An Invariant based Passive Testing approach for Protocol Data parts

Felipe Lalanne, Stephane Maag
Institut Telecom / TELECOM SudParis
CNRS UMR 5157
9, rue Charles Fourier
F-91011 Evry Cedex, France
{*Felipe.Lalanne, Stephane.Maag*}@*it-sudparis.eu*

***Abstract*—Conformance of implementations to protocol specifications is essential to assure interoperability between peers in network communications. Monitoring or passive testing techniques are used when no access to the communication interfaces is available or when the normal operation of the system cannot be interrupted. Most monitoring techniques only consider control portion of exchanged messages, usually ignoring the data part. However, as protocols become more complex and message exchange more data intensive, testing for data relations and constraints between exchanged messages becomes essential. In this paper we propose a novel approach for defining such relations as properties called invariants and show how they can be tested directly on traces using logic programming. Experimental results for SIP protocol traces are provided.**

***Keywords*-monitoring; invariant-based testing; data constraints; logic programming; network management**

I. Introduction

Communication standards for network protocols make it possible for different types of systems or different types of implementations to interoperate. Conformance to standards is essential to achieve communication on the Internet. *Formal methods* and *testing* techniques [1] allow to assess the conformance of implementations, usually by generating test cases from some type of specification based on the requirements of the protocol. These test cases are then provided as input to the Implementation Under Test (IUT) and the answers checked against those of the specification.

This approach, however, is not feasible when the IUT is in a production environment where its normal function cannot be disrupted. *Monitoring (or Passive testing)* techniques rely on observing the inputs/outputs provided by normal operation of the IUT, and then attempt to detect *faults*, either by comparing the observed events with the specification [2], [3], or by directly evaluating certain properties on the input/output trace [4], [5], [6]. For this last approach, called *invariant-based testing*, some properties (invariants) are defined, either by experts, directly from the protocol standard, or by extracting them from a formal specification. This techniques allows to test properties such as "*If x happens, then y MUST happen*" or "*For x to happen, then y MUST have happened before*" directly on the trace.

The current work deals with the application of such invariant-based techniques, particularly on the context of IP Multimedia Subsystem (IMS) applications. These kind of applications present interesting challenges for passive monitoring, given the distributed architecture of the IMS, and the utilization of a common protocol, the Session Initialization Protocol[7] (SIP) on most session management procedures, evaluating properties on a trace becomes non-trivial as shown by Figure 1. In [8] we defined some properties for testing on the Push-to-talk Over Cellular[9] (PoC) service traces, based on the protocol standard requirements, for instance

$$INVITE(CSeq = c0, From = u0, To = u1)/\theta, *,\\ \theta/OK(CSeq = c0, From = u0, To = u1)$$

indicates that a SIP *OK* response can only be received if an *INVITE* request was received first. As it can be seen in the Figure 1, the PoC trace also contains *OK* responses for *SUBSCRIBE* and *NOTIFY* requests, used by the Presence[10] service for communication. This produced *false positive* FAIL verdicts when tested on the trace, given that the property cannot distinguish between an *OK* response for an *INVITE* message from other *OK* for other type of request, or even from responses to a different *INVITE* message.

Granted, it can be argued that the defined property is rather simplistic in its definition, but shows the limitation of the expressive power of invariant properties defined this way to express data relations between messages, essential to perform more accurate testing. We believe that a more complete definition of message data and data relations for network protocol testing is necessary. This is the main motivation for the current paper.

In network protocols, communication is achieved through an exchange of messages between peers, where each peer can intermittently act as a receiver or as an emitter of the messages. As protocols evolve, messages become richer in data. Most passive testing approaches consider a small part of the message information called the *control part* (the *INVITE*, *OK* in the previous example), but very few consider the data constraints and relations for the message exchanges. A couple of examples of this are presented in [11] and [12]. In the first one, the authors use an interval refinement

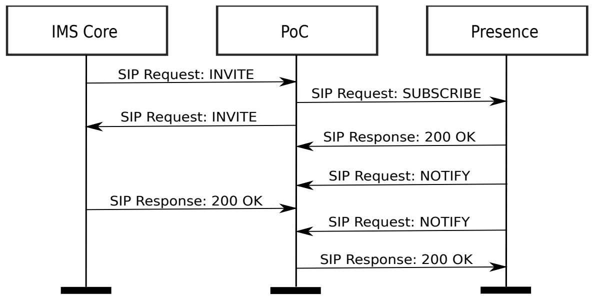

Figure 1. Interactions between applications of the IMS Subsystem. The PoC service, an IMS application, communicates with the Presence application and with the clients throught the IMS Core, all using functions of the SIP protocol.

approach to determine faults in a trace, either in data or state, by evaluation into a EEFSM (Event-driven Extended Final State Machine) formal specification. In the second one, the authors develop an algorithm to verify invariants on an EFSM specification and extract data constraints from them. However these works require a specification of the system under test, and in the industry, a complete specification is rarely available, particularly for large systems. In our work we test properties directly in the trace without the help of any formal specification.

We propose a methodology, based on first-order logic, that makes it possible to formally specify data-oriented invariants, and show how they can be tested in offline traces using a logic programming approach.

The main contributions of our work are:

- We refine the definitions of message and trace using first-order logic concepts, to better consider data domains and properties inside messages.
- With this new definition we describe how to specify data constraints and relations between messages, and define invariants as formulas in first-order logic.
- We prove the soundness of the definition to show that if an invariant verified in a formal specification does not hold in a trace, the corresponding implementation does not conform to the formal specification.
- We show how invariant formulas can be tested in real traces using logic programming and the equivalence with the first-order logic definitions.
- Finally, we present an example on the SIP protocol and show the results of testing in PoC service traces in comparison with our previous work.

II. Preliminaries

In this section we describe some necessary concepts that are the basis for the introduction of data into invariants and to demonstrate the correctness of the definitions. The Event-driven Extended Finite State Machine formalism and first-order logic.

A. Event-driven Extended Finite State Machine

For the purpose of this paper, we use the Event-driven Extended Finite State Machine (EEFSM) formalism, suitable for monitoring techniques, introduced by the authors of [11] and [13].

Definition 1. An Event-driven Extended Finite State Machine (EEFSM) is a 6-tuple $M =< S, s_0, \Sigma, \vec{x}, \vec{y}, T >$ where $S = \{s_0, s_1, \ldots, s_{n-1}\}$ is a finite set of states with $s_0 \in S$ as the initial state, Σ is a finite set of events, where for each $e(\vec{y}) \in \Sigma$, e is the event name and $\vec{y} = \{y_1, \ldots, y_q\}$ is the finite set of event parameters, $\vec{x} = \{x_1, \ldots, x_p\}$ is a finite set of internal variables, and T is a finite set of transitions. For a transition $t =< s, s', e(\vec{y}), P(\vec{x}, \vec{y}), A(\vec{x}, \vec{y}) >$, $s, s' \in S$ are the starting and ending states of the transition, $e(\vec{y})$ is the triggering event, $P(\vec{x}, \vec{y})$ is a predicate and $A(\vec{x}, \vec{y})$ is an action, which is a sequence of assignments $\vec{x} := A(\vec{x}, \vec{y})$, where $\vec{x}$ and $\vec{y}$ are the current variable and event parameter values, respectively.

Example 1. The EEFSM transition t : $s \xrightarrow{?a(y_1),[y_1>x_1],x_2:=y_1+8} s'$, indicates that when the system is in state s, upon reception of event a with parameters y_1, if $[y_1 > x_1]$ then it will assign to variable x_2 the new value of $y_1 + 8$ and finally leave the system in state s'.

B. First-order Logic

Some basic concepts of first order logic are provided since they are necessary to include data for invariants. A more complete reference on first-order theory for computer science can be found in [14].

The first-order logic vocabulary is composed by two sets: a set of *predicate* symbols $\mathcal{P}$ and a set of *function* symbols $\mathcal{F}$. Each type comes with an arity, the number of parameters it expects. *Functions* of arity 0 are called *constants*. The pair $(\mathcal{F}, \mathcal{P})$ is called a logical signature.

The first-order vocabulary rules can be defined with the introduction of terms and formulas, where a term is defined in Backus-Naur form as $t ::= x|f(t,...,t)$ where x is a variable, $f \in \mathcal{F}$ has arity n, and formula is defined as $\phi ::= P(t_1, t_2, ..., t_n)|\neg\phi|\phi \wedge \phi|\phi \vee \phi|\phi \Rightarrow \phi|\forall x\phi|\exists x\phi$ where $P \in \mathcal{P}$ is a predicate of arity n, t_i are terms and x is a variable.

In order to semantically evaluate first-order logic formulas, a *meaning* must be given to the predicate and function symbols, that is, they must be associated with real predicates and functions of a specific context or *universe*. A model $\mathfrak{M}$ is defined as a triplet $< U, (f_{\mathfrak{M}})_{f\in\mathcal{F}}, (P_{\mathfrak{M}})_{P\in\mathcal{P}} >$, where

- U is a non-empty set called universe.
- For each $f \in \mathcal{F}$ with arity n, a concrete function $f_{\mathfrak{M}}$ is defined as $f_{\mathfrak{M}} : U^n \longrightarrow U$
- For every $P \in \mathcal{P}$ with arity n, $P_{\mathfrak{M}} \subseteq U^n$

Finally, in order to evaluate formulas, variables have to be mapped to particular values in the universe. Given a set $\mathcal{X}$

of variables a function $l : \mathcal{X} \to U$ is called an *interpretation*. With this concept, given a model $\mathfrak{M}$ for a signature $(\mathcal{F}, \mathcal{P})$ and a formula ϕ, a *satisfaction relation* is defined as $\mathfrak{M} \models_l \phi$ meaning that the formula ϕ is *true* with respect to $\mathfrak{M}$ under interpretation l.

III. Data in Network Protocols

In a network protocol, a communication peer decides the course of action on the basis of two things: Locally stored state information (including internal data), and data contained in received messages from different peers. In passive testing, while the observed message data is available to the tester, state information and internal data are unknown. In order to test data, focus on the exchange of messages is fundamental, however, a formal definition must be provided first.

A protocol message is, in general, a collection of data fields of different domain, where each data field has a function in the data exchange. The *format* of the message, the function and domain of each field are defined in the requirements specification of the protocol.

The most basic domains of data we can find in protocols, are numeric and alphanumeric (string) values. Basic data domains can be combined in order to define more complex domains. For instance, an email data element can be defined as `mailbox ::= local-part "@" domain`, in order to distinguish the recipient from the sending address.

Traditional passive testing definitions, distinguish between a *control portion* and a *data portion* of messages, this is actually an abstraction necessary for compatibility with model-based testing. In practice, a trace is only a sequence of messages containing data. These messages are classified afterwards, according to their function in the protocol, determined by the data. This function is what constitutes the control portion. Although this is not really necessary for our approach, we will include the distinction between control and data in the definition of a message, in particular to make it compatible with the EEFSM formalism.

Definition 2. A message is a structure $m = e(\vec{x})$ where e is the message label belonging to some finite set L, and $\vec{x} = (x_1, \ldots, x_n)$ is a finite set of message parameters or *message variables*, where each $x_i \in D_j \cup \{\epsilon\}$, with D_j as its data domain (D_j's are not necessarily disjoint). The *null* value is represented by ϵ, where $\epsilon \notin D_j, \forall j \in 1 \ldots n$.

In the definition, e constitutes the control portion of the message and $\vec{x}$ is the data portion. The *null* value ϵ is used to allow messages with dynamic number of variables within the fixed-size set $\vec{x}$.

It is easy to see that this definition of a message is compatible with the one of *event* in an EEFSM and every possible message is an event, however it should be clear that not every possible EEFSM event is a message. From here on, we will denote as $\mathcal{M}$ the domain of all possible messages for a given EEFSM.

In order to reference a particular variable for a message m we will use the symbol '.' to reference it, for instance $m.email$ to reference the variable of name $email$ in message m. This should be considered only as a convention in order to ease the writing of formulas and not a formal definition.

IV. Incorporating data into invariants

In this section we describe how we incorporate data into invariants, by using the previously described logic concepts. We start by defining a new way to represent traces more suitable with a first-order model.

A. Trace representation

Using the traditional model-based approach, a trace can be defined in terms of an EEFSM: Given $M =< S, s_0, \Sigma, \vec{x}, \vec{y}, T >$, we say that the sequence $\mathfrak{T} =< e_1(\vec{y}_1), e_2(\vec{y}_2), \ldots, e_n(\vec{y}_n) >$ is a *trace* for M, if there exist states $s, s_1, \ldots, s_{n-1}, s' \in S$ such that we have the following transitions $s \xrightarrow{e_1(\vec{y}_1);P_1(\vec{x}_0,\vec{y}_1);A_1(\vec{x}_0,\vec{y}_1)} s_1$, $\ldots, s_{n-1} \xrightarrow{e_n(\vec{y}_n);P_1(\vec{x}_{n-1},\vec{y}_n);A_n(\vec{x}_{n-1},\vec{y}_n)} s'$, where $\vec{x}_0$ is t he internal variable state before the first transition, and $P_1, \ldots, P_n$ and $A_1, \ldots, A_n$ are respectively the predicates and actions for the transitions. In order to make the concept of trace more compatible with our current approach, we propose an alternative representation

Definition 3. Let $M =< S, s_0, \Sigma, \vec{x}, \vec{y}, T >$ be an EEFSM and $\mathcal{M}$ the domain of all possible messages for M, the structure $\mathfrak{T} =< R, m_0, Next >$ is defined, where

- $R \subseteq \mathcal{M}$ is a finite, non-empty set of messages.
- $m_0 \in R$ is the first message of the trace.
- $Next \subseteq R \times R$ is a relation that maps a message to its next on the trace. i.e $Next(x, y)$ indicates that x comes immediately after y on the trace. This relation must satisfy the property $Next(x, y) \wedge Next(x, z) \Leftrightarrow y = z$, to restrict to a single successor for each element.

$\mathfrak{T}$ is said to be a *trace* for M if there exist states $s, s_1, \ldots, s_{n-1}, s' \in S$ such that the following transitions exist: $s \xrightarrow{m_1=e_1(\vec{y}_1);P_1(\vec{x}_0,\vec{y}_1);A_1(\vec{x}_0,\vec{y}_1)} s_1, \ldots,$ $s_{n-1} \xrightarrow{m_n=e_n(\vec{y}_n);P_1(\vec{x}_{n-1},\vec{y}_n);A_n(\vec{x}_{n-1},\vec{y}_n)} s'$, where $m_i = e_i(\vec{y}_i) \in R$ and $Next(m_i, m_{i+1})$ holds for every $i = 1 \ldots n-1$.

For convenience we also define the following relations for the trace.

- $Prev(x, y)$ indicates that message x comes immediately before y in the trace.
- $After(x, y)$ the relation indicating that message x comes after message y in the trace.
- $Before(x, y)$ the relation indicating that message x comes before message y in the trace.

For the remainder of the paper, it should be assumed that the variables domain is the trace. That is, the expression $\exists u_1, \ldots, u_n$ is equivalent to $\exists u_1, \ldots, u_n \in \mathfrak{T}$.

Finally, we define $Traces(M)$ as the set of all possible traces for the EEFSM M.

B. A model for protocol traces

As our interest is to interpret invariant properties semantically, it is necessary to first define a model. We define the universe of the model as the trace, where properties are expected to hold.

Given $(\mathcal{F}, \mathcal{P})$ a logical signature, and a trace $\mathfrak{T} = (R, m_0, Next)$, we specify the sets

- $\mathcal{F}' = \mathcal{F}$
- $\mathcal{P}' = \mathcal{P} \cup \{Next, Prev, Before, After\}$.

With $(\mathcal{F}', \mathcal{P}')$ as a new signature, a model for the trace $\mathfrak{M}(\mathfrak{T})$ is constructed as follows

- The universe $U = \mathfrak{T}$.
- Each $P \in \mathcal{P}'$ is specified as a formula on the message variables for the involved messages. For instance let's suppose that each message in the trace contains a variable $index$ indicating the order where each message was collected into the trace. Then the predicate $Next$ can be stated as

$$Next(x, y) \leftarrow x.index = y.index + 1$$

- For each $f \in \mathcal{F}'$ of arity n a concrete function is defined as $f_{\mathfrak{M}(\mathfrak{T})} : \mathfrak{T}^n \rightarrow \mathfrak{T}$. We do not impose any restrictions on the kind of functions that can be defined, other than they must be within the domain of the trace, however, for the examples in this paper we will not make use of them.

C. Invariant definition

As the name implies, an invariant is a property that must be true for every trace from an implementation, formally we define

Definition 4. Let $M =< S, s_0, \Sigma, \vec{x}, \vec{y}, T >$ be an EEFSM, $\mathfrak{M}(\mathfrak{T}) =< \mathfrak{T}, (f_{\mathfrak{M}(\mathfrak{T})})_{f\in\mathcal{F}}, (P_{\mathfrak{M}(\mathfrak{T})})_{P\in\mathcal{P}} >$ be a model for a trace $\mathfrak{T}$ and $\mathcal{X}$ a set of variables, an *invariant* is any formula ϕ such that there exists *at least one* interpretation $l : \mathcal{X} \rightarrow \mathfrak{T}$ where $\mathfrak{M}(\mathfrak{T}) \models_l \phi$ for *every* trace $\mathfrak{T} \in Traces(M)$.

The fact that an invariant must hold *every* trace from the model, immediately imposes a restriction on the type of formulas that are possible, since a trace may not involve a particular property being evaluated. In particular, the definition restricts invariants to formulas of the type $\phi \Rightarrow \psi$. Due to the rules of logical implication, if the first part of the formula is *false*, the value of the formula is immediately *true*. For our approach, we will further restrict to formulas of the type: $\forall x_1, \ldots, x_p \phi \Rightarrow \exists y_1, \ldots, y_q \psi$, making it possible to describe properties such as "*Whenever x happens then y must (have) happen(ed)*".

V. Correctness of checking invariants in a trace

In this section we will first show the correctness of our approach, by using a *trace preorder* implementation relation, and then we show how to test invariants on the trace. We begin by introducing the *trace preorder* implementation relation definition as

Definition 5. Let S and I be two EEFSMs. We say that I *trace conforms* to S, denoted $I \leq_{tr} S$ if and only if $Traces(I) \subseteq Traces(S) \wedge Traces(I) \neq \emptyset$.

This definition of conformance, requires that *every* possible trace generated by the implementation, must be possible in the specification. To prove the soundness of our definitions, we also need to define correctness of an EEFSM trace with respect to an invariant as

Definition 6. Let $\mathcal{I} \overset{\text{def}}{=} \forall x_1, \ldots, x_p \phi \Rightarrow \exists y_1, \ldots, y_q \psi$ be an invariant and $\mathfrak{T}$ a trace from an EEFSM. We say that $\mathfrak{T}$ is *correct* with respect to $\mathcal{I}$ if there exists an interpretation $l : var(\mathcal{I}) \rightarrow \mathfrak{T}$, such that $\mathcal{I}$ holds.

Theorem 1. Let S and I be two EEFSM and $\mathcal{I}$ be an invariant verified on S. Let $\mathfrak{T}$ be a trace recorded from I. If the trace $\mathfrak{T}$ is not *correct* by Definition 6, with respect to $\mathcal{I}$, then I does not conform to S.

The proof is direct with the provided definitions.

Proof: Let $\mathcal{I}$ be a correct invariant and $\mathfrak{T} \in Traces(I)$ a trace that is not correct with respect to $\mathcal{I}$. By Definition 4, $\mathcal{I}$ holds for every trace in S, which means that $\mathfrak{T} \notin Traces(S)$, then $Traces(I) \nsubseteq Traces(S)$, therefore, by Definition 5, I does not conform to S. ■

We conclude this section by describing how we can check the correctness of invariants directly on a real trace by using logic programming. A description of logic programming is not provided, we recommend [15] as a good reference.

In logic programming, a *program* is composed by a set of *clauses* of the type $A_0 \leftarrow A_1 \wedge \ldots \wedge A_n$ where each A_i is an atomic formula (a predicate) that states a fact. Programs allow to test *goals*, or statements for which we want to obtain a truth value. The goal G is valid in the program P if $P \models G$.

In our approach, we can make the parallel between clauses and predicate definitions, and between goals and invariants. In order to test the invariants on a real trace, the trace messages have to be stated using logic programming. To solve this we define for the program the predicate $TraceMsg(index, msg)$ to state that the set of values in msg is a trace message in the position $index$. With this definition, a real trace consists of the set of statements

$$TraceMsg(1, msg_1)$$
$$\vdots$$
$$TraceMsg(n, msg_n)$$

where $msg_1, \ldots, msg_n$ are real message values. This way, on giving the program the goal $TraceMsg(1, x)$, it will respond

$x = msg_1$. The predicate $Next$, for instance, can be defined with the use of $TraceMsg$ as

$$\begin{array}{rcl} Next(x,y) & \leftarrow & TraceMsg(i_1,x) \wedge TraceMsg(i_2,y) \\ & \wedge & i_1 = i_2 + 1 \end{array}$$

These definitions give to the program the same information as the model for the trace defined in Section IV-B, then, a trace is correct with respect to an invariant, if such invariant holds in the corresponding program.

Finally, in order to test the invariants, it is necessary to remove the quantifiers, since most logic programming languages do not support them. As in testing it is more interesting to check the traces that fail the invariant, the quantifiers can be easily removed. Let $\mathcal{I}$ be an invariant

$$\mathcal{I} = \forall x_1, \ldots, x_p \phi \Rightarrow \exists y_1, \ldots, y_q \psi$$

by the definition of logical implication this is equivalent to

$$\mathcal{I} = \exists x_1, \ldots, x_p \neg\phi \vee \exists y_1, \ldots, y_q \psi$$

Since we want to know whether a trace fails an invariant, we need to test the negation of the invariant $\mathcal{I}^{neg} = \neg\mathcal{I}$ which leaves the formula in

$$\mathcal{I}^{neg} \leftarrow \phi \wedge \neg\psi$$

easily tested using logic programming.

VI. Invariant definition for the SIP protocol

The Session Initiation Protocol (SIP) [7] is an application-layer (control) signaling protocol for creating, modifying, and terminating sessions with one or more participants. These sessions include Internet telephone calls, multimedia distribution, and multimedia conferences.

The SIP protocol is a fundamental part in the IMS framework, where the main elements in the core IMS network behave as different SIP entities (user agent server, user agent client, proxy) and communicate between them using this protocol.

In this section we apply the concepts previously explained to the SIP protocol. First, we define the message structure, then we define some predicates and their meaning in terms of variables and finally, we use them to specify some invariants.

A. SIP Messages

A SIP Message is either a request from a client to a server, or a response from a server to a client. The basic message format is the one established by RFC 3621, as follows

```
generic-message = start-line
                  *message-header
                  CRLF
                  [ message-body ]
start-line      = Request-Line | Status-Line
```

When the message is a request, it contains a Request-Line providing the **Method Request-URI** and **SIP-Version**. SIP requests contain a Status-Line as a start-line, containing the information **SIP-Version, Status-Code** and **Reason-Phrase**.

With the available information, for our approach we define the following message variables.

- $method \in RequestMethod$, where $RequestMethod = \{$`'REGISTER'`, `'INVITE'`, `'ACK'`, `'CANCEL'`, `'BYE'`, `'OPTIONS'`$\}$. For this document we only consider the *Method* field, although a message variable could be easily defined to contain information for the request URI.
- $statusCode \in Code$, where $Code \subset \mathbb{N}$ is the set of possible status codes for the protocol.

We define a message m in SIP as at least the sequence $m = (method, statusCode)$. As it can be noticed, we are not including the control part in the message definition. Although it depends on the specification, the control part here would be a combination of the method and status code, since they define the function of the message, here we will just specify them as data variables. Along with the message definition, the following predicates will also be useful to identify a message as request or response

$$Request(m) \leftarrow m.method \neq \epsilon \wedge m.statusCode = \epsilon$$
$$Response(m) \leftarrow m.method = \epsilon \wedge m.statusCode \neq \epsilon$$

1) SIP Requests: According to the standard *"A valid SIP request formulated by a UAC MUST, at a minimum, contain the following header fields: To, From, CSeq, Call-ID, Max-Forwards, and Via; all of these header fields are mandatory in all SIP requests"*. In addition to the request line, these header fields allow the main functionality for the main SIP routing services. Adding them to the message definition, $m = (method, statusCode, to, from, cSeqNum, cSeqMethod, callId, maxForwards, via)$.

We do not detail on the on the function of this headers, but the reader is invited to refer to section 8.1.1 of [7] for further information.

2) SIP Responses: The basic procedures on how to construct a response to a request are specified in section 8.2.6.2 of the RFC. They can be translated into the following predicate

$$\begin{array}{rcl} RespondsTo(x,y) & \leftarrow & Response(x) \wedge Request(y) \\ & \wedge & x.from = y.from \\ & \wedge & x.callId = y.callId \\ & \wedge & x.cSeq.seq = y.cSeq.seq \\ & \wedge & x.cSeq.method = y.cSeq.method \\ & \wedge & x.via = y.via \\ & \wedge & x.to = y.to \end{array}$$

Meaning that if the message x is a response to the message y, then the indicated variable values must be the same in both. Using this property and, given a trace $\mathfrak{T}$ for an implementation, we can define the invariant $\mathcal{I}_1 \leftarrow \forall x \in \mathfrak{T}, Request(x) \wedge x.method \neq$ `'ACK'` $\Rightarrow \exists y \in \mathfrak{T}, RespondsTo(y,x) \wedge After(y,x)$, meaning that if there is a request message in the trace, there must exist a response message to it. In the same way, the invariant $\mathcal{I}_2 \leftarrow \forall x \in \mathfrak{T}, Response(x) \Rightarrow \exists y \in \mathfrak{T}, RespondsTo(x,y) \wedge Before(y,x)$, indicates that if there exists a response message in the trace, then there must be a request for it at some point before it in the trace. This invariant is a more general version of the one presented in the introduction, stating the relation between INVITE and OK. This one states a relation between every request and response, allowing to test more cases than the original one.

B. SIP Registration

In SIP, registration allows a user to be located by other peers. By sending a message to a server called the registrar, the user agent client (UAC) informs the server of the location (IP address, network) where a SIP user (specified by an URI) can be located.

In terms of the protocol, in order to authenticate, the user agent client sends a REGISTER request to the server and the server responds with a 200 OK response when the registration is accepted. A series of messages are sent in between, in order to perform authentication (if required by the server), determine capabilities of the server, and others. However for the purpose of this document we only consider whether a registration was successful. To achieve this, we define the following predicate

$$\begin{array}{rcl} Registration(x,y) & \leftarrow & ResponseTo(y,x) \\ & \wedge & x.method = \text{‘REGISTER’} \\ & \wedge & y.statusCode = 200 \end{array}$$

C. SIP Session Establishment

In order to establish a session between two users, one of the UACs sends an INVITE request to the server which contacts the second UAC. An exchange of messages takes place in order to setup a session (using session description protocol), different provisional responses are sent to the first UAC to indicate the status of the process (100 Trying, 180 Ringing, etc). The session is established after the negotiation where a 200 OK message is sent to the originating UAC, and an ACK request is sent back to the server to acknowledge the reception.

The ACK message, in accordance to section 17.1.1.3 in the RFC, is constructed with the Call-ID, From and Request-URI values from the original request, and the To header from the response being acknowledged. The CSeq header in the ACK must contain the same sequence number as the original request, but the method parameter must be equal to “ACK”.

Then in order to determine that the session establishment was successful we need to consider three messages as described by the following predicate.

$$\begin{array}{rcl} SessionEstabl(x,y,z) & \leftarrow & RespondsTo(y,x) \\ & \wedge & Request(z) \\ & \wedge & x.method = \text{‘INVITE’} \\ & \wedge & y.statusCode = 200 \\ & \wedge & z.method = \text{‘ACK’} \\ & \wedge & z.to = y.to \\ & \wedge & z.callId = x.callId \\ & \wedge & z.from = x.from \\ & \wedge & z.cSeq.seq = x.cSeq.seq \\ & \wedge & z.cSeq.method = \text{‘ACK’} \end{array}$$

Again we have done some simplifications according to the previous definitions with respect to the RFC. This predicate specifies when three messages conform to a session establishment.

D. Invariant examples for the PoC service

The SIP protocol is designed for extensibility and usability in different contexts, because of that, there are not many restrictions on when a session should be established, only on the message exchange in order to establish it. However, a service using SIP can define requirements for its infrastructure. For instance, the PoC service requires that a session can be established only by registered clients. Then invariant $\mathcal{I}_3$ can verify that requirement on the trace

$$\begin{array}{r} \mathcal{I}_3 \leftarrow \forall x,y,z \in \mathfrak{T}, SessionEstabl(x,y,z) \Rightarrow \\ \exists u,v \in \mathfrak{T}, Registration(u,v) \\ \wedge\, x.from = u.from \wedge Before(v,x) \end{array}$$

indicating that whenever a session establishment is found on the trace, then a registration must previously appear on the trace from the same user. This invariant in particular is interesting for its complexity, given that it requires testing several messages in order to be checked. Such kind of property is not easy to define using similar invariant testing methodologies.

VII. Experiments and Results

In this section we briefly describe the methodology used to test the invariants defined in Section VI on real protocol traces.

We defined the predicates and invariants using the Prolog programming language. We chose this language due to its definition as an ISO standard (ISO/IEC 13211-1), portability and maturity of its implementations. In particular we used the ISO compliant SWI-Prolog implementation, because of its large library of functions. A SIP message was defined as a structure msg(Request, StatusCode, ...) and predicates were defined in terms of variables as described in Section VI. Invariants were specified in the negative form $\phi \wedge \neg\psi$, for instance, for invariant $\mathcal{I}_1$ the following predicate was defined

```
negInv1(X) :-
    trace_msg(_, X), request(X),
    not(msg_method('ACK',X)), trace_msg(_, Y),
    not((
        after(Y,X), responds(Y,X)
    )).
```

where msg_method(Method, Msg) is true if the value of the variable Method equals the method of the message in Msg. In the program, the execution of the goal negInv1(X) will return all request messages for which a response message is not available in the trace.

Trace files were provided by Alcatel-Lucent for a PoC service implementation, a single trace was chosen for testing, containing the establishment of an Ad-Hoc group session by 2 clients. In order to test the traces using logical programming, we implemented a prototype tool in C to convert from the XML (PDML) format exported with the tool *Wireshark* into a set of statements of the form `trace_msg(index, msg)` as described in Section V.

In our tool, a configuration file defines how to relate a message variable in Prolog with a field of a respective packet, as follows

```
message: {
    Method = "sip.Method";
    StatusCode = "sip.Status-Code";
    To = "sip.to.addr";
    From = "sip.from.addr";
    ...
}
```

Using this configuration, along with the trace, the tool outputs a Prolog file containing the list of `trace_msg` statements, where each message is assigned an index according to its order in the trace, as follows

```
trace_msg(1, msg('INVITE', nil,
    'sip:user1@b.c', 'sip:user2@b.c', ... )).
trace_msg(2, msg(nil, 180,
    'sip:user1@b.c', 'sip:user2@b.c', ... )).
trace_msg(3, msg(nil, 200,
    'sip:user1@b.c', 'sip:user2@b.c', ... )).
```

If a particular field is not found in a trace message, it is replaced by the empty value *nil* (as the *Method* field in SIP response messages). However, if none of the defined fields are found, the message is ignored. This allows the tool to also act as a filter for messages of the studied protocol.

We performed experiments by manually introducing some errors on the implementation traces to evaluate the detection capabilities of our approach. The results were successful, the errors introduced were correctly detected by execution of the program, and no *false positive* results were produced, given that the definitions were specifically targeted to the PoC service, the process allowed to correctly filter out the Presence service messages. Even for a complex invariant such as invariant 3 the approach did not present any issues.

In addition to the detection of the introduced errors, the program also found some legitimate errors in the trace, that were not found by previous testing. These occurred for invariant 2, where some responses to an `INVITE` were found in the trace where the request could not be found. Upon further examination of the trace, we found that the error was due to a problem in the collection of the trace, since there were several responses from the server, indicating that the request must have been correctly received. This raises an issue about the reliability of trace collection tools and techniques, however this was an isolated case and since we only have access to the trace it is not possible to make a full assessment without further experimentation.

One issue that should be addressed is the performance of the approach. Since Prolog is a general purpose programming language and not a dedicated program, the number of evaluations performed in order to evaluate an invariant on the trace is large. For invariant 1, in a trace with only 407 messages, the program performed 5559 inferences (procedure calls) to arrive to a conclusion, however, the problem is better illustrated for invariant 3, the most complex one, where the number of inferences performed was of 583440 (although in only 0.09 seconds) to obtain a verdict. A dedicated algorithm could perform less inferences by limiting the range of comparison in the trace, instead of testing for every possible combination as Prolog does. This issue is even more critical as the number of messages involved in the property increases. Although some optimizations are possible, the creation of a dedicated program is indispensable.

VIII. Related Work

A short overview of works related to ours are given in this section, to complement the references cited throughout the paper.

Formal testing methods have been used for years to prove correctness of implementations by combining test cases evaluation with proofs of critical properties. In [16], [1] 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 FSM derivations are described in [2], [17]. Most of these techniques consider only control portions, in [11], [13], [3], data portion testing is approached by evaluation of traces in EEFSM 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, and using a model only for their verification. A different methodology can be found in [18], where I/O Automata and first-order logic are used to prove the correctness of the implementation w.r.t. the specification for distributed algorithms. Data relations are also taken into account. Again, this work relies on a specification to perform testing, which is not always available. There exists a considerable number of works with similar approach in literature. In [19], [4] the invariant approach was presented, and studied also in [5], [12]. A study of the application of invariants to an IMS service was also presented by us in [20], [8]. Although these invariant approaches consider in some level the data part, it is not their main objective. Some improvements have been made to invariant techniques by the authors of [6] to include temporal restrictions. Although their approach is very useful with temporal specifications, it is not extensible for testing of data. Our method improves on invariant testing by allowing to define properties with complex data relations by using a restriction on first-order logic formulas.

IX. Conclusion and Future Work

In this article we describe a new approach to invariant-based passive testing. We improve the expressiveness of invariant properties by using first-order logic, thus allowing to test data constraints between messages directly into protocol traces. We also demonstrate the soundness of our approach, showing that if a trace from an implementation is not correct with respect to an invariant verified in a specification, then the implementation does not conform to such specification. We detail on how invariants defined using our definition can be tested using logic programming. We provide an example of definition of invariants for the SIP protocol directly from the protocol standard and detail on the results obtained by testing using a Prolog implementation on a real trace obtained from a PoC service implementation.

Our experiments showed that our approach makes it possible to test complex properties directly into the trace. The expressive power of invariants defined in the way described is very large, given the few limitations imposed to the definition of formulas.

However our testing approach still requires improvement. In particular, testing using logic programming showed to be very expensive in terms of the number of procedure calls required to arrive to a conclusion. One of the reasons for this is that the evaluation of the formulas does not take into account the linearity of the trace, and always tests every message in the trace, even though it is only necessary to evaluate in a single direction in the trace, forward or backward, depending on the property.

As future work, we plan to develop an algorithm to allow more efficient testing of properties by taking into account the linearity of the trace and also by optimizing the invariant formulas. This could also allow for real time passive monitoring and testing of invariants in protocol implementations.

References

[1] R. M. Hierons, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, H. Zedan, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, and K. Kapoor, “Using formal specifications to support testing,” *ACM Computing Surveys*, vol. 41, no. 2, pp. 1–76, 2009.

[2] D. Lee, A. Netravali, K. Sabnani, B. Sugla, and A. John, “Passive testing and applications to network management,” in *Proceedings 1997 International Conference on Network Protocols*. IEEE Comput. Soc, 1997, pp. 113–122.

[3] H. Ural and Z. Xu, “An EFSM-Based Passive Fault Detection Approach,” *Lecture Notes in Computer Science*, pp. 335–350, 2007.

[4] A. Cavalli, S. Prokopenko, and C. Gervy, “New approaches for passive testing using an Extended Finite State Machine specification,” *Information and Software Technology*, vol. 45, no. 12, pp. 837–852, September 2003.

[5] E. Bayse, A. Cavalli, M. Núñez, and F. Zaïdi, “A passive testing approach based on invariants: application to the wap,” *Computer Networks*, vol. 48, no. 2, pp. 247–266, 2005.

[6] C. Andrés, M. G. Merayo, and M. Núñez, “Formal Correctness of a Passive Testing Approach for Timed Systems,” *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pp. 67–76, 2009.

[7] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, “SIP: Session Initiation Protocol. RFC 3261,” 2002.

[8] F. Lalanne, S. Maag, E. M. D. Oca, A. Cavalli, W. Mallouli, and A. Gonguet, “An Automated Passive Testing Approach for the IMS PoC Service,” in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[9] Open Mobile Alliance, “Push to Talk over Cellular Requirements. Approved Version 1.0,” Jun. 2006.

[10] Open Mobile Alliance, “Internet Messaging and Presence Service Features and Functions. Approved Version 1.2,” Jan. 2005.

[11] D. Lee and R. E. Miller, “A formal approach for passive testing of protocol data portions,” in *10th IEEE International Conference on Network Protocols, 2002. Proceedings.* IEEE Comput. Soc, 2002, pp. 122–131.

[12] B. Ladani, B. Alcalde, and A. Cavalli, “Passive testing: a constrained invariant checking approach,” in *Proc. 17th IFIP Int. Conf. on Testing of Communicating Systems*. Springer, 2005, pp. 9–22.

[13] D. Lee, D. Chen, R. Hao, R. E. Miller, J. Wu, and X. Yin, “Network protocol system monitoring-a formal approach with passive testing,” *IEEE/ACM Transactions on Networking*, vol. 14, no. 2, pp. 424–437, April 2006.

[14] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge Univ Pr, 2004.

[15] U. Nilsson and J. Maluszynski, *Logic, programming and Prolog*, 2nd ed. Wiley, 1990. [Online]. Available: http://www.ida.liu.se/˜ulfni/lpp

[16] D. Lee and M. Yannakakis, “Principles and methods of testing finite state machines: A survey,” *Proceedings of the IEEE*, vol. 84, pp. 1090–1123, 1996.

[17] M. Tabourier and A. Cavalli, “Passive testing and application to the GSM-MAP protocol,” *Information and Software Technology*, vol. 41, no. 11-12, pp. 813–821, September 1999.

[18] J. Sogaard-Andersen, S. Garland, J. Guttag, N. Lynch, and A. Pogosyants, “Computer-assisted simulation proofs,” in *Computer Aided Verification*, vol. 124, no. 1-3. Springer, March 1984, pp. 305–319.

[19] J. Arnedo, A. Cavalli, and M. Nunez, “Fast testing of critical properties through passive testing,” *Testing of Communicating Systems*, pp. 608–608, 2003.

[20] F. Lalanne and S. Maag, “From the IMS PoC service monitoring to its formal conformance testing,” in *Proceedings of the 6th International Conference on Mobile Technology, Application & Systems - Mobility ’09*. Nice, France: ACM Press, 2009, pp. 1–8.