An automated passive testing approach of real-time systems: Application to Web Services

Dung Cao, Richard Castanet, Patrick Félix

▶ **To cite this version:**

Dung Cao, Richard Castanet, Patrick Félix. An automated passive testing approach of real-time systems: Application to Web Services. IEEE Asia-Pacific Services Computing Conference APSCC 2011, Dec 2011, Jeju, South Korea. pp.78-85. hal-00997954

HAL Id: hal-00997954
https://hal.science/hal-00997954

Submitted on 11 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 automated passive testing approach of real-time systems: Application to Web Services☆

Tien-Dung Cao[a,*], Richard Castanet[b], Patrick Felix[b]

[a] *Department of Computer Science, School of Engineering - Tan Tao University, Tan Duc E-City, Duc Hoa District, Long An, Vietnam*
[b] *LaBRI - CNRS - UMR 5800, University of Bordeaux, 351 cours de la Liberation, 33405 Talence cedex, France*

Abstract

This paper proposes a new passive testing approach that verifies a timed trace with respect to a set of constraints. Our proposed approach can also be used as a runtime verification technique because it verifies message-by-message without storing them. To do this, firstly we proposed a formal syntax to define the constraints. In these constraints, the following problems are considered: time constraints (including past and future time), condition on message content, data correlation and combination of conditions by the operations AND, OR, NOT. Then, we proposed an algorithm to verify a timed trace with respect to a set of constraints. In addition to the theoretical framework we have developed a software tool, called RV4WS (Runtime Verification engine for Web Service), that helps in the automation of our passive testing approach. In particular the algorithm presented in this paper is fully implemented in the tool. Finally, we applied our tool to test a real-life case study of web service composition: Product Retriever.

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

1. Introduction

The activity of conformance testing is focused on verifying the conformity of a given implementation to its specification. In most cases testing is based on the ability of a tester that interacts directly with the implementation under test and checks the correction of the answers provided by the implementation (called: active testing). However, we cannot apply this method to test a running system, in many cases. For example, we do not have permission to access to the interface of the systems, or if we use the active method to test the function *create_new_account* of a bank service, this will make a mistake in the database of the service. Moreover, active testing does not allow us to check some security properties of the system that only happen at runtime or when many sessions are executed in parallel. Using passive testing, these problems will be solved because this is a method that collects the observable traces (or the log files) of the system by installing a probe and analyzes them with respect to a set of constraints (rules) [8, 9, 21] or a formal specification [12]. This method does not effect running system.

There are two techniques of passive testing: online and offline. The online technique (also called runtime verification technique) immediately checks an observable trace whenever an input/output event occurs. The advantages of this approach are: the faults may be found immediately and, we can stop the system to avoid the damage. On the contrary, the offline technique checks an execution trace after it is collected for a period of time, meaning the error is not found immediately if it occurs. This approach does not require added resources such as CPU, RAM or another computer to run the trace collection engine and checking engine in parallel. Depending on the concrete case, we can apply the online technique or offline technique to verify the system.

The rule is the constraints on the order of messages. We can understand a rule on a natural language as fol-

☆This Research is supported by the French National Agency of Research within the WebMov Project http://webmov.lri.fr

*Corresponding author

Email addresses: dung.cao@ttu.edu.vn (Tien-Dung Cao), castanet@labri.fr (Richard Castanet), felix@labri.fr (Patrick Felix)

low: if a message $M1$ occurs then a message $M2$ (or a suite of message $SM2$) must occur before/after $M1$ for a period of time. A generic temporal logic (like LTL) is usually used to define the constraints on the order of messages for model checking engine. Modeling constraint is required to specify permissions and prohibitions. However, the following problems must be considered when we define a set of constraints to verify a real-time system:

- Time constraints: The passive testing is the verification of messages order (before/after) in a sequence. When a message has occurred and we are waiting the next message. This message may be appear after a long duration where our system does not satisfy or does not appear. We cannot know that a message appears or not if we do not have a deadline because our system is running. Take for example a time constraint to verify a successful login if we send a loginResquest, we must receive a loginResponse within 10 seconds.

- Condition on message content: In case we do not want to verify all message in a sequence, we only verify some messages which its content satisfies some conditions. For instance, we are only interested in the messages that are sent/received to/from the machine A. This information is identified by message content (SourceIp=A or DestIp=A).

- Data correlation: A observable trace may be mixed by many traces or many sessions that are executed in parallel and we need to apply our constraints on the messages that belong to a trace or a session. In this case, firstly we must find the messages that have a correlation by its data values, then apply our constraints on these messages. For example, there are many sessions that are executed in parallel and each message has a *sessionId* field. Afterwards, we need to group the messages belonging to a session by using *sessionId* field of the messages before applying our rules to check the correctness. This is called data correlation.

- Combination of conditions: Sometime, to express a constraint, we must use the operations AND, OR, NOT to combine the conditions. For example, to express a security whenever we modify the database, before needing a login and this session still validate (i.e., do not logout).

This paper proposes a new approach that can use for both, formal passive testing (offline) or runtime verification (online), of a real-time system. Our approach verifies a timed trace with respect to a set of rules which four above properties (i.e., time constraints, condition on message content, data correlation, combination of conditions) are supported. We have proposed to extend the Nomad [14] language, by defining the constraints on each atomic action (fix conditions) and a set of data correlations between the actions, which is more convenient to use the LTL (for our objective) to define the rules. We chose this language because it provides a way to describle permissions, prohibitions that are granted (they are applied immediately) and obligrations (needing a time duration to complete) related to non-atomic actions within contexts that take time constraints. Moreover, its syntax and our natural language are quite similar. But we only support permissions and prohibitions by adding the time constraint that is bounded by an interval with time min and time max (i.e., the obligration is integrated into the permissions and the prohibitions). In addition to the theoretical framework we have developed a software tool, called RV4WS (Runtime Verification engine for Web Service), that helps in the automation of our passive testing approach. In particular the algorithm presented in this paper is fully implemented in the tool. We also applied our tool to test a real-life case study of WebMov[1] project, it is a web service composition: Product Retriever.

The rest of the paper is organized as follows. In Section 2 our notions of rule definition and the passive testing method is introduced in section 3. In section 5, we introduce the RV4WS tool and a case study that shows how to apply our tool to test a web service composition. Some discussion of existing methods for passive testing or runtime verification are presented in section 6. Finally, section 7 concludes the paper.

2. Rule definition

In our works, we consider each message as an atomic action. We use one or several messages to define a formula by using the operations AND, OR, NOT. During the formula definition, the constraint on message parameters value may be considered. Finally, from these formulas, the rule is defined in two parts: supposition (or condition) and context. The set of data correlations are included as an optional.

Definition 1. (Atomic action): We define an atomic action as one of following actions: an input message, an output message. Formally:

[1] `http://webmov.lri.fr`

$$AA := Event(Const)$$

where:

- *Event* represents an input/output message name;
- $Const := P \approx V | Const \wedge Const | Const \vee Const$ where:
  - P are the parameters. These parameters represent the relevant fields in the message.
  - V are the possible parameters values.
  - $\approx \in \{=, \neq, <, >, \leq, \geq\}$.

Definition 2. (Formula): A formula is defined recursively as following:

$$F := start(A) \mid done(A) \mid \neg F \mid F \wedge F \mid F \vee F \mid O^{d\in[m,n]}F$$

where:

- A is the atomic action.
- $start(A)$: A is being started.
- $done(A)$: A has been finished.
- $O^{d\in[m,n]}F$: F was true d units of time ago if $m > n$, F will be true d units of time if $m < n$, where m, n are two 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 and the operator = (equal) is used to compare a parameter with others. 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) are messages with p_i are the parameters where p_0^A,p_0^B,p_0^C have the same type. A data correlation set that is defined based on A, B and C is: $\{p_0^A,p_0^B,p_0^C\} \Leftrightarrow \{p_0^A = p_0^B = p_0^C\}$

Definition 4. (Rule with data correlation): Let α and β are formula, CS is a set of data correlations based on α and β (CS is defined based on the messages of α and β). A rule with data correlation is defined as: $\mathcal{R}(\alpha|\beta)/CS$[2] where $\mathcal{R} \in \{\mathcal{P}$: permission; $\mathcal{F}$: Forbidden;$\}$. The constraint $\mathcal{P}(\alpha|\beta)$ (resp. $\mathcal{F}$) means that it is permitted (resp. prohibited) to have α true when context β holds within the conditions of CS.

[2]CS is an optional part

Example 2. We only allow to create a new account on the services if we have had successfully login within maximum one day ago and have not logged out.

$$\mathcal{P}(start(createAccountReq)|O^{d\in[1,0]D}\ done(loginRes) \wedge \neg done(logoutReq))$$

In case we want to indicate the messages belonging to a session by using sessionId.

$$\mathcal{P}(start(createAccountReq)|O^{d\in[1,0]D} done(loginRes) \wedge \neg done(logoutReq))/ \{\{createAccountReq.sessionId, loginRes.sessionId, logoutReq.sessionId\}\}$$

3. Verification

3.1. Correctness of the system

In this section we explain how we can determine whether the execution traces obtained from the IUT satisfy the properties expressed by the rules.

Definition 5. (Correctness of a timed trace with respect to a finite set of rules): Let $\sigma = \sigma_0.\sigma_1.\sigma_3...$ an observable timed trace that is collected from a running system, $\Phi = \{\phi_0, \phi_1, ..., \phi_n\}$ a finite set of rules. We define: σ conforms to Φ if and only if at every occurrence of σ_i, does not exist ϕ_j such that ϕ_j is not timeout and the evaluation of ϕ_j after updating its context is not false.

3.2. Checking Algorithm

In this section, we briefly outline the computation mechanism used to determine whether a rule holds for some given input/output sequence of events. Our algorithm determines message-by-message the conformity with each rule without storing the message sequence. Here, we use two global variables: *currlist* is a list of current rules that were enabled and *rulelist* is a list of rules that are defined to verify the system. Before introducing the detail of algorithm, we present some functions to compute on the context of each rule:

- *update*: this function updates the value of context whenever a message arrives and this message exists in the context. For example, the context of a rule is $loginResponse \wedge \neg logoutRequest$. When the *loginResponse* message arrives, this context is updated as $true \wedge \neg logoutRequest$.
- *evaluate*: this function evaluates whether a context of rule is holds (true) or not. This function returns one of three values: *true*, $false$ or

undefined if a message that is not updated exists. While the evaluation, a message with the function *not* will be assigned provisionally is *true*. For example: at the time of evaluation, the expression $true \wedge \neg logoutRequest$ will be evaluated as $true \wedge true = true$.

- *correlation*: this function will return one in three values: *undefined*, *true* and *false*. *undefined* when a message *msg* is not defined in the set of data correlations of the rule. In the case that the *msg* is defined in the set of data correlations of rule, this function will query the correspondent value and compare it with the value of previous messages to return *true*/*false*.

- *contain*: to find a message in the context of a rule. This function returns *true* if the message *msg* is found in in the context of a rule and its condition is validated. For example, the context of the rule is the following expression: $msgA[msgA.id = 5]\&msgB$. When the *msgA* (with its value id=4) arrives, the contain function will be returned as *false* because the message name is found but its condition does not satisfy. In the case of *msgB* arrives, this function returns *true*.

As said earlier, there is two types of rule: future time and past time. To make this more clear, we will analyze the checking algorithm for each type.

3.3. Rule with future time

We know that each rule has two parts: the supposition part and the context part. The rule will be validated if its supposition was enabled and its context is hold (true). In a rule with future time, the context part will happen after its supposition was enabled. Our algorithm has two steps:

- Step 1) Each time that a message (called *msg*) arrives, we have a list of current rules (*currlist*) that have been enabled to wait the validation of its context. Therefore, we will firstly update the context of the current rule (noted *rule*) in this list (*currlist*) if *msg* appears in the context of *rule* and data correlation of *msg* satisfies if it is defined. Secondly, we will evaluate the context of each rule. If the context is *true* and the time constraint is satisfying, a verdict *pass*/*fail*, depending on the permission/prohibition of the rule, will be given in time *msg* arrives, and will remove this rule from current list (*currlist*). If we cannot evaluate the context, we will wait for the next message to complete the context. In this case, a *pass* verdict is given.

- Step 2) we will examine all rules in *rulelist* and enable it (add into *currlist*) if its supposition part contains the message *msg* and condition of supposition part is valid with the data of *msg*. When we enable a new rule, the properties of data correlation set will be assigned by the values that are queried from *msg*.

3.4. Rule with past time

In a rule with past time, the context part will happen before its supposition is enabled. It means that the context part must be completed and the *evaluate* function must return *true* or *false*, when its supposition is enabled. As the future time, we have also two steps:

- Step 1) we check firstly in the list of active rules (*currlist*). If its supposition part contains the message *msg* and the condition of the supposition part is valid with the data of the *msg* and the data correlation of the *msg* satisfies if it is defined. We will evaluate its context to give a verdict. On the contrary, we will check the time constraints on the rules to remove it from the list (*currlist*) if the time constraints do not satisfy. If the context of the rule contains this message (*msg*), we update the context to wait the next message.

- Step 2) we will examine all rules in the *rulelist* and enable it (add into *currlist* to wait the message in the supposition part) if its context contains the message *msg*. Like the future rule, the properties of data correlation set will also be assigned by the queried values of *msg*.

Finally, we combine it to have a complete algorithm. The detail of main checking algorithm is shown in algorithm 1. This algorithm verifies message-by-message and returns the verdict at a time of arrival message.

***Note**: this algorithm returns a *fail* verdict if it found a rule is not satisfying. This rule may be not applied to current message. To know which rule is fail at an arrival message, we propose a graphic statistics that shows the current test status.

Example 3. For example, we have an execution timed trace with 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)... The security rules that are defined to assess the system are:

$r_1 = \mathcal{P}(start(a_1)|O^{d\in[0,10]}\ done(b_1) \vee done(c_1))$,

$r_2 = \mathcal{P}(start(b_2)|O^{d\in[+\infty,0]}\ done(a_2) \wedge \neg done(c_2))$

The table 1 shows the results of the algorithm.

Algorithm 1: 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 consider only one time (i.e. one session);
6      if rule.id ∉ list then
7          if rule is future time then
8              res ←verify_future(rule, msg, t, res);
9          else
10             res ←verify_past(rule, msg, t, res);
11         list.add(rule.id);
12 //step 2: check in rulelist to enable new rule;
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.assignValue4Properties(msg);
19             currlist.add(r1); //add into enabled list;
20         else if rule.correlation(msg)≠ false ∧ rule.evaluate()! = true ∧ rule.id ∉ list then
21             res ← false;
22     else if rule is past time ∧ rule.id ∉ list ∧ rule.context.contain(msg) then
23         r1 ← rule; //create a new rule;
24         r1.active_time ← t; // set active time;
25         r1.update(msg) //update context;
26         r1.assignValue4Properties(msg);
27         currlist.add(r1); //add into actived list;
28 return res;
```

Algorithm 2: verify_future(*rule*, *msg*, *t*, *result*)

Require: *currlist*: is a global variable

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

```
1  if verifyTime(t, rule.active_time) = false ∧ rule.type =′ P′ then
2      result ← false;
3      currlist.remove(rule);
4  else if r.context.contain(msg) ∧ rule.correlation(msg)≠ false then
5      rule.update(msg) //update context;
6      if rule.evaluate() = true then
7          currlist.remove(rule);
8          if rule.type =′ F′ ∧ verifyTime(t, rule.active_time) = true then
9              result ← false ;
10     else if rule.evaluate()=false then
11         currlist.remove(rule);
12         if rule.type =′ P′ then
13             result ← false;
14 return result;
```

Algorithm 3: verify_past(*rule*, *msg*, *t*, *result*)

Require: *currlist*: is a global variable

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

```
1  if msg ∈ rule.supposition() ∧ rule.condition(msg)=true ∧ rule.correlation(msg)≠ false then
2      currlist.remove(rule);
3      if rule.evaluate() = true then
4          if rule.type =′ F′ ∧ verifyTime(t, rule.active_time) = true then
5              result ← false;
6      else
7          if rule.type =′ P′ then
8              result ← false;
9  else
10     if verifyTime(t, rule.active_time) = false then
11         currlist.remove(rule);
12     else if rule.context.contain(msg) ∧ rule.correlation(msg)≠ false then
13         rule.update(msg);
14 return result;
```

message	enabled rule list	verdict	add/remove (+/-)
$(a_1, 0)$	$\{r_1^+ = \mathcal{P}(true \vert O^{d\in[0,10]} done(b_1) \vee done(c_1))\}$	true	$+r_1$
$(a_2, 2)$	$\{r_1 = \mathcal{P}(true \vert O^{d\in[0,10]} done(b_1) \vee done(c_1));$ $r_2^+ = \mathcal{P}(start(b_2) \vert O^{d\in[+\infty,0]} true \wedge \neg done(c_2))\}$	true	$+r_2$
$(a_1, 3)$	$\{r_1 = \mathcal{P}(true \vert O^{d\in[0,10]} done(b_1) \vee done(c_1));$ $r_2 = \mathcal{P}(start(b_2) \vert O^{d\in[+\infty,0]} true \wedge \neg done(c_2));$ $r_1^+ = \mathcal{P}(true \vert O^{d\in[0,10]} done(b_1) \vee done(c_1))\}$	true	$+r_1$
$(b_2, 8)$	$\{r_1 = \mathcal{P}(true \vert O^{d\in[0,10]} done(b_1) \vee done(c_1));$ $r_1 = \mathcal{P}(true \vert O^{d\in[0,10]} done(b_1) \vee done(c_1))\}$	true	$-r_2$
$(b_1, 9)$	$\{r_1 = \mathcal{P}(true \vert O^{d\in[0,10]} done(b_1) \vee done(c_1))\}$	true	$-r_1$
$(a_2, 12)$	$\{r_1 = \mathcal{P}(true \vert O^{d\in[0,10]} done(b_1) \vee done(c_1));$ $r_2^+ = \mathcal{P}(start(b_2) \vert O^{d\in[+\infty,0]} true \wedge \neg done(c_2))\}$	true	$+r_2$
$(b_3, 15)$	$\{r_2 = \mathcal{P}(start(b_2) \vert O^{d\in[+\infty,0]} true \wedge \neg done(c_2))\}$	false*	$-r_1$
$(c_1, 16)$	$\{r_2 = \mathcal{P}(start(b_2) \vert O^{d\in[+\infty,0]} true \wedge \neg done(c_2))\}$	true	

Table 1: An example of runtime verification

*at the message $(b_3, 15)$, we receive a $false$ verdict because rule r_1, which has the last enabled message is $(a_1, 3)$, is fail at the time 15.

4. RV4WS tool

RV4WS (Runtime Verification for Web services) is implemented to verify a web service at runtime based on a set of constraints that are declared by the defined syntax in section 2. 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 detail of architecture is shown in figure 1.

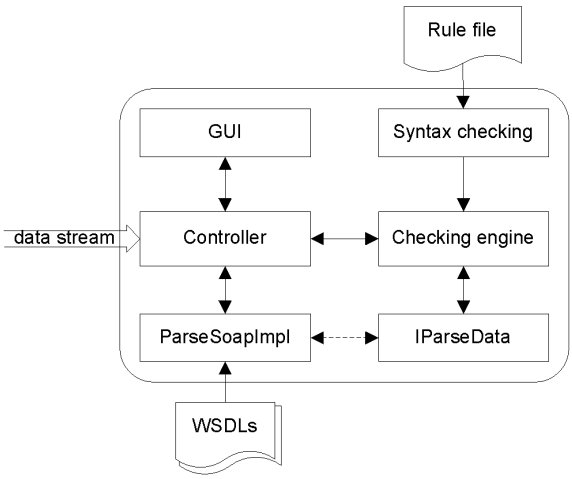

Figure 1: Architecture of the RV4WS tool

One of the most interesting components in this architecture is the checking engine component that implemented the runtime verification algorithm 1. The engine allows us to verify each incoming message without any constraint of order dependencies, so we can apply this approach to both of online and offline testing. Also, this algorithm verifies the validation of current message without using any storage memory. In order to use this engine for the other systems, there is a difference between the systems is the data structure of input/output messages, we define an interface (i.e., *IParseData*, shown in the figure 2) as an adapter to parse the incoming data of RV4WS. The methods in *IParseData* are for gathering information from incoming message. *getMesssageName*() returns the message name from its content and *queryData*() allows us to query a data value from a field of message content. In each concrete case, we will implement this interface. For example, in the case of Web services, its implementation is the class *ParseSoapImpl*. This engine has been designed as a java library and is controlled by a component called Controller, which received a data stream coming from TCP/IP port.

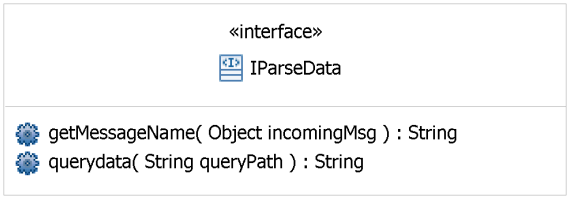

Figure 2: ParseData Interface of RV4WS

The input format for this tool is a xml file that has been defined in figure 4. A rule with a *true* verdict represents a permission and a $false$ verdict represents a prohibition. A context of rule will be expressed as an ex-

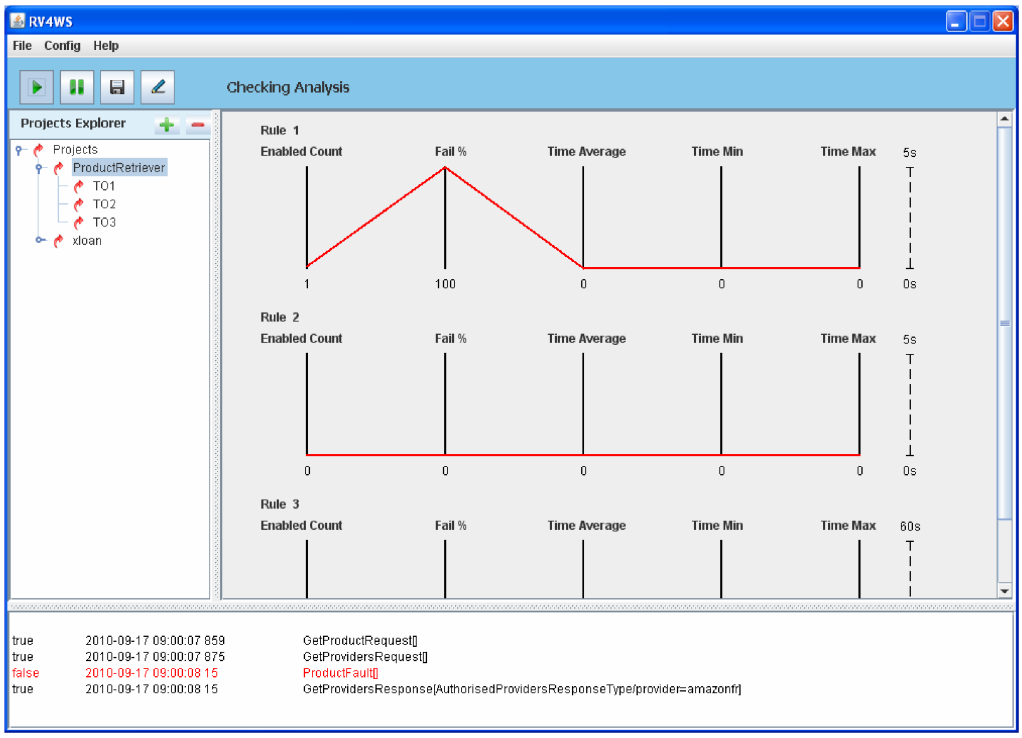

Figure 3: The main GUI and checking analysis of RV4WS tool

pression with three operators *AND*, *OR* and *NOT*. Each data correlation is defined as a property with some query expressions from the different SOAP messages. In the case of web services, we have developed a Graphic User Interface (GUI) that allows us to easily define a set of rules from WSDL files.

```
<?xml version="1.0" encoding="UTF-8"?>
<rules>
      <rule applyProperty="true" id="1" name="" verdict="true">
      <if>
            <message> requestRequest[]</message>
      </if>
      <then time_max="1" time_min="0" time_type="m" type="after">
            <context>
                  <expression> xLoanConfirmRequest[]</expression>
            </context>
      </then>
      <properties>
            <property name="correlation_Id" type="int">
                  <query>requestRequest.requestInfo/id</query>
                  <query>xLoanConfirmRequest.confirmIn/id</query>
            </property>
      </properties>
      </rule>
</rules>
```

Figure 4: Rule format example

If a rule is found to be not satisfying, the checking algorithm returns a fail verdict. This rule may be not applied to the current message. To know which rule has failed at an arrival message, we have also presented a Graphic User Interface (GUI) that is used to visualize some statistical properties, calculated at any moment of testing process. Whenever a rule is activated, this means that its conditions have been satisfied and a statistical property such as type counter will be used to compute the percentage of un-satisfying time when applying the rule on the input data stream. If the rule was satisfied, we need to know the time duration from the activating moment to its context's holding moment. We have three statistical properties about time (time-min, time-max and time-average) for each rule.

Now we need to know the values of these statistical properties and also visualize the relationships between them. For example, one rule executing shows its fail percentage in proportion to its duration time or to others properties. If we had used a histogram view and applied it for each, we would not have been able to get this information because of the different scales of these properties. We built a visual interface which is based on the idea of parallel coordinates scheme, introduced by Inselberg [19]. In information visualization, parallel coordinates view is used to show the relationships between items in a multidimensional dataset. Each axes in this view parallel to each other and a point in n-dimensional space is represented as a polyline with vertices on these axes. Considering that list of statistical properties of our testing process as a multivariate/multi dimension

data, we have applied this visualization to RV4WS tool and made it possible to explore the result of our checking algorithms. As said earlier, we have implemented the checking algorithms inside RV4WS tool which enables a user-tester to verify these conditions defined in rules. Then the user-tester discovers that rule's properties change over time and he or she often needs a complete view of these traces of testing process. There are the parallel coordinates views correspondent to rules. In the figure 3, each scheme of parallel coordinates represents a time-log of statistical values as these polylines crossing properties axes. Within each view, there is a single polyline per time instance. The lines of current time are always highlighted. So this view enables the tester to visualize rapidly if these changes of executing rule's properties are interesting or not. Because of this problem, this visualization is refreshed after a duration. It means that it does not run in real-time.

5. A case study

In this section, we present a real-life case study, named Product Retriever [23], of WebMov project and how to apply our tool (i.e., RV4WS) to test the Product Retriever. This case study is a BPEL process that allows users to automate part of the purchasing process. It enables you to retrieve one searched product sold by a preauthorised 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 (firstname, lastname and department) and searched product (keyword, max price, category). This process has 4 partner services, named *AmazonFR*, *AmazonUK*, *CurrencyExchange* and *PurchaseService* that are developed by Montimage[3] and available at `http://80.14.167.59:11404/servicename`, and its overview behaviour, illustrated in figure 5, is described by the following:

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 a case where there is no authorized provider, an announcement will be sent to the client by a fault message response.

[3] `http://www.montimage.com/`

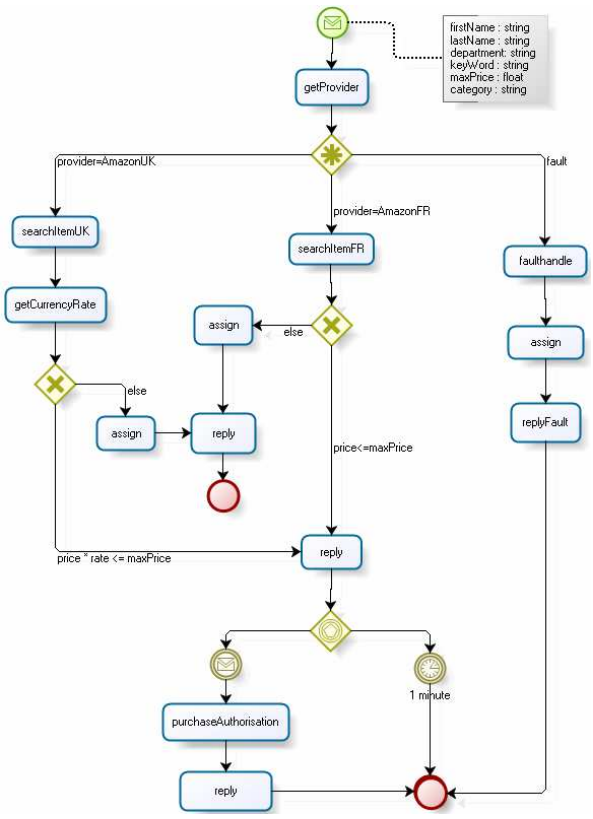

Figure 5: ProductRetriever - BPMN specification

3. Depending on the authorized provider result, the process contacts either the AmazonFR or the AmazonUK 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 a link to where it can be ordered. If a matching product is not found, a response with unsatisfying 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 (i.e., one minute) of time.

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

5.1. Test Product Retriever by RV4WS tool

In this section, we present some preliminary results we got after conducting our first experimentations on the Product Retriever case study using RV4WS tool.

SoapUI [24] is a well known test tool for web services based. We used it in our experiments as a client of Product Retriever service, sending requests to activate the web service (i.e., BPEL process). To collect the communicating messages between the Product Retriever service and its partners (including SoapUI), we have developed a proxy that allows us to forward a message to a specified destination. This allows us to receive and forward from/to some sources and destinations. Each connection is handled on a different port. Afterwards, this message and its time occurrence are also sent to our tool (i.e., RV4WS) to check its correctness. SoapUI and Product Retriever service were configured to make connections through the proxy. The connection information (service name) is also sent to RV4WS to help this tool to easily identify which message belongs to which service. Figure 6 shows our testbed architecture.

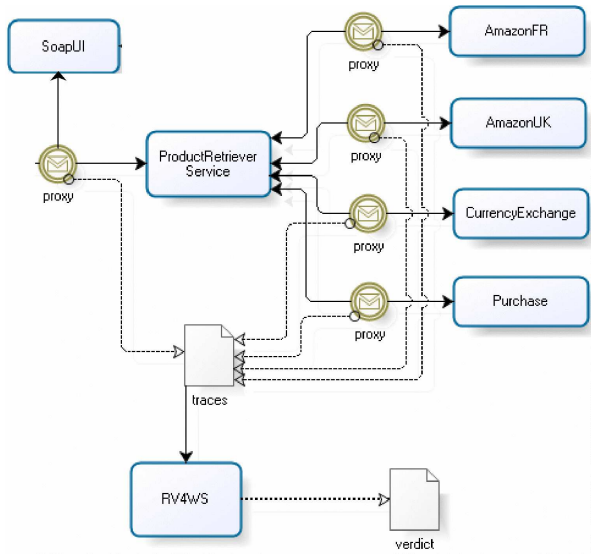

Figure 6: Testbed architecture

5.1.1. Rule definition

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

1. During the execution of service, if the client receives a *ProductFault* message, so before the Purchase service must return a *ProviderFault* message. The time constraint of this test purpose is less important, so we define the maximum interval time between two messages as 10 seconds.

$$\mathcal{P}(start(ProductFault)|O^{d\in[10,0]s}\ done(ProviderFault))$$

2. If the Purchase service introduces the provider service AmazonUK then the orchestration must contact the CurrencyExchange service within maximum of 10 seconds.

$$\mathcal{P}(start(getProviderResponse[provider = AmazonUK])|O^{d\in[0,10]s}\ done(getCurrencyRateRequest))$$

3. When the client sends an authorization request message to confirm the purchase of a product, so it must receive a product response message with the *EmptyResponseProduct* field be null within maximum one minute ago. In this rule, the data correlation is used by *userId*.

$$\mathcal{P}(start(getAuthorizationRequest)|O^{d\in[1,0]m}\ done(getProductResponse\ [EmptyResponseProduct = null]))/\ (getAuthorizationRequest.userid,\ getProductResponse.userid)$$

5.1.2. Checking results

Figure 7 presents the checking analysis of the Product Retriever. This figure indicates: 1) the fault messages that are defined in rule 1 do not occur. 2) the *getProviderResponse* message with *provider* = *AmazonUK* appeared two times, but the tool did not found a message *getCurrencyRateRequest* within 10 seconds from the occurrence time of the message *getProviderResponse*. See figure 8, we found the interval time between them is 11 seconds for the first case and 25 seconds for the second case. 3) the message *getAuthorizationRequest* appeared three times. Before that, the *getProductResponse* message also appeared with the field *EmptyResponseProduct* is empty and the interval time between them is less than one minute. Figure 8 returns the $false$ verdict when the *itemSearchResponse* arrives because at the occurrence time of *itemSearchResponse*, the time constraint for the second rule (i.e., 10 seconds) does not satify.

6. Discussion

This section will discuss some passive testing approaches.

6.1. Passive testing of systems

Bayse et al [9] and Cavalli et al [12] proposed a passive testing approach based on invariants of a Finite State Machine (FSM). For a FSM $M = (\mathcal{S}, s_{in}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ where $\mathcal{S}$ is a set of finite states, s_{in} is an initial state, $\mathcal{I}$

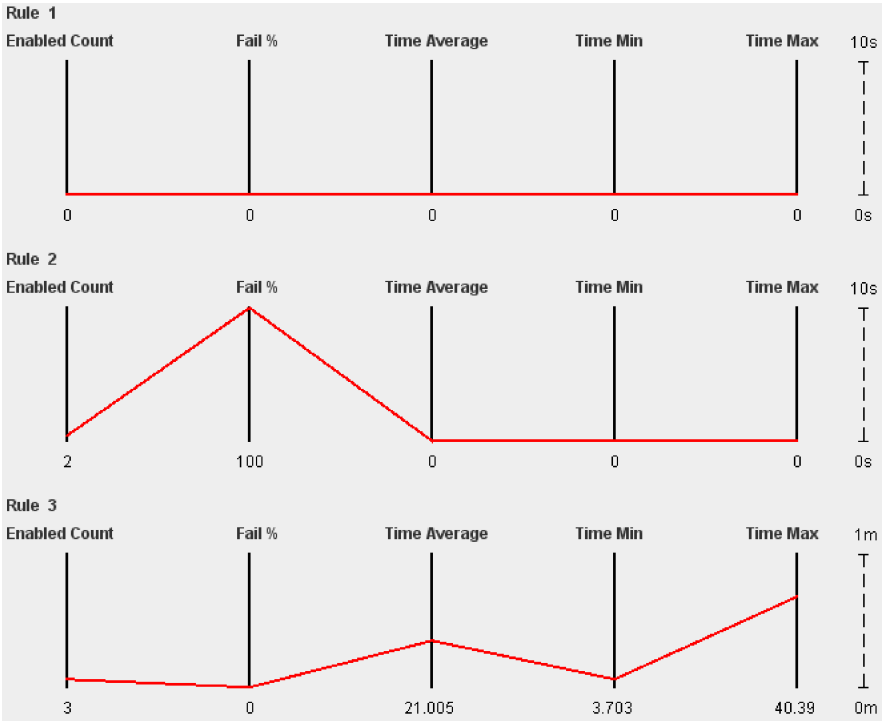

Figure 7: Checking analysis of Product Retriever

true	2010-09-30 04:49:15 78	GetProvidersRequest[]
true	2010-09-30 04:49:15 62	GetProductRequest[]
true	2010-09-30 04:49:15 390	GetProvidersResponse[AuthorisedProvidersResponseType/provider=AmazonFR]
true	2010-09-30 04:49:15 406	itemSearchRequest[]
true	2010-09-30 04:49:18 703	itemSearchResponse[]
true	2010-09-30 04:49:48 718	GetProductResponse[productOut/EmptyResponseProduct=null]
true	2010-09-30 04:49:52 421	GetAuthorisationRequest[]
true	2010-09-30 04:49:52 437	PurchaseAuthorisationRequest[]
true	2010-09-30 04:49:54 375	PurchaseAuthorisationResponse[]
true	2010-09-30 04:50:24 390	GetAuthorisationResponse[]
true	2010-09-30 04:51:24 93	GetProvidersRequest[]
true	2010-09-30 04:51:24 93	GetProductRequest[]
true	2010-09-30 04:51:26 359	GetProvidersResponse[AuthorisedProvidersResponseType/provider=AmazonUK]
true	2010-09-30 04:51:26 375	itemSearchRequest[]
false	2010-09-30 04:51:37 156	itemSearchResponse[]
true	2010-09-30 04:51:37 171	CurrencyExchangeRequest[]
true	2010-09-30 04:51:50 390	CurrencyExchangeResponse[]
true	2010-09-30 04:52:20 406	GetProductResponse[productOut/EmptyResponseProduct=null]
true	2010-09-30 04:52:39 328	GetAuthorisationRequest[]
true	2010-09-30 04:52:39 343	PurchaseAuthorisationRequest[]
true	2010-09-30 04:52:47 843	PurchaseAuthorisationResponse[]
true	2010-09-30 04:53:17 859	GetAuthorisationResponse[]
true	2010-09-30 04:54:45 906	GetProductRequest[]
true	2010-09-30 04:54:45 921	GetProvidersRequest[]
true	2010-09-30 04:54:52 515	GetProvidersResponse[AuthorisedProvidersResponseType/provider=AmazonUK]
true	2010-09-30 04:54:52 531	itemSearchRequest[]
false	2010-09-30 04:55:17 765	itemSearchResponse[]
true	2010-09-30 04:55:17 765	CurrencyExchangeRequest[]
true	2010-09-30 04:55:47 0	CurrencyExchangeResponse[]
true	2010-09-30 04:56:17 0	GetProductResponse[productOut/EmptyResponseProduct=null]
true	2010-09-30 04:56:57 390	GetAuthorisationRequest[]

Figure 8: Trace collection of Product Retriever

is the set of input actions, O is the set of output actions and $\mathcal{T}$ is the set of transitions. The authors define two types of invariants:

- *Simple invariant*: a trace such as $i_1/o_1, i_1/o_1, ..., i_{n-1}/o_{n-1}, i_n/O$ is a simple invariant of M if each time that the trace $i_1/o_1, i_1/o_1, ..., i_{n-1}/o_{n-1}$ is observed if we obtain the input i_n then we necessarily get an output belonging to O,where $O \subseteq \mathcal{O}$.
- *Obligation invariant*: to express properties such as "if y happens then we must have that x had happened before".

Next, the authors present two algorithms to check from left-to-right and right-to-left a finite trace to give a verdict. This approach does not consider the time constraints on the traces. TIPS [11](Test Invariant of Protocols and Services) is an implementation tool of this approach.

An extention of simple invariant with time constraints, called Timed Invariant which allows us to express temporal properties is introduced in the works of C. Andrés et al [3, 1, 2]. There are some limitations of the Timed Invariant model:

- Supports only the future time, the past time was not admit. Its semantic is: if a pair input/output event (the interval time between an input and an output is also considered) or some event pairs have been happened and we continue obtain an input, then an output must be happened after a duration;
- Not support the operations as NOT, AND, OR to combine some conditions to a Timed Invariant;
- Not consider the constraints on the content of each event, so the data correlation problem between the events is also not considered;
- Finally, the tool PasTe [1] that is implemented to check the correctness of a log with respect to a set of time invariant does not allow us to verify an execution trace in parallel with the trace collection engine (it means the runtime verification or the online checking).

Mallouli et al [16] proposed the security rule using the Nomad language to express the constraints on the traces with *obligations*, *prohibitions* and *permissions*. A prohibition or a permission rule is granted and it applies immediately on the trace, an obligation rule needs a deadline and the works are not completed before this deadline. This approach solves the time constraints of invariant approach. An algorithm to check the correction of the trace following these security rules is introduced. This approach does not consider the correlation of messages by its data values, an important problem of passive testing.

M. Tabourier and A. Cavalli [22] proposed an approach to verify the traces actually belong to the accepted specification that is provided by a finite state machine. This method is composed of two stages:

- Firstly, passive homing sequence is applied to determine the current state. Initially, all states are the candidates. When an input/output arrives, these current states will be updated by the destination state of corresponding transition if it is the source state of transition. If not, it is 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 has been detected.
- Secondly, fault detection is used by applying the search technique from the current state and the current input/output pair. If a state which does not accept the following transition is reached, there is an error. If not, the end of the trace is reached, no error was detected.

In the case where the trace is collected from the execution of multi-sessions that run in parallel, we can not used this approach. Moreover, this method does not consider the time constraints on the traces.

6.2. Passive testing of web services

In recent years, many methods and tools are proposed and developed for passive testing of a web service (including a composite of web services) [17, 4, 5, 6, 13, 20]. These works 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] propose the utilization of Stream X-machines for constructing formal behavioral specifications of Web services. The authors also present a runtime monitoring and verification architecture and discuss how it can be integrated into different types of service-oriented infrastructures. But the authors do 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] present a monitoring framework for BPEL orchestration which is obtained by integrating two approaches namely Dynamo and Astro. These approaches are used for dynamic statistics of some properties of BPEL process from single instance or multi instances. These works 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] propose a trace collection mechanism for SOA by integrating modules within BPEL engine and a tool [13, 16] that checks offline an execution trace. This approach uses the Nomad [14] language to define the security rule. But it does not allow us to check real-time (i.e., "online") whenever a message happened. Moreover, this work does not consider the data correlation between the messages in the rules.

The works of Li et al [20, 21] present the pattern and scope operators as the rule-based to define the interaction constraints of Web services. The authors use the finite state automata (FSA) 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. Moreover, this work does not consider the time constraints.

7. Conclusions

This paper presents a passive test method for systems, in particular, Web services with : the definition of a language including logic expressions for constraints, a verification methodology and a tool implementing the verification algorithm. This tool has been integrated in the WebMov tool chains. In order to show the application of the proposed methodology on real systems, a real case study, a web service composition, named Product Retriever, has been extensively studied.

Extensions are planned for this research: first a system for calculating the test coverage (corresponding tio real need of the implementor of the web services), an extension to test more complex distributed systems from such type of 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 Nguyen Thi Kim Dung, a master student of PUF (Pole Universitaire Français) in Ho Chi Minh city, who help us to develop the RV4WS tool in the context of her internship in LaBRI.

References

[1] C. Andrés, Mercedes G. Merayo, and M, Núnez, "Formal correctness of a passive testing approach for timed systems", *IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pp. 67-76, Apr 01- 04, 2009, Denver, Colorado , USA

[2] C. Andrés, Mercedes G. Merayo, M. Núnez, "Passive Testing of Stochastic Timed Systems", *International Conference on Software Testing Verification and Validation*, pp. 71 - 80, Apr 01-04, 2009, Denver, Colorado , USA

[3] C. Andrés, Mercedes G. Merayo, M. Núnez, "Passive Testing of Timed Systems", *International Symposium on Automated Technology for Verification and Analysis*, pp. 418 - 427, vol. 5311, LNCS, 2008.

[4] L. Baresi, S. Guinea,M. Pistore, and M. Trainotti, "Dynamo + Astro: An integrated Approach for BPEL Monitoring", *2009 IEEE International Conference on Web Service*, pp. 230 - 237, July 6-10, 2009, Los Angeles, CA, USA.

[5] L. Baresi, S. Guinea, R. Kazhamiakin, and M. Pistore, "An Integrated Approach for the Run-Time Monitoring of BPEL Orchestrations", *1st European Conference on Towards a Service-Based Internet*, pp. 1 - 12, 2008, Madrid, Spain.

[6] L. Baresi and S. Guinea, "Towards Dynamic Monitoring of WS-BPEL Processes", *Third International Conference on Service-Oriented Computing*, pp. 269 - 282, Dec 12-15, 2005, Amsterdam, The Netherlands.

[7] A. Benharref, R. Dssouli, Mohamed A. Serhani, A. En-Nouaary, and R. Glitho, "New Approach for EFSM-Based Passive Testing of Web Services", *Testing of Software and Communicating Systems*, pp. 13-27, vol. 4581, 2007.

[8] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-Based Runtime Verification", *5th International Conference on Verification, Model Checking, and Abstract Interpretation*, Jan 11-13, 2004, Venice, Italy.

[9] E. Bayse, A. Cavalli, M. Nunez, and F. Zaidi, "A passive testing approach based on invariants: application to the WAP", *Computer Networks 48 (2005) pp. 247 - 266.*

[10] T.-D. Cao, T.-T. Phan-Quang, P. Felix, and R. Castanet, "Automated Runtime Verification for Web services", *IEEE International Conference on Web services*, pp. 76-82, July 5-10, 2010, Miami, FL, USA.

[11] A. Cavalli, Edgardo Montes De Oca, W. Mallouli, and M. Lallali, "Two Complementary Tools for the Formal Testing of Distributed Systems with Time Constraints", *12th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Canada, Oct 27 - 29, 2008.

[12] A. Cavalli, C. Gervy, and S. Prokopenko, "New approaches for passive testing using an extended finite state machine specification", *Infomation and Software technology*, Vol. 45, No. 12, pp. 837-852, 2003.

[13] A. Cavalli, A. Benameur, W. Mallouli, and K. Li, "A Passive Testing Approach for Security Checking and its Pratical Usage for Web Services Monitoring", *NOTERE 2009*, Montreal, Canada, 2009.

[14] F. Cuppens, N. Cuppens-Boulahia, and T. Sans, "Nomad: a security model with non atomic actions and deadlines", *18th IEEE Workshop on Computer Security Foundations*, pp. 186 - 196, 20-22 June 2005, Aix-en-Provence, France.

[15] S. Halle, R. Villemaire, and O. Cherkaoui, "Specifying and Validating Data-Aware Temporal Web Service Properties" *IEEE Transactions on Software Engineering*, Vol. 35, No. 5 (2009), pp. 669-683.

[16] W. Mallouli, F. Bessayah, A. Cavalli, and A. Benameur, "Security Rules Specification and analysis Based on Passive Testing" *IEEE Global Telecommunications Conference, 2008*, pp. 1 - 6, Nov 30 - Dec 4, 2008, New Orleans, LA, USA.

[17] D Dranidis, E. Ramollari, and D. Kourtesis, "Run-time Verification of Behavioural Conformance for Conversational Web Services", *2009 Seventh IEEE European Conference on Web Services*, pp. 139 - 147, Nov 9 - 11, 2009, Eindhoven, The Netherlands.

[18] A . Goldberg and K. Havelund, "Automated Runtime Verification with Eagle", *Verification and Validation of Enterprise Information Systems*, May 24, 2005, Miami, USA.

[19] Alfred Inselberg, "The plane with parallel coordinates", *The Visual Computer*, pp. 69 - 91, Vol 1, No 2, Springer Berlin / Heidelberg, August, 1985.

[20] Z. Li, Y. Jin, and J. Han, "A Runtime Monitoring and Validation Framework for Web Service Interactions", *Proceedings of the Australian Software Engineering Conference*, pp. 70 - 79, Apr 18 -21, 2006, Sydney, Australia.

[21] Z. Li, J. Han, and Y. Jin, "Pattern-Based Specification and Validation of Web Services Interaction Properties", *In Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC'05)*, pp. 73 - 86, Dec 12-15, 2005, Amsterdam, The Netherlands.

[22] M. Tabourier and A. Cavalli, "Passive testing and application to the GSM-MAP protocol", *Information ans software technology*, pp. 813 - 821, Vol 41, 1999.

[23] W. P. Consortium, "D5.1 webmov case studies: definition of functional requirements and test purposes", WebMov, Tech. Rep. WEBMOV-FC-D5.1/T5.1, 2009.

[24] Eviware, `http://www.eviware.com/`.