in all the properties. The time complexity of evaluation is proportional to the number of inconclusive verdicts. Take property 1 and its results for example, its worst time complexity of evaluation is $O(n^2)$ (where n is the number of packets). If the variable n is doubled, the expected evaluation time should be 4 times greater than the previous one. However, from the Table 3.2, we can see there is hardly any inconclusive verdict, and the actual evaluation time is only about twice greater than the previous one, as the Figure 3.5 shows. This means that the actual time complexity in the evaluation is close to its best complexity $O(n)$.

In addition, we test the same property with the same number of traces where numerous inconclusive verdicts can be observed (the inconclusive verdicts accounted for 100% of the total verdicts). The result can be seen from Figure 3.6, which illustrates the evaluation time practically equal to our expected time. In other words, the actual time complexity of evaluation is almost equal to $O(n^2)$. This phenomenon can be used to estimate the evaluation time and the number of inconclusive verdicts in the future.

Integration for Performance Testing

We also found out some possible improvements for integrating our approach to performance testing when we worked on the relevant RFCs. As defined in the RFC1242 [65] and RFC2544 [66], the performance indicators can be indicated as:

- Accessibility: whether the packet can reach a destination.
- Communication bandwidth: the data transfer rate between two nodes.

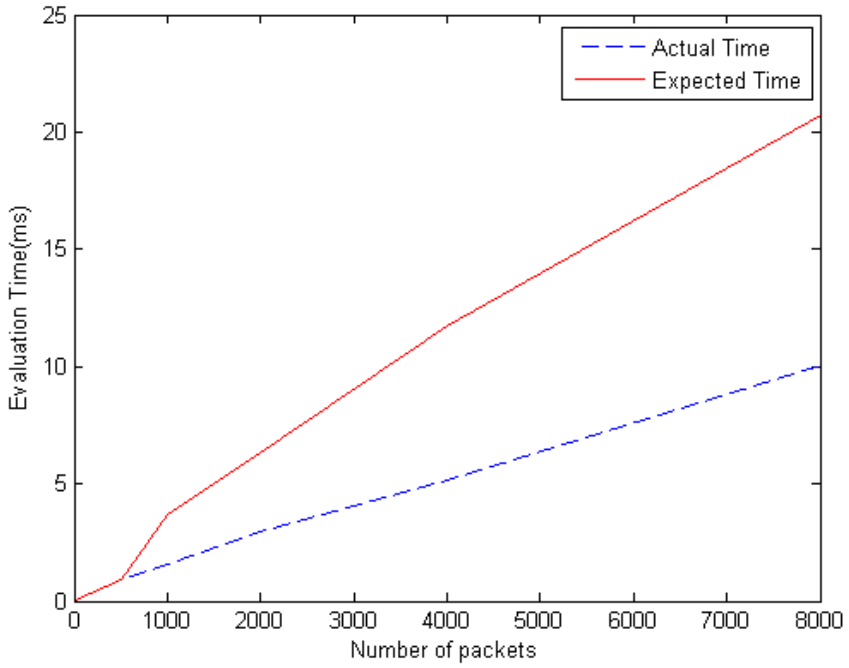


FIGURE 3.5: Evaluation Time Table

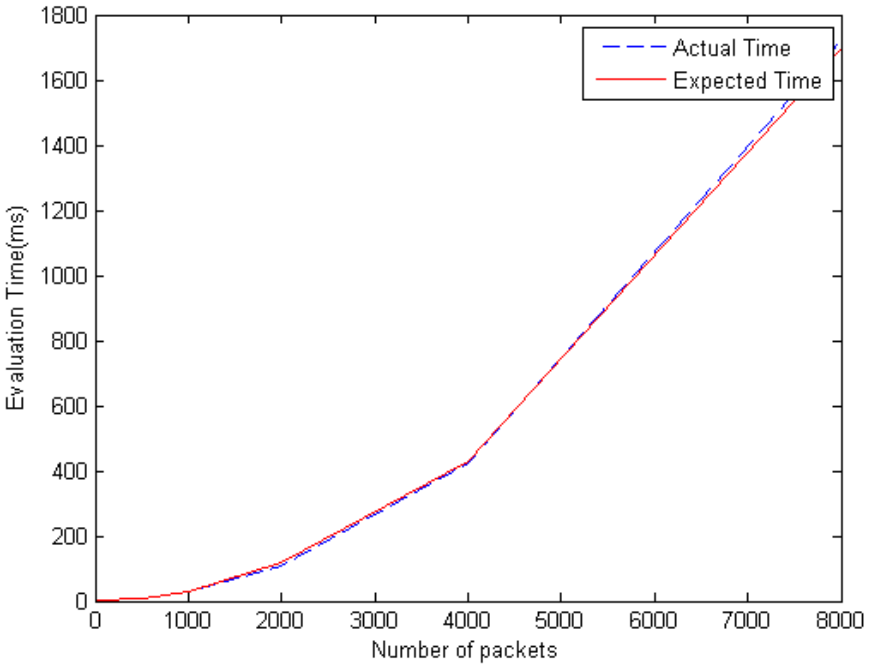


FIGURE 3.6: The evaluation time table of numerous inconclusive verdicts

- Maximum frame rate: the maximum transmission rate of the device under test.
- Communication latency: the time required for delivering the packet to destination.

- Frame loss rate: the ratio of loss packets and sent packets during the data transport.

We can integrate our approach to performance testing if we are able to measure all these norms. Currently in our approach, the result of property 1 already showed that we could certainly test the Accessibility and Frame loss rate by detecting the number of resent packets from the inconclusive verdicts. However, if we want to test the Communication latency, Communication bandwidth and Maximum frame rate, a *timer* function need to be added in our approach in order to test the arrival time.

3.5.7 Conclusion

In this chapter, we presented our initial approach for conformance testing of IMS services, through a real communicating environment. The results are positive, the implemented approach allows to define and test complex data relations efficiently, and evaluate the properties successfully. Besides, as described in the Section 3.4.6, some improvements are proposed as future works for performance testing. Meanwhile, we firstly published our preliminary work in [67] with the syntax and semantics, and then we published our following work in [55] with completed algorithm and comprehensive experiment results.

Chapter 4

Formal Approach for Performance Testing

"An investment in knowledge always pays the best interest."
– Benjamin Franklin (1706 – 1790)

In the previous chapter, we elaborated how our approach works on conformance testing, and also we raised some interesting challenges. Since many performance related properties cannot be specified, and many benefits can be brought to the test process if both conformance and performance testing inherit from the same approach, here we extend our proposed methodology to present a passive performance testing approach for communicating protocols based on the formal specification of the time related requirements. Also for solving the indeterminacy problems existed in non-positive verdicts, we introduce a four-valued semantics {'*Pass*', '*Con-Fail*', '*Per-Fail*', '*Inconclusive*'} in our formalism. Finally, we implement our approach in a complex network environment to test its functionality and flexibility. In this chapter, we will introduce these works into details.

4.1 Performance Testing

In our previous work, some interesting issues have been raised on how to test the communication latency and how to observe the duration of an interaction. These issues are mainly due to the fact we did not consider time constraints in the tested protocol properties. We introduce here some timing aspects which answer to these issues.

4.1.1 Basics and Syntax

For each $m \in M_p$, we add a real number $t_m \in \mathbb{R}^+$ which represents the time when the message m is received or sent by the monitored entity.

Example 3. A possible message for the SIP protocol, specified using the previous definition could be

$$m = \{(method, \textbf{`INVITE'}), (time, \text{`644.294133000'}), (status, \epsilon), (from, \\ \text{`alice@a.org'}), (to, \text{`bob@b.org'}), (cseq, \{(num, 7), (method, \textbf{`INVITE'})\})\}$$

representing an INVITE request from *alice@a.org* to *bob@b.org*. The value of *time* '644.294133000' is a relative value ($t_0 + 644.294133000$) since the P.O started its timer (initial value t_0) when capturing traces.

A *trace* ρ is a sequence of messages of the same domain containing the interactions of a monitored entity in a network, through an interface (the P.O), with one or more peers during an arbitrary period of time. The P.O also provides the relative time set $T \subset \mathbb{R}^+$ for all messages m in each *trace* ρ.

The *timed atom* is a particular atom defined as $p(\overbrace{term_t, ..., term_t}^{k})$, where $term_t \in T$.

Example 4. Let us consider the message m of the previous example, a time constraint on m can be defined as 'm.time $<$ 550ms'. By using this definition, requirements relevant to timing aspects can be formalized to atoms, which can be used to solve the problems mentioned in previous chapter.

4.1.2 Semantics

In conformance testing, since a finite trace is a finite segment of an infinite execution, it is not possible to declare a '$\top$' ('*Pass*') result for $\forall_x \phi$ as in the infinite case since we do not know if a '$\bot$' ('*Fail*') may come after the end of ρ. Equivalently, for $\exists_x \phi$, it is unknown whether '$\bot$' ('*Fail*') in ϕ becomes '$\top$' ('*Pass*') for future values of x. The semantics for trace quantifiers requires then the introduction of a new truth value '?' (inconclusive) to indicate that no definite response can be provided.

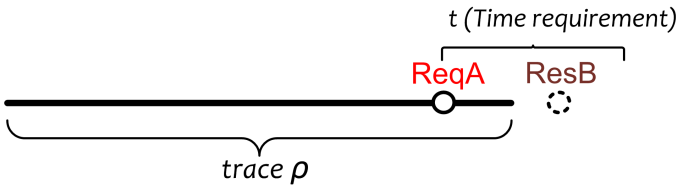


FIGURE 4.1: "Inconclusive" condition in performance testing. (ReqA is a request message in trace ρ, ResB is a possible response to ReqA in the future trace and t represents the time requirement bound)

However, different from conformance testing, performance requirements in performance testing have strict required time bounds. It indicates that there is no indeterminacy in semantics of quantifiers for finite traces, except one condition: the bounds of time requirements exceed the end of finite traces (as shown in Figure 4.1). In this case, '?' (inconclusive) verdicts will be used for indicating that no definite response can be provided. In [57], it is proved that if M is a *minimal* model[1] for a clause set K and a trace ρ, if $M(\phi) = \top(Pass)$, then ϕ is '$\top$' ('*Pass*') for K and ρ. Let operator $\hat{M}$ be the semantics of formulas, and the semantics of quantifiers $\forall_x$ and $\exists_x$ are redefined as follows:

$$\hat{M}(\forall_x \phi) = \begin{cases} \top(Pass) & \text{if } \hat{M}(\phi\theta) = \top, \forall\theta \text{ where } x/m \in \theta \text{ and } m \in \rho \\ \bot\ (Fail) & \text{if } \exists\theta \text{ with } x/m \in \theta \text{ and } m \in \rho, \text{where } \hat{M}(\phi\theta) = \bot \\ ? & \text{otherwise} \end{cases}$$

$$\hat{M}(\exists_x \phi) = \begin{cases} \top(Pass) & \text{if } \exists\theta \text{ with } x/m \in \theta \text{ and } m \in \rho, \text{where } \hat{M}(\phi\theta) = \top \\ \bot\ (Fail) & \text{if } \hat{M}(\phi\theta) = \bot, \forall\theta \text{ where } x/m \in \theta \text{ and } m \in \rho \\ ? & \text{otherwise} \end{cases}$$

The semantics for $\forall_{y>x}$, $\forall_{y<x}$, $\exists_{y>x}$ and $\exists_{y<x}$ are equivalent to the two formulas shown above. Based on the semantics, an algorithm for evaluating formulas is provided in Section 4.1.4.

4.1.3 Performance testing verdicts

We formalize the performance requirements of the IUT by using the syntax above described, and the truth values {*Pass*, *Fail*, ?} are provided to the interpretation of the obtained formulas on real protocol execution traces. However, from our analysis of standards [66] [7], most of the performance requirements are composed of conformance

[1] Obtained as $\cap M$, the intersection of all models for K and ρ

requirements and strict time constraints. By using the introduced syntax before, we define the formalized performance requirement, the constituent conformance requirement, the set of formalized performance requirements and constituent conformance requirements by using ϕ_{con}, ϕ_{per}, R_{per} and R_{con} respectively.

Definition 6. The set of constituent conformance requirements R_{con} is defined as $R_{con} = \{\phi_{con_1}, \phi_{con_2}, ..., \phi_{con_n} \mid n \in \mathbb{N}\}$, where $\phi_{con_i} = \{(A_1 \wedge A_2 \wedge ... \wedge A_m) \mid m \in \mathbb{N} \mid i = 1...n\}$. The set of performance requirements R_{per} can be defined as $R_{per} = \{\phi_{per_1}, \phi_{per_2}, ..., \phi_{per_n} \mid n \in \mathbb{N}\}$, where $\phi_{per_i} = \{\phi_{con_i} \wedge (A_{t_1} \wedge ... \wedge A_{t_k}) \mid k \in \mathbb{N}\}$ and $A_{t_i} = p(term_t, ..., term_t)$.

Example 4. The performance requirement "*the message response time should be less than 5ms*", represented by formalized formula $\phi_{per_1} = \forall_x(request(x) \rightarrow \exists_{y>x}(response(y,x) \wedge within time(x,y,5ms)))$, is composed of the conformance requirement "*The SUT receives a response message*" represented by formalized formula $\phi_{con_1} = \forall_x(request(x) \rightarrow \exists_{y>x}(response(y,x)))$, and a required time constraint $within time(x,y,5ms) = term_{t_y} - term_{t_x} < 5ms$.

Once a '*Pass*' truth value is given to a performance requirement, without doubt, both the performance and conformance requirements are satisfied. In the Example 4, if a '*Pass*' is given to ϕ_{per_1}, it means the SUT received a response message and the response time of this message is less than 5ms, the constituent ϕ_{con_1} and $within time(x,y,5ms)$ are also sufficed.

However, if a '*Fail*' truth value is returned to a performance requirement, we cannot find out the real cause of it. Since we cannot distinguish whether it is due to the violation of the constituent conformance requirement or the required time constraint. For instance, in Example 4, if a '*Fail*' is given to ϕ_{per_1}, we cannot distinguish whether it is due to "*The SUT received a response message, but the response time is greater than 5ms*", "*There is an error in the data portion of this response message*" or "*The SUT never received a response message*".

Due to these issues, a method is required for clearly differentiating the cause of non-pass results. Since in our approach, we have a common methodological ground for conformance testing and performance testing, we can simultaneously test performance requirements and their constituent conformance requirements, and then combine the obtained verdicts together to have further analysis. Let $comb(\phi_{per}, \phi_{con})$ represents

the combination of verdicts obtained from performance requirements and conformance requirements. It follows the rules shown in Table 4.1.

ϕ_{per}	Pass	Pass	Pass	Fail	Fail	Fail	?	?	?
ϕ_{con}	Pass	Fail	?	Pass	Fail	?	Pass	Fail	?
$comb(\phi_{per}, \phi_{con})$	Pass	Con-Fail	Pass	Per-Fail	Con-Fail	Per-Fail	?	Con-Fail	?

TABLE 4.1: Combinations of conformance and performance testing verdicts

Inspired from the work of [27] who defined a four-valued semantics for LTL to better explain "*Inconclusive*" verdicts, we introduce two new definitions of verdicts "*Con-Fail*" and "*Per-Fail*" in our performance testing formalism for better explaining "*Fail*" verdicts. "*Con-Fail*" represents the failures caused by the constituent conformance requirements, and "*Per-Fail*" represents the failures truly caused by the violation of the performance requirements. As shown in the table, "*Fail*" verdicts finally are separated to these two kinds of verdicts according to their causes. Followed with the rules in Table 4.1, the semantics of operator '$comb(\phi_{per}, \phi_{con})$' can be formally defined as follows

$$comb(\phi_{per}, \phi_{con}) = \begin{cases} Pass & \text{if } eval(\phi_{per}, \theta, \rho) = Pass \mid \phi_{per} \in R_{per} \text{ and} \\ & eval(\phi_{con}, \theta', \rho) = Pass \text{ or } ? \mid \phi_{con} \in R_{con} \\ Con\text{-}Fail & \text{if } eval(\phi_{con}, \theta, \rho) = Fail \mid \phi_{con} \in R_{con} \\ Per\text{-}Fail & \text{if } eval(\phi_{per}, \theta, \rho) = Fail \mid \phi_{per} \in R_{per} \text{ and} \\ & eval(\phi_{con}, \theta', \rho) = Pass \text{ or } ? \mid \phi_{con} \in R_{con} \\ ? & \text{otherwise} \end{cases}$$

where $eval(\phi, \theta, \rho)$ expresses the evaluation of a formula ϕ on a finite trace ρ, by using the substitution θ for performance requirements and θ' for conformance requirements.

4.1.4 Evaluating Algorithm

We use a recursive algorithm for evaluating the formalized performance requirements on a real trace. It is coupled with a modification of the Selective Linear Definite-clause (SLD) resolution algorithm [58] for the evaluation of Horn clauses presented in our Algorithm 3. The algorithm starts by checking the existence of a trace ρ and a formalized performance requirement ϕ_{per}. If any of it does not exist, the algorithm will terminate since nothing can be tested (Line 1-2, 33-34). Then if ϕ_{per} contains sub formulas, they will be sequentially tested by using recursive calls (Line 4, 30). For testing a formula ϕ on a finite trace ρ, the algorithm will first assign the values to substitution θ from each message m in the trace (Line 6). Then the obtained θ' will be used to

Algorithm 2: Algorithm for eval(ϕ_{per}, θ, ρ)

Input: Formalized requirement ϕ_{per}, Substitution θ with initial bindings, and finite trace ρ
Output: *Pass* if the formula has a solution, *Fail* if exist a violation of the requirements, '?' if no definite response can be provided

```
1  if ρ is not empty, φ_per is not empty then
2      verdict ← ⊤;
3      if φ_per = φ1 → φ2 then
4          for (m0 ∧ ... ∧ mn) ∈ ρ do
5              θ' ← θm_i;
6              for (C0 ∧ ... ∧ Cn) ∈ φ1 where verdict ≠⊥ do
7                  for (A0 ∧ ... ∧ An) ∈ C_j where verdict ≠⊥ do
8                      if θ'A0 ∧ ... ∧ θ'An = ⊤ then
9                          verdict ← verdict ∧ ⊤, next C_j, return verdict;
10                     end
11                     else
12                         verdict ← verdict∧ ⊥, return verdict;
13                     end
14                 end
15             end
16             if verdict = ⊤ then
17                 next m_i, logfile ← pass, return logfile;
18             end
19             else
20                 logfile ← m_i, next m_i, check_end_of_file(m_i);
21                 if check_end_of_file(m_i) = ⊤ then
22                     logfile ← inconclusive, return logfile;
23                 end
24                 else
25                     logfile ← fail, return logfile;
26                 end
27             end
28         end
29         save logfile(φ1), eval(φ2, θ, ρ);
30     end
31     Go to loop for (m0 ∧ ... ∧ mn) ∈ ρ with φ1=φ_per, return logfile;
32 end
```

compare with each atom in the formula ϕ_{per} (Line 7-9). If all the atoms in ϕ_{per} are satisfied, a truth value '$\top$' will be assigned and the algorithm will step to test the next message m. Otherwise, any violation of the atoms will result to a truth value '$\bot$' and the algorithm will immediately terminate the comparing process and step to test the next message m (Line 10-14). The truth values '$\top$' and '$\bot$' will be eventually transformed to the verdict '*Pass*', '*Fail*' or '*Inconclusive*' as the semantics defined in Section 3 (Line 17-26). Finally, a final report will be provided when all the sub formulas of ϕ_{per} are tested through the trace ρ.

Similar to the algorithm we used in conformance testing, the complexity of the algorithm is decided by the number of quantifiers used in the formula being tested. In the worst-case, the time complexity of our algorithm with k quantifiers is still $\mathcal{O}(n^k)$ to analyze the full trace, where n represents the number of messages in the trace. Although the complexity seems high, it should be emphasized that it is the worst case complexity,

where the evaluation of every quantifier for a trace returns '?' and '$\perp$'. And also the complexity corresponds to an analysis of the whole trace, not for obtaining individual solutions, which depends on the type of quantifiers used. For instance, for a property $\forall_x request(x)$, individual results are obtained in $\mathcal{O}(1)$.

4.1.5 Experiments

In this section, our approach has been implemented into an IMS framework. We provide some experiments results evaluated on large traces, in order to verify the functionality of our approach.

4.1.5.1 Environments

In our experiments, SIPp is still used for obtaining traces. Different from the wired LAN environment used in previous conformance testing experiments, a simple ad-hoc based wireless environment has been implemented and tested here. Since compared with wired network, wireless ad-hoc network supports mobility and freedom in the networks, and it has been widely used in personal area network (PAN) and wireless sensor networks (WSNs). The ad-hoc technology almost has been implemented to all the new released laptops and cellphones. Using the ad-hoc based wireless environment here, can make our experiment more close to the real daily-life communication environment.

Unlike wired transmission, the wireless transmission in ad-hoc may deal with problems caused by the characteristic of the electronic wave. The obstacles existing in the physical environment can cause shadowing, reflection, scattering, fading, refraction, diffraction of the wave. These propagation may lead to transmitted packets being garbled and thus received in error, which satisfy our need of variability on the data traffic. Data errors could happen in the experiments. Besides, the characteristic of wave prevents wireless communication to transmit data better than wired communication. In other words, the ad-hoc networks have lower data transmission rate. Since it is our preliminary experiments on performance testing, the aim of the experiments is to test the accuracy and the functionality of our approach. The ad-hoc networks still satisfy our needs, and also it is the reason why the sizes of trace sets in the experiments are much smaller than the ones in conformance testing. The structure of our environment is shown in Figure 4.2.

The experiments have been performed on two laptops (2.5GHz Intel Core 2 Duo with 4GB RAM and 2GHz Intel Core 2 Due with 2GB RAM) and a Table PC (2.5GHz AMD Duo Core with 2GB RAM). The laptop with higher specification plays the role of User

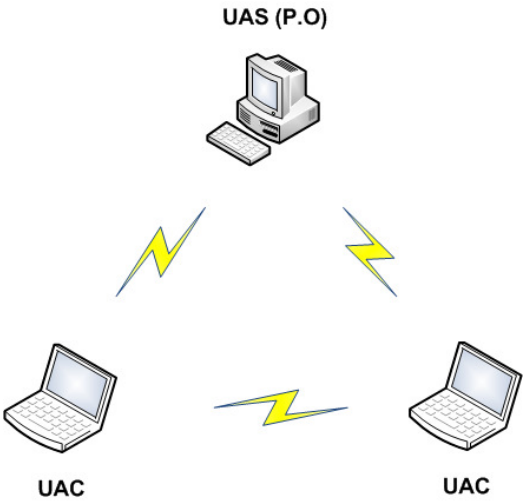


FIGURE 4.2: Ad-hoc environment

Agent Server (UAS), while the other two machines act as User Agent Clients (UAC). The traffic is obtained from the P.O (UAS) by using monitoring tool Wireshark[2]. In the traces we collected, only the information of Session Layer has been used in our experiments. The traces collected in different environments can be found at http://www-public.it-sudparis.eu/~che_xiao/TSPSIPpOption.html.

We simulate two scenarios for the data traffic: one is the data traffic under normal condition (called **normal** for short), which means sufficient bandwidth is provided and quite few re-transmissions occur; while the other one is under high data traffic congestion (called **high** for short), which simulates the condition that numerous users are calling at the same time, where numerous re-transmissions and packet-losses occur. Several sets of traces under **normal** and **high** conditions have been collected for the following experiments.

In the following subsections, performance properties collected from RFC 3261 are formalized to formulas. They are evaluated through numerous execution traces, and the testing verdicts {*Pass, Con-Fail, Per-Fail, Inconclusive*} are provided in the following.

4.1.5.2 Properties and Results

Property 1: For every request there must be a response, each response should be received within 0.5 s

This property can be used for a monitoring purpose, which reflects the current traffic latency condition. By using the syntax mentioned in the Section 3, this performance property consists of a conformance property '*For every request there must be a response*'

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

and a time requirement '*Response should be received within 0.5 s*', where the conformance property can be formalized as follows ϕ_{con_1}:

$$\begin{aligned} &\forall_x(request(x) \wedge x.method\,! = \textbf{'ACK'} \\ &\rightarrow \exists_{y>x}(nonProvisional(y) \wedge responds(y,x))) \end{aligned} \tag{4.1}$$

where $nonProvisional(x)$ accepts all non provisional responses (with $status \geq 200$) to requests with method different than **ACK**, which does not require a response. The response time for each request is a crucial indicator for performance, and based on the previous formula, the performance property can be formalized as follows ϕ_{per_1} (let t be the value of strict time requirement):

$$\begin{aligned} &\forall_x(request(x) \wedge x.method\,! = \textbf{'ACK'} \\ &\rightarrow \exists_{y>x}(nonProvisional(y) \wedge responds(y,x) \wedge withintime(x,y,t))) \end{aligned} \tag{4.2}$$

where $withintime$ is defined as

$$withintime(x,y,t) \leftarrow y.time < x.time + 0.5s$$

Initially, the properties have been tested through the **normal** traces, the results are as follows.

Trace	Messages	ϕ_{con_1}			ϕ_{per_1}			Time(s)
		#Pass	#Fail	#Inc.	#Pass	#Fail	#Inc.	
1	500	150	0	0	150	0	0	1.468
2	1000	318	0	1	318	0	1	1.714
3	1500	504	0	1	504	0	1	2.335
4	2000	674	0	0	674	0	0	2.919
5	2500	798	0	1	798	0	1	3.576

TABLE 4.2: Test results for "For every request there must be a response" and"For every request there must be a response within 0.5 s " (normal)

Trace	Messages	$comb(\phi_{per_1}, \phi_{con_1})$			
		#Pass	#Con-Fail	#Per-Fail	#Inc.
1	500	150	0	0	0
2	1000	318	0	0	1
3	1500	504	0	0	1
4	2000	674	0	0	0
5	2500	798	0	0	1

TABLE 4.3: Final results for "For every request there must be a response within 0.5 s " (normal)

As expected, most of the results shown in Table 4.2 show only '*Pass*' verdicts for the property evaluation. The column "Time" represents the evaluation time of each trace.

However, as shown in Table 4.3, still three inconclusive verdicts can be observed after the combination of both conformance and performance results. Thoroughly looking at trace 2, this inconclusive verdict corresponds to the **INVITE** request message, this message is at the end of the trace, which could indicate that the client closed the connection before receiving the corresponding response message. The same phenomenon happens for trace 3 and 5.

After analyzing the traces under **normal** condition, we step to test the traces under **high** condition, the results after combination are shown in the Figure 4.3. As mentioned before, 'ACK' requests are considered as irrelevant messages for this property, and they account for the rest proportion of messages which are not shown in the figure. Non-positive verdicts (i.e. that are not *Pass*) can be observed from the results. The '*Inconclusive*' verdicts indicate the messages can not be determined which are at the end of the trace, while the '*Per-Fail*' verdicts indicate the response messages received by the SUT but exceed the expected time 't=0.5 s'. More crucially, different from no '*Fail*' verdict in the previous **normal** condition, numbers of '*Per-Fail*' verdicts can be observed in the **high** condition, but no '*Con-Fail*' verdict has been found. They indicate that the traffic is in a high latency situation.

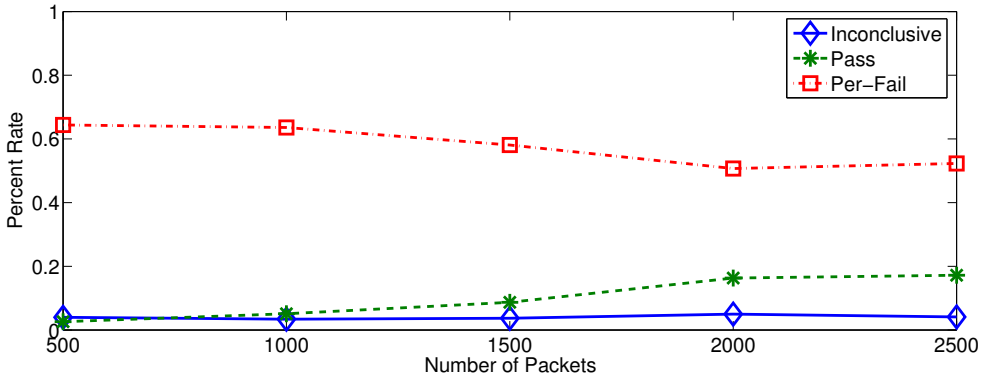


FIGURE 4.3: "For every request there must be a response with in t = 0.5s" (high)

Besides, since in SIP, different values of t correspond to different frequencies of re-transmission of a message, varying the value of t can be used to detect the frequency of re-transmission of specific messages. As illustrated in Figure 4.4, for each trace, the distribution map of the response times is depicted explicitly by different colored bars. These bars not only represent the response times, but also correspond to different re-transmission times, where t=0.5s denotes no re-transmission, t=1s denotes one re-transmission and so on, until the maximum timer t=16s denotes five re-transmissions. Let us have a look at trace 1 (500 packets) for example, the bars illustrate that 61% tested messages are responded within 0.5s while the rest 39% are responded between 0.5s and 1s.

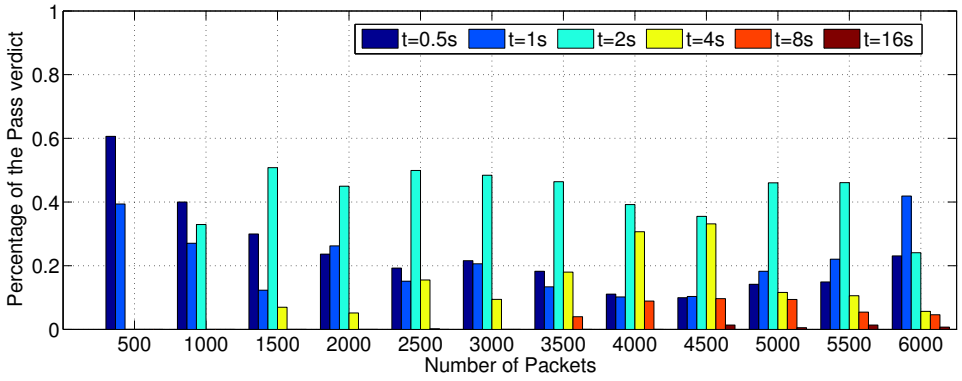


FIGURE 4.4: *Pass* percentage for different time intervals. (The x-axis represents different traces, the y-axis roughly estimates the percentage of *Pass* verdicts)

Moreover, from the figure, more re-transmissions can be observed in the latter traces, which conform to the condition when we collected the traces. Since the traces are collected at the beginning of the Ad-hoc communications, as time goes on, more traffic congestion occurs in the environment, which leads to more re-transmissions and larger latency of the response messages. It can be observed that response time are getting faster after the trace of 4500 messages, which indicates the traffic condition in Ad-hoc network is getting better at that period.

Property 2: Session Establishment Duration

This performance property is used for monitoring the time duration of establishing a session. It is based on the establishment of a session which can be formalized to a conformance property ϕ_{con_2} '*For each INVITE request, there should be a 2xx response if the session has been successfully established*', as follows:

$$\begin{aligned} &\forall_x(request(x) \wedge x.method = \textbf{'INVITE'} \\ &\rightarrow \exists_{y>x}(response(y,x) \wedge y.statusCode = 200)) \end{aligned} \tag{4.3}$$

and the performance property ϕ_{per_2} "Session Establishment Duration" (with $t = 1.5s$) can be expressed as:

$$\begin{aligned} &\forall_x(request(x) \wedge x.method = \textbf{'INVITE'} \\ &\rightarrow \exists_{y>x}(response(y,x) \wedge y.statusCode = 200 \wedge withintime(x,y,t))) \end{aligned} \tag{4.4}$$

Initially, we still use the **normal** traces to test the properties, the results are as follows.

From the Table 4.4, numbers of non-positive verdicts can be observed. As we penetrate deeply into these verdicts with Table 4.5, all the '*Fail*' verdicts are caused by the violation of required time constraint 't=1.5 s', and they are concluded as '*Per-Fail*'.

Trace	Messages	ϕ_{con_2}			ϕ_{per_2}			Time(s)
		#Pass	#Fail	#Inc.	#Pass	#Fail	#Inc.	
1	500	60	0	10	60	9	1	1.488
2	1000	109	0	18	109	15	3	3.210
3	1500	139	0	37	139	32	5	6.768
4	2000	180	0	53	181	52	2	10.971
5	2500	267	0	51	260	53	5	14.548

TABLE 4.4: Test results for "For each INVITE request there should be a 2xx response" and "For each INVITE request there should be a 2xx response, within 1.5s" (normal)

Trace	Messages	$comb(\phi_{per_2}, \phi_{con_2})$			
		#Pass	#Con-Fail	#Per-Fail	#Inc.
1	500	60	0	9	1
2	1000	109	0	15	3
3	1500	139	0	32	5
4	2000	180	0	52	2
5	2500	260	0	53	5

TABLE 4.5: Final results for "For each INVITE request there should be a 2xx response within 1.5 s" (normal)

However, different from the previous property, we can observe that most of them are due to the packet-loss during the transmission, only one '*Per-Fail*' verdict in trace 4 and seven '*Per-Fail*' verdicts in trace 5 are truly caused by their large latency. Although it seems that there is no apparent difference between the results returned for ϕ_{per_2} and $comb(\phi_{per_2}, \phi_{con_2})$, the combination of verdicts provide the availability for further analyses which can help us to have a precise result on other relevant performance properties, such as packet-loss or average latency, etc. Similarly as the previous property, several "*Inconclusive*" verdicts are observed and they are still caused by the same reason. "Unfortunately", still no '*Con-Fail*' verdict can be observed under this **normal** condition.

For drawing further conclusions, we test this property under **high** conditions. Likewise, all the requests and responses other than 'INVITE' and its response are considered as irrelevant messages, which account for the rest proportion of total messages. As the Figure 4.5 shows, many '*Fail*' verdicts are returned for the traces. It is mainly due to the high traffic congestion in the Ad-hoc network environment. It is worthwhile to note that one '*Con-Fail*' verdict is observed in the trace of 1500 messages. This verdict is caused by an error in the data portion of a response message. When we take a closer look at the message, the data error is due to an unexpected byte in the data portion. As we introduced in section 4.1.5.1, it is very likely that this phenomenon is caused by the impact of impulsive noise on the electronic wave. Besides, the '*Per-Fail*' verdicts are still mostly due to packet-losses, and few are caused by the violation of required time $t = 1.5s$. Nevertheless, the variety of non-positive verdicts obtained eventually

proves the functionality of our approach. It can precisely detect all kinds of failures as mentioned in Section 3.

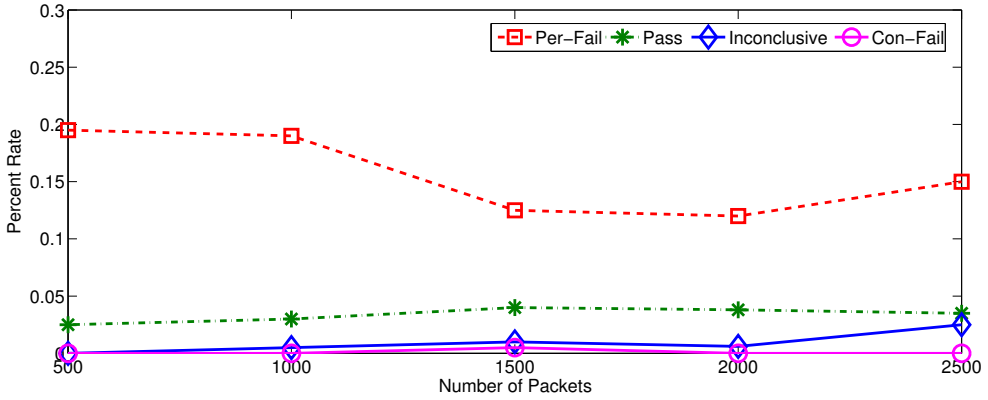


FIGURE 4.5: "For each INVITE request, there should be a 2xx response within $t = 1.5s$" (high)

Property 3: Registration Duration Measurement

This performance property is used for monitoring the time duration of successful registrations. It is based on a session which begins with 'REGISTER' request and ends with a 200 response, which can be formalized to a conformance property ϕ_{con_3} '*For each successfully registration, it should begin with a REGISTER request and end with a 200 response*', as follows:

$$\begin{aligned} &\forall(request(x) \wedge x.method = \textbf{'REGISTER'} \\ &\rightarrow \exists_{y>x}(success(y) \wedge responds(y,x)))) \end{aligned} \tag{4.5}$$

and the performance property ϕ_{per_3} "Registration Duration should less than $t = 1s$" can be expressed as:

$$\begin{aligned} &\forall(request(x) \wedge x.method = \textbf{'REGISTER'} \\ &\rightarrow \exists_{y>x}(success(y) \wedge responds(y,x) \wedge withintime(x,y,t))) \end{aligned} \tag{4.6}$$

From the evaluation results of previous properties, it can be concluded that compared to the **normal** condition, more diversified results can be observed in the **high** condition. Therefore, for this property, we only illustrate the verdicts obtained from the traces in **high** condition, as Tables 4.6 and 4.7 illustrate below.

From the Table 4.6, numerous '*Fail*' verdicts can be observed in the traces. Moreover, obvious differences between the obtained '*Pass*' verdicts in ϕ_{con_3} and ϕ_{per_3} can be observed. After the combination process, all the "*Fail*" verdicts are separated into different sets according to their causes. Due to the high congestion and packet-loss in the network

Trace	Messages	ϕ_{con_3}			ϕ_{per_3}			Time(s)
		#Pass	#Fail	#Inc.	#Pass	#Fail	#Inc.	
1	500	18	0	66	18	40	26	2.534
2	1000	31	0	149	20	117	43	7.983
3	1500	62	0	201	24	184	55	16.952
4	2000	180	1	219	81	243	75	30.792
5	2500	222	5	373	63	421	116	69.042

TABLE 4.6: Test results for "For each successfully registration, it should begin with a REGISTER request and end with a 200 response" and "For each successfully registration, the duration should be within 1 s" (high)

Trace	Messages	$comb(\phi_{per_3}, \phi_{con_3})$			
		#Pass	#Con-Fail	#Per-Fail	#Inc.
1	500	18	0	40	26
2	1000	31	0	117	43
3	1500	62	0	184	55
4	2000	81	1	243	75
5	2500	63	5	421	116

TABLE 4.7: Final results for "For each successfully registration, the duration should be within 1 s" (high)

environment, numbers of '*Per-Fail*' verdicts are returned indicating that lots of registration duration exceeded the time requirement 1s. This also well explains the differences between '*Pass*' verdicts in ϕ_{con_3} and ϕ_{per_3}. Although some registrations exceeded the time requirement, they still satisfy the conformance requirement and should be assigned to '*Pass*' verdicts in ϕ_{con_3}. Besides, several '*Con-Fail*' verdicts can be observed in trace 4 and 5, they are caused by unexpected bytes because of the same reason mentioned in property 2. We also expand the experiment by testing continuous traces while varying the time requirement t, a bar chart of the '*Pass*' verdicts in different time intervals can be concluded in Figure 4.6.

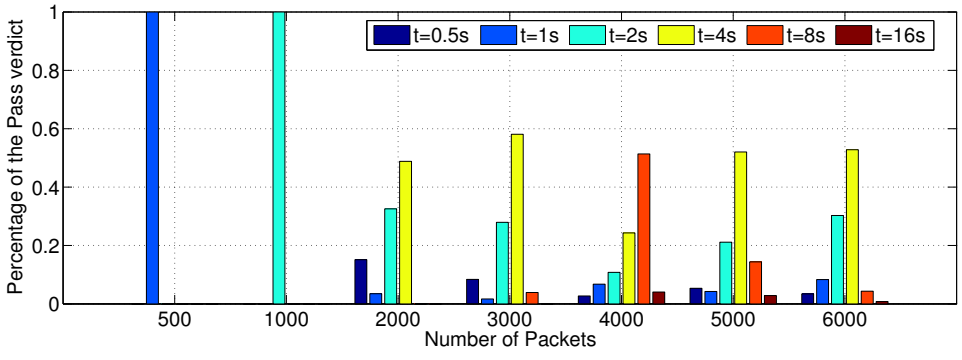


FIGURE 4.6: *Pass* percentage for different time intervals. (The x-axis denotes the different traces, the y-axis roughly represents the percentage of *Pass* verdicts)

Similarly to property 2, this figure explicitly illustrates the registration performance by measuring the percentage of '*Pass*' verdicts in different time intervals. For instance,

in the trace of 500 messages, all of the registrations are performed within 1s; while in the trace of 2000 messages, only less than 20% registrations satisfy this condition, the rest are either in the range $[1s, 2s]$ or $[2s, 4s]$. The more '*Pass*' verdicts appear in the rearward time intervals, which means larger response time of the messages, the worse registration performance will be.

4.1.5.3 Discussions

In this section, a performance benchmark system for SIP is proposed according to RFC1242 [65] and RFC2544 [66], and some relevant results are discussed. Instead of simply measuring the global throughput and latency, they are extended into detailed measuring indicators: Session Attempt Number / Rate / Successful Rate, Session Establishment Number / Rate / Duration, Session Packet loss Number / Rate, Session Packet response Latency, Registration Number / Rate / Duration. Sessions are the basic testing unit we used here, due to the reason that they are the most crucial units of communications in SIP.

By using our approach introduced before, these indicators can be formalized to formulas. These formulas will be tested through the testers. After evaluating each formula ϕ on a trace ρ, N_p, N_{cf}, N_{pf} and N_{in} will be returned which represent the number of '*Pass*', '*Con-Fail*', '*Per-Fail*' and '*Inconclusive*' verdicts respectively. In the testing process, constituent conformance requirement ϕ_{con} is used for differentiating non-positive results. Besides, let t_{test} be the time used for capturing a trace ρ, which is the time duration between the first and the last captured messages, where $\rho = \{m_0, ..., m_n\}$. The definition of these symbols are shown below.

$$N_p(\phi) = \begin{cases} \sum[eval(\phi_{con}, \theta, \rho) = Pass \text{] if } \phi_{con} \in R_{con}] \\ \sum[comb(\phi_{per}, \phi_{con}) = Pass \text{] if } \phi_{per} \in R_{per} \text{ and } \phi_{con} \in R_{con}] \end{cases}$$

$$N_{cf}(\phi) = \begin{cases} \sum[eval(\phi_{con}, \theta, \rho) = Fail \text{] if } \phi_{con} \in R_{con}] \\ \sum[comb(\phi_{per}, \phi_{con}) = Con\text{-}Fail \text{] if } \phi_{per} \in R_{per} \text{ and } \phi_{con} \in R_{con}] \end{cases}$$

$$N_{pf}(\phi) = \sum[comb(\phi_{per}, \phi_{con}) = Per\text{-}Fail \text{] if } \phi_{per} \in R_{per} \text{ and } \phi_{con} \in R_{con}]$$

$$N_{in}(\phi) = \begin{cases} \sum[eval(\phi_{con}, \theta, \rho) = Inconclusive \text{] if } \phi_{con} \in R_{con}] \\ \sum[comb(\phi_{per}, \phi_{con}) = Inconclusive \text{] if } \phi_{per} \in R_{per} \text{ and } \phi_{con} \in R_{con}] \end{cases}$$

$$t_{test} = m_n.time - m_0.time$$

Except the formalized conformance and performance requirements mentioned in section 4 (ϕ_{con_1}, ϕ_{per_1}, ϕ_{con_2}, ϕ_{per_2}, ϕ_{con_3}, ϕ_{per_3}), the following requirements are also used for the indicators.

$$\phi_{con_4} = \begin{cases} \forall(request(x) \wedge x.method = \textbf{`INVITE'} \\ \rightarrow \exists_{y>x}(nonProvisional(y) \wedge responds(y,x))) \end{cases}$$

$$\phi_{con_5} = \begin{cases} \forall(request(x) \wedge x.method = \textbf{`REGISTER'} \\ \rightarrow \exists_{y>x}(nonProvisional(y) \wedge responds(y,x))) \end{cases}$$

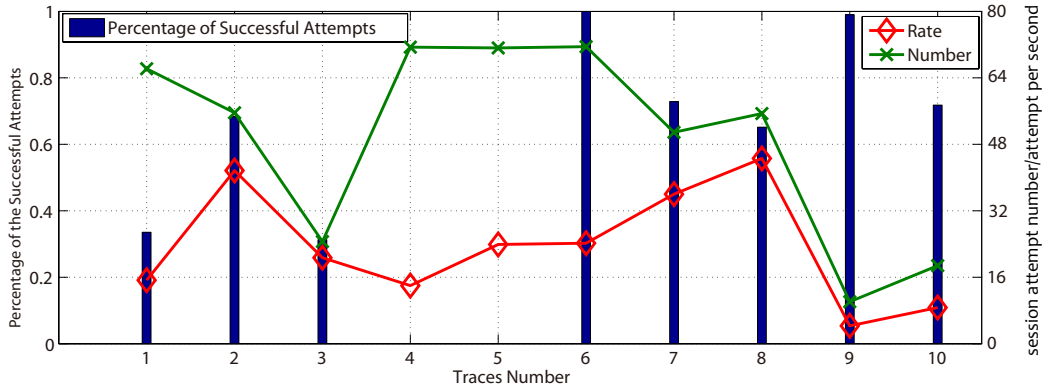
According to the definitions and the formulas above, these indicators can be formally described as:

- Session Attempt Indicators: Session Attempt Number $N_p(\phi_{con_4}) + N_{in}(\phi_{con_4})$, Session Attempt Rate $(N_p(\phi_{con_4}) + N_{in}(\phi_{con_4}))$ / t_{test}, Session Attempt Successful Rate $N_p(\phi_{con_2})$ / $(N_p(\phi_{con_4}) + N_{in}(\phi_{con_4}))$.
- Session Establishment Indicators: Session establishment Number $N_p(\phi_{con_2})$, Session establishment Rate $N_p(\phi_{con_2})$ / t_{test}, Session establishment Duration $N_p(\phi_{per_2})$.
- Session Global Indicators: Session Packet loss Number $N_{in}(\phi_{con_1})$, Session Packet loss Rate $N_{in}(\phi_{con_1})$ / $N_p(\phi_{con_1}) + N_{cf}(\phi_{con_1}) + N_{in}(\phi_{con_1})$, Session Packet latency $N_p(\phi_{per_1})$.
- Session Registration Indicators: Registration Number $N_p(\phi_{con_3})$, Registration Rate N_p (ϕ_{con_3}) / t_{test}, Registration Duration $N_p(\phi_{per_3})$.

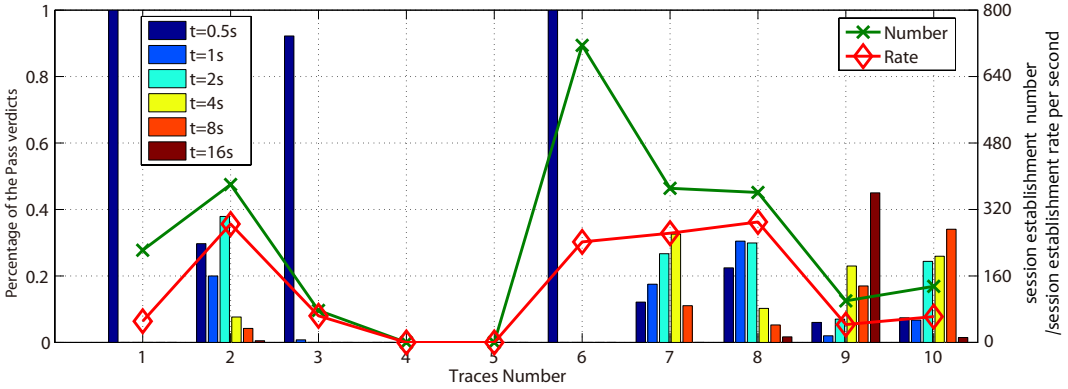
By using the formalized indicators above, an explicit performance analysis can be given to the trace being tested, as Figure 4.7 shows. Sampling from the traces of 50,000 messages, we obtained ten sets of 5000 messages for each. These sets have been analyzed in order to test the functionality and efficiency of our approach. In Figure 4.7(a), the histogram illustrates the percentages of the successful attempts of each trace, while the double color curves demonstrate the session attempt numbers and rates (per second). Then in Figure 4.7(b) and Figure 4.8(a)(b), the histograms display the distribution maps of time duration/latency of each trace, while the green curves (with cross) demonstrate the throughput numbers and the red curves (with diamond) represent the rates.

In addition, from Figure 4.7(a), the successful attempt rates of trace 4 and 5 are zero while the numbers/rates of attempts are not, which denotes that in these two traces, lots of session attempt requests are sent but none of them is responded with '200' success response. The '0' session establishment number of trace 4 and 5 in Figure 4.7(b) also proves this phenomenon. Meanwhile, in Figure 4.8(a)(b), the registration requests

FIGURE 4.7: Performance Indicators (1)



(a) Session Attempt Number/Rate/Successful Rate



(b) Session Establishment Number/Rate/Duration

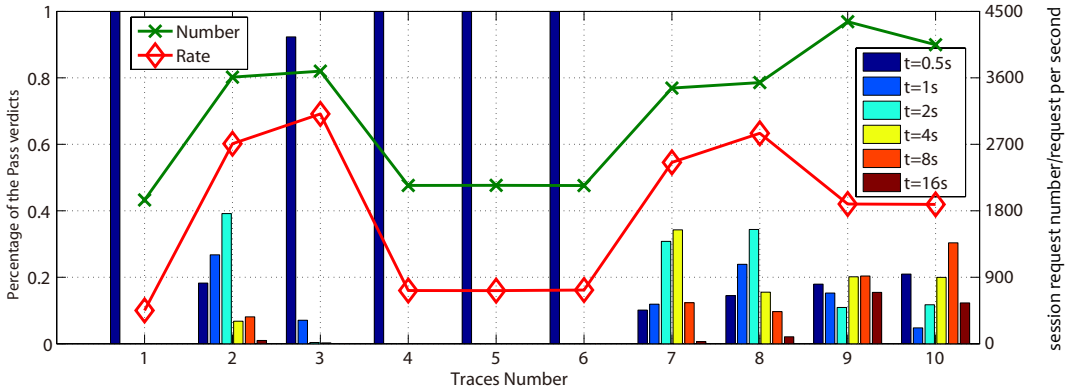
are responded quickly and not influenced. Which can be concluded that the low performance of session establishment in trace 4 and 5 is due to this service of the server reach to the maximum load, rather than the massive packet loss during the transmission. Note that the results of each trace are obtained in a brief time. The aggregate results illustrated in the figures show that our approach is suitable for complex performance testing environment with numerous specific performance requirements.

Future works

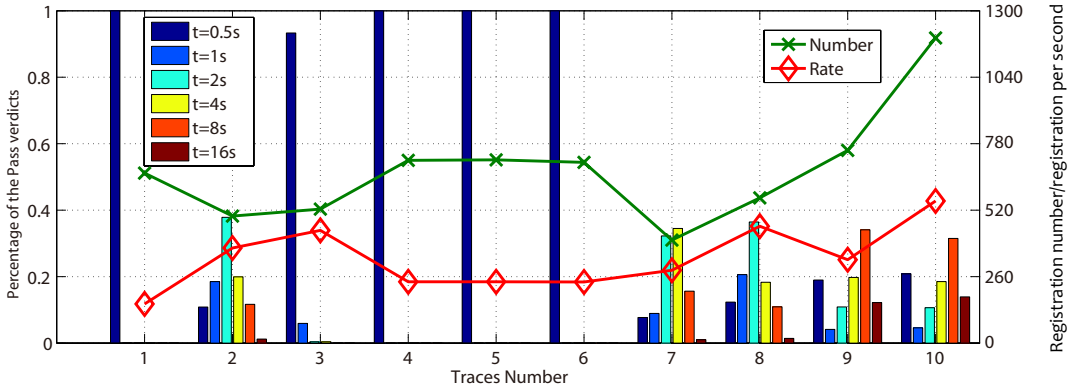
As the tables shown in the experiments, all the results have been obtained in short times (evaluation time of a complex property 3 on 2500 messages is less than 70s). However, in some cases, if the complexity of a formula increases, the result for a trace which contains numerous messages will be obtained in hours. Let us take the property 'Every 2xx response for **INVITE** request must be responded with an **ACK** in 16s' for example, which can be formalized by the formula:

$$\forall_x(request(x) \wedge x.method = \textbf{INVITE} \rightarrow \exists_{y>x}(responds(y,x) \wedge success(y))$$
$$\rightarrow \exists_{z>y}(ackResponse(z,x,y) \wedge withintime(z,y,t)))$$

FIGURE 4.8: Performance Indicators (2)



(a) Session Request Packets Number/Rate/Response Latency



(b) Session Registration Number/Rate/Duration

The results for testing this formula are shown in Table 4.8. As the table shows, the execution time will increase to 1 hours when analyzing 64000 messages. Although we almost managed to reduce 40% of the evaluation time compared to our previous conformance testing algorithm, still some improvements can be done and these will be the issues we will working on in the future. When we test the performance of protocols, lots

Trace	No.of messages	Pass	Con-Fail	Per-Fail	Inconclusive	Time
1	500	60	0	0	0	1.146s
2	1000	109	0	0	0	2.199s
3	2000	183	0	0	0	7.083s
4	4000	405	0	0	0	20.264s
5	8000	784	0	2	1	78.366s
6	16000	1459	0	2	0	313.023s
7	32000	2904	0	4	1	1342.652s
8	64000	5864	0	12	0	1.257h
9	128000	11555	0	35	1	5.17h
10	256000	23154	0	184	0	22.43h
11	450000	43205	0	255	0	85.53h
12	512000	?	?	?	?	?

TABLE 4.8: "Every 2xx response for **INVITE** request must be responded with an **ACK**"

of P.Os are implemented on each IUT in the networks. It provides a perfect environment for performing distributed testing. Then, how to properly synchronize different testers is the next aspect we focus on. Also, except SIP, we should test other protocols to prove the universality of our approach.

4.2 Distributed Performance Testing

Aiming to solve the problems raised from previous section, we introduce a distributed performance testing method in this section. And the experiments results on Extensible Message Presence Protocol are introduced afterwards.

4.2.1 Distributed Testing Framework

For the aim of distributively testing conformance and performance requirements, we introduce a passive distributed testing architecture. Based on the standardized active testing architectures [4] (master-slave framework), we adapted it with several P.Os.

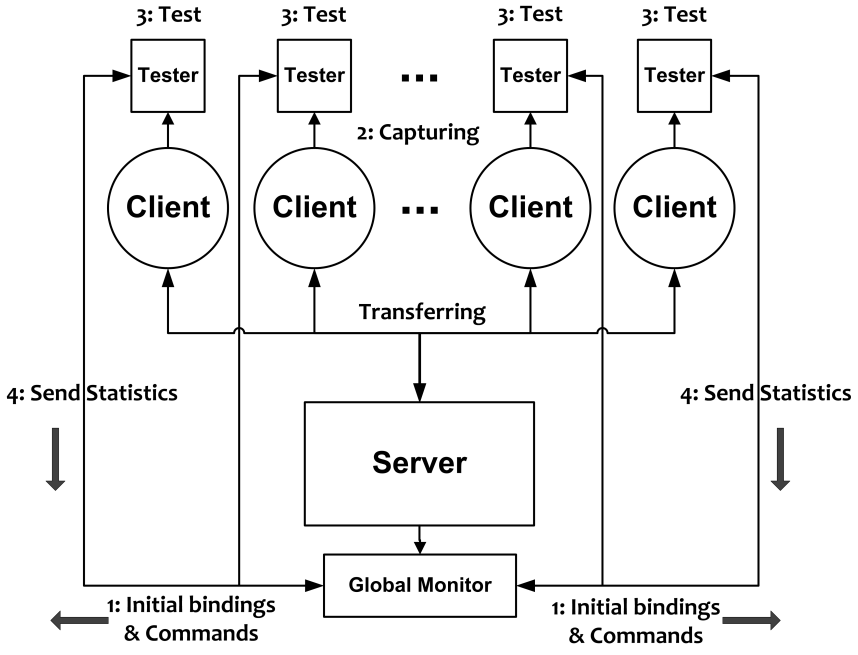


FIGURE 4.9: Distributed testing architecture

As Figure 4.9 depicts, the framework consists of one global monitor and several sub testers. The global monitor is used as a server tester, a console to control sub testers and a terminal to reflect real-time results. The sub testers are linked to the nodes to be tested, in order to capture and test the transporting messages. Once the traces are captured, they will be tested through the predefined requirement formulas, and the test

results will be sent back to the global monitor. On the other side, the global monitor is attached to the server to be tested, aiming at collecting and testing the traces from the server and receiving statistic results from sub testers. The collected aggregate results are analyzed and displayed on the global monitor. It can reflects the real-time conformance and performance condition of the protocol during testing procedures.

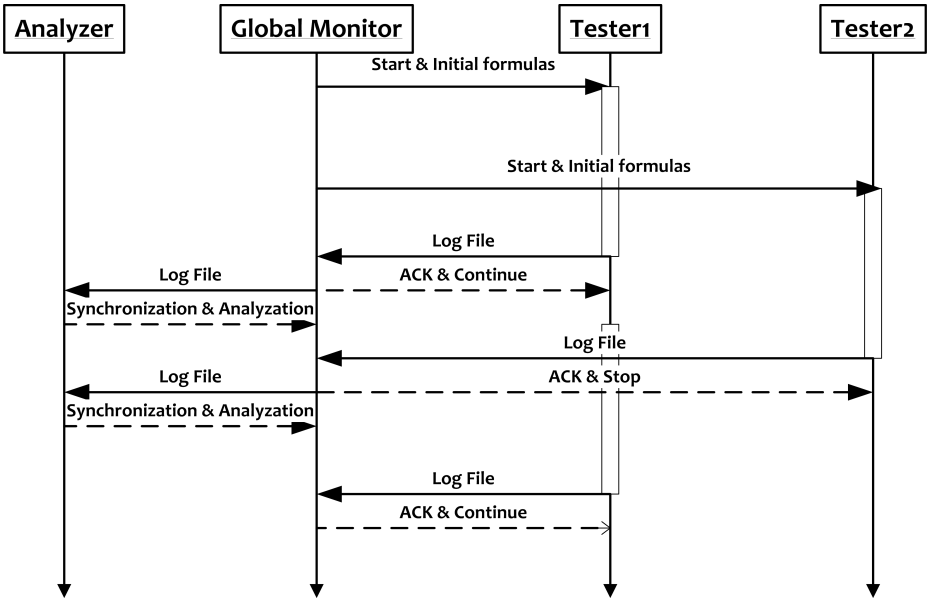


FIGURE 4.10: Sequence Diagram between Testers

Initially, as the Figure 4.10 shows, the global monitor sends initial bindings (formalized requirement formulas, testing parameters) to the sub testers. When the testers receive these information, they initialize capturing packets and save the traces to readable files during each time slot. Once the readable files are generated, the testers will test the traces through the predefined requirements formulas and send the results back to the global monitor.

The analyzer mentioned here is a part of the Global Monitor, for precisely describing the testing procedure, we illustrate it separately. This testing procedure will keep running until the global monitor has to stop or pause a tester, it will send a **Stop** command to the tester needed to be stopped.

4.2.2 Synchronization

As we introduced before, synchronization in the distributed testing environment is a crucial problem to be solved. Several synchronization methods are provided in distributed environment [68]. Network Time Protocol (NTP) [69] is the current standard for synchronizing clocks on the Internet. Applying NTP, time T_{ij}^{k} is stamped on packet k by

the sender i upon transmission to node j. The receiver j stamps its local time R_{ij}^k upon receiving a packet, and time T_{ji}^k upon re-transmitting the packet back to source. The source i stamps its local time R_{ji}^k upon receiving the packet back. Each packet k will eventually have four time stamps on it T_{ij}^k, R_{ij}^k, T_{ji}^k and R_{ji}^k. The computed round-trip delay for packet k is $RTT_{ij}^k = (R_{ij}^k - T_{ij}^k) + (R_{ji}^k - T_{ji}^k)$. Node i estimates its own clock offset relative to node j's clock as $(1/2)[(R_{ij}^k - T_{ij}^k) + (R_{ji}^k - T_{ji}^k)]$, and the transmission process is shown in Figure 4.11.

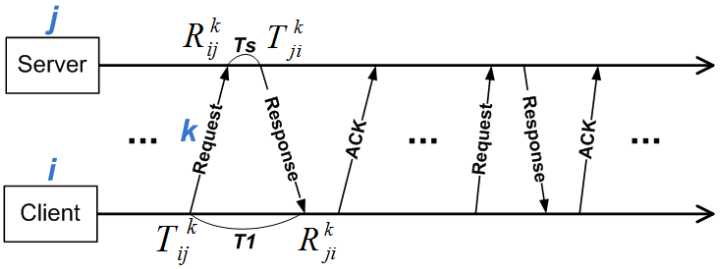


FIGURE 4.11: Synchronization

NTP is designed for synchronizing a set of entities in the networks. In our framework, timers are used for all the testers. However, the non-synchronization between these timers are ineluctable, especially the non-synchronization between the global monitor timer and sub tester timers would affect the results, when real-time performance are analyzed under the influence of network events. Accordingly, the global monitor and sub testers need to be synchronized, and synchronizations between neighbor testers are not required. For satisfying the needs, slight modifications have been made to the transmission process. Rather than exchanging the four time stamps in NTP, two time duration are computed and exchanged in our approach. Initially, we will use an existing successful transaction from the captured traces, since the messages are already tagged with time stamps when captured by the monitors, the redundant tag actions can be omitted.

As illustrated in the Figure 4.11, the T_s represents the service time of the server (time for reacting when receiving a message), and T_1 represents the time used for receiving a response in the client side. Benefiting from capturing traces from both Server and Client sides, the sum $(R_{ij}^k - T_{ij}^k) + (R_{ji}^k - T_{ji}^k)$ can be transformed to $(R_{ij}^k - T_{ji}^k) - (T_{ij}^k - R_{ji}^k) = T_1 - T_s$. Although relative timers are still used for each device, they are merely used for computing the time duration.

After capturing the traces, two sets of messages are generated by the global monitor and sub tester:

Set_{server}={Req_i,Res_i,...,Req_{i+n}, Res_{i+n}}

Set_{client}={Req_j,Res_j,...,Req_{j+m},Res_{j+m} | $j \leq i, j+m \leq i+n$}

As we mentioned before, a successful transaction (Req_k, $Res_k| \ k \leq j+m$) will be chosen from the Set_{client} for the synchronization. The time duration T_1 of the transaction can be easily computed and sent to the global monitor with the testing results. Once the chosen transaction sequence has been found in the Set_{server}, the time duration T_s can be obtained, and the time offset $(1/2)(T_1 - T_s)$ between the global monitor and a sub tester can be handled.

Algorithm 3: Algorithm for Testers

Input: Command
Output: Statistic Logs

```
1  Listening Port n;
2  switch Receive do
3      case Start & Initial bindings:
4          Set Initial bindings to formulas, TimeSlot;
5          Capture(), Test();
6          Send log(i) to Global Monitor;
7          //Send log file to the Global Monitor;
8          Pending;
9      endsw
10     case Continue:
11         Capture(), Test();
12         Send log(i) to Global Monitor;
13         Pending;
14     endsw
15     case Stop:
16         return;
17     endsw
18     case others:
19         Send UnknownError to Global Monitor;
20         Pending;
21     endsw
22 endsw
23 Procedure Capture(timeslot)
24 for (timer=0;timer≤time maximum;timer++) do
25     Listening Port (5060) & Port (5061);
26     //Capture packets;
27     if timer%timeslot==0 then
28         Buffer to Tester(i).xml;
29         //Store the packets in testable formats;
30     end
31 end
32 Procedure Test(formulas)
33 for (j=0;j≤max;j++) do
34     Test formula(j) through Tester(i).xml;
35     //Test the predefined requirement formulas;
36     Record results to log(i);
37     //Save the results to log file;
38     Record first transaction to log(i);
39     //Use the first transaction for synchronization;
40 end
```

4.2.3 Testing Algorithm

The distributed testing algorithms are described in Algorithm 3 and 4. Algorithm 3 describes the behaviors of sub testers when receiving different commands. When the tester receives the initial bindings and a "*Start*" command, firstly it initializes the testing parameters (line 4). Then it starts capturing the traces and tests them (as mentioned in previous sections) when traces are translated to readable xml files (lines 23-40). Finally the results are sent back to the global monitor with the chosen transaction for synchronization.

Algorithm 4: Algorithm for Global Monitor

```
Input: Log files
Output: Performance Graphs
1  Capture(), Test();
2  Display real-time conformance and performance condition;
3  for (i=0;i<tester-number;i++) do
4  |  Send Initial bindings to Tester[i];
5  |  //Send initial bindings to all sub testers
6  end
7  switch receive do
8  |  case log:
9  |  |  if command==Continue then
10 |  |  |  Send Continue to Tester[i];
11 |  |  end
12 |  |  else
13 |  |  |  Send Stop to Tester[i];
14 |  |  end
15 |  |  Synchronize(Log[i].transaction);
16 |  |  Analyze(Log[i].results);
17 |  |  Display real-time conformance and performance condition;
18 |  endsw
19 |  case others:
20 |  |  Send Continue to Tester;
21 |  endsw
22 endsw
23 Procedure Synchronize(Log[i].transaction)
24 for (a=0; a≤Message-Number, quit!=1; a++) do
25 |  find Client.Request(k) in Server.Request(a);
26 |  if (exists==True) then
27 |  |  for (b=a; b≤Message-Number, quit!=1; b++) do
28 |  |  |  find Client.Response(k) in Server.Response(b);
29 |  |  |  if (exists==True) then
30 |  |  |  |  Calculate T_s;
31 |  |  |  |  Handle timer deviation (T_1 − T_s)/2;
32 |  |  |  |  quit=1;
33 |  |  |  end
34 |  |  |  else
35 |  |  |  |  Return transaction error;
36 |  |  |  |  quit=1;
37 |  |  |  end
38 |  |  end
39 |  end
40 end
```

The Algorithm 4 sketches the global monitor behaviors and the synchronization function. Initially, the monitor starts to capture and test as the other testers do. Meanwhile, it sends initial bindings to all the sub testers and waits for their responses (lines 1-5). Once the server receives the response, it reacts according to the content of the response, and the synchronization is made during this time (lines 20-37). In the *synchronize*() procedure, the monitor finds the chosen transaction in its captured traces, and rectifies the time offset $(1/2)(T_1 - T_s)$.

4.2.4 Experiments

Since Internet of Things (IoT) can provide complex wireless communication environment, it becomes a perfect platform to verify the functionality of our approach. In this section, we will introduce the experimental results when our approach are implemented into an IoT environment.

4.2.4.1 Internet of Things

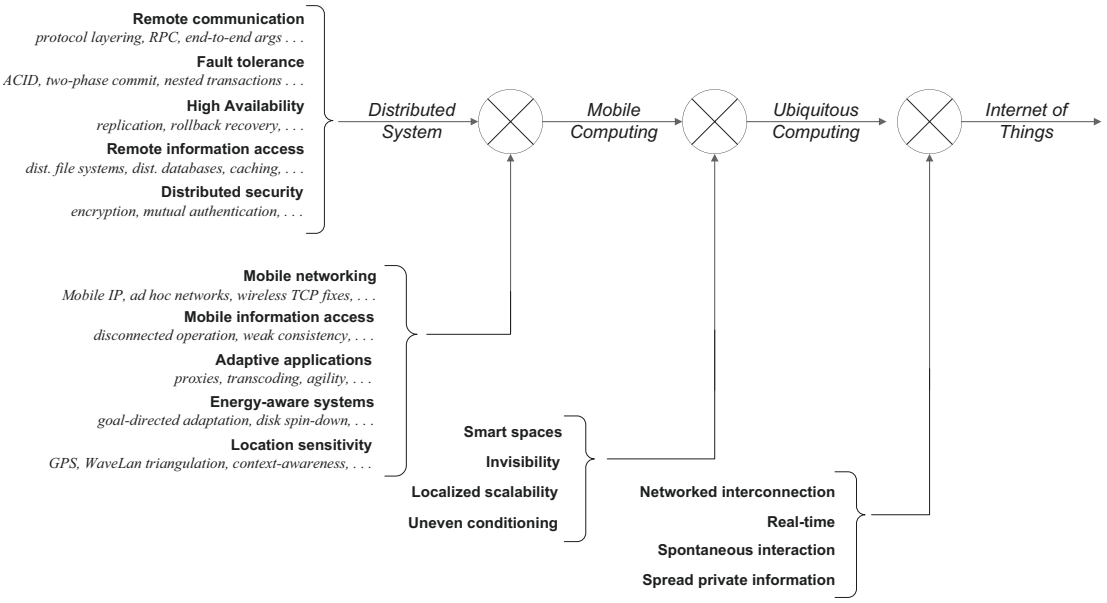


FIGURE 4.12: The evolution of Internet of Things from distributed computing, mobile computing and ubiquitous computing

The Internet of Things (IoT) refers to a networked interconnection of daily objects, this requires the objects not only for being interacted, but also for cooperating with each other at anytime or in anyplace [70] [71]. It opens the door of Internet to the physical world such that objects can be managed remotely and act as physical access points to Internet services [72]. IoT transforms the manner of daily activities by real-time tracking

physical objects. Correspondingly, it opens up massive opportunities for economy and individuals, accompanying immense technical challenges and risks. The evolution of IoT from distributed computing, mobile computing and ubiquitous computing is shown in Figure 4.12.

IoT is established on the basis of proliferation of wireless sensor network, MobiComp (*Mobile Computing*), UbiComp (*Ubiquitous Computing*) and information technologies [73]. Thanks to their diminishing size, declining price and falling energy consumption, sensors are being increasingly integrated into everyday objects. Thus, IoT is applicable in a wide spectrum of fields. To get a heightened awareness of real-time events, it deploys sensors in infrastructures [74]. For achieving an enhanced situational awareness, it employs Radio Frequency IDentification (RFID) to capture object contexts (e.g., location) [75]. For guaranteeing safe driving and green travel, it uses motes to track transportation systems [76]. For getting user preferences, IoT takes advantage of a recommendation service in recommend systems (i.e., [77] a kind of virtual sensors). As the trend goes, we foresee that IoT eventually links the majority of objects into the virtual space and allows objects to interact in the same place. In this case, the communication protocols used in the IoT will be a crucial part for testing.

In the recent years, the Extensible Messaging and Presence Protocol (XMPP) [78] has gained more attention as communication protocol in the Internet of Things, which is a standardized protocol by the IETF and well established in the Internet. XMPP is available for common used programming languages and device platforms. Several studies have investigated the potentialities of applying XMPP in IoT [79] [80] [81]. The authors of [81] introduce a service platform based on the XMPP protocol for the development and provision of services for pervasive infrastructures (Figure 4.13), their work perfectly illustrates the usages of XMPP in IoT.

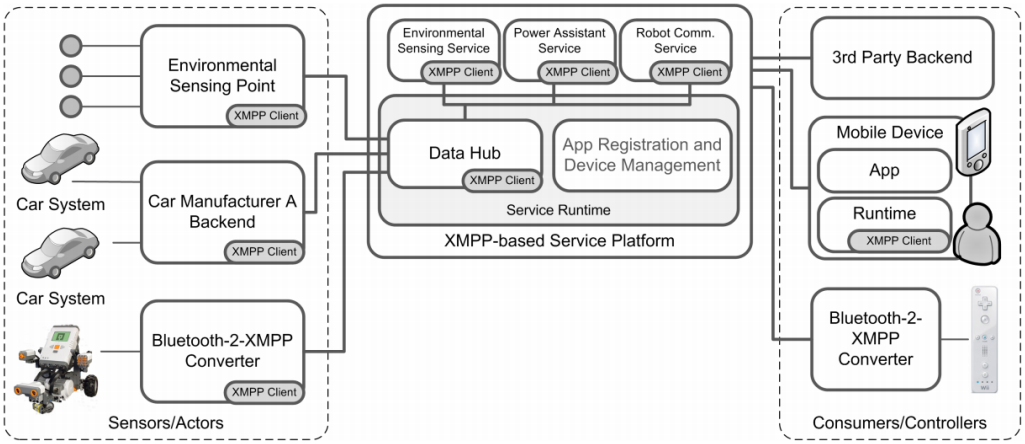


FIGURE 4.13: Architecture of XMPP Service in Internet of things [81]

XMPP is used for communication in this system. All entities including the services are XMPP clients which can be identified by a Jabber Identifier (JID) in a system-wide unique manner. XMPP servers are usually necessary to mediate the communication between XMPP clients. With the tendency that the XMPP is more and more widely used in many aspects of IoT, the problem of formally testing it in a wireless environment is coming out in the wash.

4.2.4.2 Extensible Messaging and Presence Protocol

The XMPP is an application profile of the Extensible Mark-up Language that enables the near-real-time exchange of structured yet extensible data. The purpose of XMPP is to enable the exchange of relatively small pieces of structured data (called "XML stanzas") over a network between any two (or more) entities. XMPP is typically implemented using a distributed client-server architecture, wherein a client needs to connect to a server in order to gain access to the network and thus be allowed to exchange XML stanzas with other entities (which can be associated with other servers). The process whereby a client connects to a server, exchanges XML stanzas, and ends the connection is:

- Determine the IP address and port at which to connect, typically based on resolution of a fully qualified domain name
- Open a Transmission Control Protocol [TCP] connection
- Open an XML stream over TCP
- Preferably negotiate Transport Layer Security [TLS] for channel encryption
- Authenticate using a Simple Authentication and Security Layer [SASL] mechanism
- Bind a resource to the stream
- Exchange an unbounded number of XML stanzas with other entities on the network
- Close the XML stream
- Close the TCP connection

The communication process is shown in Figure 4.14. In the process, two fundamental concepts make possible the rapid, asynchronous exchange of relatively small payloads of structured information between XMPP entities: XML streams and XML stanzas.

An XML stream is a container for the exchange of XML elements between any two entities over a network. The start of an XML stream is denoted unambiguously by an

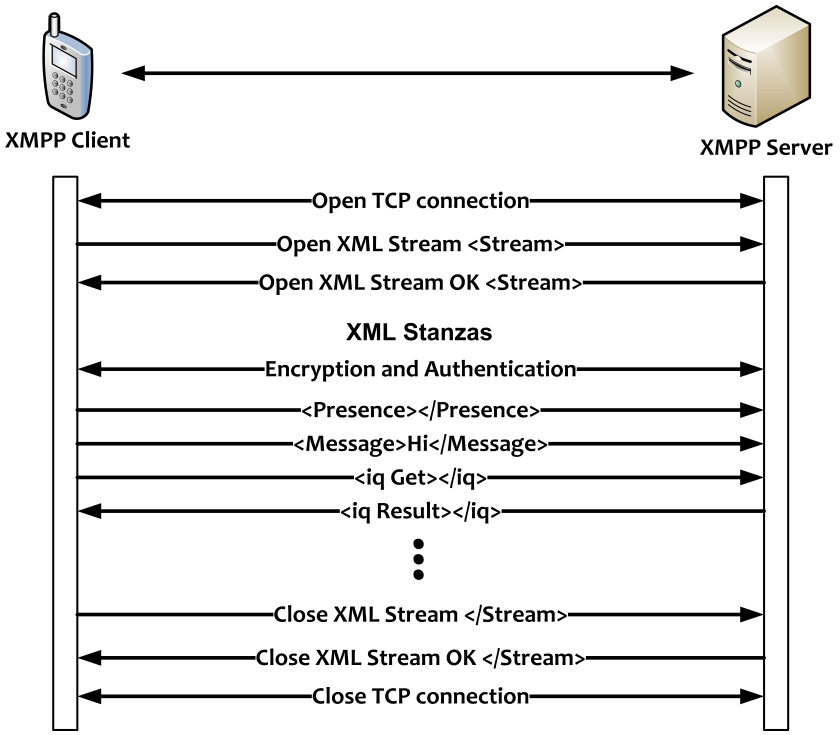


FIGURE 4.14: Connection process of a XMPP client to a XMPP server

opening "stream header" (i.e., an XML <stream> tag with appropriate attributes and namespace declarations), while the end of the XML stream is denoted unambiguously by a closing XML </stream> tag. During the life of the stream, the entity that initiated it can send an unbounded number of XML elements over the stream, either elements used to negotiate the stream (e.g., to complete TLS negotiation or SASL negotiation) or XML stanzas. The "initial stream" is negotiated from the initiating entity (typically a client or server) to the receiving entity (typically a server), and can be seen as corresponding to the initiating entity's "connection to" or "session with" the receiving entity. The initial stream enables unidirectional communication from the initiating entity to the receiving entity; in order to enable exchanges of stanzas from the receiving entity to the initiating entity, the receiving entity MUST negotiate a stream in the opposite direction (the "response stream").

The attributes of the root stream element are defined in the following:

- **From**: The 'from' attribute specifies an XMPP identity of the entity sending the stream element. For initial stream headers in client-to-server communication, the 'from' attribute is the XMPP identity of the principal controlling the client, i.e., a JID of the form $< localpart@domainpart >$.

- **To**: For initial stream headers in both client-to-server and server-to-server communication, the initiating entity MUST include the 'to' attribute and MUST set

its value to a domainpart that the initiating entity knows or expects the receiving entity to service.

- **id**: The 'id' attribute specifies a unique identifier for the stream, called a "stream ID". The stream ID MUST be generated by the receiving entity when it sends a response stream header and MUST BE unique within the receiving application (normally a server).
- **xml:lang**: The 'xml:lang' attribute specifies an entity's preferred or default language for any human-readable XML character data to be sent over the stream.
- **version**: The inclusion of the version attribute set to a value of at least "1.0" signals support for the stream-related protocols defined in this specification, including TLS negotiation, SASL negotiation, stream features, and stream errors.

An XML stanza is the basic unit of meaning in XMPP. A stanza is a first-level element whose element name is "message", "presence", or "iq" and whose qualifying namespace is 'jabber:client' or 'jabber:server'. By contrast, a first-level element qualified by any other namespace is not an XML stanza (stream errors, stream features, TLS-related elements, SASL-related elements, etc.), nor is a <message/>, <presence/>, or <iq/> element that is qualified by the 'jabber:client' or 'jabber:server' namespace but that occurs at a depth other than one (e.g., a <message/> element contained within an extension element for reporting purposes), nor is a <message/>, <presence/>, or <iq/> element that is qualified by a namespace other than 'jabber:client' or 'jabber:server'. An XML stanza typically contains one or more child elements (with accompanying attributes, elements, and XML character data) as necessary in order to convey the desired information, which MAY be qualified by any XML namespace.

The five common attributes of the XML message, presence, and iq stanzas are defined in the following:

- **To**: The 'to' attribute specifies the JID of the intended recipient for the stanza.
- **From**: The 'from' attribute specifies the JID of the sender.
- **id**: The 'id' attribute is used by the originating entity to track any response or error stanza that it might receive in relation to the generated stanza from another entity.
- **type**: The 'type' attribute specifies the purpose or context of the message, presence, or iq stanza. The particular allowable values for the 'type' attribute vary depending on whether the stanza is a message, presence, or iq stanza.

- **xml:lang**: A stanza SHOULD possess an 'xml:lang' attribute if the stanza contains XML character data that is intended to be presented to a human user. The value of the 'xml:lang' attribute specifies the default language of any such human-readable XML character data.

An example of a presence stanza m from Romeo to Juliet using the introduced attributes is shown below.

$$\begin{aligned}\mathbf{m} = &< presence\ from = `romeo@example.net/orchard' \\ &\quad to = `juliet@im.example.com' \\ &\quad xml : lang = `en' > \\ &\quad\quad < show > dnd < /show > \\ &\quad\quad < status > WooingJuliet < /status > \\ &< /presence >\end{aligned}$$

4.2.4.3 Testing framework and Tsung

For our experiments, XMPP traces were obtained from Tsung[3]. Tsung is a distributed load testing tool which is protocol independent and can be used to stress Hypertext Transfer Protocol (HTTP), WebDAV, Simple Object Access Protocol (SOAP), PostgreSQL, MySQL, Lightweight Directory Access Protocol (LDAP), and XMPP servers. It has the ability to simulate a huge number of simultaneous users from a single machine. When used on cluster, impressive load can be generated on a server with a modest cluster, easy to set up and maintain.

The implementation has been performed using Java and is composed of two main modules, as shown in Figure 4.15. The *trace processing* module receives the raw traces

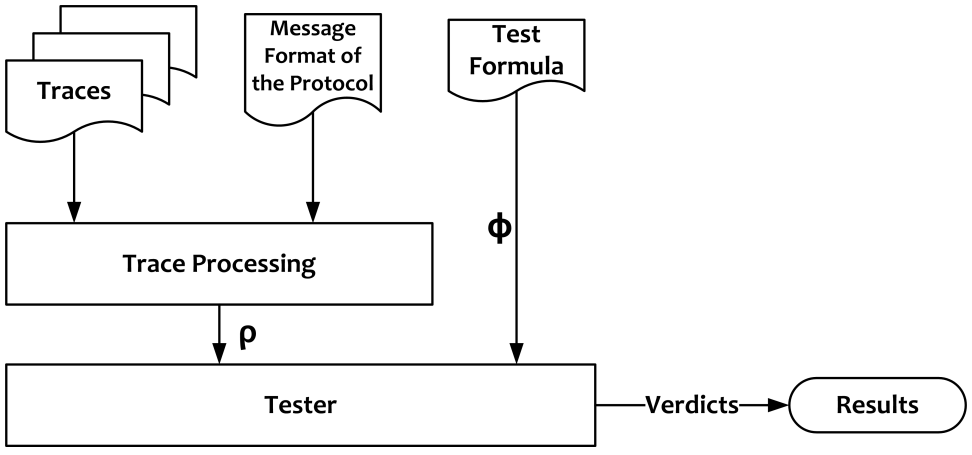


FIGURE 4.15: Our architecture for our testing framework.

[3]http://tsung.erlang-projects.org/

collected from the network exchange and converts the messages from the input format into a list of messages compatible with the clause definitions. Although the module can be adapted to multiple input formats, in our experiments, the inputs are XML files obtained from Tsung traces.

The *tester* module takes the resulting trace from the trace evaluation along with the formula to test, and it returns a set of satisfaction results for the formula in the trace, as well as the variable bindings and the messages involved in the result. The results from the experiments are presented in the following sections.

4.2.4.4 Environments

In the experiments, we designed a simulation on wireless ad hoc architecture for testing. For ensuring the accuracy and authenticity of the results, we construct a wireless environment using real laptops. Each laptop is implemented with a XMPP server and several XMPP clients. This environment can be used to test the correctness, robustness, and reliability of XMPP protocol under tremendous number of messages. The observation points being on the XMPP server and clients are shown in Figure 4.16. The configuration of laptops are CPU- Intel Core i5-2520M 2.50 GHz, 4GB DDR3; CPU- AMD Atholon 64 X2 5200+, 2GB DDR2; CPU- Intel Core2 Duo T6500 2.10 GHz, 2GB DDR2; and CPU- Intel Core2 Duo T6500 2.10 GHz, 4GB DDR2.

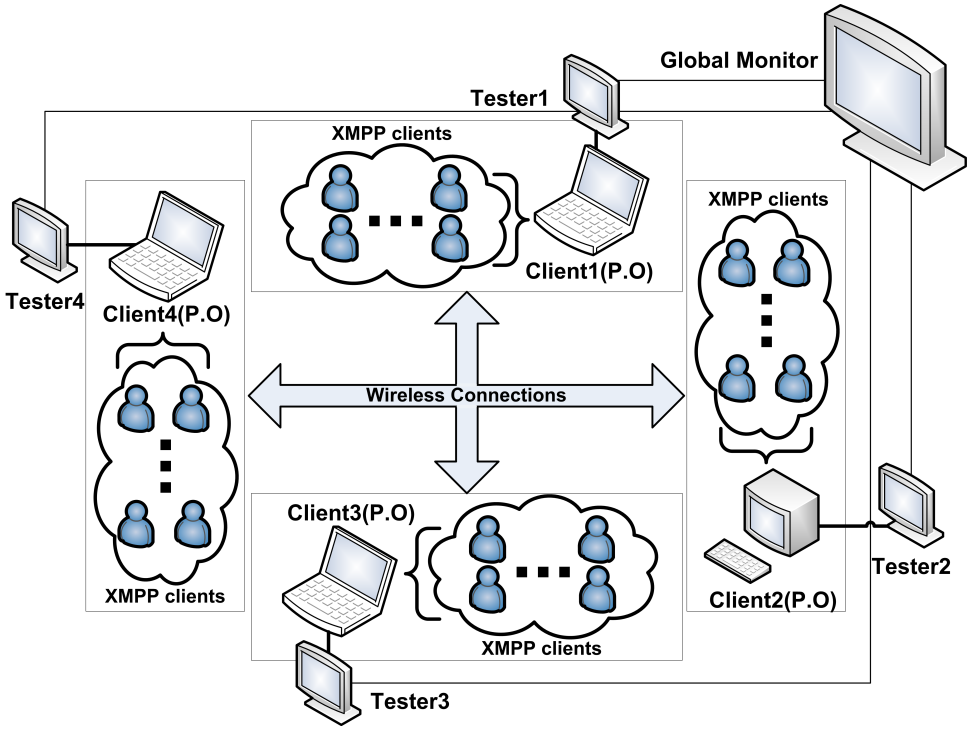


FIGURE 4.16: XMPP testing architecture.

4.2.4.5 Properties and Results

To formally design the properties to be passively tested, we studied the RFC 6120 of XMPP [78]. We designed several properties for the experiments; for the evaluation of each property, we used a set of traces collected from P.O ($Client_1$) containing {500, 1000, 2000, ... , 64,000, 128,000} packets to get exhaustive results.

Property 1: For every Roster-GET request there must be a response
This conformance property can be used for a monitoring purpose. Due to the issues related to testing on finite traces for finite executions, a $fail$ results can never be given for this context. However, *inconclusive* results can be provided and conclusions may be drawn from further analysis of the results. The property evaluated is as follows:

$$\forall_x(request(x) \wedge x.iq.type = \textbf{'GET'} \rightarrow \exists_{y>x}(responds(y,x)))$$

where $responds(y,x)$ only accepts the responses to **GET** requests. To verify the efficiency of our approach, we first provide the testing results from a single P.O $Client_1$, as shown in Table 4.9.

As expected, most traces show only pass results for the property evaluation, but inconclusive results can also be observed. After analyzing trace 2, the inconclusive verdict is found caused by a missing response message to **GET** request; this **GET** message is at the end of the trace, which could indicate that the client closed the connection before receiving the response message. The same phenomenon can be observed on the trace 6. Besides, other inconclusive verdicts in traces 4, 8, and 9 are caused by the real lost responses to the **GET** requests in the transportation.

Trace	Number of messages	Pass	Fail	Inconclusive	Time (s)
1	500	25	0	0	0.842
2	1000	43	0	1	1.434
3	2000	90	0	0	2.851
4	4000	167	0	5	5.940
5	8000	343	0	0	10.219
6	16,000	679	0	1	20.160
7	32,000	1328	0	0	39.906
8	64,000	2175	0	7	72.489
9	128,000	4031	0	12	157.451

TABLE 4.9: "For every **Roster-GET** request, there must be a response."

Property 2: No "Presence" message can be received without a previous subscription

Here, a more complex conformance property is tested, which can verify that only users successfully subscribed with the XMPP Server can receive the "Presence" message. It is defined using our syntax as follows:

$$\forall_x(presence(x) \rightarrow \exists_{y<x}(\exists_{z>y} subscription(y,z)))$$

where $presence(x)$ and $subscription(y,z)$ are defined as

$$\begin{aligned} presence(x) &\leftarrow x.presence! = \text{'Null'} \\ subscription(y,z) &\leftarrow request(y) \wedge responds(z,y) \\ &\quad \wedge y.presence.type = \textbf{'subscribe'} \\ &\quad \wedge z.presence.type = \textbf{'subscribed'} \end{aligned}$$

Still, we use the same trace collected from $Client_1$ for testing this property. As shown in Table 4.10, it can be shown that this property and the framework allow to detect when the tested property holds on the trace. From the results in Table 4.10, we can observe

Trace	Number of messages	Pass	Fail	Inconclusive	Time
1	500	82	0	0	18.960 s
2	1000	135	0	0	51.841 s
3	2000	210	0	1	128.364 s
4	4000	392	0	0	402.215 s
5	8000	623	0	0	1179.275 s
6	16,000	1145	0	0	1.032 h
7	32,000	2176	0	2	3.147 h
8	64,000	4081	0	1	6.078 h
9	128,000	8135	0	2	12.984 h

TABLE 4.10: No "Presence" message can be received without a previous subscription.

that most of the traces satisfy this property. The inconclusive verdicts in traces 3, 7, 8 and 9 are still caused by the same reasons mentioned in the first conformance property. It can be seen that the evaluation of this property is much more time consuming than the one in Table 4.9. This is expected given the complexity of the evaluation (n^2 for property 1 and n^3 for the current one).

Property 3: For every request, the response should be received within 8 s

After testing two functional conformance requirements, a nonfunctional performance property is tested which can be used for reflecting the current packet-delay condition. The property is designed as follows:

$$\forall_x(request(x) \rightarrow \exists_{y>x}(responds(y,x) \wedge withintime(x,y,8s)))$$

where $withintime(x, y, 8s)$ will find out the requests responded over the 8s limitation. However, this performance requirement ϕ_{per_6} is based on the conformance requirement ϕ_{con_6}-"For every request, the response should be received," and the evaluation results for $(comb(\phi_{per_6}, \phi_{con_6}))$ are shown in Table 4.11.

		ϕ_{con_6}			$comb(\phi_{per_6}, \phi_{con_6})$				
Trace	Number of messages	Pass	Fail	Incon	Pass	Con-Fail	Per-Fail	Incon	Time (s)
1	500	162	0	7	154	4	8	3	0.745
2	1000	314	0	16	293	13	22	2	1.213
3	2000	671	0	6	658	2	15	2	2.472
4	4000	1189	0	24	1163	10	37	3	4.914
5	8000	2546	0	10	2524	1	29	2	11.462
6	16,000	4397	0	18	4356	5	52	5	21.650
7	32,000	8316	0	32	8248	18	79	3	40.564
8	64,000	15464	0	51	15378	10	125	2	85.721
9	128,000	31981	0	87	31857	22	187	2	173.163

TABLE 4.11: For every request, the response should be received within 8 s.

From the results in Table 4.11, the formalized performance requirement can be perfectly tested. Due to the help of simultaneously testing the constituent conformance requirement ϕ_{con_6}, the negative verdicts can be differentiated. The messages exceeded time limitations are reported as Per-Fail in $comb(\phi_{per_6}, \phi_{con_6})$, while the messages are reported as Con-Fail due to the reason of detecting unexpected bytes in data portions. These unexpected bytes are caused by the same reason when testing SIP messages: the impact of impulsive noise on the electronic wave. Also, the reported inconclusive verdicts for $comb(\phi_{per_6}, \phi_{con_6})$ indicate that the messages cannot be checked since they are at the end of traces.

Global monitor

Apart from testing a single client, as we described in the previous subsections, each client will be tested through predefined properties. All the results returned from different P.Os ($Client_1, Client_2, Client_3$ and $Client_4$) will be aggregated to the global monitor for a global view of the testing information.

Figure 4.17-4.19 illustrate an example of the aggregated testing information from four testers. Charts 4.17 and 4.18 represent the percentages of 'Pass' verdicts on properties "Each Request must be responded with a response" and "For each Request, the response should be received within 8 s," respectively. From these two figures, we can observe the 'Pass' rate of each client for the two properties. Figure 4.19 illustrates the results of a performance indicator "Number of Requests per second". From this figure, we can observe the performances of server and clients at the same time, the information returned from global server and sub testers are illustrated in the same graph. In this way, the

current conformance and performance requirement conditions can be intuitively reflected to the users.

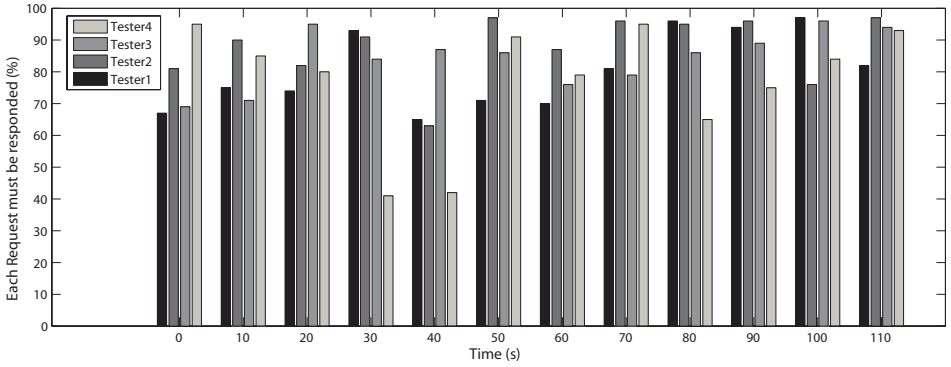


FIGURE 4.17: Testing information on global monitor: "Each Request must be responded with a response."

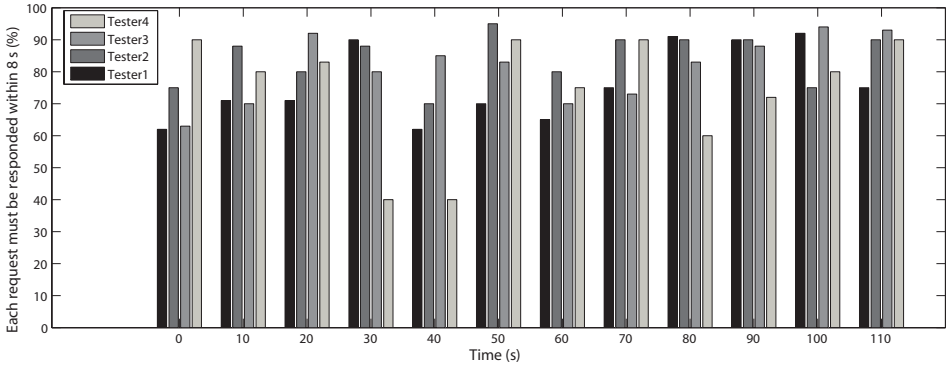


FIGURE 4.18: Testing information on global monitor: "For each Request, the response should be received within 8 s."

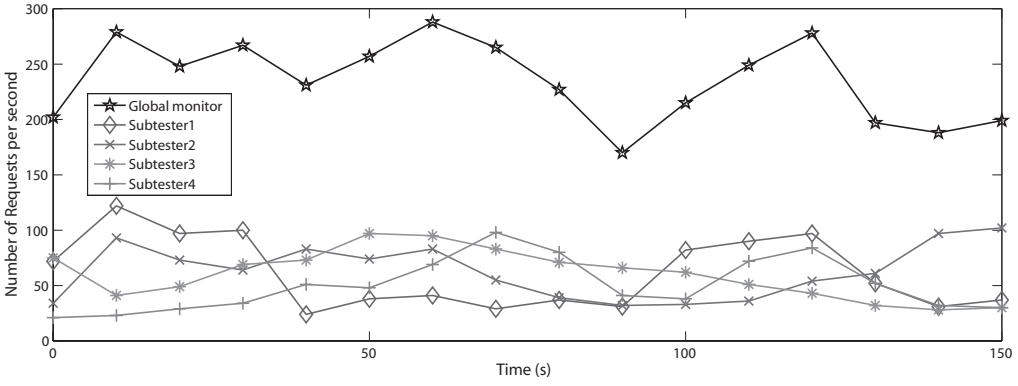


FIGURE 4.19: Testing information on global monitor: Number of Requests per second.

4.2.4.6 Discussions

As shown in the experiments, the results from testing several properties on large traces have been obtained with success. We solved the problem of synchronization and we successfully performed our approach on another protocol XMPP. The functionality and flexibility of our approach are shown.

Consequently, building a standardized performance testing benchmark system for protocols would be the work we will focus on in the future. In that case, the efficiency and processing capacity of the system when massive sub testers are performed would be the crucial point to handle, leading to an adaptation of our algorithms to more complex situations.

Currently, our research is based on collecting traces from the implemented frameworks, in other words, we are performing an off-line testing process. For satisfying the increasing need of run-time monitoring and testing process, a good way for practicing our approach would be to put it online.

4.2.5 Conclusion

In this chapter, we presented a passive performance testing approach for communicating protocols based on the formal specification of the time related requirements. We detailed on the modified syntax and semantics of formulas for satisfying the needs in formalizing performance requirements. Also for solving the indeterminacy problems existed in non-positive verdicts, we introduce a four-valued semantics {'*Pass*', '*Con-Fail*', '*Per-Fail*', '*Inconclusive*'} in our formalism. Then we explained our evaluating algorithm and the preliminary experiments results. The results proved the functionality and flexibility of our approach, and this preliminary work is published in [82] and [83]. Besides, for solving the distributed testing issues, we proposed a distributed passive testing framework, and we implemented and tested it through XMPP properties in IoT environments. The results from testing several properties on large traces have been obtained with success. This following work is published in [84] and [85].

Chapter 5

Online Testing Approach

"What one man can invent, another can discover."
– Arthur Conan Doyle (1859 – 1930)

As the question raised in the previous chapter, online testing approaches are crucial in complex systems. By that way, testing a protocol at run-time may be performed during a normal use of the system without disturbing the process. The traces are observed and analyzed on-the-fly to provide test verdicts and no trace sets should be studied a posteriori to the testing process. In this chapter, we describe the architecture and testing process of our approach for online testing. We also explain the new definitions of online testing verdicts into details.

5.1 Architecture of the approach

The architecture of our online testing approach is illustrated in Figure 5.1. In our approach, the Horn logic [56] is still used for formally expressing properties as formulas. A syntax tree generated from the formulas is used for filtering incoming traces and optimizing evaluation processes, in order to reduce the cost of resources. For the evaluation part, we use the SLD-resolution algorithm for evaluating formulas.

5.2 Testing Process

As shown in Figure 5.1, the testing process consists of eight parts: Formalization, Construction, Capturing, Generating Filters/Setup, Filtering, Transfer/Buffering, Load Notification and Evaluation. We describe these processes in the following.

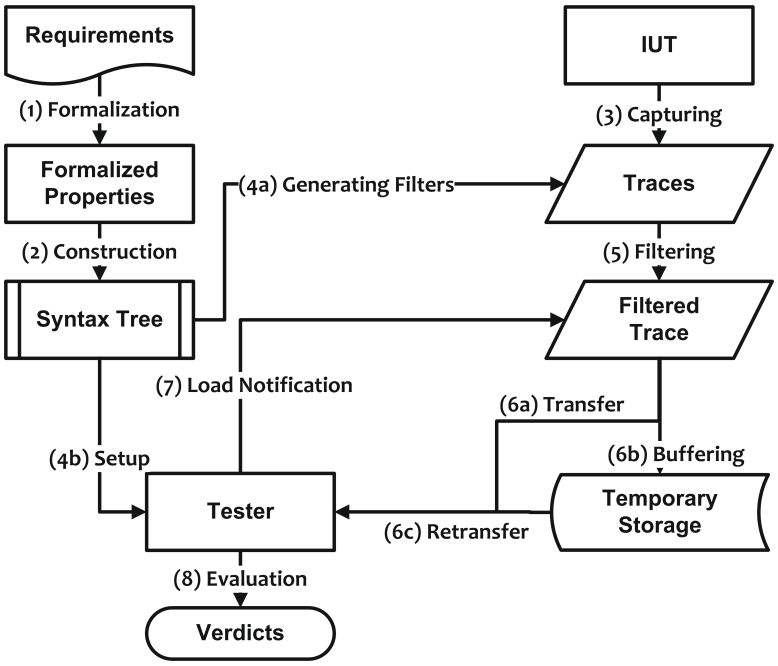


FIGURE 5.1: Architecture of our online testing approach

Formalization: Initially, informal protocol requirements are formalized using the syntax and semantics mentioned in previous sections. Then the verdicts {'*Pass*', '*Con-Fail*', '*Per-Fail*', '*Time-Fail*', '*Inconclusive*', '*Data-Inc*'} are provided to the interpretation of obtained formulas on real protocol execution traces. However, different from offline testing, definite verdicts should be immediately returned in online testing process. This indicates that only '*Pass*', '*Con-Fail*', '*Per-Fail*' and '*Time-Fail*' should be emitted in the final report, and indefinite verdicts '*Data-Inc*' and '*Inconclusive*' will be used as temporary unknown status, but finally must be transformed to one of the definite verdicts at the end of the testing process.

Construction: From formalized formulas, a syntax tree is constructed for further testing processes. In this process, each formula representing a requirement will be transformed to an Abstract Syntax Tree (AST) using the TREEGEN algorithm [86]. The standard BNF representation of each formula is the input to construct an AST. An abstract syntax tree example for formula

$$\forall_x(request(x) \wedge x.method = \textbf{'INVITE'} \rightarrow \exists_{y>x}(success(y) \wedge responds(y,x)))$$

(representing the requirement "Every **INVITE** request must be responded with a 200 response") is shown in Figure 5.2.

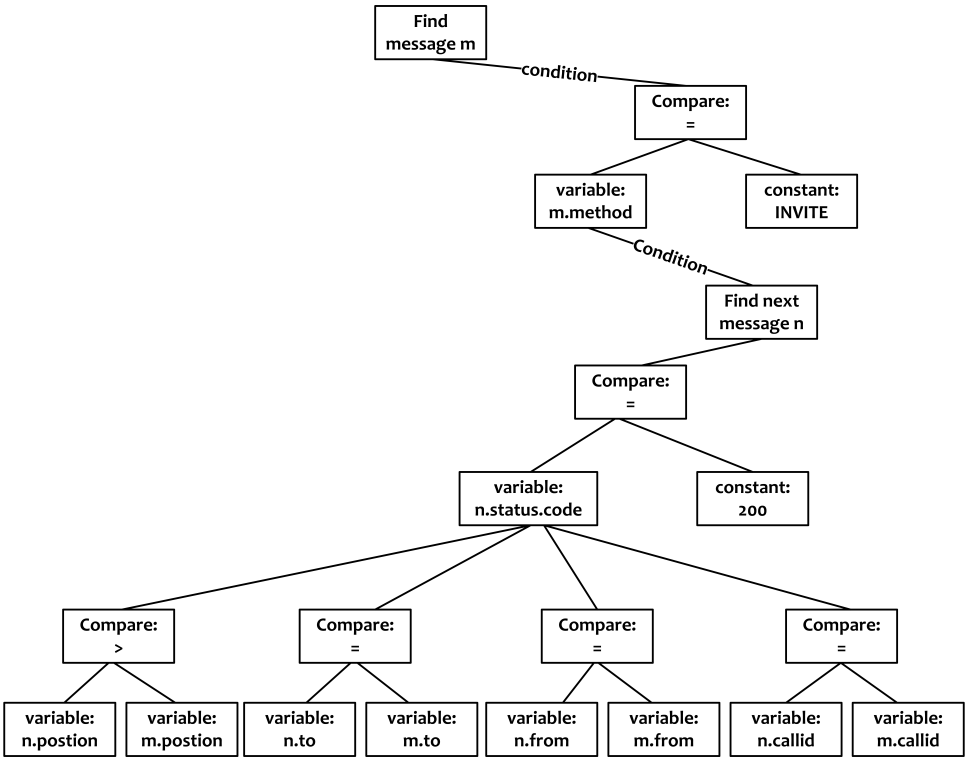


FIGURE 5.2: An example of an abstract syntax tree

All the generated ASTs are finally combined to a syntax tree using a fast merging algorithm [87], as shown in Figure 5.3. The syntax tree will be transferred to the tester as requirements and will be used to filter the captured traces.

Capturing: The monitor consecutively captures traces of the protocol to be tested from points of observations (P.Os) of the IUT, until the testing process finishes. When messages are captured, they are tagged with a time-stamp t_m in order to test the properties with time constraints and to provide verdicts on the performance requirements of the IUT.

Generating Filters and Setup: Once the syntax tree is constructed, it will be applied to captured traces for playing the role of a filter. Meanwhile, the tree will also be sent to the tester with the definition of verdicts. According to different conditions, verdicts are defined as below:

- PASS: The trace satisfies the requirements.
- CON-FAIL: The trace does not satisfy the conformance requirements. Different from our approach in off-line testing, the Con-Fail verdict here is only used to report violation of data portion requirements.

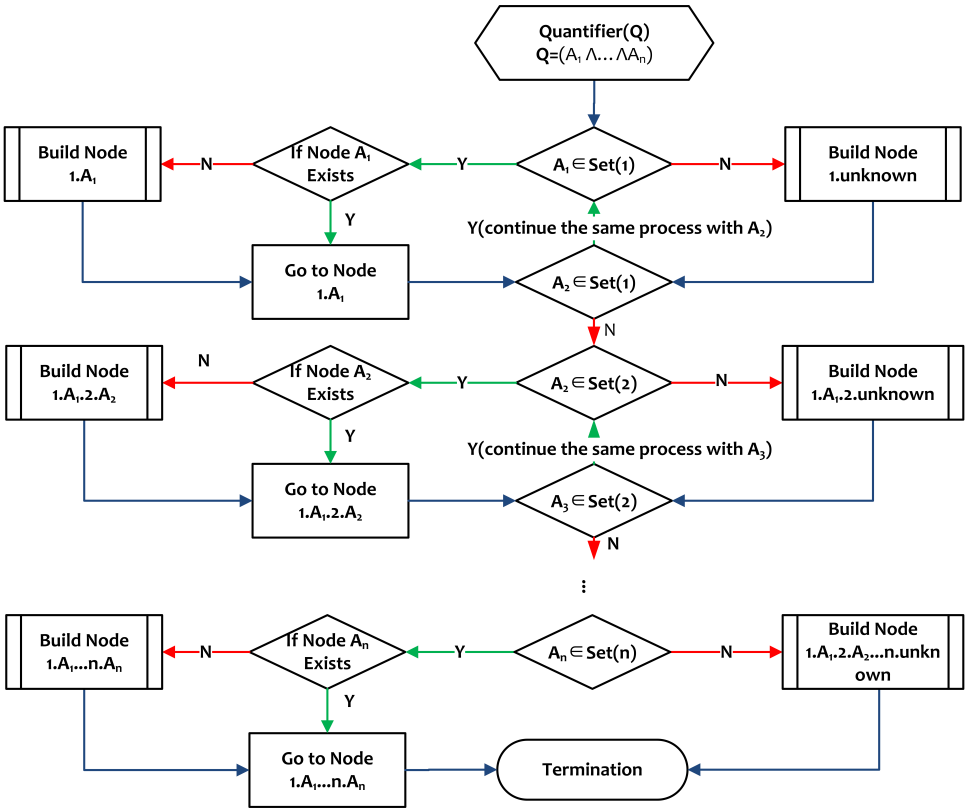


FIGURE 5.3: The process of a syntax tree generated from formulas

- PER-FAIL: The trace does not satisfy the performance requirements. The same as in off-line testing, the Per-Fail is still used to report violation of performance requirements.

- TIME-FAIL: The target message cannot be observed within the maximum time limitation. Since we are working on online testing, a timeout is used to stop searching target message in order to provide the real-time status. The timeout value should be the maximum response time written in the protocol standard. If we cannot observe the target message within the timeout time, then a *Time-Fail* verdict will be assigned to this property. It has to be noticed that this verdict is only provided when no time constraint is required in the requirement. If any time constraint is required, the violation of this requirement will be concluded as *Per-Fail*, not as a *Time-Fail* verdict.

- INCONCLUSIVE: Uncertain status of the properties. Different from offline testing, this verdict will not appear in the final results. It only exists at the beginning of the test or when the test is paused, in order to describe the indeterminate state of the properties (e.g. a property that requires a special occurrence on the protocol that did not occur yet).

- DATA-INC (Data Inconclusive): In the testing process, some properties may be evaluated through traces containing only control portion (there is no data portion or the latter case mentioned in Step 'Transferring'). If any property requires for testing the data portion, *Data-Inc* verdicts will be assigned to the property, due to the fact that no data portion can be tested.

  However, these *Data-Inc* verdicts will be eventually updated to *Pass* or *Fail* based on the data (coming from complete traces) analyzed on the tested properties. Currently we are using worst-case solution (all concluded as *Fail* verdicts). It will not affect the overall results, since *Data-Inc* verdicts only represent a tiny proportion (less than 0.1%) of the whole traces in our experiments. However, expecting eventual contingencies, we plan to apply a support vector machine (SVM) approach [88] in the future.

Filtering: The incoming captured traces will go through the filtering module, and messages in the traces are filtered into different sets. As shown in Figure 5.4, the unnecessary messages irrelevant to any of the requirements are filtered into the "Unknown" set, and they will not go through the testing process. Finally, traces will be filtered to multiple optimized streams. This step will obviously reduce the processing time, since futile comparisons with irrelevant messages are omitted.

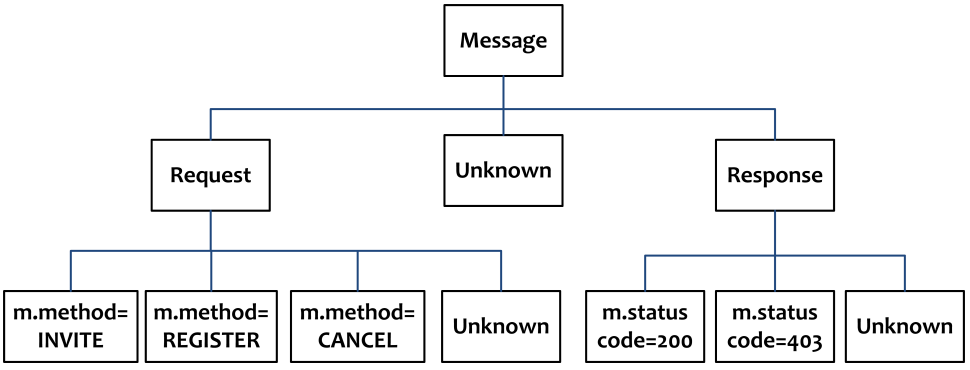


FIGURE 5.4: Example of Filtered messages

Transferring: The filtered traces are transferred (6a) to the tester when the tester is capable for testing. If the tester priority has to be decreased (e.g. the CPU and RAM must be used for another task on this computer of the end-user), a "load notification" (7) is provided to the monitor in order to transfer/store incoming traces. Based on the message format of the protocols to be tested, different buffering methods will be applied.

- If in the message format, the size of its header is larger than its body, as shown in Figure 5.5. Then the whole message will be buffered in the temporary storage.

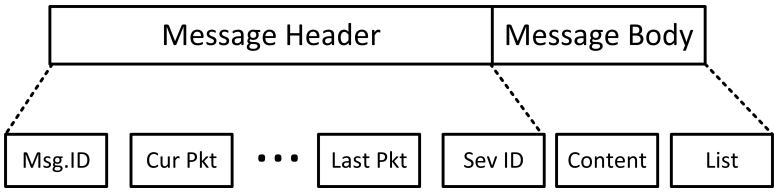


FIGURE 5.5: Example of a header larger than body

- On the contrary, if the size of its header is equal or less than its body, as shown in Figure 5.6, and the requirements have no specific needs on the data portion, then if necessary only the control portion of the packets are buffered (6b) in the temporary storage. Since not all the protocol requirements have specific needs on the data portion, only buffering the control portion will save a lot of memory space when dealing with millions of messages. Also it is shown in our experiments, these ignored data portions will not influence the general information we get.

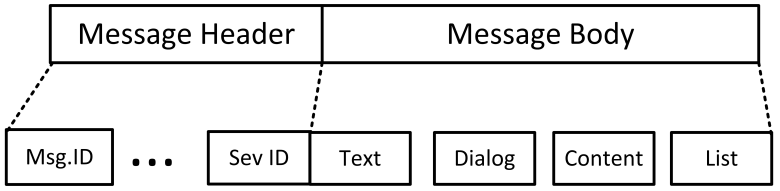


FIGURE 5.6: Example of a header shorter than body

When the tester is available (notification obtained), the stored traces are retransferred (6c) to the tester. In the latter case mentioned above, only the control portion of packets are provided. In both cases, the continuity of traces is ensured, since no packet will be dropped in any condition. If the protocol requirement has specific needs on the data portion, then the new verdict *Data-Inc* can be given and will be eventually updated to final verdicts by future analysis with the entire traces (the tester is indeed available again).

Load Notification: When the tester reaches its limit regarding the amount of data processable or is given a lower priority (e.g. to discharge the CPU / RAM), it sends a "Load Notification Y" to pause incoming filtered traces and store them in the temporary storage. When the tester is available back, a "Load Notification N" to release stored traces and to pursue incoming packets is sent. A brief description of processes 6 and 7 is shown in Figure 5.7.

As the figure illustrates, when captured traces from the IUT are transferred to the tester buffer, a checking overflow function will be called. If the buffer already reached to its maximum capacity, it will notify the IUT to redirect incoming traces to temporary

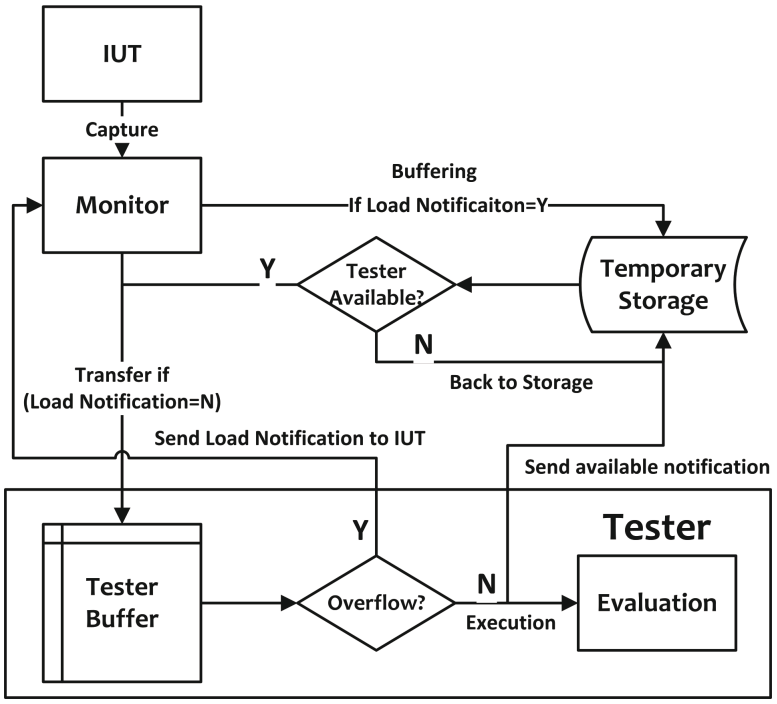


FIGURE 5.7: Process of buffering and notification

storage in order to avoid the overflow. On the contrary, if the buffer is in a stable condition, it will send the available notification N to the temporary storage for releasing stored messages and to the IUT for returning back to normal transport process.

Evaluation: The tester checks whether the incoming traces satisfy the formalized requirements, and provides the final verdicts *Pass, Con-Fail, Per-Fail* or *Time-Fail* and temporary verdicts *Inconclusive* or *Data-Inc*.

5.3 Testing algorithm

The online testing algorithm for a tester is described in the Algorithm 5. It describes the behaviors of an online tester.

Firstly, the tester will capture packets from the predefined interface by using libpcap[1], and report the live condition by using *thread_init* (*report_live_status*). Then the program will continue with the main thread, and tag time stamps to all the captured packets at the same time (Line 1-3). The *last_observed_packet_time* is used for controlling package timeouts.

[1] http://www.tcpdump.org/

Algorithm 5: Algorithm of online tester

```
Input: open_live_capture_on_interface(INTERFACE_NAME) //Using libpcap
Output: property verdicts report
1  thread_init(timeout_thread) //thread to remove packets from packet queues given global timeout.
   thread_init(report_live_status) //thread to report the live
2  for each packet on live_capture do
3      last_observed_packet_time ← get_time(packet);
4      for each prototype on prototype_packets do
5          property ← get_prototype_property(prototype);
6          if match_properties_of(prototype, packet) then
7              prototype_list ← get_prototype_list(prototype);
8              for each prototype_dependency on dependencies(prototype) do
9                  matched_dependency ← FALSE;
10                 for each stored_packet on get_dependency_prototype_list(prototype_dependency) do
11                     if match_properties_dependency(prototype_dependency, packet, stored_packet) then
12                         associate(packet, stored_packet, property), matched_dependency ← TRUE;
13                         goto next_dependency;
14                     end
15                 end
16                 if !matched_dependency then
17                     goto next_prototype
18                 end
19             end
20             if prototype_determines_property(prototype) then
21                 associations_list ← get_associations(packet) report_property_pass(property, packet,
                   associations_list) delete_from_prototype_lists(associations_list)
22             end
23             else
24                 push(prototype_list, packet)
25             end
26         end
27         next_prototype;
28     end
29 end
30 Function timeout_thread(), sleep(global_timeout_value);
31 for each prototype on prototype_packets do
32     property ← get_prototype_property(prototype);
33     for each stored_packet on get_prototype_list(prototype) do
34         associations_list ← get_associations(stored_packet);
35         report_property_fail(property, stored_packet, associations_list);
36         delete_from_prototype_lists(associations_list);
37     end
38 end
```

After, it will load all the properties (formalized requirements) that have to be tested (Line 4-5), and match each packet with the properties in chronological order due to the grammar lead nodes match properties. In this step, only the packets needed for the current property will be saved and tackled. The other irrelevant packets will be discarded in order to accelerate the testing process. In line 6-7, the program will check the properties that can be matched without the use of any dependencies, and list where to store packets if it needs to be stored.

In the following loop (Line 8-19), if prototype doesn't have dependencies then this will never go inside. In the loop, the program will check all the captured packets with the

dependencies. Meanwhile, if it is not a type of prototype package, the program will not keep looking. (Line 11-13). After this loop, the program will report the success and dropped the packets already tested (Line 20-22). This process will keep running until all the properties have been checked. When finishing the checking process, it will report the testing result and empty the buffer immediately in order to make good use of the limited memory (Line 23-29).

Also, we use a *timeout_thread* to execute timeout function on the packets (Line 30-38), and the global timeout value is determined by an expert on the protocol.

5.4 Experiments

In the experiments, to verify and test the approach, our methodology are implemented into a real-time IMS communications environment, and results from testing several properties online are obtained.

5.4.1 Environment

We still use IMS as our environment, since it aims at facilitating the access to voice or multimedia services in an access independent way, which is a perfect platform for online testing. Most communication with its core network and between the services is done using the Session Initiation Protocol (SIP) [7].

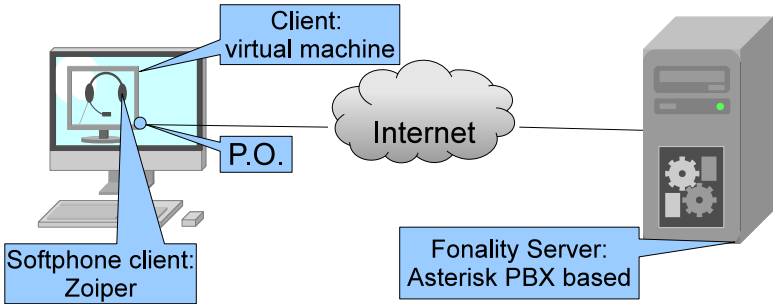


FIGURE 5.8: Experiments environment

Different from our other experiments on SIPp, the communication traces here were obtained through ZOIPER[2] which is a VoIP soft client, meant to work with any IP-based communication systems and infrastructure. SIPp is a load testing tool which simulate client behaviors, and ZOIPER is a real VoIP soft client tool with real human actions on the softphone. As we mentioned before, we are always pursuing the most

[2]http://www.zopier.com/softphone/

suitable environment which satisfies our testing aims. Since we are performing online testing, apparently the traces obtained from real human actions on the softphones are more suitable here.

We run four ZOIPER VoIP clients on the virtual machines using VirtualBox for Mac version 4.2.16. On the other side, the server is provided by Fonality[3], which is running Asterisk PBX 1.6.0.28-samy-r115. As Figure 5.8 shows, the tests are performed in the virtual machines by opening a live capture on the client local interface. This live capture is processed by the clients using an implementation of the formal approach above mentioned and was developed in C code.

5.4.2 Test Results

For better understanding how our approach works, we illustrate a simple use case tested on one of the clients. As shown in Figure 5.9, we have a SIP requirement to be tested: "Every 2xx response for **INVITE** request must be responded with an ACK within 2s", which can be formalized to a formula:

$$\forall_x(request(x) \wedge x.method = \textbf{INVITE} \rightarrow \exists_{y>x}(responds(y,x) \wedge success(y))$$
$$\rightarrow \exists_{z>y}(ackResponse(z,x,y) \wedge withintime(z,y,2s)))$$

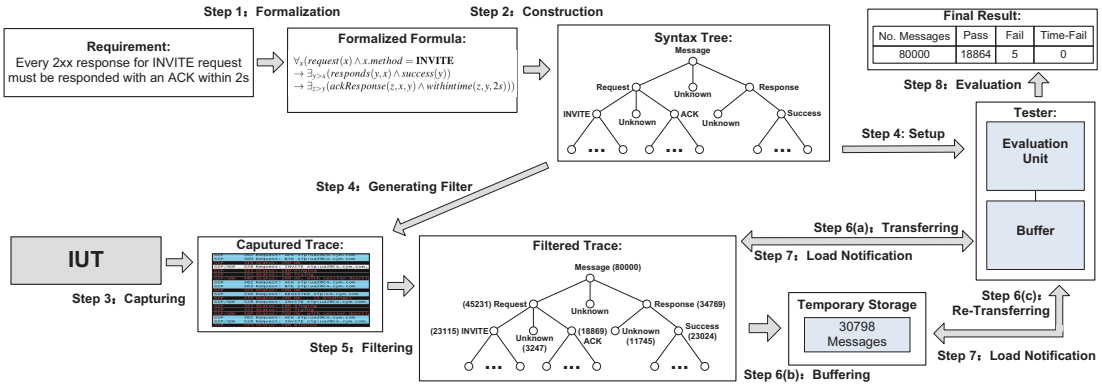


FIGURE 5.9: Use case for Testing Process

This formula is transformed to a syntax tree. When the syntax tree is generated and transferred to the IUT monitor, it starts to capture the trace and apply the syntax tree as a filter (step 3 and 4) for captured messages. Meanwhile, the syntax tree will be applied in the tester as requirement. Once the captured trace is filtered into different sets (step 5), it checks the Load Notification value first. Currently, the Load Notification value equals to N, which makes the tester available to test incoming traces. Then all incoming traces are sent to the tester directly (step 6a).

[3]http://www.fonality.com

As soon as the tester receives the trace, it tests the trace through the formalized property. When the tester is almost reaching to its maximum capacity, it sends a load notification value Y back to the monitor (step 7 and 8). In this case, all incoming traces will be stored in the temporary storage (step 6b) until the tester recovers to an available state (step 6c). Finally, after our 2 hours testing process, we got 18,864 '*Pass*' verdicts, 5 '*Fail*' verdicts caused by violation of the time constraint and no *Time-Fail* verdicts.

Secondly, we test our approach in a more complex environment. It has been performed to concurrently test five properties on a huge set of messages: "Prop.1: Every request must be responded", "Prop.2: Every request must be responded within 8s", "Prop.3: Every **INVITE** request must be responded", "Prop.4: Every **INVITE** request must be responded within 4s" and "Prop.5: Every **REGISTER** request must be responded".

Properties	Total Msgs	Filtered Msgs	Rate	Pass	Con-Fail	Per-Fail	Time-Fail	Incon	Data-Inc
Prop.1	2,324,506	1,631,797	70.19%	631,271	0	0	61,432	52	2,164
Prop.2	2,324,506	1,631,797	70.19%	498,124	0	194,579	0	52	2,164
Prop.3	2,324,506	1,979,904	85.17%	314,923	0	0	29,673	14	1,086
Prop.4	2,324,506	1,979,904	85.17%	247,257	0	97,339	0	14	1,086
Prop.5	2,324,506	2,259,032	97.18%	61,550	0	0	3,924	6	371

TABLE 5.1: Online Testing result for Properties

The table 5.1 shows a snapshot of temporary testing verdicts after 3 hours online continuously testing. Benefited from the filtering function, more than 70% irrelevant messages are filtered out before testing process, which apparently reduces the cost of computing resources. Further, numbers of *Per-Fail* and *Time-Fail* verdicts can be observed. *Time-Fail* verdicts in Prop.1, Prop.3 and Prop.5 indicate that there are 61432, 29673 and 3924 messages respectively that cannot be observed within the timeout, in other words, they are lost during the communication between the client and the server.

Besides, the '0' *Fail* verdict indicates there is no error observed in the data portion for these three properties currently. On the other side, *Per-Fail* verdicts reported in Prop.2 and Prop.4 indicate that there are 194579 and 97339 messages that cannot satisfy the time requirement. These *Per-Fail* verdicts include the *Time-Fail* verdicts reported in Prop.1 and Prop.3, since lost messages also violate the time requirement. In the whole experiment, no *Con-Fail* verdict is reported which indicates that no error has been found in the data portion during the test.

Moreover, several '*Inconclusive*' verdicts indicating the numbers of pending procedures for each property can be observed. We also used the control-portion-only buffering mechanism to test the usage of '*Data-Inc*'. All the buffered messages without data portion are successfully reported as '*Data-Inc*' shown in Table 5.1. Since they take a

tiny proportion of whole traces (between 0.015% and 0.09%), we conclude them as *Fail* in the worst-case. Meanwhile, during the experiments, the CPU occupancy rate of the machine we used as tester is always less than 20%, and the memory usage is below 1GB. During the whole testing process, our approach successfully handled this huge set of messages and did not suspend.

5.5 Conclusions

In this chapter, we introduces a novel online approach to test conformance and performance of network protocol implementation. Our approach allows to define relations between messages and message data, and then to use such relations in order to define the conformance and performance properties that are evaluated on real protocol traces. The evaluation of the property returns a *Pass*, *Con-Fail*, *Per-Fail*, *Time-Fail*, *Data-Inc* or *Inconclusive* result, derived from the given trace.

The approach also includes an online testing framework. To verify and test the approach, we design several SIP properties to be evaluated by our approach. Our methodology has been implemented into an environment which provides the real-time IMS communications, and the preliminary results from testing several properties online have been obtained successfully. This preliminary work on online testing is published in [89].

From the results, we find out that applying our approach under billions of messages and extending more testers in a distributed environment will be our future works. In that case, the efficiency and processing capacity of the approach will be scalably tested. Meanwhile, we will work on the optimization of our algorithms to severe situations in case of several related P.Os, and try to use SVM for predicating *Data-Inc* verdicts.

Chapter 6

General Conclusion

"Prediction is very difficult, especially about the future."
– Niels Bohr (1885 – 1962)

The main objective of the presented work is to address some of the issues related to passive testing for conformance and performance, particularly in the context of message-based protocols.

We firstly presented a state of the art of conformance, performance testing techniques in the Chapter 2. In modern message-based protocols, while the control part still plays an important role, data is essential for the execution flow. Input/output causality cannot be assured since many outputs may be expected for a single input. Moreover, when traces are captured on centralized services, many equivalent messages can be observed due to interactions with multiple clients. Although the traces are finite, the number of related packets may become huge and the properties to be verified complex. Thus, we found out that a passive testing approach for communicating protocols based on the formal specification of functional requirements is required.

For solving these issues, we presented our initial approach for conformance testing of IMS protocols, through a real communicating environment in the Chapter 3. The results are positive, the implemented approach allows to define and test complex data relations efficiently, and evaluate the properties successfully. Besides, as described in the Section Discussion, some improvements can be proposed as future works for performance testing, such as: Testing the accessibility and loss rate of traces by measuring the time complexity, Introducing a timer function to the approach for testing the communication latency.

Moreover, we guess that some properties need, for various reasons as mentioned in the work, to be specified using timers. Since many performance related properties cannot be

specified, and many benefits can be brought to the test process if both conformance and performance testing inherit from the same approach, it raised our interest to create a passive performance testing approach for communicating protocols based on the formal specification of the time related requirements.

Then, for satisfying the needs, we presented our main contribution – a passive performance testing approach for communicating protocols in Chapter 4. We detailed the modified syntax and semantics of formulas for satisfying the needs in formalizing performance requirements. Also for solving the indeterminacy problems existed in non-positive verdicts, we introduce a four-valued semantics {'*Pass*', '*Con-Fail*', '*Per-Fail*', '*Inconclusive*'} in our formalism. Then, we explained our evaluating algorithm and the relevant experiments results. The results showed the functionality and flexibility of our approach. Meanwhile, we proposed several performance indicators for SIP, and tested them in the same environment.

During the experiments, we observed that when we are testing the performance of protocols, lots of P.Os are implemented on each IUT in the networks. It can provide a perfect environment for performing distributed testing. Then, it raised our interest on properly synchronizing different testers and test other protocols to prove the universality of our approach. For solving these issues, we proposed a distributed passive testing framework, and we implemented and tested it through XMPP properties in IoT environments. The results from testing several properties on large traces have been obtained with success. The problem of synchronization is tackled and the functionality and flexibility of our approach are shown.

Since our research is based on collecting traces from the implemented frameworks, we are performing off-line testing processes. For satisfying the increasing need of run-time monitoring and testing process, we have to practice our approach online. With online testing approaches, the collection of traces is avoided and the traces are eventually not finite. Indeed, testing a protocol at run-time may be performed during a normal use of the system without disturbing the process. The traces are observed and analyzed on-the-fly to provide test verdicts and no trace sets should be studied a posteriori to the testing process. In this case, we described the architecture and testing process of our approach for online testing in Chapter 5. We also explained the new definitions of online testing verdicts '*Time-Fail*', '*Data-Inc*' and '*Inconclusive*' into details.

Our online framework is designed to test them at run-time, with new verdicts '*Time-Fail*', '*Data-Inc*' and '*Inconclusive*' representing unobserved message within timeout, untested data portion and uncertain status respectively. In order to demonstrate the efficiency of our online approach, we successfully applied it on a real IMS communicating environment. The premier results proved the preciseness and efficiency of our

approach. From the experiments, we found out that applying our approach under billions of messages and extending more testers in a distributed environment will be our future works. In that case, the efficiency and processing capacity of the approach will be scalably tested. Meanwhile, we will work on the optimization of our algorithms to severe situations in case of several related P.Os, and try to use SVM for predicating *Data-Inc* verdicts.

6.1 Perspectives

In this subsection, we will briefly describe the perspectives of our future works.

Terminate State

As we mentioned in Chapter 3, the terminate state in a property like "The session MUST be terminated after a **BYE** request" is complicated to be formalized, due to the difficulty of detecting the 'terminated' state. Indeed, in our case we do not have any complete formal specification available and we can not stimulate the IUT. Moreover, we should ensure that no more messages will be exchanged after the 'terminated' state, which indicates that we need to keep monitoring the transaction even after it terminates. It is time consuming and unpredictable. But it is still an interesting work we will focus on in the future.

Standardized Benchmark System on Protocols

As we concluded in Chapter 4, building a standardized performance testing benchmark system for protocols would be the work we will focus on in the future. Since we already shown that our approach can test the performance of SIP and XMPP, then extending more performance indicators based on the RFCs and building benchmark systems for both protocols will be interesting. In that case, the efficiency and processing capacity of the system when massive sub testers are performed would be the crucial point to handle, leading to an adaptation of our algorithms to more complex situations. Also, we will try to test other interesting protocols, in order to build a universal protocol testing tool.

Online distributed testing

Also from the results we got in Chapter 5, we find out that applying our approach under billions of messages and extending more testers in a distributed online environment will be another future work for us. In that case, the efficiency and processing capacity of the approach will be scalably tested. Meanwhile, we will work on the optimization of our

algorithms to severe situations in case of several related P.Os, and try to use SVM for predicating *Data-Inc* verdicts.

Bibliography

[1] The Institute of Electrical and Eletronics Engineers. IEEE Standard glossary of software engineering terminology. IEEE Standard, September 1990.

[2] ETSI. Methods for testing and specification (MTS); Conformance test specification for SIP. 2004. Version: ETSI TS 102 027-2 V4.1.1.

[3] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20 (4):14, 2011.

[4] ISO/IEC 9646-1. ISO/IEC information technology - open systems interconnection - Conformance testing methodology and framework - Part 1: General concepts. Technical report, ISO, January 1994.

[5] E. Bayse, A. Cavalli, M. Nunez, and F. Zaidi. A passive testing approach based on invariants: application to the wap. *Computer Networks*, pages 48(2):247–266, 2005.

[6] Robert M Hierons, Paul Krause, Gerald Luttgen, and Anthony J. H. Simons. Using formal specifications to support testing. *ACM Computing Surveys*, page 41(2):176, 2009.

[7] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, and J. Peterson. SIP: Session initiation protocol. 2002. RFC-3261.

[8] V. Stolz. Temporal assertions with parametrized propositions. *Journal of Logic and Computation*, pages 20(3):743–757, 2008.

[9] Gerardo Morales, Stephane Maag, Ana Cavalli amd Wissam Mallouli, and EM De Oca. Timed extended invariants for the passive testing of web services. In *Proceedings of 2010 IEEE International Conference on Web Services*, pages 592 – 599, 2010.

[10] D. Lee and R.E. Miller. Network protocol system monitoring-a formal approach with passive testing. *IEEE/ACM Transactions on Networking*, pages 14(2):424–437, 2006.

[11] H. Ural and Z. Xu. An EFSM-based passive fault detection approach. In *Proceedings of the 19th IFIP TC6/WG6.1 international conference*, pages 335–350, 2007.

[12] Elaine J. Weyuker and Filippos I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, 2000.

[13] Margus Veanes, Colin Campbell, Wolfram Schulte, and Nikolai Tillmann. Online testing with model programs. In *Proceedings of the 10th European Software Engineering Conference*, pages 273–282, 2005.

[14] Tien-Dung Cao, Patrick Félix, Richard Castanet, and Ismail Berrada. Online testing framework for web services. In *Proceedings of third International Conference on Software Testing, Verification and Validation*, pages 363–372, 2010.

[15] Gordon Fraser, Franz Wotawa, and Paul Ammann. Testing with model checkers: a survey. *Software Testing and Verification Reliability*, 19(3):215–261, 2009.

[16] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, pages 78(5):293–303, 2009.

[17] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, Dec 2004.

[18] Marie-Claude Gaudel. Checking Models, Proving Programs, and Testing Systems. In *Tests and Proofs*, pages 1–13, 2011.

[19] Felipe Lalanne and Stephane Maag. A formal data-centric approach for passive testing of communication protocols. *IEEE / ACM Transactions on Networking*, 21 (3):788–801, 2013.

[20] D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *Proceedings of 1997 International Conference on Network Protocols*, pages 113–122, 1997.

[21] Raymond E Miller. Passive testing of networks using a CFSM specification. In *Proceedings of IEEE International Performance, Computing and Communications*, pages 111–116, 1998.

[22] Felipe Lalanne, Stephane Maag, Edgardo Montes De Oca, and Ana Cavalli. An automated passive testing approach for the IMS PoC service. In *Proceedings of 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 535–539, 2009.

[23] Tien-Dung Cao, Trung-Tien Phan-Quang, Patrick Felix, and Richard Castanet. Automated runtime verification for web services. In *Proceedings of IEEE International Conference on Web Services*, pages 76–82, 2010.

[24] F. Cuppens, N. Cuppens-Boulahia, and T. Sans. Nomad. A security model with non atomic actions and deadlines. In *Proceedings of 18th IEEE Computer Security Foundations Workshop*, pages 186–196, 2005.

[25] Z. Li, Y. Jin, and J. Han. A runtime monitoring and validation framework for web service interactions. In *Proceedings of the Australian Software Engineering Conference*, pages 70–79, 2006.

[26] Zheng Li, Jun Han, and Yan Jin. Pattern-based specification and validation of web services interaction properties. In *Proceedings of International Conference on Service Oriented Computing*, pages 73–86, 2005.

[27] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proceedings of the 7th International Workshop on Runtime Verification*, pages 126–138, 2007.

[28] Sylvain Halle and Roger Villemaire. Runtime monitoring of message-based work flows with data. In *Proceedings of 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 63–72, 2008.

[29] J. Calvar, R. Tremblay-Lessard, and S. Halle. A runtime monitoring framework for event streams with non-primitive arguments. In *Proceedings of IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 499–508, 2012.

[30] Feng Chen and Grigore Roşu. Java-mop: A monitoring oriented programming environment for java. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 546–550, 2005.

[31] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. *In Verification, Model Checking, and Abstract Interpretation*, 2937:277–306, 2004.

[32] Klaus Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, 10:14–19, 2012.

[33] Boris Beizer. *Software Testing Techniques (2nd Ed.)*. Van Nostrand Reinhold Co., 1990.

[34] Jan Tretmans. Testing concurrent systems: A formal approach. In *Proceedings of 10th International Conference on Concurrency Theory*, pages 46–65, 1999.

[35] Gilles Bernot. Testing against formal specifications: A theoretical view. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development on Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Development (CCPSD): Vol. 2*, pages 99–119, 1991.

[36] César Andrés, Mercedes G Merayo, and Manuel Núñez. Formal passive testing of timed systems : theory and tools. *Software Testing, Verification and Reliability*, 22 (6):365–405, 2012.

[37] Jerry Zayu Gao, Jacob Tsao, Ye Wu, and Taso H.-S. Jacob. *Testing and Quality Assurance for Component-Based Software*. Artech House, Inc., 2003.

[38] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed performance monitoring: Methods, tools and applications. *IEEE Transactions on Parallel and Distributed Systems*, 5:585–597, 1994.

[39] Robert V. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[40] M.G. Merayo. Passive testing of timed systems with timeouts. In *Proceedings of 12th International Conference on Quality Software (QSIC)*, pages 69–78, Aug 2012.

[41] Sylvain Hallé and Roger Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, 2012.

[42] Michela Taufer and Thomas Stricker. A performance monitor based on virtual global time for clusters of pcs. In *Proceedings of IEEE International Conference on Cluster Computing*, pages 64–72, 2003.

[43] Catalin Dumitrescu, Ioan Raicu, Matei Ripeanu, and Ian Foster. Diperf: An automated distributed performance testing framework. In *Proceedings of 5th International Workshop in Grid Computing*, pages 289–296. IEEE Computer Society, 2004.

[44] Scott Pakin. The design and implementation of a domain-specific language for network performance testing. *IEEE Transactions on Parallel Distributed System*, 18(10):1436–1449, 2007.

[45] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 407–418, 2006.

[46] Marcelo R. N. Mendes, Pedro Bizarro, and Paulo Marques. A framework for performance evaluation of complex event processing systems. In *Proceedings of the Second International Conference on Distributed Event-Based Systems*, pages 313–316, 2008.

[47] Fusheng Wang, Shaorong Liu, and Peiya Liu. Complex RFID event processing. *VLDB Journal*, 18(4):913–931, 2009.

[48] C. Barna, M. Litoiu, and H. Ghanbari. Model-based performance testing: NIER track. In *Proceedings of 33rd International Conference on Software Engineering (ICSE)*, pages 872 –875, May 2011.

[49] P. Arpaia, L. Fiscarelli, G. La Commara, and C. Petrone. A model-driven domain-specific scripting language for measurement-system frameworks. *IEEE Transactions on Instrumentation and Measurement*, 60(12):3756 –3766, Dec. 2011.

[50] C.-H.P. Yuen and S.-H.G. Chan. Scalable real-time monitoring for distributed applications. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2330 –2337, dec. 2012.

[51] E. Knauss and D. Damian. V:issue:lizer: Exploring requirements clarification in online communication over time. In *Proceedings of 35th International Conference on Software Engineering (ICSE)*, pages 1327–1330, 2013.

[52] Huu Nghia Nguyen, Pascal Poizat, and Fatiha Zaïdi. Online verification of value-passing choreographies through property-oriented passive testing. In *Proceedings of 14th International IEEE Symposium on High-Assurance Systems Engineering*, pages 106–113, 2012.

[53] Bachar Wehbi, Edgardo Montes de Oca, and Michel Bourdellès. Events-based security monitoring using MMT tool. In *Proceedings of IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 860–863, 2012.

[54] Fevzi Belli and Michael Linschulte. Event-driven modeling and testing of real-time web services. *Service Oriented Computing and Applications*, 4(1):3–15, 2010.

[55] Xiaoping Che, Felipe Lalanne, and Stephane Maag. A logic-based passive testing approach for the validation of communicating protocols. In *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 53–64, 2012.

[56] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951.

[57] MH Van Emden and RA Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, pages 23(4):733–742, 1976.

[58] K.R. Apt and M.H. Van Emden. Contributions to the theory of logic programming. *Journal of the ACM (JACM)*, 29(3):841–862, 1982.

[59] Rajeev Alur and Thomas A. Henzinger. Real-time logics: complexity and expressiveness. *INFORMATION AND COMPUTATION*, 104:390–401, 1993.

[60] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Proceedings of 13th Formal Methods for Industrial critical systems*, pages 135–149, 2008.

[61] Open Mobile Alliance. Internet messaging and presence service features and functions. 2005. Version: OMA-IMPS-WV-Features-Functions-V1_2-20050125-A.

[62] Open Mobile Alliance. Push to talk over cellular requirements. 2006. Version: OMA-RD-PoC-V2_0-20080421-C.

[63] Hewlett-Packard. *SIPp*. http://sipp.sourceforge.net/, 2004.

[64] J. Rosenberg, H. Schulzrinne, and O. Levin. A Session initiation protocol (SIP) event package for conference state. 2006. RFC-4575.

[65] S. Bradner. Benchmarking terminology for network interconnection devices. 1991. RFC-1242.

[66] S. Bradner and J. McQuaid. Benchmarking methodology for network interconnect devices. 1991. RFC-2544.

[67] Felipe Lalanne, Xiaoping Che, and Stephane Maag. Data-centric property formulation for passive testing of communication protocols. In *Proceedings of the 13th IASME/WSEAS*, ACC'11/MMACTEE'11, pages 176–181, 2011.

[68] Minsu Shin, Mankyu Park, Deockgil Oh, Byungchul Kim, and Jaeyong Lee. Clock Synchronization for One-Way Delay Measurement: A Survey. *Advanced Communication and Networking*, 199(1):1–10, 2011.

[69] David L. Mills. Internet time synchronization: the Network time protocol. *IEEE Transactions on Communications*, 39:1482–1493, 1991.

[70] N Gershenfeld, R Krikorian, and D Cohen. The Internet of Things. *Scientific American*, 291(4):76–81, 2004.

[71] Conner Margery. Sensors empower the "Internet of Things". *Electrical Design News*, pages 32–38, May 2010.

[72] Harald Sundmaeker Woelffle, Patrick Guillemin, Peter Friess, and Sylvie. *Vision and challenges for releasing the Internet of Things.* Publications Office of the European Union, Luxembourg, March 2010.

[73] M Chui, M Löffler, and R Roberts. The Internet of Things. *McKinsey Quarterly*, 3(2):1–9, 2010.

[74] G Kortuem, F Kawsar, D Fitton, and V Sundramoorthy. Smart objects as building blocks for the Internet of Things. *IEEE Internet Computing*, 14(1):44–51, 2009.

[75] E. Welbourne, L. Battle, G. Cole, K. Gould, K. Rector, S. Raymer, M. Balazinska, and G. Borriello. Building the Internet of Things using RFID: The RFID ecosystem experience. *IEEE Internet Computing*, 13(3):48–55, 2009.

[76] Antonio Puliafito, Angelo Cucinotta, Antonino Longo Minnolo, and Angelo Zaia. *Making the Internet of Things a reality: the WhereX Solution*, pages 99–108. Springer New York, 2010.

[77] K Aberer, M Hauswirth, and A Salehi. Middleware support for the Internet of Things. In *Proceedings of the 5th GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, Stuttgart, Germany, 2006. EPFL.

[78] P. Saint-Andre. Extensible messaging and presence protocol (XMPP): Core. 2011. IETF RFC 6120.

[79] R. Klauck and M. Kirsche. Chatty things - making the internet of things readily usable for the masses with XMPP. In *Proceedings of 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 60–69, 2012.

[80] Michael Kirsche and Ronny Klauck. Unify to bridge gaps: Bringing XMPP into the internet of things. In *Proceedings of 2012 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 455–458, 2012.

[81] Sven Bendel, Thomas Springer, Daniel Schuster, Alexander Schill, Ralf Ackermann, and Michael Ameling. A service infrastructure for the internet of things based on XMPP. In *Proceedings of IEEE International Conference on Pervasive Computing and Communications (PerCom)*, San Diego, CA, USA, 2013.

[82] Xiaoping Che and Stephane Maag. Passive testing on performance requirements of network protocols. In *Proceedings of International Workshop on Network Management and Monitoring (NetMM 2013)*, pages 1439 – 1444, 2013.

[83] Xiaoping Che and Stephane Maag. A formal passive performance testing approach for distributed communication systems. In *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 74–84, 2013.

[84] Xiaoping Che and Stephane Maag. Testing protocols in internet of things by a formal passive technique. *SCIENCE CHINA Information Sciences*, 57(3):1–13, 2014.

[85] Xiaoping Che and Stephane Maag. A passive testing approach for protocols in internet of things. In *Proceedings of IEEE International Conference on Internet of Things and Cyber, Physical and Social Computing (iThings/CPSCom)*, pages 678–684, Aug 2013.

[86] Robert E. Noonan. An algorithm for generating abstract syntax trees. *Computer Languages*, 10(3-4):225–236, 1985.

[87] Mark R. Brown and Robert E. Tarjan. A fast merging algorithm. *Journal of ACM*, 26(2):211–226, 1979.

[88] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[89] Jorge Lopez, Xiaoping Che, and Stephane Maag. An online passive testing approach for communication protocols. In *Proceedings of the 9th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 32–37, 2014.