Automated Runtime Verification for Web Services

Tien-Dung Cao, Trung-Tien Phan-Quang, Patrick Félix, Richard Castanet

Automated Runtime Verification for Web Services

Tien-Dung Cao, Trung-Tien Phan-Quang, Patrick Felix, and Richard Castanet
LaBRI - CNRS - UMR 5800, University of Bordeaux 1
351 cours de la libération, 33405 Talence cedex, France.
Email: {cao,phanquan,felix,castanet}@labri.fr

***Abstract*—This paper presents a methodology to perform passive testing of behavioural conformance for the web services based on the security rule. The proposed methodology can be used either to check a trace (offline checking) or to runtime verification (online checking) with timing constraints, including future and past time. In order to perform this: firstly, we use the Nomad language to define the security rules. Secondly, we propose an algorithm that can check simultaneously multi instances. After that, with each security rule, we propose a graphical statistics, with some fixed properties, that helps to tester easy assess about the service. 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 are fully implemented in the tool. We also present a mechanism to collect the observable trace in this paper.**

***Keywords*-Web Services, Runtime verification, Passive testing, Rule specification, Nomad language.**

I. 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 to 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 case. For example, if we use the active method to test the function $create\_new\_account$ of a bank service, this will effect to a database of the service. With a composite web service, we can only use the method of active testing for unit testing by simulating its partners to guarantee that does not effect to its partners while testing. But the composite web service is a system that is integrated at runtime and its result depends on its partners or the real environment. In this case, the passive testing method is used to verify the result of partner services or the interval time between a message request and a message response. The passive testing is a method that collects the observable traces of the system by installing a probe and analyzes its to give a verdict. This method does not effect to running system.

There are two approaches of passive testing, $online$ and $offline$. The online approach checks immediately an execution trace whenever an input/output event is happened. The advantage of this approach is: the faults may be found immediately and from that we can require stop the system to avoid the damage. On the contrary, the offline approach checks an execution trace after it is collected for a period of time. It means that an error is not found immediately if it happened. Depending on the concrete case, we can apply online approach or offline approach to verify the conformance of system.

This paper presents an approach for passive testing of behavioural conformance for a web service. We focus on black-box testing (in case of the service composition, we called gray-box testing because we know that interactions exist between a service and its partners). To passive testing, firstly we must define the constraints on the order of event and/or on data (called security rule). We can understand a security rule on a natural language as follow: if an event $E1$ has happened (may be including the constraints on data) then an event $E2$ (or a suite of event $SE2$) must be happened before/after $E1$ for a period of time. In this paper, we proposed use the Nomad language [11] which is more convenient in use than a generic temporal logic (like LTL) to define the security rules for web services. This language is ready used to define the security rules in the method of [9]. Secondly, we present an algorithm that can be use to check $online$ or $offline$, including future and past time, from a sequence of input/output event. An important of this method, the analyze is done on a event-by-event basic, without storing the execution trace. This algorithm is fully implemented in the RV4WS tool (Runtime Verification engine for Web Services). While this tool analyzes the execution trace, with each security rule, it presents a graphical statistic, with some fixed properties, that helps to tester easy assess about the service. Finally, we propose a mechanism to collect the observable execution trace from a service under test.

The rest of paper is organized as follows. Section II discusses about some previous works of passive testing for web services. Section III, we give the notation of security rule in the Nomad language. Section IV presents the detail of the our methodology and some tools support. Finally, Section V concludes the paper and presents future works.

II. Related works

The recent years, many the methods, the tools are proposed and developed for passive testing of a web service

(including a composite of web service) [5–9, 14]. These works focus on either checking on a trace file to give a verdict [9] or proposed a method for dynamic statistics [6, 8] of some properties of web services.

Dranidis et al [5] propose the utilization of Stream X-machines for constructing formal behavioural specifications of Web services. The authors present also a runtime monitoring and verification architecture and discuses 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 [6, 7] 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 behavioural properties of composition processes expressed in BPEL rather than on individual Web services.

Cavalli et al [9] propose a trace collection mechanism for SOA by integrating modules within BPEL engine and a tool [9, 10] that checks offline an execution trace. This approach uses also the Nomad language to define the security rule. As Baresi et al [6, 7], this approach proposes a trace collection mechanism that depends on BPEL engine. We cannot use it for a web service that is developed by another language, for example Java, C#, php.

In the works of Li et al [14, 15] 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.

III. Preliminaries

This section presents an overview of Nomad language and how to use it to define the security rule in our approach. We choice this language because it provides a way to describe permissions, prohibition that are granted (they are applied immediately) and obligations (needing a time duration to complete) related to non-atomic actions within contexts that takes time constraints. Moreover, its syntax and our natural language are quite near. In our approach, we consider an obligation rule as a permission rule because we are considering time constraints as time intervals (i.e. time min and time max).

A. Nomad syntax

We present only, in this section, the notions that are used in our approach.

Definition 1: (Atomic action): We define an atomic action as one of following actions: an input message, an output message. If A is an action that can be performed within a system S, then $not(A)$ (which means "the non occurrence of A") is an action.

*** Note**: Some constraints on message parameters value are also considered in the action syntax description:

$$Event(\{Par_0\ lop\ Val_0\}\ op\ ...\ op\ \{Par_n\ lop\ Val_n\})$$

Where:

- $Event$ represents an input/output message name.
- $Par\ (i \in \{1, ...n\})$ are the parameters. These parameters represent the relevant fields in the message.
- $Val\ (i \in \{1, ...n\})$ are the possible parameters values.
- $lop \in \{=, \neq, <, >, \leq, \geq\}$.
- We use the operators $op \in \{\wedge, \vee\}$ to combine together some constraints.

Definition 2: (Formula): If A is an action then $start(A)$ (A is being started), $done(A)$ (A has been finished) are formula.

- If α and β are formula then $\neg\alpha$, $(\alpha \wedge \beta)$,$(\alpha \vee \beta)$ are formula.
- If α is formula then $O^{d\in[m,n]}\alpha$ (α was true d units of time ago if $m > n$, α will be true d units of time if $m < n$) is a formula too, where m, n are two natural numbers.
- If α and γ are formula then $(\alpha|\gamma)$ is a formula whose semantics is: in the context γ, the formula α is true.

Definition 3: (Security rule): If α and β are formula then $\mathcal{R}(\alpha|\beta)$ is a security rule where $\mathcal{R} \in \{\mathcal{P}$: permission; $\mathcal{F}$: Forbidden;$\}$. The security $\mathcal{P}(\alpha|\beta)$ (resp. $\mathcal{F}$) means that it is permitted (resp. prohibited) to have α true when context β holds.

B. Examples of security rule

We introduce some examples of security rule that are defined to verify the web services.

Example 1: We only allow to create a new account on the services if we have login within maximal one day ago and do not logout.

$$\mathcal{P}(start(createAccountRequest)|O^{d\in[1,0]D}$$
$$done(loginResponse) \wedge \neg done(logoutRequest))$$

Example 2: In the case of web service composition, we can define a rule to verify the interval time (i.e. 10 seconds) between a request message and a response message from a partner service to assess about the successful rate if this partner is installed on a far host.

$$\mathcal{P}(start(msgRequest)|O^{d\in[0,10]S}done(msgResponse))$$

IV. Passive Testing methodology

A passive testing method composes three steps: 1) define the passive testing architecture to collect the execution traces on a running system. 2) define the security rules. We use the Nomad language that is described in the section III to do

this. 3) analyze (online or offline) the execution traces to give the verdict. This verdict is $PASS$ if the system trace respects the specified security policy and $FAIL$ if it does not. The $INCONCLUSIVE$ verdict is possible in case of offline checking and the tester cannot extract the necessary information if the execution trace is short.

A. Trace collection

Many trace collection architectures are presented by the previous works [5, 6, 9]. In these architectures, a probe, that is used to collect the execution trace, is normally integrated either at the consumer's site or the provider's site. In this section, we introduce two architectures for the trace collection based on the notion Point Observation (PO), one for web service based and another for web service composition. Our PO will collect the execution trace at network level. That allows us to collect the trace without depend on the SOAP API. Our trace collection architectures are shown in figure 1. With a web service, we set a PO between the client and the service to collect the SOAP messages. With a web service composition, we can also test the communication between a web service and its partners. So that, to collect all input/output messages of a web service composition, each connection between the service and its partner will be setted a PO. All messages that are collected by the POs will be sent to a checking engine to analyze and give the verdict.

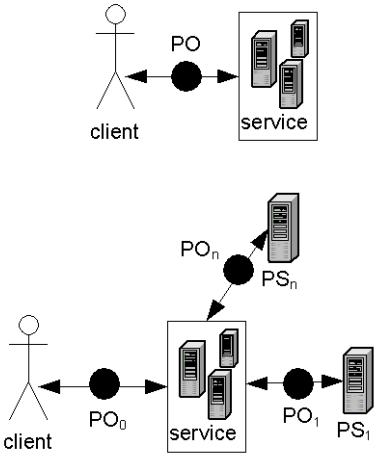

Figure 1. Trace collection architecture for service based (top) and for a composite of service (bottom)

B. Checking algorithm

In this section, we briefly outline the computation mechanism used to determine whether a security rule holds for some given input/output sequence of events. Our algorithm determines event-by-event to conform with each security rule without storing the message sequence. Before introduce the detail of algorithm, we present some functions to compute on the context of each rule: 1) $update$: 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 $true \wedge \neg logoutRequest$. 2) $evaluate$: this function evaluates a context of rule is either holds (true) or not. At a time, this function returns one in three values: $true$, $false$ or $undefined$ if exists at least a message that is not updated. While the evaluation, a message with the function not will be assigned provisionally is $true$. For example: at a time of evaluation, the expression $true \wedge \neg logoutRequest$ will be evaluated $true \wedge true = true$. 3) $contain$: to find a message in the context of a rule. 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.

There is two types of rule: future time and past time. To easy understand, we will analyze the checking algorithm for each type.

1) 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) at 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. So that, we will firstly update the context of the current rules in this list ($currlist$). Secondly, we evaluate the context of each rule. If the context is $true$ and the time constraint is satisfying, a verdict $pass/fail$, depends on the permissions/prohibition of rule, will be given in time msg arrives, and remove this rule from current list ($currlist$). If we cannot evaluate the context, we will wait 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.

2) 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, the $evaluate$ function returns $true$ 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 condition of supposition part is valid with the data of msg. 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. else 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 $rulelist$ and enable it (add into $currlist$ to wait the message in the supposition part) if its context contains the message msg.

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

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, arrival time t.
Output : $true/false$

$res := true$;
$list := \emptyset$; //a list;

//step 1: check in currlist to give a verdict;
foreach $rule$ ***in*** $currlist$ **do**
  //if a rule is enabled many times, we consider only one time (i.e. one session);
  if $rule.id \notin list$ **then**
    if $rule$ *is future time* **then**
      $res := verify\_future(rule, msg, t, res)$;
    else
      $res := verify\_past(rule, msg, t, res)$;
    $list.add(rule.id)$;

//step 2: check in rulelist to enable new rule;
foreach $rule$ ***in*** $rulelist$ **do**
  if $msg \in rule.supposition \wedge rule.condition(msg) = true$ **then**
    if $rule$ *is future time* **then**
      $r1 := rule$; //create a new rule;
      $r1.active\_time := t$; // set active time;
      $cl.add(r1)$; //add into actived list;
    else if $rule.evaluate()! = true \wedge rule.id \notin list$ **then**
      $res := false$;
      $rule.fail++$;
  else if $rule$ *is past time* $\wedge\ rule.id \notin list \wedge rule.context.contain(msg)$ **then**
    $r1 := rule$; //create a new rule;
    $r1.active\_time := t$; // set active time;
    $r1.update(msg)$ //update context;
    $cl.add(r1)$; //add into actived list;
return res;

***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.

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

Require: $currlist$: is a global variable

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

if $t - rule.active\_time > rule.time\_max \wedge rule.type =' P'$ **then**
  $rule.fail++$;
  $result := false$;
  $currlist.remove(rule)$;
else if $r.context.contain(msg)$ **then**
  $rule.update(msg)$ //update context;
  if $rule.evaluate() = true$ **then**
    $currlist.remove(rule)$;
    if $rule.type =' F' \wedge t - rule.active\_time \in [rule.time\_min, rule.time\_max]$ **then**
      $result := false$;
      $rule.fail++$;
  else if $rule.evaluate() = false$ **then**
    $currlist.remove(rule)$;
    if $rule.type =' P'$ **then**
      $result := false$;
      $rule.fail++$;
return $result$;

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

Require: $currlist$: is a global variable

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

if $msg \in rule.supposition \wedge rule.condition(msg) = true$ **then**
  $currlist.remove(rule)$;
  if $rule.evaluate() = true$ **then**
    if $rule.type =' F' \wedge t - rule.active\_time \in [rule.time\_min, rule.time\_max]$ **then**
      $result := false$;
      $rule.fail++$;
  else
    if $rule.type =' P'$ **then**
      $result := false$;
      $rule.fail++$;
else
  if $t - rule.active\_time > rule.time\_max$ **then**
    $currlist.remove(rule)$;
  else if $rule.context.contain(msg)$ **then**
    $rule.update(msg)$;
return $result$;

C. Tool support

To support for our approach, we have developed two software tools. One allows us to verify either online or offline a trace execution, called RV4WS (Runtime Verification engine for Web Services). Another, called SOAP_Sniffer, that allows us to capture all input/output SOAP messages of a web service (including a service composition).

1) RV4WS tool: RV4WS is a tool implemented as a solution to demonstrate these theories presented in this paper. The detail of architecture is shown in the figure 2.

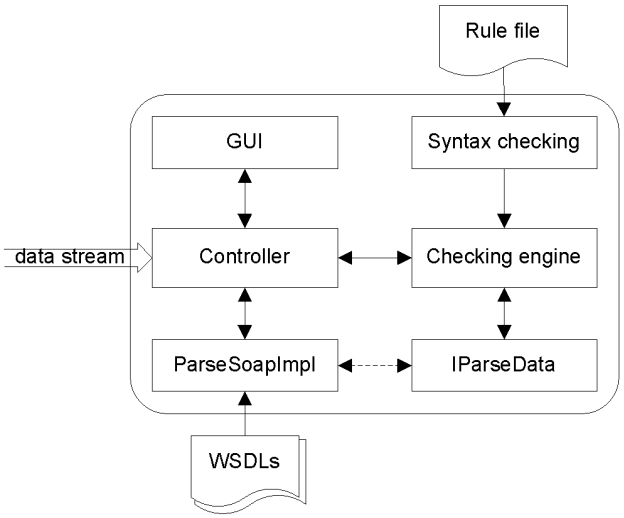

Figure 2. RV4WS architecture

One of the most interested components in this architecture is checking engine component that implemented the runtime verification algorithm 1. The engine allows us to verify each of 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 needing any storage in memory. To use this engine for the other systems, because of the difference between the systems is the data structure of input/output messages, so we define an interface (i.e., $IParseData$, shown in the figure 3) 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 controlled by a component called Controller which received a data stream coming from TCP port (online) or from log file (offline).

The input format for this tool is a xml file that has been

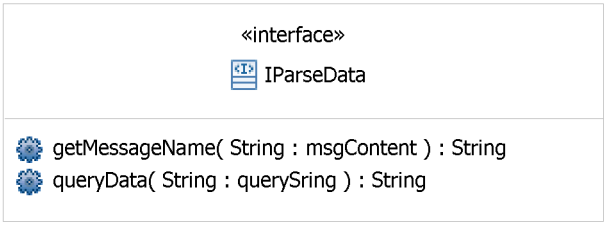

Figure 3. ParseData Interface

defined as in the figure 4. A rule with verdict $true$ represents a permission and a verdict $false$ represents a prohibition. A context of rule will be expressed as an expression with two operators AND and OR.

```
<rule id="0" name="" verdict="true">
   <if>
      <message>createAccountRequest</message>
      <!-- The condition field is optional -->
      <condition></condition>
   </if>
   <!--time type: s=second; m=minute; h=hour; d=day-->
   <then type="before" time_min="0" time_max="1"
                                        time_type="d">
      <context>
         <expression>loginResponse &amp;
                        !logoutRequest</expression>
      </context>
   </then>
</rule>
```

Figure 4. An example of rule define for RV4WS

To support the visualization of testing results, we have also presented a Graphic User Interface (GUI) that used to visualize some statistical properties calculated at any moment of testing process. Whenever a rule is activated, means that its conditions have been satisfied, a statistical property 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 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 that its fail percentage in proportion to its duration time or to others properties. If we had used a histogram view applied for each, we would not have been able to get these informations cause of the difference scales of these properties. We built a visual interface which based on the idea of parallel coordinates scheme, introduced by Inselberg [17]. In information visualization, parallel coordinates view is used to show the relationships between items in a multidimensional dataset. Each of 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

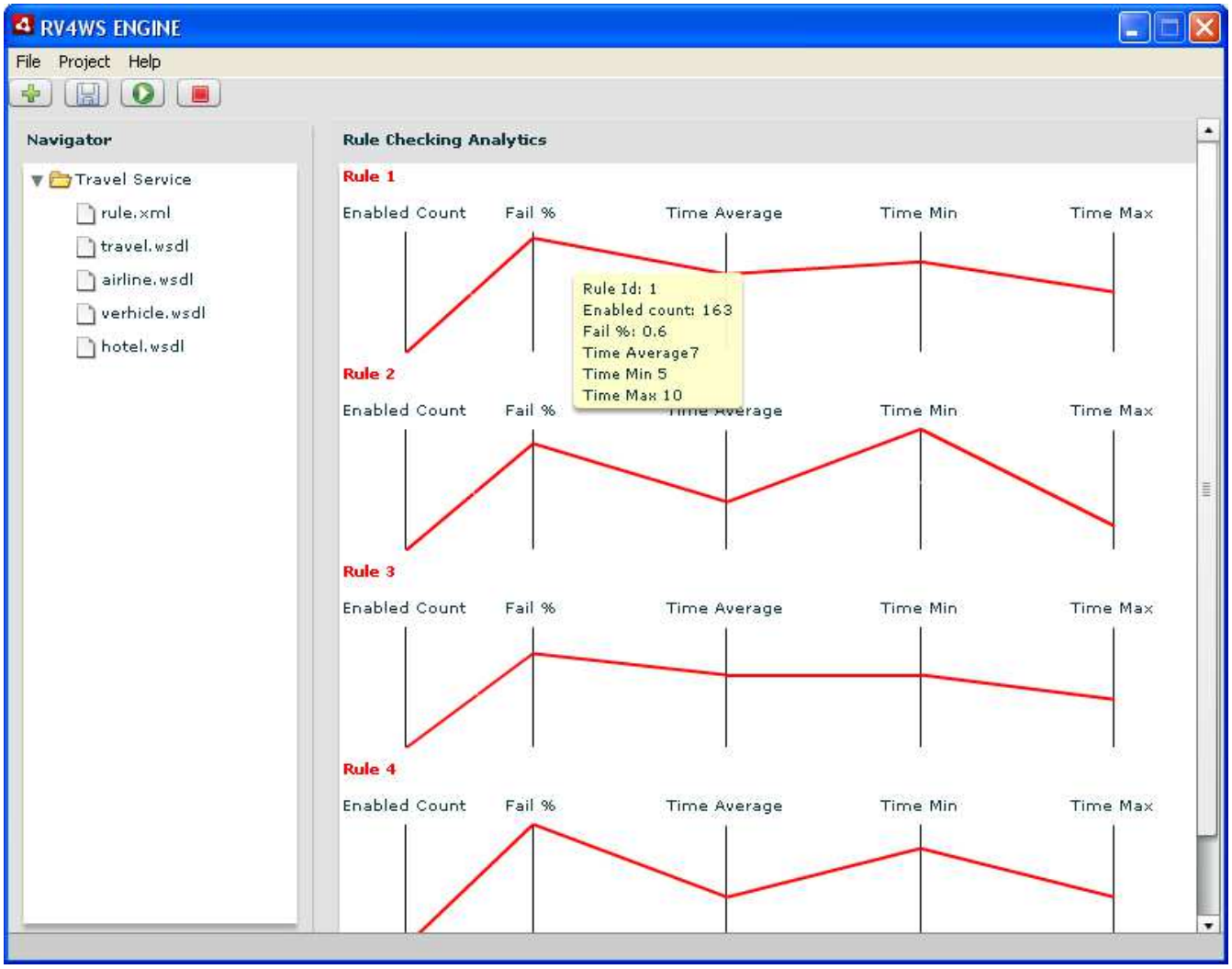

Figure 5. Test result

testing process as a multivariate/multi dimension data, we have applied this visualization to RV4WS tool and make 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 on these traces of testing process. There are the parallel coordinates views correspondent to rules. In the figure 5, 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 each 10 seconds. It means that it does not run in real-time.

2) SOAP_Sniffer tool: To collect the input/output SOAP messages of a service under test, we are developing a tool, called SOAP_Sniffer. We use the *pcap* [16] library to capture all packages that pass over a network card and filter its by applying the port number, source IP, destination IP, the transition protocol (i.e. TCP/IP) etc. Next, we analyse the package content to filter the packages that are transmitted by HTTP/XML protocol. Finally, these messages and the informations: source IP, source port, destination IP, destination port will be sent to the RV4WS tool by TCP/IP protocol to identify its name and verify it. Because a web service composition will send/receive the SOAP message to/from many its partner. To identify these messages are sent and receive to/from which partner, we need the informations as source IP, source port, destination IP, destination port. On the other hand, these informations may be used in the

condition of rule. With a web service based, we can install this tool on either server side or client side to collect the trace execution. We will install this tool on the server to collect all input/output SOAP messages of a service composition.

V. Conclusions and Future works

There are two approaches to test a system: active testing which tester interacts directly to the implementation under test and checks the correction of the answers to give the verdict, and passive testing that tester collects trace execution and analyses it to give the verdict. This paper focus on the problem of passive testing of behavioural conformance for the Web services. We have proposed to use the Nomad language to define the security rules and an algorithm to verify the correction of a trace execution that is a sequence of input/output message. This algorithm has been fully implemented in the RV4WS tool for two approaches: online verification and offline verification. In particular, this tool also proposed a graphic statistical analyse that shows the current test status. To support the collection of trace execution, we present a trace collection architecture and implement a support tool, SOAP_Sniffer.

There is a limitation of the RV4WS. The current version is not support yet the data correlation between the condition of supposition part and the condition of context. In the future works, we will study to solve this problem and finish the SOAP_Sniffer tool.

Acknowledgment

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

References

[1] Web Services Description Language 1.1. http://www.w3.org/TR/wsdl. Last accessed on November 2, 2009.

[2] OASIS. Web Services Business Process Execution Language (BPEL) Version 2.0, April 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html. Last accessed on December 25, 2009.

[3] Active Endpoints. The Active-Bpel engine. http://www.activevos.com/community-open-source.php. Last accessed on November 2, 2009.

[4] 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.*

[5] 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.

[6] 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.

[7] 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, Madrid, Spain, 2008.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] Programming with pcap: http://www.tcpdump.org/pcap.htm. Last accessed on January 25, 2010.

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