Data-centric Property Formulation for Passive Testing of Communication Protocols

Felipe Lalanne, Xiaoping Che and Stephane Maag
Telecom SudParis
CNRS UMR 5157
9 rue Charles Fourier, 91011 Évry Cedex
FRANCE
{Felipe.Lalanne, Xiaoping.Che, Stephane.Maag}@it-sudparis.eu

Abstract: Passive testing techniques are used whenever the system under test cannot be interrupted, or access to its interfaces is unavailable. Under such conditions, communication traces are extracted from points of observation and compared with the expected behavior formally specified as properties. Since most works on the subject come from a formal model context, they are optimized for testing the control part of the communication with a secondary focus on the data parts. In the current work we provide a data-centric approach for black-box testing of network protocols. A syntax and semantics are provided to express and evaluate complex properties in a bottom-up fashion starting from expected data relations in messages. The approach has been implemented into a framework, and some experimental results are briefly discussed.

Key–Words: passive testing, data , conformance, IMS, protocols

1 Introduction

Passive testing is based on the observation of input and output events of an implementation under test (IUT) in run-time. Differently from active testing, it does not modify the natural behavior of the IUT (hence passive). Although it 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 production systems when the IUT cannot be stopped. In order to check conformance of the IUT the record of the observation during runtime (called trace) is compared with the expected behavior, defined by either a formal model (when available) or by one or more expected functional properties obtained from the requirements. Passive testing is many times conflated [7] with runtime monitoring [14], however, while both attempt to detect the conformance of a trace with respect to a property, passive testing aims ultimately to provide a verdict on the implementation, where the evaluation of the property allows to confirm/falsify a previously defined test hypothesis.

Both passive testing and runtime monitoring approaches derive respectively from model-based testing and model checking techniques. Due to this, they are usually propositional in nature, i.e. they represent inputs and outputs as part of a (usually) finite set of symbols or values accounting for the control part of the communication, allowing techniques from regular language theory to be used. Since modern, particularly application-layer, protocols depend heavily on data exchange, some works extend the propositional approach by using a concept as “parameterized propositions”[20], where propositions are augmented with parameters (variables) of different domains. Such extension allows to successfully test simple data relations, however, as the number of parameters increases, it does it in detriment of succinctness and readability of formulas. It is the premise of the current work that, particularly for application-layer protocols, where the main functionality of the protocol is contained in the data and not in the control part, a data-centric approach, i.e. a definition of properties from the basis of data fields and their expected relations, provides a more effective solution for testing of network protocols.

In this paper we formally define a message as a collection of data fields and the base unit in a trace (Section 2). With such definition we propose a logic syntax and semantics (Section 3) based in Horn logic in order to express properties from the bottom-up: starting with relations between individual data fields within messages, grouped to allow for constructs of varying complexity. Temporal (order) constraints between messages are expressed by means of trace quantifiers, allowing for definitions for the past or future of the trace. The implementation of the syntax and semantics as a framework and some experimental...
results are also briefly discussed (Section 4).

2 Messages and traces

2.1 Messages in network protocols

A message in a communication protocol is, using the most general view possible, a collection of data fields belonging to multiple domains. Data fields in messages, are usually either atomic, i.e. the information they provide comes from using them as a unit (e.g. timestamp, name, port), or compound, i.e. they are composed of multiple elements (e.g. 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 \(v = \{(l_i, v_i) \mid l_i \in L \land v_i \in D_i \cup \{\epsilon\}, i = 1 \ldots k\}\), where \(L = \{l_1, \ldots, l_k\}\) 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 \(\epsilon\) (null value). A compound domain is then the set of all values with the same set of labels and domains.

Definition 2. Given \(L\) a set of labels of length \(k\) and \(D_1, \ldots, D_k\) a group of data domains (not necessarily disjoint), with \(k > 0\). A compound domain \(C\) is identified by the tuple \(L, D_1, \ldots, D_k\). Each element \(v \in C\) is a compound value of length \(k\) with labels \(l_i\) in \(L\) and corresponding \(v_i \in D_i \cup \{\epsilon\}\), for \(i = 1 \ldots k\).

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 \(m \in M_P\), that is, a message is any value which is valid with respect to the protocol specification.

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

\[
m = \{\text{(method, ‘INVITE’), (status, \epsilon), (from, ‘john@b.org’), (to, ‘paul@b.org’), (cseq, \{(num, 10), (method, ‘INVITE’)\})}\}
\]

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, \(\{(\text{num}, 10), (\text{method}, ‘INVITE’)\}\).

Accessing data inside messages is a basic requirement for the current approach. In order to reference elements inside a compound value, the syntax \(v, l_1, l_2, \ldots, l_n\) 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, is associated to \(\epsilon\).

2.2 Definition of a trace

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). Such definition makes a trace potentially infinite, however, testing of properties can only occur in a finite segment of the trace. A definition of trace and trace segment is provided below.

Definition 4. Given the domain of messages \(M_P\) for a protocol \(P\). A trace is a sequence \(\Gamma = m_1, m_2, \ldots\) of potentially infinite length, where \(m_i \in M_P\).

Definition 5. Given a trace \(\Gamma = m_1, m_2, \ldots\), a trace segment is any finite sub-sequence of \(\Gamma\), that is, any sequence of messages \(\rho = m_i, m_{i+1}, \ldots, m_j\) \((j > i)\), where \(\rho\) is completely contained in \(\Gamma\) (same messages in the same order). The order relations \(\{<, >\}\) are defined in a trace, where for \(m, m' \in \rho\), \(m < m' \iff \text{pos}(m) < \text{pos}(m')\) and \(m > m' \iff \text{pos}(m) > \text{pos}(m')\) and \(\text{pos}(m) = i\), the position of \(m\) in the trace \((i \in \{1, \ldots, \text{len}({\rho})\})\).

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 Syntax and semantics

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 [1], for deductive databases, however, extended to allow for message variables and temporal relations. Both syntax and semantics are described in the current section.

3.1 Syntax

Formulas in this logic can be defined with the introduction of terms and atoms, as defined below.
Definition 6. A term is either a constant, a variable or a selector variable. In Backus-Naur form (BNF):

\[ t ::= c \mid x \mid t_1\ldots t_l \]

where \( c \) is a constant in some domain (e.g., a message in a trace), \( x \) is a variable, \( l \) represents a label, and \( t_1\ldots t_l \) is called a selector variable, and represents a reference to an element inside a compound value, as defined in Section 2.1.

Definition 7. An atom is defined as

\[ A ::= p(t_1,\ldots, t_l) \mid t = t \mid t \neq t \]

where \( t \) is a term and \( p(t_1,\ldots, t_l) \) 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 \( A_0 \leftarrow A_1 \land \ldots \land A_n \), where \( A_0 \), called the head of the clause, has the form \( A_0 = p(t_1^*, \ldots, t_k^*) \), where \( t_i^* \) are a restriction on terms for the head of the clause (\( t^* = c \mid x \)). The expression \( A_1 \land \ldots \land A_n \), called the body of the clause, where \( A_i \) are atoms. Disjunction is achieved by using the same head for multiple clause definitions. The following set of declarations

\[ A_0 \leftarrow A_{1,1} \land \ldots \land A_{1,n_1} \\
A_0 \leftarrow A_{2,1} \land \ldots \land A_{2,n_2} \\
\vdots \\
A_0 \leftarrow A_{p,1} \land \ldots \land A_{p,n_p} \]

is equivalent to \( A_0 = (A_{1,1} \land \ldots \land A_{1,n_1}) \lor \ldots \lor (A_{p,1} \land \ldots \land A_{p,n_p}) \).

Finally a formula is defined by the following BNF

\[ \phi ::= A_1 \land \ldots \land A_n \mid \phi \lor \phi \mid \forall_x \phi \mid \forall_y > \phi \\
\mid \forall_y < \phi \mid \exists_x \phi \mid \exists_y > \phi \mid \exists_y < \phi \]

where \( A_1, \ldots, A_n \) 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 ‘<’ indicating the order relation from Definition 5. These type of quantifiers are called trace quantifiers.

3.2 Semantics

The semantics used on this work is related to the traditional Apt–Van Emden–Kowalsky semantics for logic programs [22], 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 [16]).

Definition 8. A substitution is a finite set of bindings \( \theta = \{x_1/t_1, \ldots, x_k/t_k\} \) where each \( t_i \) is a term and each \( x_i \) 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 \( \theta \) 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 substitution \( \theta \) to a selector variable \( x.l_1 \ldots l_k \) is defined as

\[ x.l_1 \ldots l_k \theta = \begin{cases} t.l_1 \ldots l_k & \text{if } x/t \in \theta \text{ with } t \\ x.l_1 \ldots l_k & \text{otherwise} \end{cases} \]

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 \( \theta \) on an expression \( E \), denoted by \( E\theta \) is the application of all bindings in \( \theta \) to all terms appearing in \( E \).

Given \( K = \{C_1, \ldots, C_p\} \) a set of clauses and \( \rho = m_1, \ldots, m_n \) a trace. An interpretation\(^3\) is any function \( I \) mapping an expression \( E \) that can be formed with elements (clauses, atoms, terms) of \( K \) and terms from \( \rho \) 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\(^2\) \( A = p(c_1, \ldots, c_k) \) is true, iff \( A \in I \).
- An atom \( A \) is true, iff every ground instance of \( A \) is true in \( I \).
- The expression \( A_1 \land \ldots \land A_n \), 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 \( I \).

\(^2\)Called a Herbrand interpretation in logic programming.
\(^3\)An atom where no unbound variables appear.
Recent Researches in Mathematical Methods in Electrical Engineering and Computer Science

4 Examples and Experiments

The Session Initiation Protocol (SIP) [19] is an application-layer protocol that relies on request and response messages for communication, and it is an essential part for communication within the IMS (IP Multimedia Subsystem) framework. Messages contain a header which provides session, service and routing information, as well as an (optional) body part to complement or extend the header information. Several RFCs have been defined to extend the protocol with to allow messaging, event publishing and notification. These extensions are used by services of the IMS such as the Presence service [17] and the Push-to-talk Over Cellular (PoC) service [18].

For the experiments, traces for an ad-hoc PoC session establishment were obtained from a production IMS implementation, provided by Alcatel-Lucent and extracted from the interfaces of the IMS core, containing exchange between the client and the IMS presence and PoC services. Tests were performed using a prototype implementation of the concepts presented in the current work, using an algorithm developed by us and described in [8]. The implementation, defined properties and network traces can be found in http://www-public.int-evry.fr/~lalanne/datamon.html.

Properties, extracted from the requirements for the PoC and Presence service were evaluated in the traces and the approach showed successful in determining the appearance of such properties. Two of the properties tested are provided as example in the following.

For every request there must be a response. This property can be used for a monitoring purpose, in order to draw further conclusions from the results. Due to the issues relative to testing on finite traces for infinite executions, false results can never be given for this context. However, inconclusive results can be returned and conclusions may require further analysis of the results (for instance if the same type of message is always without a response). The property evaluated is as follows:

\[
\forall_x (\text{request}(x) \land x.\text{method} \neq \text{ACK} \rightarrow \exists_{y>x} (\neg \text{nonProvisional}(y) \land \text{responds}(y, x)))
\]

where \(\neg \text{nonProvisional}(x)\) accepts all non provisional responses (non-final responses, with status \(\geq 200\)) to requests with method different than \text{ACK}, which does not require a response. The predicate \(\text{responds}(y, x)\)
accepts all pairs of messages \((y,x)\) where \(y\) is a response to \(x\).

**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

\[
\forall x (\exists y > x \text{sessionEstablished}(x,y) \rightarrow \exists u < x (\exists v > u \text{registration}(u,v)))
\]

where \(\text{sessionEstablished}\) and \(\text{registration}\) are defined as

\[
\text{sessionEstablished}(x,y) \leftarrow x.\text{method} = \text{INVITE} \\
\land y.\text{statusCode} = 200 \\
\land \text{responds}(y,x)
\]

\[
\text{registration}(x,y) \leftarrow \text{request}(x) \land \text{responds}(y,x) \\
\land x.\text{method} = \text{REGISTER} \\
\land y.\text{statusCode} = 200
\]

From the experiments, a predominance of inconclusive results were found. In the second property, for example, inconclusive results indicate that a session initialization was found on the trace, but no registration procedure was captured. If prior to testing, the assumption is made, due to the trace capturing methodology, that a session registration should be available, then such inconclusive results are telling and indicate a possible fault in the implementation. Unfortunately, such assumption could not be made in our case.

Another result, and a possible limitation of the syntax was found during experimentation. For the previous property, the current evaluation algorithm must test all combination of messages as bindings for \((x,y)\) and then all combinations for \((u,v)\) until a solution is found, this means a complexity of \(n^4\) with \(n\) the length of the trace. A more flexible syntax might allow for fewer iterations. Again, the results and their discussion are provided in more detail in [8].

5 Related Work

Formal testing methods have been used for years to prove correctness of implementations by combining test cases evaluation with proofs of critical properties. In [10] 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. Although most works consider only control portions, in [13, 21], data portion testing is approached by evaluation of traces in EEFSM and SEFSM (Event-based/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. In [2] the invariant approach was presented, and studied also in [6, 11]. A study of the application of invariants to an IMS service was also presented by us in [12]. In more recent work, the authors of [15] define a methodology for the definition and testing of time extended invariants, also considering data in their definitions. We improve on this approach by allowing to express relations between multiple messages, while they mostly focus on binary relations.

Runtime monitoring and runtime verification techniques have gained momentum in the latest years, particularly using model checking techniques for testing properties on the trace. The authors of [14] provide a good survey and introduction of methodologies in this area. The usual approach, consists on the definition of some logic (LTL is commonly used), which is used to create properties from which a monitor is defined to test on the trace. The authors of [4] describe the definition of monitors as finite state machines for LTL formulas, they introduce a 3-valued semantics (true, false, inconclusive) in order to test formulas for finite segments of the trace. They expand their analysis on inconclusive results in [5], by proposing a 4-value semantics. Regarding the inclusion of data, the authors of [20] and [9] provide two different approaches to it, with parameterized propositions for LTL formulas in the first, and with quantifiers for specific message data fields \(\forall x.\phi\) to indicate that the value of \(x\) comes from the method field, for the second. Our work improves on the simplicity of formulas expressible through their syntax, by allowing to group data relations within data definitions. Finally, in [3], the authors propose a highly expressive logic for runtime monitoring of programs, which also allows for relations between data variables. However, their logic is propositional in nature and their representation of data is aimed at characterizing variables and variable expressions in programs, which makes it less than ideal for testing message exchanges in a network protocol as required in our work.

6 Conclusion and Future Work

This paper introduces a novel approach to passive testing of network protocol implementations, with a particular focus on IMS services. Motivated by the fact that modern (particular application-layer) protocols are highly dependant on data to set-up and coordinate...
communication, our approach reduces the focus on the testing of control part of the communication, and provides a means for directly testing data relations between messages in a network trace. The 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. A three-valued semantics for evaluation is proposed, where true, false or inconclusive can be given, whenever the property is confirmed, falsified or no useful information can be derived, respectively on a given trace. The approach has been implemented into a framework and results from testing properties on a IMS PoC service traces are provided.

Future improvements on the work are the inclusion of real time constraints, which would allow to reduce the number of inconclusive verdicts by providing an alternative signal to stop evaluation (other than reaching the end of the trace). Improvements to allow online testing, i.e. evaluation of properties as the SUT is being run, would also be desirable, for instance for the testing of security properties. Modifications on the syntax would also allow for more flexibility and possibly, improve the complexity of evaluation by reducing the number of messages evaluated. Finally further study should be given to possible race conditions occurring during evaluation of formulas, when multiple messages with the same timestamp are encountered on a particular trace.

References:


