A Logic-based Passive Testing Approach for the Validation of Communicating Protocols

Xiaoping Che, Felipe Lalanne and Stephane Maag
Telecom SudParis, CNRS UMR 5157, 9 Rue Charles Fourier, 91011 Evry, France

Keywords: Formal Methods, Passive Testing, Monitoring, Protocols, IMS/SIP.

Abstract: Conformance testing of communicating protocols is a crucial step to the validation of systems. Formal approaches provide many keys to test efficiently these protocols. These approaches are divided in two main sets: active and passive testing techniques. While they both have their own advantages and drawbacks, passive testing techniques are the only ones that can be applied when the controllability of the system interfaces is unavailable or when the implementation under test cannot be stimulated in runtime. In this paper, we present a novel logic-based passive testing approach. We aim at formally specifying protocol properties in order to check them on real execution traces. Based on algorithms defined in a previous paper, a prototype is here developed and experienced. In order to evaluate and assess our technique, we present experiments through a set of IMS/SIP properties and long-size execution traces. We finally provide relevant verdicts and discussions.

1 INTRODUCTION

Two main types of formal approaches can be applied to test the conformance 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. They are called *passive* since they do not modify the runtime behavior of the IUT. 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 (Lee and Miller, 2006) (when available) or as a set of formally specified properties (Bayse et al., 2005) 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 (Hierons et al., 2009), 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. SIP (Rosenberg, J. and Schulzrinne, H. and Camarillo, G. and Johnston, A. and Peterson, J., 2002)), 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 this work, we present a passive testing approach for communicating protocols based on the formal specification of the requirements and their analysis on collected runtime execution traces. In Section 2, we present the related works. While most of the approaches are based on the study of protocol control parts, we herein focus on the causality of observed events (i.e. packets). In Section 3, a Horn based logic is defined to specify the properties to be

Che X., Lalanne F. and Maag S..
A Logic-based Passive Testing Approach for the Validation of Communicating Protocols.
DOI: 10.5220/0003990100530064
In *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2012), pages 53-64
ISBN: 978-989-8565-13-6
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

verified. Both the syntax and a three-valued semantics are provided. The semantics defines satisfaction within the truth values $\{$*true, false, inconclusive*$\}$, respectively indicating that the property is satisfied on the trace, not satisfied and that no conclusion can be provided. An algorithm has been defined in a previous work (Maag and Lalanne, 2011) to evaluate the satisfaction of properties on off-line traces. Our approach has been implemented and relevant experiments are depicted. They have been performed in an customized architecture to simulate the type of communication performed in the IP Multimedia Subsystem (IMS) architecture. The test architecture allows us to effectively assess our approach by tackling several interesting issues in Section 4. Besides, we discuss in Section 5 the obtained verdicts as well as the efficiency and performance of our approach. Finally, we conclude our paper in Section 6.

2 RELATED WORK

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 (Hierons et al., 2009) the authors present a description of the state of the art and theory behind these techniques. Passive testing techniques are used to test already deployed platforms or when direct access to the interfaces is not available. Most of these techniques only consider control portions (Lee and Miller, 2006; Ural and Xu, 2007), data portion testing is approached by evaluation of traces in state based 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, and only consider a model (if available) for potential verification of the properties. Our research is also inspired from the runtime monitoring domain. Though runtime 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 (Leucker and Schallhart, 2009) provide a good survey in this area.

In (Bayse et al., 2005), an invariant approach taking into account control parts has been presented. In (Morales et al., 2010), 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.

Although closer to runtime monitoring, the authors of (Cao et al., 2010) propose a framework for defining and testing security properties on Web Services using the Nomad (Cuppens et al., 2005) 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. In (Barringer et al., 2004), the authors propose a logic for runtime monitoring of programs, called **EAGLE**, that uses the recursive relation from LTL $\mathbf{F}\phi \equiv \phi \vee X\phi$(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. Constraints 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.

3 FORMAL PASSIVE TESTING APPROACH

3.1 Basics

A message in a communication protocol is, using the most general possible view, a collection of data fields belonging to multiple domains. Data fields in messages, are usually either *atomic* or *compound*, i.e. they are composed of multiple elements (e.g. a URI sip: name@domain.org). Due to this, we also divide the types of possible domains in *atomic*, defined as sets of numeric or string values[1], or *compound*, as follows.

Definition 1. *A compound value v of length $k > 0$, is defined by the set of pairs*

[1]Other values may also be considered atomic, but we focus here, without loss of generality, to numeric and strings only.

$\{(l_i, v_i) \mid l_i \in L \wedge v_i \in D_i \cup \{\varepsilon\}, i = 1...k\}$,where $L = \{l_1, ..., l_k\}$ is a predefined set of labels and D_i are data domains, not necessarily disjoint.

In a *compound* value, in each element (l, v), the label l represents the functionality of the piece of data contained in v. The length of each compound value is fixed, but undefined values can be allowed by using ε (null value). A *compound domain* is then the set of all values with the same set of labels and domains defined as $\langle L, D_1, ..., D_k \rangle$. Notice that, D_i being domains, they can also be either *atomic* or *compound*, allowing for recursive structures to be defined. Finally, given a network protocol P, a compound domain M_p can generally be defined, where the set of labels and element domains derive from the message format defined in the protocol specification. A *message* of a protocol P is any element $m \in M_p$.

Example. A possible message for the SIP protocol, specified using the previous definition is

$$m = \{(method, \textbf{`INVITE'}), (status, \varepsilon), (from, \text{`john@b.org'}), (to, \text{`paul@b.org'}), (cseq, \{(num, 10), (method, \textbf{`INVITE'})\})\}$$

representing an **INVITE** request (a call request) from *john@b.org to paul@b.org*. Notice that the value associated to the label cseq is also a compound value, {(num,10), (method, **'INVITE'**}.

Accessing data inside messages is a basic requirement for the current approach. In order to reference elements inside a compound value, the syntax $v.l_1, l_2, ..., l_n$ is used, where v is a compound value, and l_i are labels. In the previous example, *m.cseq.num* references the value associated with the label *num* inside the value for *cseq* (the value 10). If the reference does not exist, it is associated to ε.

A *trace* is a sequence of messages of the same domain (i.e. using the same protocol) containing the interactions of an entity of a network, called the *point of observation* (P.O), with one or more peers during an indeterminate period of time (the life of the P.O).

Definition 2. Given the domain of messages M_p for a protocol P. A trace is a sequence $\Gamma = m_1, m_2, ...$ of potentially infinite length, where $m_i \in M_p$.

Definition 3. Given a trace $\Gamma = m_1, m_2, ...$ a *trace segment* is any finite sub-sequence of Γ, that is, any sequence of messages $\rho = m_i, m_{i+1}, ..., m_{j-1}, m_j (j > i)$, where ρ is completely contained in Γ (same messages in the same order). The order relations $\{<, >\}$ are defined in a trace, where for $m, m' \in \rho, m < m' \Leftrightarrow pos(m) < pos(m')$ and $m > m' \Leftrightarrow pos(m) > pos(m')$ and $pos(m) = i$, the position of m in the trace $(i \in \{1, ..., len(\rho)\})$.

As testing can only be performed in trace segments, in the rest of the document, trace will be used to refer to a trace segment unless explicitly stated.

3.2 Syntax and Semantics for the Tested Properties

A syntax based on Horn clauses is used to express properties. The syntax is closely related to that of the query language Datalog, described in (Abiteboul et al., 1995), for deductive databases, however, extended to allow for message variables and temporal relations. Both syntax and semantics are described in the current section.

3.2.1 Syntax

Formulas in this logic can be defined with the introduction of terms and atoms, as defined below.

Definition 4. A *term* is either a constant, a variable or a *selector variable*. In BNF: $t ::= c \mid x \mid x.l.l...l$ where c is a constant in some domain (e.g. a message in a trace), x is a variable, l represents a label, and $x.l.l...l$ is called a *selector variable*, and represents a reference to an element inside a compound value, as defined in Definition 1.

Definition 5. An *atom* is defined as

$$A ::= p(\overbrace{t, ..., t}^{k}) \mid t = t \mid t \neq t$$

where t is a term and $p(t, ..., t)$ is a predicate of label p and arity k. The symbols = and $\neq$ represent the binary relations "equals to" and "not equals to", respectively.

In this logic, relations between terms and atoms are stated by the definition of clauses. A *clause* is an expression of the form $A_0 \leftarrow A_1 \wedge ... \wedge A_n$, where A_0, called the head of the clause, has the form $A_0 = p(t_1^*, ..., t_1^k)$, where t_1^* are a restriction on terms for the head of the clause (t^* = c | x). $A_1 \wedge ... \wedge A_n$ is called the body of the clause, where A_i are atoms.

A formula is defined by the following BNF:

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

where $A_1, ..., A_n$ are atoms, $n \geq 1$ and x, y are variables. Some more details regarding the syntax are provided in the following

- The $\rightarrow$ operator indicates causality in a formula, and should be read as *"if-then"* relation.

- The $\forall$ and $\exists$ quantifiers, are equivalent to its counterparts in predicate logic. However, as it will be seen on the semantics, here they only apply to messages in the trace. Then, for a trace $\rho, \forall_x$ is equivalent to $\forall(x \in \rho)$ and $\forall_{y<x}$ is equivalent to $\forall(y \in \rho; y < x)$ with the '$<$' indicating the order relation from Definition 3. These type of quantifiers are called *trace temporal quantifiers*.

3.2.2 Semantics

The semantics used on this work is related to the traditional Apt–Van Emdem–Kowalsky semantics for logic programs (Emden and Kowalski, 1976), however we introduce an extension in order to deal with messages and trace temporal quantifiers. We begin by introducing the concept of substitution (as defined in (Nilsson and Maluszynski, 1990)).

Definition 6. A *substitution* is a finite set of bindings $\theta = \{x_1/t_1, ..., x_k/t_k\}$ where each t_i is a term and x_i is a variable such that $x_i \neq t_i$ 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.

Given $K = \{C_1, ..., C_p\}$ a set of clauses and $\rho = m_1, ..., m_n$ a trace. An *interpretation*[2] *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).
– A ground *atom*[3] $A = p(c_1, ..., c_k)$ 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.
– 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 (Emden and Kowalski, 1976) 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 (Bayse et al., 2005) and runtime monitoring (Bauer and Leucker, 2007), for evaluations on finite traces. This issue is further detailed in one of our technical reports. (Maag and Lalanne, 2011).

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 $\exists_x$

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

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

[2] Called an *Herbrand Interpretation*.

[3] An atom where no unbound variables appear.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

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

3.3 Evaluation Complexity

An algorithm for evaluation of formulas is provided in (Maag and Lalanne, 2011). The algorithm uses a recursive procedure to evaluate formulas, coupled with a modification of SLD (Selective Linear Definite-clause) resolution algorithm (Apt and Van Emden, 1982) for evaluation of Horn clauses. In the same work it is shown that the worst-case time complexity for a formula with k quantifiers is $O(n^k)$ to analyze the full trace, where n is the number of messages in the trace. Although the complexity seems high, this 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 $O(1)$, and for a property $\forall_x \exists_y q(x,y)$, results are obtained in the worst case in $O(n)$. Finally, it can also be shown that a formula with a '$\rightarrow$' operator, where Q are quantifiers

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

has a worst-case time complexity of $O(n^{k+max(l,m)})$, which has advantages with respect to using formulas without the '$\rightarrow$' operator. For instance, evaluation of the formula $\forall_x(\exists_y p(x,y) \rightarrow \exists_z q(z))$ has a complexity of $O(n^2)$, while the formula $\forall_x \exists_y \exists_z (p(x,y) \wedge q(z))$ has a complexity of $O(n^3)$ in the worst case.

4 EXPERIMENTS

The concepts described in the Section 3, along with the above mentioned evaluation algorithm (for a reason of space, the reader interested in this algorithm is invited to see (Maag and Lalanne, 2011)), form part of our implemented framework.[4] The implementation has been performed using Java and is composed of two main modules, as shown by the Figure 1.

The *trace processing* module receives the raw traces 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 input format used was PDML, an XML format that can be obtained from Wireshark[5] 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 *tester* module takes the resulting trace from the trace evaluation along with the clause set and 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.

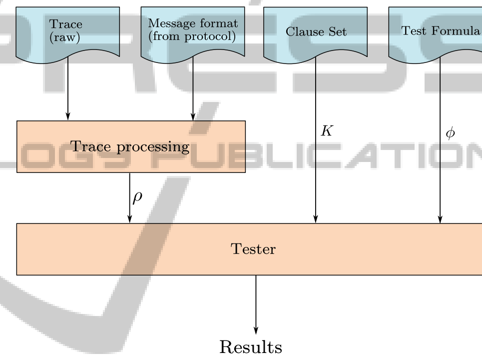

Figure 1: Architecture for the framework.

4.1 IP Multimedia Subsystem Services and SIPp

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 on 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 communications with the core network and between the

[4] Available at http://www-public.int-evry.fr/~lalanne/datamon.html

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

services are done using the Session Initiation Protocol (Rosenberg, J. and Schulzrinne, H. and Camarillo, G. and Johnston, A. and Peterson, J., 2002). Figure 2 shows the core functions of the IMS framework and the protocols used between the different entities.

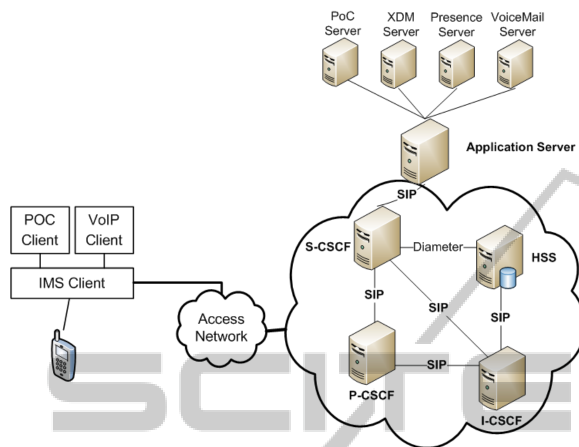

Figure 2: Core functions of IMS framework.

SCIENCE AND TECHNOLOGY PUBLICATIONS

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 a body part (optional) 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 (Open Mobile Alliance, 2005) and the Push to-talk Over Cellular (PoC) service (Open Mobile Alliance, 2006).

For the experiments, traces were obtained from SIPp (Hewlett-Packard, 2004). SIPp is an Open Source implementation of a test system conforming to the IMS, and it is also a test tool and traffic generator for the SIP protocol, provided by the Hewlett-Packard company. It includes a few basic user agent scenarios (UAC and 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 (e.g. subscription including SUBSCRIBE and NOTIFY events). It also support 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. The traces obtained from SIPp contain all communications between the client and the SIP core. Based on these traces and properties extracted from the SIP RFC, tests were performed using our above mentioned methodology and tool.Tests were performed using a prototype implementation of the formal approach mentioned above, using an algorithm developed by us and described in (Maag and Lalanne, 2011).

4.2 Architectures

In the experiments, we designed a simulation on Local Area Network (LAN) architecture for testing. For ensuring the accuracy and authenticity of the results, we construct the environment by using real laptops. 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 on the UAS (Fig.3). The HW configuration of UAS is a CPU- Intel Core i5-2520M 2.50 GHz, 4GB DDR3 and the ones of UACs: CPU- AMD Atholon 64 X2 5200+, 2GB DDR2 and CPU- Intel Core2 Duo T6500 2.10 GHz, 2GB DDR2.

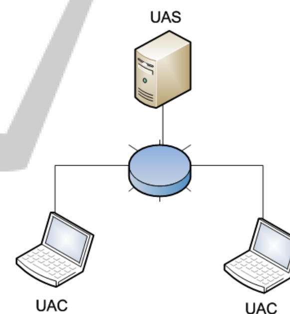

Figure 3: Architecture of our LAN.

4.3 Properties

In order to formally design the properties to be passively tested, we got inspired from the TTCN-3 test suite of SIP (ETSI, 2004) and the RFC 3261 of SIP (Rosenberg, J. and Schulzrinne, H. and Camarillo, G. and Johnston, A. and Peterson, J., 2002). We designed seven properties for the experiments, for the evaluation of each property we used a set of traces containing {500, 1000, 2000, ... , 512000 packets} in order to get exhaustive results.

4.3.1 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 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 (for instance if the same type of message is always without a response). The property evaluated is as follows:

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

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 1. As expected, most traces show only true results for the property evaluation, but inconclusive results can also be observed. Taking a closer look at trace 10, the inconclusive verdict corresponds to **REGISTER** message, with an Event header corresponding to a conference event (Rosenberg et al., 2006), 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 tool crashed after four hours execution.

Table 1: For every request there must be a response.

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	?	?	?	?

4.3.2 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

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

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

However, the analysis of the results 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 traces collected such condition does not hold, therefore a definitive verdict cannot be provided. However it can be shown that the property and the framework allows to detect when the tested property holds on the trace, as Table 2 illustrates.

Table 2: No session can be initiated without a previous registration.

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	?	?	?	?

From the results on Table 2, it can also be seen that the evaluation of this property is much more time consuming than the one on Table 1. By extrapolation from the same time complexity, the trace 9 will take about 23 days for the evaluation where the same trace took only 155s in property 1. Although this is expected given the complexity of the evaluation (n^2 for 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 more efficient to look first for a message with method **INVITE**, then look if 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. But, the syntax as currently specified does not allow that type of definition.

4.3.3 Subscription to Events and Notifications

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

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

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, then their value is set by the evaluation of the *subscribe* clause, shown in the following

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

Here, the = operator, compares the two terms, however if one of the terms 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. 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 update notification was correctly detected by the property evaluation.

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

Table 3: Whenever an update event happens, subscribed users must be notified on the set of traces.

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	?	?	?	?

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

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 4. Notice that 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, one pair **SUBSCRIBE - PUBLISH** was sufficient 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.

Table 4: If an update event is found, then if a previous subscription exists, then a notification must be provided.

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	?	?	?	?

4.3.4 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 evaluated property is as follows:

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

where *success* is defined as

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

and *ackResponse* is defined as

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

Table 5: Every 2xx response for **INVITE** request must be responded with an **ACK**.

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	?	?	?	?

The inconclusive messages observed in traces 5,7,9 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 2 and 5 illustrate, 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 $sequence_a$: "**REGISTER** → 200 → **INVITE**" in property 2, on the other side, in property 4 we are searching a $sequence_b$: "**INVITE** → 200 → **ACK**". As described in property 2, each **INVITE** must have a previous **REGISTER** message. We can infer a new sequence: "**REGISTER** → 200 → **INVITE** → 200 → **ACK**", which means each **ACK** message in the transaction must be corresponded to one **REGISTER** request. Since multiple user conference is not considered in the experiments, we could only conclude: under our particular architecture, the verdict numbers of $sequence_a$ should be equal to the ones of $sequence_b$.

4.3.5 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:

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

where $fail$ is defined as

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

and *ackResponse* is defined as

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

Table 6: Every 300-699 response for INVITE request must be responded with an ACK.

Trace	No.of messages	Pass	Fail	Inconclusive	Time
1	500	10	0	0	3.445
2	1000	18	0	0	10.798
3	2000	49	0	0	34.331
4	4000	91	0	0	137.083
5	8000	165	0	0	557.803
6	16000	367	0	1	1950.656
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	?	?	?	?

The only inconclusive verdict in trace 6 is due to the same phenomenon described in property 1. This property has the same evaluation time complexity as the previous one ($O(n^3)$), which should equal or be close to the ones in the property 4. However, the actual evaluation time does not respect it. From the Table 5 and 6, we can observe that the evaluation times of property 5 are always one higher level than the times of property 4. In order to clarify this issue, we can use the following formula to derive it. Assuming that, the processing time of a message is t, the total number of messages is n, the number of **INVITE** is k, there are x success responses and y fail responses. The evaluation time of property 4 can be derived as

$$T_e = x*t + \{\{n - [\frac{2x}{y} + \frac{(n-k-x-y)}{y}]\} + \{n - 2*[\frac{2x}{y} + \frac{(n-k-x-y)}{y}]\} + \{n - 3*[\frac{2x}{y} + \frac{(n-k-x-y)}{y}]\} + , \ldots , + \{n - (y-1)*[\frac{2x}{y} + \frac{(n-k-x-y)}{y}]\} + \{n - y*[\frac{2x}{y} + \frac{(n-k-x-y)}{y}]\}\}$$

where we know under regular condition $k = x + y$, and the equation becomes:

$$T_e = x*t + t*[n*y - (\sum_{a=1}^{y} a) * \frac{(2x+n-2k)}{y}]$$
$$= t*[y^2 + \frac{(n-2)y}{2} + \frac{n}{2} + k - y]$$

This always gives a positive result and the *evaluation time is proportional to the value of y (the number of fails)*. Conversely in property 5, *the evaluation time is proportional to the value of x (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 be well explained.

4.3.6 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:

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

where *continues* is defined as

$$continues(y,x) \leftarrow y.to = x.to$$
$$\wedge y.Call-ID = x.Call-ID$$
$$\wedge y.From = x.From$$
$$\wedge y.Request-URI = x.Request-URI$$
$$\wedge y.TopVia = x.TopVia$$

and *invite* is defined as

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

As Table 7 illustrates, there is no inconclusive verdict. The evaluation time of each trace almost respects the linear increment of $y = 2x$ (x being the evaluation time of the current trace, y being the evaluation time of next trace), which means the complexity of evaluation is $O(n)$.

5 DISCUSSION

In this section, we discuss an interesting property and possible improvements for performance testing.

Table 7: A CANCEL request should not be sent to cancel a request other than INVITE.

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	?	?	?	?

5.1 The Session MUST be Terminated After a BYE Request

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 as in the last received final response, the same Call-ID, From headers as in the original **INVITE** message, an incremented CSeq value and a method field in the CSeq header set to "BYE".

Differently as the properties before, this one is complicated to formalize, due to the difficulty of detecting the 'terminated' state. Indeed, 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 indicate that we need to keep monitoring the transaction even after it terminates. It is time consuming and unpredictable.

5.2 Time Complexity

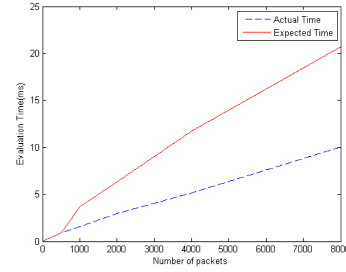

Figure 4: Evaluation time table.

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 one 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 1, 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 4 shows. It means that the actual evaluation time complexity 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 5, 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.

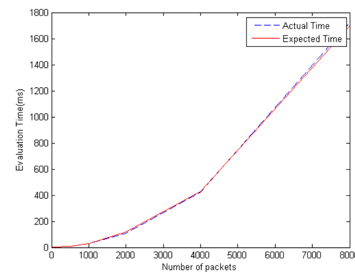

Figure 5: The evaluation time table of numerous inconclusive verdicts.

5.3 Performance Testing

As defined in the RFC1242 (Bradner, S., 1991) and RFC2544 (Bradner, S. and McQuaid, J., 1991), a performance benchmarking can be indicated as:

– Accessibility: if the packet can reach a destination.
– Communication bandwidth: the data transfer rate between two nodes.
– 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 build a passive performance benchmarking system if all these norms are measurable. Currently, the result of property 1 already showed we could certainly test the Accessibility and Frame loss rate by detecting the number of resent packets from the inconclusive verdicts. But if we want to test the Communication Latency, a *timer* function needs to be added in order to test the arrival time.

6 CONCLUSIONS

This paper introduces a novel approach to passive testing of network protocol implementation, with a particular focus on IMS services. This approach allows to define high-level relations between messages or message data, and then use such relations in order to define properties that will be evaluated on the trace. The property evaluation returns a *pass*, *fail* or *inconclusive* result for a given trace. To verify and test the approach, we design several properties for the evaluation. The approach has been implemented into a framework, and results from testing these properties on tremendous traces collected from IMS service.

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 5, 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 this paper, to be specified using timers. In this objective, other perspectives are to enhance our syntax and semantics as well as study the more relevant way of formulating an RFC property.

REFERENCES

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Datalog and Recursion*. Addison-Wesley, 2nd edition.

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

Barringer, H., Goldberg, A., Havelund, K., and Sen, K. (2004). Rule-based runtime verification. *In Verification, Model Checking, and Abstract Interpretation*, pages 277–306.

Bauer, A. and Leucker, M. (2007). Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering and Methodology*, pages 1–68.

Bayse, E., Cavalli, A., Nunez, M., and Zaidi, F. (2005). A passive testing approach based on invariants: application to the wap. *Computer Networks*.

Bradner, S. (1991). Benchmarking terminology for network interconnection devices.

Bradner, S. and McQuaid, J. (1991). Benchmarking methodology for network interconnect devices.

Cao, T.-D., Phan-Quang, T.-T., Felix, P., and Castanet, R. (2010). Automated runtime verification for web services. *IEEE International Conference on Web Services*, pages 76–82.

Cuppens, F., Cuppens-Boulahia, N., and Nomad, T. S. (2005). A security model with non atomic actions and deadlines. *IEEE*.

Emden, M. V. and Kowalski, R. (1976). The semantics of predicate logic as a programming language. *Journal of the ACM*, pages 23(4):733–742.

ETSI (2004). Methods for testing and specification (mts); conformance test specification for sip.

Hewlett-Packard (2004). *SIPp*. http://sipp.sourceforge.net/.

Hierons, R. M., Krause, P., Luttgen, G., and Simons, A. J. H. (2009). Using formal specifications to support testing. *ACM Computing Surveys*, page 41(2):176.

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

Leucker, M. and Schallhart, C. (2009). A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, pages 78(5):293–303.

Maag, S. and Lalanne, F. (2011). A formal data-centric approach for passive conformance testing of communication protocols. Technical report, Telecom Sud-Paris.

Morales, G., Maag, S., Cavalli, A., Mallouli, W., and De Oca, E. M. (2010). Timed extended invariants for the passive testing of web services. *IEEE International Conference of Web Services*.

Nilsson, U. and Maluszynski, J. (1990). *Logic, programming and Prolog*. Wiley, 2nd edition.

Open Mobile Alliance (2005). Internet messaging and presence service features and functions.

Open Mobile Alliance (2006). Push to talk over cellular requirements.

Rosenberg, J., Schulzrinne, H., and Levin, O. (2006). A session initiation protocol (sip) event package for conference state.

Rosenberg, J. and Schulzrinne, H. and Camarillo, G. and Johnston, A. and Peterson, J. (2002). Sip: Session initiation protocol.

Ural, H. and Xu, Z. (2007). An efsm-based passive fault detection approach. *Lecture Notes in Computer Science*, pages 335–350.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS