An Approach to Automated Runtime Verification for Timed Systems: Applications to Web Services
Dung Cao, Richard Castanet, Patrick Felix, Kevin Chiew

To cite this version:
Dung Cao, Richard Castanet, Patrick Felix, Kevin Chiew. An Approach to Automated Runtime Verification for Timed Systems: Applications to Web Services. JOURNAL OF SOFTWARE, 2012, 7 (6), pp.1338-1350. 10.4304/jsw.7.6.1338-1350. hal-00997948

HAL Id: hal-00997948
https://hal.archives-ouvertes.fr/hal-00997948
Submitted on 12 Jun 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
An Approach to Automated Runtime Verification for Timed Systems: Applications to Web Services

Tien-Dung Cao¹*, Richard Castanet², Patrick Felix², and Kevin Chiew¹

¹ School of Engineering, Tan Tao University
Email: {dung.cao, kevin.chiew}@ttu.edu.vn
² LaBRI - CNRS - UMR 5800, University of Bordeaux
351 cours de la libération, 33405 Talence cedex, France.
Email: {castanet, felix}@labri.fr

Abstract—Software testing plays an important role in verifying and assessing the quality of a software application. There are various testing approaches proposed for different application scenarios. In this paper, we propose a new passive testing approach to verifying a timed trace with respect to a set of constraints. With the extension of Nomad language, we are able to formally describe all constraints and combine conditions by logical operations AND and OR into expressions. By well organizing and evaluating the expressions, we are able to carry out runtime verification message by message in a timed trace. In addition to the theoretical framework, we have also developed a software tool known as RV4WS (Runtime Verification for Web Services) for the automation of our testing approach, and implemented all algorithms in the paper with this tool. We conduct a case study of web service composition to verify the effectiveness of our approach and tool.

Keywords—Runtime verification, Passive testing, Rule specification, Web services.

I. INTRODUCTION

The activity of conformance testing focuses on verifying the conformity of a given implementation based on its specification. It can be classified into two categories, namely active testing and passive testing. (a) Active testing requires a tester to interact directly with the implementation under test (in short as IUT) and check the correctness of answers by the implementation. However, this method is not applicable to a running system due to some reasons like (1) testers do not have permission to access to the interface of a running system; (2) if testers use the active method to test functions like create_new_account(...) or update_debit(...) of a banking system, it may incur errors like false accounts or updates in the database of the system; and (3) active testing does not allow us to check several security properties of a system that can only be captured at runtime or when several sessions are executed in parallel. Moreover, because testing cannot find all faults, even if a system has passed an active test, we still need to verify its conformity at running time or to analyze its log files for improving the reliability of a system. (b) Passive testing collects the observable traces (i.e., the log files) of the running system by installing a probe and analyzes them based on a set of rules [8], [9], [21] or a formal specification [12]. Without a tester directly interacting with the IUT, passive testing does not affect the system running, and is widely adopted for system verification.

Passive testing can be carried out either on-line or offline. The on-line technique, a.k.a. runtime verification technique, immediately checks an observable trace once an input/output event occurs so that potential damage can be prevented by terminating the system running whenever any fault is detected; whilst the off-line technique checks an observable trace after it is collected for a period of time, and does not usually require additional resources such as CPU, RAM or another computer to run both of the trace collection engine and checking engine in parallel.

For a complex system such as an SOA application or a cloud computing application, the communications across system components are carried out by signals, events, and messages, whose timed traces may be collected from a distributed environment and need to be well synchronized during verification. Therefore, we suggest to address the following factors when we define a set of constraints to verify a timed system.

- Time constraint. The passive testing verifies the message sequence in a trace. However, when a system is running, we do not know when the next message will arrive after the previous one. Thus we may have to set time constraint for each message. For example, we can set 10 seconds of time constraint for receiving a loginResponse after sending a loginRequest.
- Condition on message content. Sometimes we are only interested in some messages of which the contents satisfy some conditions. For example, we can identify the messages sent to or received from machine A by their contents (SourceIP = A) or...
(DestIP = A).

- **Data correlation.** For any observable trace mixed by several traces or sessions that are executed in parallel, we need to apply our constraints on the messages that belong to an individual trace or session. To do so, we firstly find the messages that have a correlation by their data values (known as data correlation), and then apply our constraints on these messages. For example, we can assign sessionId fields to messages belonging to different sessions running in parallel, can group these messages by the values of sessionId field before applying our rules for correctness verification.

- **Combination of conditions.** A constraint can also be represented by a combination of several conditions with logical operations such as AND, OR, and NOT.

In this paper we propose a new approach to passive testing either on-line or off-line for a timed system by verifying a timed trace based on a set of rules which contains the constraints on message sequence, the interval time between any two messages, and the contents of messages. To formally describe constraints for the specifying permissions and prohibitions, we propose to extend the Nomad [14] language by defining the constraints on each atomic action (fixed conditions) and a set of data correlations between the actions, so as to describe permissions and prohibitions both of which are atomic actions and should be applied immediately and obligations which are related to non-atomic actions within contexts and need a time duration to complete. For example, let $x$ be a positive integer, a prohibition or a permission rule is evaluated to be true at time $t$ if $t \in [0, x]$; whereas an obligation rule is evaluated to be true at time $t$ if $t \geq x$, meaning that the obligation needs at least a duration $x$ to complete the work. Besides the theoretical framework, we develop a software tool known as RV4WS (Runtime Verification for Web Service) to implement the automation of our passive testing approach. In particular, the algorithms presented in this paper are fully implemented by this software tool. We also apply our tool to a case study of WebMov project which provides design and composition mechanisms for web services.

The remaining sections are organized as follows. We first present some discussions about software testing and existing method for passive testing or runtime verification in Section II, and then introduce the syntax and semantics of our rules in Section III followed by an algorithm for verifying a timed system based on a set of rules in Section IV. In Section V, we introduce the RV4WS tool together with a case study in Section VI before concluding the paper in Section VII.

II. DISCUSSION

A. Software Testing

Testing is an important step to verify and assess the quality of a software application, and an appropriate testing type should be chosen for an individual application.

We classify the types of testing into four categories based on the characteristics of the application, the phases of the development, the available information of specifications and the capability of application controls, and use a schema with four axes to show the classification as depicted in Figure 1.

1) The characteristics:

- **Conformance testing.** It is used to test the conformance of an implementation based on its specification.
- **Robustness testing.** It is used to test the capability to deal with the unexpected data.
- **Performance testing.** It refers to the assessment of the performance of an application in different cases in terms of the speed and effectiveness.
- **Security testing.** It is a process to determine that an information system protects data and maintains functionality as intended. Some security concepts that need to be covered by security testing are listed as follows.
  - **Authentication**, which is the process of establishing the identification of a user.
  - **Authorization**, which is the process of determining that a requester is allowed to receive a service or perform an operation.
  - **Availability**, which is to assure information and communications services be ready for use upon requests or the information kept available to authorized users when they need it.
  - **Integrity**, which is a measure by which receivers can determine the correctness of the information provided by the system.

2) The phases of the development:

- **Unit testing.** It is to verify the operation of an individual component or module in isolation to the rest of the system.
- **Integrated testing.** It is to test the interactions
amongst components of a system. In other words, it tests a system at the interface level of each component.

- **System testing.** It is to verify the global behavior of the system.

3) The accessibility:

- **Black-box testing.** It allows testers to generate test cases from system specifications for functional testing without knowing the internal structure of system.
- **Gray-box testing.** It is used when some information of the internal structure are available for testers.
- **White-box testing.** It is used when testers know the internal structure of the system (i.e., the code) and allows testers to verify the structure by testing different paths in the code.

4) The controllability:

- **Active testing.** It allows testers to interact directly with the system under test by sending requests and receiving responses for analysis.
- **Passive testing.** It allows testers to assess a system from input/output events or log files without interacting with the system under test.

B. Passive Testing of systems

Due to its side-effect to a system, passive testing is usually used as a monitoring technique to detect and report errors when we cannot use an active testing method. Another area of its applications is in network management for the detection of configuration problems, fault identification, or resource provisioning. This section reviews some passive testing approaches.

Bayse et al. [9] and Cavalli et al. [12] proposed a passive testing approach based on invariants of a Finite State Machine (FSM). They defined two types of invariants in the following for an FSM $M = (S, s_0, T, O, \mathcal{T})$ where $S$ is a set of finite states, $s_0$ an initial state, $T$ the set of input actions, $O$ the set of output actions, and $\mathcal{T}$ the set of transitions.

- **Simple invariant.** Trace $i_1/o_1, i_2/o_1, ..., i_{n-1}/o_{n-1}$, $i_n/O$ is a simple invariant of FSM $M$ given that we necessarily get an output $O$ where $O \subseteq O$ if we obtain the input $i_n$ under the premise that each time the trace $i_1/o_1, i_2/o_1, ..., i_{n-1}/o_{n-1}$ is observed.
- **Obligation invariant.** It is used to express properties such as “if y happens then we must have that x had happened”.

They presented two algorithms to check from left-to-right and right-to-left a finite trace to give a verdict without considering the time constraints on the traces. TIPS [11] (Test Invariant of Protocols and Services) is an implementation tool of this approach.

To express temporal properties, Andrés et al. [1]–[3] introduced Timed Invariant as an extension of simple invariant with time constraints between an input and an output. There are some limitations of their Timed Invariant model as listed below.

- It only supports future times not past time. This is because its semantic is defined as “if we obtain a trace of the pair of input/output event (the interval time between an input and an output is also considered) and we continue to obtain an input (after this trace), then we must obtain an output after a fixed interval time”.
- It does not support combining several conditions into a Timed Invariant by the logical operations such as AND and OR.
- It does not consider the constraints on the content of each event, therefore the data correlation problem between the events is also not considered.
- Finally, the tool PaTe [1] that is implemented to check the correctness of a log w.r.t. a set of time invariant does not allow us to verify an execution trace in parallel with the trace collection engine, i.e., not supporting runtime verification or on-line checking.

Mallouli et al. [16] proposed the security rule using the Nomad language to express the constraints on a trace with obligations, prohibitions and permissions. That is, a prohibition or permission rule is granted and applied immediately to a trace; while a obligation rule delimits the completion deadline of a task. They also introduced an algorithm to check the correction of the trace following these security rules. Their approach solves the time constraints caused by invariant approach though, it does not consider the correlation of messages by its data values which is an important issue for passive testing.

Tabourier and Cavalli [22] proposed an approach to verify the traces which actually belong to the accepted specifications provided by an FSM. This method is composed of two stages:

- Firstly, passive homing sequence is applied to determine the current state. Initially, all states are put into a candidate list. When an input/output arrives, the current state will be updated by the destination state of the corresponding transition if it is the source state of the transition or otherwise removed from the candidate list. After a number of iterations, either a single current state is obtained and we move to the second step to detect the fault, or an input/output pair is not accepted by any candidate state. In the latter case, a fault is detected.
- Secondly, fault detection is carried out by applying the search technique to the current state and the current input/output pair. If a state which does not accept the following transition is reached, then there is an error; otherwise, then the end of the trace is reached, and no error is detected.

This method does not consider the time constraints on the traces and is not applicable to the case where the trace is collected from the execution of multi-sessions that run in parallel.
C. Passive Testing of Web Services

In recent years, many methods have been proposed together with tools developed for passive testing of web services [4]–[6], [13], [17], [20]. These work focus on either checking the order of messages and/or its occurrence time on a trace file to give a verdict [13], [20], [21] or proposing a method for dynamic statistics [4], [6] of some properties of web services.

Dranidis et al. [17] proposed the utilization of Stream X-machines for constructing formal behavioral specifications of web services. They also presented a runtime monitoring and verification architecture and discussed how it can be integrated into different types of service-oriented infrastructures. However, they did not present an algorithm or a tool to verify an execution trace using the Stream X-machines specification of web services.

Baresi et al. [4], [5] presented a monitoring framework for BPEL orchestration which is obtained by integrating two approaches namely Dynamo and Astro, which are used for dynamic statistics of some properties of BPEL processes from single instance or multi instances. These work focus on the behavioral properties of composition processes expressed in BPEL rather than on individual web services. Moreover, an assessment (a verdict true/false) about service is not considered in this work.

Cavalli et al. [13] proposed a trace collection mechanism for SOA by integrating modules within BPEL engine and a tool [13], [16] that checks off-line execution traces. This approach uses the Nomad [14] language to define the security rule though, it does not allow us to check real-time (i.e., “on-line”) whenever a message happens. Moreover, this work does not consider the data correlation between the messages in the rules.

Li et al. [20], [21] presented the pattern and scope operators as the rule-based to define the interaction constraints of web services. The authors use FSM as semantic representation of interaction constraints. In this approach, the validation process runs in parallel with the trace collection. This approach is limited by the pattern number, while it does not consider the time constraints.

III. RULE DEFINITION

A. Syntax

In our work, we consider each message as an atomic action, and use one or several messages to define a formula with logical operations AND and OR. We also use the operation NOT to indicate that a message is not permitted to appear in the trace within a duration. During the formula definition, the constraint on message parameters values may be considered. Finally, from these formulas, the rule is defined in two parts, namely supposition (or condition) and context. The set of data correlations are included as an option.

Definition 1. Atomic action. An atomic action is either an input message or an output message, formally denoted as

\[ AA := Event(Const) | \neg AA \]

where
- \( Event \) represents an input/output message name;
- \( Const := P \approx V \mid Const \land Const \mid Const \lor Const \)
- \( P \) are the parameters. These parameters represent the relevant fields in the message;
- \( V \) are the possible parameters values;
- \( \approx \in \{ =, \neq, <, >, \leq, \geq \} \);
- \( \neg A \) means not(A).

Definition 2. Formula. A formula is recursively defined as

\[ F := start(A) | done(A) | F \land F | F \lor F | O^{de}[m,n]F \]

where
- \( A \) is the atomic action;
- \( start(A) \): A is being started;
- \( done(A) \): A has been finished;
- \( O^{de}[m,n]F \): F was true in d units of time ago if \( m > n \), and F will be true in the next d units of time if \( m < n \) where \( m \) and \( n \) are natural numbers.

Definition 3. Data correlation. A data correlation is a set of parameters that have the same data type where each different parameter represents a relevant field in a different message, for which the operator = (equal) is used to compare the equality amongst parameters. A data correlation is considered as a property on data.

Example 1. Let \( A(p_0^A, p_1^A), B(p_0^B, p_1^B, p_2^B) \) and \( C(p_0^C) \) be messages with \( p_1 \), \( p_2 \) the parameters where \( p_0^A \), \( p_0^B \) and \( p_0^C \) have the same data type. A data correlation set that is defined based on A, B and C is \( \{ p_0^A, p_0^B, p_0^C \} \iff \{ p_0^A = p_0^B = p_0^C \} \).

By putting the time constraints into an interval, we support only two types of rules, namely permission and forbidden. Permission means that all traces must satisfy the constraints; whereas forbidden is the negation of a permission constraint.

Definition 4. Rule with data correlation. Let \( \alpha \) and \( \beta \) be formulas, and \( CS \) be a set of data correlations based on \( \alpha \) and \( \beta \) (CS is defined based on the messages of \( \alpha \) and \( \beta \)). A rule with data correlation is defined as \( R(\alpha | \beta)/CS^2 \) where \( R \in \{ \mathcal{P}; Permission; \mathcal{F}; Prohibition \} \). The constraint \( \mathcal{P}(\alpha | \beta) \) or \( \mathcal{F}(\alpha | \beta) \) (where \( \mathcal{F}(\alpha | \beta) = \mathcal{P}(NOT \alpha | \beta) \)) respectively means that it is permitted or prohibited to have \( \alpha \) true when context \( \beta \) holds within the conditions of CS.

Example 2. We create a new account on the services if we successfully logged in within maximal one day ago and have not yet logged out by now. The rule with data correlation for this event can be denoted as

\[ \mathcal{P}(start(createAccountReq))O^{de}[1,0]D \land done(loginRes) \land done(\neg logoutReq). \]

In case we want to indicate the messages belonging to a session by using sessionId, we can denote it as

\[ CS \text{ is an optional part.} \]
A model of rules corresponds to a pair \( r = (P_r, C_r) \) where

- \( P_r \) is a total function that associates every integer \( x \) with a propositional formula.
- \( C_r \) is a total function that associates every integer \( x \) with a pair \((\alpha, d)\) where \( \alpha \) is a formula and \( d \) a positive integer.

Intuitively, \( \forall x, p \in P_r(x) \) means that proposition \( p \) is true at time \( x \); while \( (\alpha, d) \in C_r(x) \) means that context of formula \( \alpha \) holds (is evaluated true) at time \( t \) where

- \( t \in [x, x + d] \) if we focus on future time.
- \( t \in [x - d, x) \) if we focus on past time.

IV. Verification

A. Correctness of a System

The following definition is a formal description for the correctness of a system. That is, a system is correct if the execution traces obtained from the IUT satisfy the properties expressed by the rules, and a system fails if a rule is timeout or its content is evaluated to be false.

**Definition 5.** Correctness of a timed trace w.r.t. a finite set of rules. Let \( \sigma = \sigma_0, \sigma_1, \sigma_2, \ldots \) be an observable timed trace that is collected from a running system where \( \sigma_i = (m_i, t_i) \) denotes the message and its occurrence time, and let \( \Phi = \{ \phi_0, \phi_1, \ldots, \phi_n \} \) be a finite set of rules, define \( \sigma \) conforms to \( \Phi \) if and only if \( \forall \sigma_i \in \Phi \), there is \( \exists \phi_j \) such that \( \phi_j \) is timeout at \( t_i \) or the evaluation of \( \phi_j \) after updating its context is false.

B. Checking Algorithm

In this section, we give the outline for the computation mechanism used to determine whether a rule holds for some given input/output sequence of events. Our algorithm verifies message-by-message the conformity with each rule without storing the message sequence. Here, we use two global variables, namely \( \text{currlist} \) and \( \text{rulelist} \). \( \text{currlist} \) is a list of enabled rules that have been activated, while \( \text{rulelist} \) is the list of defined rules that are used to verify the system. Before introducing the detail of our algorithm, we present some functions running on the context of each rule.

- Function \text{correlation}. This function will return one of three values, namely either \( \text{undefined} \) or true or false. Value \( \text{undefined} \) is returned when a message is not defined in the set of data correlations of rule. If a message is defined in the set of data correlations of rule, then this function will query the corresponding value and return true/false after comparing it with the value of the previous messages.

- Function \text{contain}. This function verifies whether a message is contained by the context of a rule. It returns true if a message is found in the context of a rule and its conditions are validated (if they are defined). For example, given the context of the rule \( \text{msgA} \) (with its value \( id = 5 \)) and \( \text{msgB} \), when message \( \text{msgA} \) arrives, this function returns false because but its condition (i.e., \( id = 4 \)) does not match that in the rule even if the message name is found; whereas when message \( \text{msgB} \) arrives, this function returns true.

- Function \text{update}. This function updates the value of context whenever a message arrives and is found in the context (verified by function \text{contain}). For example, the context of a rule is \( \text{loginResponse} \land \neg \text{logoutRequest} \). When message \( \text{loginResponse} \) arrives, this context is updated as \( \text{true} \land \neg \text{logoutRequest} \).

- Function \text{evaluate}. This function evaluates whether not a context of rule holds (true) by returning one of three values, namely either true or false or \( \text{undefined} \). The \( \text{undefined} \) value is returned if there is at least one message name in the context of the rule. For instance, context true\( \land \neg \text{logoutRequest} \) is evaluated to be \( \text{undefined} \). During the evaluation, a message with the function NOT\(^3\) will be provisionally assigned as true. For example, at the time of evaluation, the expression \( \text{true} \land \neg \text{logoutRequest} \) will be evaluated as true.

As foregoing, there are two types of rules, namely future time and past time rules. To make this more clear, we will analyze the checking algorithm for each type.

1) Rules with future time:

Given that each rule has two parts (i.e., the supposition and context parts), a rule will be evaluated as either true or false or \( \text{undefined} \) if its supposition has been enabled and the current message belongs to its context. At any occurrence time \( t \) of message \( \text{msg} \), our algorithm checks the correctness of a rule by two steps.

- Step 1. Examine the list of enabled rules \( \text{currlist} \) to evaluate their context if the time constraints are valid. If the context of a rule is evaluated to be true/false, then it will be removed from the enabled list \( \text{currlist} \) and the corresponding verdict is returned. Otherwise (i.e., the context is \( \text{undefined} \), meaning incomplete context), we wait for the arrival of the next message and return true to the verdict.

- Step 2. Examine the list of rules \( \text{rulelist} \) to activate them if their supposition contains the current message \( \text{msg} \).

Algorithm 1 shows how to checks the correctness of a message with a set of future time rules, in which we assume that the rules are Permission (the Forbidden rules are the negation of the verdict of the Permission rules), and do not consider data correlation.

---

3This function only applies to atomic actions.
to show the current test states. Upon an arrival of message, we propose a graphic statistics applicable to current message. To identify which rule fails if the algorithm finds a rule that is not satisfied and not true/false be evaluated to be before its supposition, meaning that the context part must handles the current message. Upon the arrival of any timed event happens. Two functions verify event-by-event and returns the verdict whenever complete checking algorithm as shown in algorithm 3. It verify and verify

Algorithm 1: Checking algorithm for future time rules

Input : timed event: (msg, t)
Output: true/false
decide ← true
1) For each \( r \in \text{currlist} \)
   - IF the time constraints of \( r \) at \( t \) are validation
     - IF msg belongs to the context of \( r \)
       * Update context of \( r \) by msg
       * IF the evaluation of the context of \( r \) is true/false
       - Remove \( r \) from currlist
       - verdict ← verdict\& true/false
     - ELSE: verdict ← false
   2) For each \( r \in \text{rulelist} \)
     - IF msg belongs to the supposition of \( r \)
       - Update the activated time for \( r \) by \( t \)
       - Add \( r \) into currlist (activated)

2) Rule with past time:
For a rule with past time, the context part will happen before its supposition, meaning that the context part must be evaluated to be true/false whenever its supposition handles the current message. Upon the arrival of any timed event \((msg, t)\), our algorithm checks correctness of a rule with past time by two steps.

- Step 1. Examine the list of enabled rules currlist to check the correctness of current message msg. If \( t \) satisfies their time constraints and \( msg \) belongs to their supposition, then remove them from list currlist. At the same time, if their context is evaluated to be false/undefined, then a false verdict will be assigned; otherwise, a true verdict is admitted. On the other hand, if \( msg \) does not belong to their supposition and \( msg \) is found in their context, then we update their context by \( msg \) and wait the next message to evaluate these rules.
- Step 2. Examine the list of rules rulelist and activate them if their context contains the current message msg.

Algorithm 2 shows how to checks the correctness of a message with a set of past time rules under the assumption that the rules are Permission.

Be combining the above two algorithms, we give the complete checking algorithm as shown in algorithm 3. It verifies event-by-event and returns the verdict whenever a timed event happens. Two functions verify\_future() and verify\_past() called by algorithm 3 are shown in algorithms 4 & 5.

There is an exception that a fail verdict is returned if the algorithm finds a rule that is not satisfied and not applicable to current message. To identify which rule fails upon an arrival of message, we propose a graphic statistics to show the current test states.

Example 3. We have an execution of timed trace with

Algorithm 2: Checking algorithm for past time rules

Input : timed event: (msg, t)
Output: true/false
verdict ← true
1) For each \( r \in \text{currlist} \)
   - IF the time constraints of \( r \) at \( t \) are validation
     - IF \( msg \) belongs to the supposition of \( r \)
       * Remove \( r \) from currlist
     - IF the evaluation of the context of \( r \) is false/undefined
       - verdict ← verdict\& false
     - ELSE IF the context of \( r \) contains \( msg \)
       * Update the context of \( r \) by \( msg \)
     - ELSE: verdict ← false
   2) For each \( r \in \text{rulelist} \)
     - IF \( msg \) belongs to the context of \( r \)
       - Update the activated time for \( r \) by \( t \)
       - Add \( r \) into currlist (activated)

2) For each \( a \in \text{rulelist} \)
   - IF \( msg \) belongs to the context of \( a \)
     - Add \( a \) into currlist (activated)

The message name and its time occurrence as: \((a_1, 0), (a_2, 2), (a_1, 3), (b_2, 8), (b_1, 9), (a_2, 12), (b_3, 15), (c_1, 16), \ldots. \) The security rules defined to assess the system are:

\[ r_1 = P(\text{start}(a_1)[0|\leq|0.10] done(b_1) \lor done(c_1)), \]
\[ r_2 = P(\text{start}(b_2)[0|\leq|0.10] done(a_2) \land done(c_2)). \]

The table 1 shows the results of the algorithm execution. In the table, a false verdict is returned at message \((b_3, 15)\) due to the failure of rule \( r_1 \) at time 15 of which the last enabled message is \((a_1, 3)\).

V. RV4WS TOOL

RV4WS (Runtime Verification for Web services) is a software tool implemented to verify a web service at runtime based on a set of constraints defined by the syntax in Section III. This tool receives a sequence of messages (message content and its occurrence time) via a TCP/IP port, then verifies the correctness of this sequence. The architecture of RV4WS is shown in Figure 2.


<table>
<thead>
<tr>
<th>message</th>
<th>enabled rule list</th>
<th>verdict</th>
<th>add/remove (+/-)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a1, 0)</td>
<td>( r_1^1 = P(\text{true}[0, 10] \text{done}(b_1) \lor \text{done}(c_1)) )</td>
<td>true</td>
<td>+r1</td>
</tr>
<tr>
<td>(a2, 2)</td>
<td>( r_1 = P(\text{true}[0, 10] \text{done}(b_1) \lor \text{done}(c_1)) ); ( r_2 = P(\text{start}(b_2)[0, 10] \text{true} \land \text{done}(\neg c_2)) )</td>
<td>true</td>
<td>+r2</td>
</tr>
<tr>
<td>(a1, 3)</td>
<td>( r_1 = P(\text{true}[0, 10] \text{done}(b_1) \lor \text{done}(c_1)) ); ( r_2 = P(\text{start}(b_2)[0, 10] \text{true} \land \text{done}(\neg c_2)) ); ( r_3 = P(\text{true}[0, 10] \text{done}(b_1) \lor \text{done}(c_1)) )</td>
<td>true</td>
<td>+r1</td>
</tr>
<tr>
<td>(b2, 8)</td>
<td>( r_1 = P(\text{true}[0, 10] \text{done}(b_1) \lor \text{done}(c_1)) ); ( r_1 = P(\text{true}[0, 10] \text{true} \land \text{done}(\neg c_2)) ); ( r_1 = P(\text{true}[0, 10] \text{done}(b_1) \lor \text{done}(c_1)) )</td>
<td>true</td>
<td>-r2</td>
</tr>
<tr>
<td>(b2, 9)</td>
<td>( r_1 = P(\text{true}[0, 10] \text{done}(b_1) \lor \text{done}(c_1)) ); ( r_1 = P(\text{true}[0, 10] \text{true} \land \text{done}(\neg c_2)) )</td>
<td>true</td>
<td>-r3</td>
</tr>
<tr>
<td>(b2, 12)</td>
<td>( r_1 = P(\text{true}[0, 10] \text{done}(b_1) \lor \text{done}(c_1)) ); ( r_2 = P(\text{start}(b_2)[0, 10] \text{true} \land \text{done}(\neg c_2)) )</td>
<td>true</td>
<td>+r2</td>
</tr>
<tr>
<td>(b3, 15)</td>
<td>( r_2 = P(\text{start}(b_2)[0, 10] \text{true} \land \text{done}(\neg c_2)) )</td>
<td>false</td>
<td>-r1</td>
</tr>
<tr>
<td>(c1, 16)</td>
<td>( r_2 = P(\text{start}(b_2)[0, 10] \text{true} \land \text{done}(\neg c_2)) )</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

**Algorithm 3:** The detail of runtime verification algorithm

**Require:** currlist is the list of current rules that were enabled,
rulelist is list of rules that are defined to verify the system.

**Input:** message msg, occurrence time t.

**Output:** true/false

1. res ← true;
2. list ← ∅; // a list;
3. // step 1: check in currlist to give a verdict;
4. foreach rule in currlist do
5.   // if a rule is enabled many times, we just pick up the first one to consider and use a variable list to handle this problem;
6.   if rule.id ∉ list then
7.     if rule is future time then
8.       res ← res ∧ verify_future (rule, msg, t);
9.     else
10.    res ← res ∧ verify_past (rule, msg, t);
11.   list.add (rule.id);
12. // step 2: check in rulelist to enable new rules;
13. foreach rule in rulelist do
14.   if msg ∈ rule.supposition () ∧ rule.condition (msg) = true then
15.     if rule is future time then
16.       r1 ← rule; // create a new rule;
17.       r1.active_time ← t; // set active time;
18.       r1.getDataCorrelationValue (msg);
19.       currlist.add (r1); // add into enabled list;
20.     // the rule is not processed in the first step (rule.id ∉ list) if it is a past time rule;
21.     else if rule.correlation (msg) ≠ false ∧ rule.evaluate ()! = true ∧ rule.id ∉ list then
22.       res ← false;
23.     // the rule is not processed in first step (rule.id ∉ list) if it is a past time rule;
24.     else if rule is past time ∧ rule.id ∉ list ∧ rule.context.contain (msg) then
25.       r1 ← rule; // create a new rule;
26.       r1.active_time ← t; // set active time;
27.       r1.update (msg) // update context;
28.       r1.getDataCorrelationValue (msg);
29.       currlist.add (r1); // add into the list of enabled rules;
30. return res;
Algorithm 4: verify\_future(rule, msg, t)

Require: currlist is a global variable

Input : rule: a rule, msg: a message, t: occurrence time
Output : true/false

1 result ← true;

2 // the time condition is FALSE and the type of rule is Permission;
3 if verifyTime(t, rule.active\_time) = false ∧ rule.type = 'P'
4 then
5 result ← false;
6 currlist.remove(rule);

7 else if r.context.contain(msg) ∧ rule.correlation(msg) ≠ false
8 then
9 rule.update(msg) // update context;
10 if rule.evaluate() = true then
11 currlist.remove(rule);
12 // the time condition is TRUE and the type of rule is Prohibition;
13 if rule.type = 'F' ∧ verifyTime(t, rule.active\_time) = true
14 then
15 result ← false;
16 else if rule.evaluate() = false
17 then
18 currlist.remove(rule);
19 // type of rule is Permission;
20 if rule.type = 'P'
21 then
22 result ← false;
23 else
24 if verifyTime(t, rule.active\_time) = false
25 then
26 currlist.remove(rule);
27 else if rule.context.contain(msg) ∧ rule.correlation(msg) ≠ false
28 then
29 rule.update(msg);
30 return result;

Algorithm 5: verify\_past(rule, msg, t)

Require: currlist is a global variable

Input : rule: a rule, msg: a message, t: occurrence time
Output : true/false

1 result ← true;

2 if msg ξ rule.supposition() ∧ rule.condition(msg) = true ∧
3 rule.correlation(msg) ≠ false
4 then
5 currlist.remove(rule);
6 // the time condition is TRUE and type of rule is Prohibition;
7 if rule.type = 'F' ∧ verifyTime(t, rule.active\_time) = true
8 then
9 result ← false;
10 else
11 // the type of rule is Permission;
12 if rule.type = 'P'
13 then
14 result ← false;
15 else
16 if verifyTime(t, rule.active\_time) = false
17 then
18 currlist.remove(rule);
19 else if rule.context.contain(msg) ∧ rule.correlation(msg) ≠ false
20 then
21 rule.update(msg);
22 return result;
The checking engine in the architecture implements the runtime verification algorithm 3. It allows us to verify each incoming message without any constraint of order dependencies, and is applicable to both of on-line and off-line testing. Moreover, it verifies the validation of current message without using any storage memory. In order to use this engine for other systems, we define an interface IParseData shown in Figure 3 as an adapter to parse the incoming data for RV4WS if the data structure of input/output messages from another system is different from ours. The methods in IParseData are for gathering information from incoming messages. Method getMesssageName() returns the message name from its content, while method queryData() allows us to query a data value from a field of message content. This interface is implemented for each application case. For example, its implementation is class ParseSoapImpl for a web service application. This engine has been designed as a java library and is controlled by a component known as Controller which receives a data stream coming from a TCP/IP port.

The input format for this tool is an XML file as defined in Figure 4. A rule with a true or false verdict respectively represents a permission or prohibition. The context of a rule will be expressed as an expression with three operators AND, OR and NOT. Each data correlation is defined as a property with some query expressions from different SOAP messages. For web service applications, we have developed a Graphic User Interface (GUI) that allows us to easily define a set of rules from WSDL files.

The checking algorithm returns a fail verdict if a rule is found not satisfied, meaning that this rule is not applicable to the current message. To identify which rule fails at an arrival of message, we have developed a Graphic User Interface (GUI) for visualizing some statistical properties that are calculated at any moment of testing process. Whenever a rule is activated which means that its conditions have been satisfied, a statistical visualization does not have to be refreshed in real-time, rather, it can be refreshed after a duration.

The checking engine in the architecture implements the runtime verification algorithm 3. It allows us to verify each incoming message without any constraint of order dependencies, and is applicable to both of on-line and off-line testing. Moreover, it verifies the validation of current message without using any storage memory. In order to use this engine for other systems, we define an interface IParseData shown in Figure 3 as an adapter to parse the incoming data for RV4WS if the data structure of input/output messages from another system is different from ours. The methods in IParseData are for gathering information from incoming messages. Method getMesssageName() returns the message name from its content, while method queryData() allows us to query a data value from a field of message content. This interface is implemented for each application case. For example, its implementation is class ParseSoapImpl for a web service application. This engine has been designed as a java library and is controlled by a component known as Controller which receives a data stream coming from a TCP/IP port.

The input format for this tool is an XML file as defined in Figure 4. A rule with a true or false verdict respectively represents a permission or prohibition. The context of a rule will be expressed as an expression with three operators AND, OR and NOT. Each data correlation is defined as a property with some query expressions from different SOAP messages. For web service applications, we have developed a Graphic User Interface (GUI) that allows us to easily define a set of rules from WSDL files.

The checking algorithm returns a fail verdict if a rule is found not satisfied, meaning that this rule is not applicable to the current message. To identify which rule fails at an arrival of message, we have developed a Graphic User Interface (GUI) for visualizing some statistical properties that are calculated at any moment of testing process. Whenever a rule is activated which means that its conditions have been satisfied, a statistical visualization does not have to be refreshed in real-time, rather, it can be refreshed after a duration.

The checking engine in the architecture implements the runtime verification algorithm 3. It allows us to verify each incoming message without any constraint of order dependencies, and is applicable to both of on-line and off-line testing. Moreover, it verifies the validation of current message without using any storage memory. In order to use this engine for other systems, we define an interface IParseData shown in Figure 3 as an adapter to parse the incoming data for RV4WS if the data structure of input/output messages from another system is different from ours. The methods in IParseData are for gathering information from incoming messages. Method getMesssageName() returns the message name from its content, while method queryData() allows us to query a data value from a field of message content. This interface is implemented for each application case. For example, its implementation is class ParseSoapImpl for a web service application. This engine has been designed as a java library and is controlled by a component known as Controller which receives a data stream coming from a TCP/IP port.

The input format for this tool is an XML file as defined in Figure 4. A rule with a true or false verdict respectively represents a permission or prohibition. The context of a rule will be expressed as an expression with three operators AND, OR and NOT. Each data correlation is defined as a property with some query expressions from different SOAP messages. For web service applications, we have developed a Graphic User Interface (GUI) that allows us to easily define a set of rules from WSDL files.

The checking algorithm returns a fail verdict if a rule is found not satisfied, meaning that this rule is not applicable to the current message. To identify which rule fails at an arrival of message, we have developed a Graphic User Interface (GUI) for visualizing some statistical properties that are calculated at any moment of testing process. Whenever a rule is activated which means that its conditions have been satisfied, a statistical visualization does not have to be refreshed in real-time, rather, it can be refreshed after a duration.

VI. A CASE STUDY

In this section, we present a real-life case study known as Product Retriever [23] from WebMov project, and talk how to apply our RV4WS tool to test Product Retriever. This case study is a BPEL process that allows users to automate part of the purchasing process. It enables users to retrieve one searched product sold by an authorized provider. The search is limited by specifying a budget range and one or more keywords characterizing the product. The searched product is done through the operation getProduct and the parameter RequestProductType that is composed of information about the user (first-name, last-
name and department) and searched product (keyword, 
max price, category).

The process contains four partner services, namely 
AmazonFR, AmazonUK, CurrencyExchange and 
PurchaseService. They are developed by Montimage4 
and available online5. The overview behavior of the 
process is illustrated in Figure 6 and described as follows.

1) Receives a message from the client with the product 
and keywords of the characteristics of the product.
2) Contacts the PurchaseService partner to obtain 
the list of authorized providers for that product. 
In case there is not any authorized provider, an 
announcement be sent to the client by a fault 
message response.
3) Depending on the authorized provider result, the 
process contacts either the AmazonFR or Ama-
azonUK service to search a product that matches the 
price limit by Euro and the keywords.
4) Sends back to the client the product information 
and the name of the provider where the product was 
found, and the link from which it can be ordered. 
If a matching product is not found, a response with 
unsatisfied product will be sent back to the client.
5) After receiving the product information, the client 
can send an authorization request to confirm the 
purchase of the product within a certain duration of 
time (e.g., one minute).

The Product Retriever service is built with Netbeans 
6.5.1 and deployed by a Sun-Bpel-engine within a Glass-
fish 2.1 web server.

4http://www.montimage.com/

A. Test Product Retriever by RV4WS tool

In this section, we present some preliminary results 
from our first experiment on the case study of Product
we introduce three test purposes:

1) Rule definition: We can define many test purposes to verify the interaction order with partner services. Here we introduce three test purposes:

- During the execution of service, if the client receives a `ProductFault` message, then the Purchase service must have already returned a `ProviderFault` message. The time constraint for this test purpose is less important, so we can define the maximal time interval between two messages as 10 seconds.

\[ P(\text{start}(\text{ProductFault}) | O \in [10, 0]) \]

- If the Purchase service introduces the provider service `AmazonUK`, then the orchestration must contact the `CurrencyExchange` service within 10 seconds.

\[ P(\text{start}(\text{getProviderResponse} | \text{provider} = \text{AmazonUK}) | O \in [0, 10]) \]

- When the client sends an authorization request message to confirm the purchase of a product, then it must have received a product response message with field `EmptyResponseProduct` being null one minute ago. In this rule, the data correlation is used by `userId`.

\[ P(\text{start}(\text{getAuthorizationRequest}) | O \in [1, 0]) \]

\[ \text{done}(\text{getProductResponse}) \]

\[ [\text{EmptyResponseProduct} = \text{null}] / (\text{getAuthorizationRequest}.\text{userid}, \text{getProductResponse}.\text{userid}) \]

2) Checking results: Figure 8 illustrates the checking analysis of the Product Retriever, which indicates

- The fault messages that are defined in rule 1 do not occur (see the percentage in `fail` column of rule 1).
- Message `getProviderResponse` with `provider = AmazonUK` appeared three times (see value in `enabled count` column of rule 2), however there are two times when the tool did not found message `getCurrencyRateRequest` within 10 seconds from the occurrence time of message `getProviderResponse`. In Figure 9, we found the interval time between them is 26 seconds for the first fail case and 42 seconds for the second fail case. Then the tool produces the fail verdicts (the `fail` column of rule 2).
- Message `getAuthorizationRequest` appeared two times (see value on `enabled count` column of rule 3). Before that, message `getProductResponse` also appeared with field `EmptyResponseProduct` being empty and the interval time between them less than one minute.

In Figure 9, a `false` verdict is returned when the `itemSearchRequest` arrives because at the occurrence time of `itemSearchResponse`, the time constraint of rule 2 (i.e., 10 seconds) is not satisfied.

**VII. CONCLUSIONS**

This paper presents a passive test method for systems in particular for web services with (1) the definition of a language including logic expressions for constraints and (2) a verification method and a tool implementing the verification algorithm. This tool has been integrated in the WebMov tool chains. To verify the practicability of the proposed method on real systems, a real case study which is a web service composition known as Product Retriever has been extensively studied.

Extensions planned for this research include (1) a system for calculating the test coverage (corresponding to real need of the implementor of the web services), (2) an extension to test more complex distributed systems such as cloud computing architecture by integrating a set of distributed observers with recoveries of all the traces that need to be synchronized.

**ACKNOWLEDGMENT**

We would like to thank Ms. Nguyen Thi Kim Dung, a master student from PUF (Pole Universitaire Français) in Ho Chi Minh city, for helping us develop the RV4WS tool during her internship in LaBRI. We also thank Montimage for their case study Product Retriever.
Fig. 8. Checking analysis of Product Retriever

Fig. 9. A part of collected trace of Product Retriever

© 2012 ACADEMY PUBLISHER
REFERENCES


