Analysis of Composite Web Services using Logging Facilities

Mohsen Rouached, Claude Godart

Analysis of Composite Web Services using Logging Facilities

Mohsen Rouached and Claude Godart

LORIA-INRIA-UMR 7503
BP 239, F-54506 Vandœuvre-les-Nancy Cedex, France
{mohsen.rouached, claude.godart}@loria.fr

Abstract. Web services are becoming more and more complex, involving numerous interacting business objects within considerable processes. In order to fully explore Web service business opportunities while ensuring a correct and reliable modelling and execution, analyzing and tracking Web services interactions will enable them to be well understood and controlled. Then, given the resulting event log we want to verify certain specified properties, to provide knowledge about the context of and the reasons for discrepancies between services'behaviours and related instances.
This paper advocates a novel technique to log composite Web services and a formal approach, based on an algeabric specification of the discrete event calculus language $\mathcal{DEC}$, to check behavioural properties of composite Web services regarding their execution log. An automated induction-based theorem prover SPIKE is used as verification back-end.

1 Introduction

Creating new services by combining a number of existing ones is becoming an attractive way of developing value added web services. This pattern is not new but it does pose some new challenges which have yet to be addressed by current technologies and tools for Web service composition.

In order to satisfy current users and to attract new customers, services providers need to pay special attention to the quality of their services. In particular, they need to trace executions of these services in order to ensure explainability in case of failure or auditing, as well as to support decision-making aimed at improving the structure and dynamics of the services. These traces of ongoing and past executions of services provide also the information required to detect services whose executions tend to fail, and to conduct routine or ad-hoc checks involving the executions of a service.

In the research related to Web services, several initiatives have been conducted with the intention to provide logging facilities. Despite all these efforts, the Web service logging activity is a highly complex task. The complexity, in general, comes from the following sources. First, the number of services available over the Web increases dramatically during the recent years, and one can expect to have a huge Web service repository to be searched. Second, Web services can be created and updated on the fly, thus the composition system needs to detect the updating at runtime and the decision should be made based on the up to date information.

To a service composer, it is desirable to be able to verify that the composition is well formed: for example that it does not contain any deadlocks or livelocks which would cause the composition to not terminate under certain conditions; and that the composition uses each web service *correctly*. It is possible to verify the former using formal notations and model checkers but for the latter it is necessary to precise what is meant by *correctly*. One aspect of using a web service correctly is invoking the operations in the order in which the provider intended. However, the WSDL description of a web service does not specify any ordering information for the operations which are exposed by the service. To allow a service composer to verify this aspect of correctness of the composition, we focus in this paper on defining ordering information about services' behaviours regrding the execution log. Indeed, we consider behavioural properties where ordering and timing are relevant and we check whether certain properties hold or not assuming that the information system at hand left a "footprint" in some event log. To do this, both behavioural properties and the log are expressed in a novel algeabric specification of $\mathcal{DEC}$. Then, an automated induction-based theorem prover SPIKE is used as verification back-end.

The remainder of the paper is structured as follows. Section 2 discusses existing Logging facilities for Web services and introduces our technique to collect composite Web services executions. Section 3 presents an algeabric specification of the $\mathcal{DEC}$ language. An illustrative example is used to illustrate our ideas. In Section 4, an overview of SPIKE is given. The encoding of $\mathcal{DEC}$ in SPIKE is explained in Section 5. Using this encoding, behavioural properties are checked in Section 6. The related work is discussed in section 7. Finally, Section 8 concludes the paper and outlines some future directions.

2 Web Service Logging

In this section we examine and formalize the logging possibilities in service-oriented architectures.Then, we introduce our technique to log Web services executions and more specifically the composite ones. The levels of logging vary in the richness of the information that is logged and in the additional development effort that is needed when implementing the respective features.

2.1 Web service collecting solutions and Web log structure

The first step in the Web service analysis process consists of gathering the relevant Web data, which will be analyzed to provide useful information about the Web Service behaviour. We discuss how these log records could be obtained by using existing tools or specifying additional solutions. Then, we show that the analysis process is tightly related to what of information provided in Web service log and depend strongly on its richness.

Existing logging solutions provide a set of tools to capture web services logs. These solutions remain quite "poor" to analyze advanced web service behaviours. That is why *advanced logging solutions* should propose a set of developed techniques that allows

us to record the needed information to analyze more advanced behaviour. This additional information is needed in order to be able to distinguish between Web services composition instances.

2.2 Existing logging solutions

There are two main sources of data for Web log collecting, corresponding to the interacting two software systems: data on the Web server side and data on the client side (see Figure 1). The existing techniques are commonly achieved by enabling the respective Web server's logging facilities. There already exist many investigations and proposals on Web server log and associated analysis techniques. Actually, papers on Web Usage Mining WUM [11] describe the most weel-known means of web log collection. Basically, server logs are either stored in the *Common Log Format* [1] or the more recent *Combined Log Format* [2]. They consist primarily of various types of logs generated by the Web server. Most of the Web servers support as a default option the *Common Log Format*, which is a fairly basic form of Web server logging. The log entry recorded in Apache Tomcat when a request is sent to a Web service *ExampleService* may look as follows:

```
127.0.0.1 - - [15/Mar/2005:19:50:13 +0100] "POST /axis/
services/ExampleService HTTP/1.0" 200 819 "-" "Axis/1.1"
```

The log entry contains the requestor's IP address, a timestamp, the request line, the HTTP code returned by the server, i.e., 200 for OK, the size of the returned resource, and the User-Agent, i.e., Axis/1.1. The empty element, i.e. "-", indicates that no referer-information is available. Such log records allow for tracking of the service consumer, determining which service is called how often (but not which operation of the service), or analyzing service failure rates.

Level	Logged information	Logging facility
(1) Standard HTTP-server logging	consumer'IP,invoked WS,timestamp,HTTP'status code	Web server
(2) Logging of complete HTTP requests and responses	(1)+ SOAP request and response,timestamps	HTTP listener and logger
(3) Logging at Web service container level	invoked WS and operation,SOAP request and response,timestamps	WS container, SOAP handlers
(4) Logging client activity	(3)+ consumer-side activity	WS container,SOAP handlers
(5) Providing for process information	(4)+ workflow information	(4)+ Web services

Fig. 1. Summary of logging features

However, the emerging paradigm of Web services requires richer information in order to fully capture business interactions and customer electronic behaviour in this new Web environment. Since the Web server log is derived from requests resulting from users accessing pages, it is not tailored to capture service composition or orchestration. That is why, we propose in the following a set of advanced logging techniques that allows to record the additional information to analyze more advanced behaviour.

[1] http://httpd.apache.org/docs/logs.html
[2] http://www.w3.org/TR/WD-logfile.html

2.3 Advanced logging solutions

Identifying Web service composition instance : Successful analysis for advanced architectures in Web services models requires composition (choreography/orchestration) information in the log record. Such information is not available in conventional Web server logs. Therefore, the advanced logging solutions must provide for both a choreography or orchestration identifier and a case identifier in each interaction that is logged.

A known method for debugging, is to insert logging statements into the source code of each service in order to call another service or component, responsible for logging. However, this solution has a main disadvantage: we do not have ownership over third parties code and we cannot guarantee they are willing to change it on someone else behalf. Furthermore, modifying existing applications may be time consuming and error prone.

Since all interactions between Web services happen through the exchange of SOAP message (over HTTP), an other alternative is to use SOAP headers that provides additional information on the message's content concerning **choreography**. Basically, we modify SOAP headers to include and gather the additional needed information capturing **choreography** details. Those data are stored in the special <`WSHeaders`>. This tag encapsulates headers attributes like: `choreographyprotocol`, `choreographyname`, `choreographycase` and any other tag inserted by the service to record optional information; for example, the <`soapenv:choreographyprotocol`> tag, may be used to register that the service was called by *WS-CDL* choreography protocol. The SOAP message header may look as shown in Figure 2. Then, we use SOAP intermediaries [2] which are an application located between a client and a service provider. These intermediaries are capable of both receiving and forwarding SOAP messages. They are located on web services provider and they intercept SOAP request messages from either a Web service sender or captures SOAP response messages from either a Web service provider. On Web service client-side, this remote agent can be implemented to intercept those messages and extract the needed information. The implementation of client-side data collection methods requires user cooperation, either in enabling the functionality of the remote agent, or to voluntarily use and process the modified SOAP headers but without changing the Web service implementation itself (the disadvantage of the previous solution).

Concerning **orchestration** log collecting, since the most Web services orchestration are using a WSBPEL engine, which coordinates the various orchestration's web services, interprets and executes the grammar describing the control logic, we can extend this engine with a sniffer that captures orchestration information, i.e., the orchestration-ID and its instance-ID. This solution provides is centralized, but less constrained than the previous one which collects choreography information.

Using these advanced logging facilities, we aim at taking into account Web services' neighbors in the analysis process. The term neighbors refers to other Web services that the examined Web service interacts with. The concerned levels deal with analyzing Web service choreography interface (abstract process) through which it communicates with others web services to accomplish a choreography, or discovering the set of interactions exchanged within the context of a given choreography or composition.

$< soapenv : Header >$
$< soapenv : choreographyprotocol$
$soapenv : mustUnderstand =$ "0"
$xsi : type =$ "$xsd : string$" $> WS - CDL$
$< /soapenv : choreographyprotocol >$
$< soapenv : choreographyname$
$soapenv : mustUnderstand =$ "0"
$xsi : type =$ "$xsd : string$" $> OTA$
$< /soapenv : choreographyname >$
$< soapenv : choreographycase$
$soapenv : mustUnderstand =$ "0"
$xsi : type =$ "$xsd : int$" > 123
$< /soapenv : choreographycase >$
$< /soapenv : Header >$

Fig. 2. The SOAP message header

Collecting Web service composition Instance : The focus in this section is on collecting and analysing **single** web service composition instance. The issue of identifying several instances has been discussed in the previous section. The exact structure of the web logs or the event collector depends on the web service execution engine that is used. In our experiments, where we have used the engine bpws4j [3] uses log4j[4] to generate logging events. Log4j is an OpenSource logging API developed under the Jakarta Apache project. It provides a robust, reliable, fully configurable, easily extendible, and easy to implement framework for logging Java applications for debugging and monitoring purposes. The event collector (which is implemented as a remote log4j server) sets some log4j properties of the bpws4j engine to specify level of event reporting (INFO, DEBUG etc.), and the destination details of the logged events. At runtime bpws4j generates events according to the log4j properties set by the event collector. More details about the implementation can not be presented here due to lack of space but can be found in [13].

3 Analyzing Web services'behaviours

In this section, we focus on the process of analyzing Web services'behaviours regarding their execution logs. Indeed, given an event log, we want to verify certain behavioural properties, to provide knowledge about the context of and the reasons for discrepancies between services 'behaviours and related instances.

3.1 Illustrative example

As an illustrative example, we consider a scenario of a device purchase order shown in Figure 3. This scenario models a 3-party composition, in which a supplier coordinates with its warehouse in order to sell and ship electronic devices.

[3] http://alphaworks.ibm.com/tech/bpws4j
[4] http://logging.apache.org/log4j

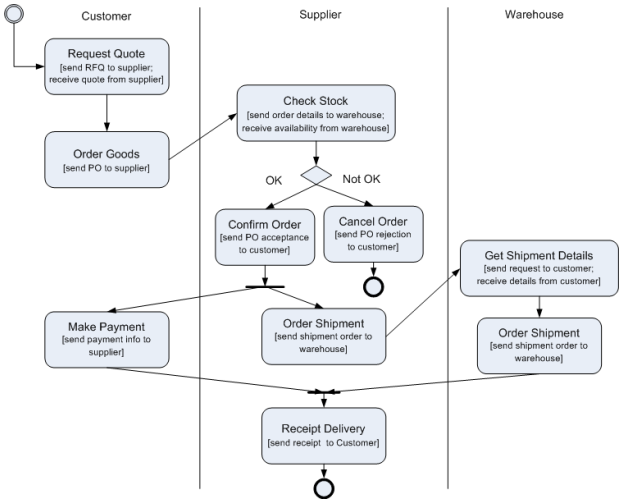

Fig. 3. Device purchase order

The interaction starts when a *Customer* communicates a purchase order to the *Supplier*. *Supplier* reacts to this request asking the *Warehouse* about the availability of the ordered item. Once received the response, *Supplier* decides to cancel or confirm the order, basing this choice upon Item's availability and *Customer*'s country. In the former case, the execution terminates, whereas in the latter one a concurrent phase is performed: *Customer* sends an order payment, while *Warehouse* handles the item's shipment. When both the payment and the shipment confirmation are received by *Supplier*, it delivers a final receipt to the *Customer*. The specification of this scenario is given as follows. The events are represented in the form $msgType(sender, receiver, content_1, ..., content_n)$, where the $msgType$, $sender$, $receiver$ and $content_i$ retain their intuitive meaning.

During the rest of the paper, we focus on a simple execution instance of the previously described example. In this instance, inspired by Disney characters, the criminal *bigTime* (BT) beagle wants to buy a device from the online shop *devOnline* (DO), whose warehouse is *devWare* (DW). Figure 4 contains the log of the scenario from the viewpoint of *devOnline*; note that messages are expressed in an high level way, abstracting from the SOAP exchange format using the technique introduced in Section 2.3.

In the device purchase scenario, we can distinguish several behavioural properties that should be respected. However, due to lack of space, we just mention the following ones. (BP1) specifies that, when *Customer* sends to *Supplier* the purchase order, including the requested *Item* and his/her *Country*, *Supplier* should request Item's availability to *Warehouse*. (BP2) indicates that *Warehouse* should respond within 6 minutes to *Supplier*'s request giving the corresponding quantity Qty. The deadline is a constraint over the variable $Tqty$, that represents the time which the response is sent to.

message	sender	receiver	content	t_s	t_e
1.purchase_order	BT	DO	$[dev]$	1	2
2.isAvailable	DO	DW	$[dev]$	3	9
3.inform	DW	DO	$[dev, 2]$	10	11
4.accept_order	DO	BT	$[dev]$	12	13
5.shipment_order	DO	DW	$[dev]$	14	15
6.confirm_shipment	DW	DO	$[dev]$	16	18
7.payment	BT	DO	$[dev]$	19	20
8.delivery	DO	BT	$[dev, r]$	21	22

Fig. 4. A fragment of SOAP messages exchanged in the device purchase order

3.2 Discrete Event Calculus: $\mathcal{DEC}$

Given the fact that we consider behavioural properties where ordering and timing are relevant and we adopt an event driven reasoning, the Event Calculus ($\mathcal{EC}$) [9] seems to be a solid basis to start from. $\mathcal{EC}$ is a temporal formalism based on a first order logic, that can be used to specify the *events* that appear within a system and the effect (or the *fluents*) of these events. It includes an explicit *time structure* that dates the system changes caused by the occurrence of the events.

For our purpose, we have used the discrete Event Calculus ($\mathcal{DEC}$) that is enough expressive to cope with the runtime analysis of composite Web services. $\mathcal{DEC}$ includes the predicates $Happens$, $Initiates$, $Terminates$ and $HoldsAt$, as well as some auxiliary predicates defined in terms of these. $Happens(a, t)$ indicates that event (or action) a actually occurs at time-point t. $Initiates(a, f, t)$ (resp. $Terminates(a, f, t)$) means that if event a were to occur at t it would cause fluent f to be $true$ (resp. $false$) immediately afterwards. $HoldsAt(f, t)$ indicates that fluent f is true at t. The auxiliary predicate $Clipped(t1, f, t2)$ expresses whether a fluent f was terminated during a time interval $[t1, t2]$. The following four axioms capture the behaviour of fluents once initiated or terminated by an event:

1. $Happens(a, t_1) \wedge (t_1 < t_2) \wedge Terminates(a, f, t_2) \rightarrow Clipped(t_1, f, t_2)$
2. $Happens(a, t_1) \wedge (t_1 < t_2) \wedge Initiates(a, f, t_2) \rightarrow \neg Clipped(t_1, f, t_2)$
3. $Happens(a, t_1) \wedge (t_1 < t_2) \wedge \neg Clipped(t_1, f, t_2) \rightarrow HoldsAt(f, t_2)$
4. $Happens(a, t_1) \wedge (t_1 < t_2) \wedge Clipped(t_1, f, t_2) \rightarrow \neg HoldsAt(f, t_2)$

Thus, the event log fragment depicted in Figure 4 can be easily translated in $\mathcal{DEC}$ formalism as follows:

$$L1 : Happens(purchase\_order(BT, DO, dev, country), 2)$$
$$L2 : Happens(isAvailable(DO, DW, dev), 3)$$
$$L3 : Happens(inform(DW, DO, dev, 3), 10)$$
$$L4 : Happens(accept\_order(DO, BT, dev), 12)$$
$$L5 : Happens(shipment\_order(DO, DW, dev, BT), 13)$$
$$L6 : Happens(confirm\_shipment(DW, DO, dev), 16)$$
$$L7 : Happens(payment(BT, DO, dev), 19)$$
$$L8 : Happens(delivery(DO, BT, dev, rec), 21)$$

In the same way, the behavioural properties can be expressed formally. For instance, the properties introduced in Section 3.1 are described in $\mathcal{DEC}$ formalism as follows:

$$(BP1) : Happens(purchase\_order(cu, s, i), T_{po}) \wedge Happens(isAvailable(s, w, i), T_{ca})) \implies T_{po} < T_{ca}$$

$$(BP2) : Happens(isAvailable(s, w, i), T_{ca}) \wedge Happens(inform(w, s, i, Qty), T_{qty})) \implies T_{qty} < T_{ca} + 6$$

4 Overview of SPIKE

Theorem provers have been applied to the formal development of software. They are based on logic-based specification languages and they provide support to the proof of correctness properties, expressed as logical formulas. In this work, we use the SPIKE induction prover [3]. SPIKE was chosen for the following reasons: (i) its high automation degree (to help a Web service designer), (ii) its ability on case analysis (to deal with multiple operations and many case of transformations), (iii) its *refutational completeness* (to find counter-examples), (iv) its incorporation of *decision procedures* (to automatically eliminate arithmetic tautologies produced during the proof attempt[5]). SPIKE proof method is based on cover set induction. Given a theory, SPIKE computes in a first step induction variables where to apply induction and induction terms which basically represent all possibles values that can be taken by the induction variables. Typically for a nonnegative integer variable, the induction terms are 0 and $x + 1$, where x is a variable.

Given a conjecture to be checked, the prover selects induction variables according to the previous computation step, and substitute them in all possible way by induction terms. This operation generates several instances of the conjecture which are then *simplified* by rules, lemmas, and induction hypotheses.

5 Encoding $\mathcal{DEC}$ in SPIKE

In this section, we describe a method for representing $\mathcal{DEC}$ in SPIKE language. In the sequel, we assume that all formulas are universally quantified.

5.1 Ingredients of our encoding

Data. All data information manipulated by the system is ranged over a set of sorts. This data concerns generally the argument types of events and fluents. For instance, the sets of customers, suppliers, items and countries are defined respectively by the sorts $Customer$, $Supplier$, $Item$ and $Country$. The sort $Bool$ represents the boolean values, where $true$ and $false$ are its constant constructors.

[5] like $x + z > y = false \wedge z + x < y = false \implies x + z = y$

Events. We consider that all events of the system are of sort $Event$, where the event symbols are the constructors of this sort. These constructors are free as all event symbols are assumed distincts. For instance, the event symbol $purchase\_order(x, y, z, t)$ is a constructor of $Event$ such that x, y, z and t are variables of sorts $Customer$, $Supplier$, $Item$ and $Country$ respectively. We define also an *idle* event which when occuring it lets the system unchanged. We represent it by the constant constructor $Noact$.
Fluents. The sort $Fluent$ respresents the set of fluents. All fluent symbols of the systems are the constructors of sort $Fluent$, that are also free. The fluent symbol $EqualItem(x, y)$, for example, means that the variables x and y, of sort $Item$, are equal.
Time. We use the sort of natural numbers, Nat, which is reflected by constructors 0 and successor $succ(x)$ (meaning $x + 1$). We have modified the code of SPIKE in order to enable handling of Peano numbers. For example, now we can directly write 17 instead of $s(s(...(0)...))))$ as it was in the previous versions of SPIKE .
Axioms. We express all predicates used in $\mathcal{DEC}$ as boolean function symbols. The signatures of these function symbols and others additional functions are as follows:

$Happens : Event \times Nat \rightarrow Bool$
$Initiates : Event \times Fluent \times Nat \rightarrow Bool$
$Terminates : Event \times Fluent \times Nat \rightarrow Bool$
$HoldsAt : Fluent \times Nat \times Nat \rightarrow Bool$
$Clipped : Fluent \times Nat \times Nat \rightarrow Bool$
$p : Event \times nat \rightarrow EventTime$
$Cons : EventTime \times List \rightarrow List$
$member : EventTime \times List \rightarrow bool$
$Happens : EventTime \rightarrow Bool$

$HoldsAt$ and $Clipped$ are defined within a time range. For instance, $HoldsAt(f, t_1, n)$ is defined within the range $[t_1, t_1 + n]$. In addition, we define the functions symbols p, $Cons$, and $member$. p is a constructor that associates an event to its occurrence time. $Cons$ is used to group the list of events in the constant $ListEvent$. This provide a certain flexibility in the construction of the log. Then, $member$ is a boolean function that permits to test if an event appears in the log. After defining the p constructor, the signature of the function associated to the predicate $Happens$ is changed from $Happens : Event \times Nat \rightarrow Bool$ to $Happens : EventTime \rightarrow Bool$.

Finally, the four axioms given in Section 3.2 are expressed in conditional equations as follows:

(A1) $event \neq Noact \wedge Happens(p(event, t_1)) = true \wedge Initiates(event, f, t_1) = true \Rightarrow HoldsAt(f, t_1, 0) = true$
(A2) $HoldsAt(f, t_1, t) = true \wedge Clipped(f, t_1 + t, s(0)) = false \Rightarrow HoldsAt(f, t_1, s(t)) = true$
(A3) $event \neq Noact \wedge Happens(p(event, t_1)) = true \wedge Terminates(event, f, t_1) = true \Rightarrow Clipped(f, t_1, s(0)) = true$
(A4) $event \neq Noact \wedge Happens(p(event, t_1 + t + s(0))) = true \wedge Terminates(event, f, t_1 + t + s(0)) = true \Rightarrow Clipped(f, t_1, s(s(t))) = true$

(A5) $Happens(p(Noact, t_1 + t + s(0))) = true \implies Clipped(f, t_1, s(s(t))) = Clipped(f, t_1, t + s(0))$
(A6) $Happens(x) = member(x, ListEvent)$
(A7) $member(x, Nil) = false$
(A8) $x = y \Rightarrow member(x, Cons(y, l)) = true$
(A9) $x \neq y \Rightarrow member(x, Cons(y, l)) = member(x, l)$

Log. Using the function $Cons$ we define the log in equational form:

$$
\begin{array}{l}
ListEvent = Cons(p(purchase\_order(BT, DO, dev, country), 2), \\
Cons(p(isAvailable(DO, DW, dev), 3), \\
Cons(p(inform(DW, DO, dev, 3), 10), \\
Cons(p(accept\_order(DO, BT, dev), 12), \\
Cons(p(shipment\_order(DO, DW, dev, BT), 13), \\
Cons(p(confirm\_shipment(DW, DO, dev), 16), \\
Cons(p(payment(BT, DO, dev), 19), \\
Cons(p(delivery(DO, BT, dev, rec), 21), Nil)))))))
\end{array}
$$

Behavioural properties. In the same way, we can express the bahavioural properties in equational form. For instance, the properties ($BP1$) and ($BP2$)given in Section 3.2, are written as follows:

$$
\begin{array}{l}
(BP1) : Happens(purchase\_order(x, y, i), t_1) = true \wedge \\
\qquad Happens(isAvailable(y, w, i), t_2) = true \\
\qquad \implies (t_1 < t_2) = true
\end{array}
$$

$$
\begin{array}{l}
(BP2) : Happens(isAvailable(s, w, i), t_1) = true \wedge \\
\qquad Happens(inform(w, s, i, q), t_2) = true \\
\qquad \implies (t_2 < t_1 + 6) = true
\end{array}
$$

where t_1, t_2, x, y, w, q and i are variables.

5.2 Methodology

In the following, we propose a methodology for reasoning about composite Web services specifications with deductive methods based on induction and rewriting. In the first step, we build an algebraic specification from $\mathcal{DEC}$ specification. The sorts, the signatures of the functions and the axioms (conditional equations) of the algebraic specification are obtained by

1. associating to each argument of events and fluents a sort;
2. definig events and fluents as constructors of sorts $Event$ and $Fluent$ respectively;
3. encoding the log in a set of conditional equations (as in the above section);
4. adding the axioms given in Section 5.1.

Only the first three steps are to be defined for every composite Web serivces. Once building the algebraic specification, we can check all behevioural properties by means the powerful deductive techniques (rewriting and induction) provided by SPIKE .

6 Checking Behavioural Properties

All the generated axioms can be directly given to the prover SPIKE , which automatically orientes these axioms into conditional rewrite rules. When SPIKE is called, either the behavioural properties proof succeed, or the SPIKE 's proof-trace is used for extracting all scenarios which may lead to potential deviations. There are two possible scenarios. The first scenario is meaningless because conjectures are valid but it comes from a failed proof attempt by SPIKE . Such cases can be overcome by simply introducing new lemmas. The second one concerns cases corresponding to real deviations. The trace of SPIKE gives all necessary informations (events, fluents and timepoints) to understand the inconsistency origin. Consequently, these informations help designer to detect behavioural problems in the composite Web service.

Let consider the example illustrated in this paper. SPIKE has found that ($BP1$) is true and the reader can be confirm that by analyzing the log. But when submitting ($BP2$), SPIKE has discovered an error. In the following, we describe how the prover checks the behavioural property ($BP2$).

Firstly, SPIKE simplifies ($BP2$) using the axioms (A6) and (A7) introduced in Section 5.1:

$$p(isAvailable(s, w, i), t_1) \wedge \quad (1)$$

$$p(inform(w, s, i, q), t_2) \quad (2)$$

$$\Longrightarrow\ (t_2 < t_1 + 6) = true \quad (3)$$

Using the literals $L2$ and $L3$ given by the log (that replace t_1 and t_2 by 3 and 10 respectively), this conjecture becomes $(10 < 3 + 6) = true$ that is always false. Consequently, the prover has detected an anomaly in the log. Below, we present a fragment of the SPIKE trace when checking property $BP2$.

```
Uncaught exception: Failure("fail induction on [ 10973 ] inform (u2, u1, u3, u5)
 <> purchase_order (e1, e2, e3, e4) /\\ inform (u2, u1, u3, u5) <> isAvailable (
e2, e5, e3) /\\ u2 = e5 /\\ u1 = e2 /\\ u3 = e3 /\\ u5 = 3 /\\ u6 = 10 /\\ isAva
ilable (u1, u2, u3) <> purchase_order (e1, e2, e3, e4) /\\ u1 = e2 /\\ u2 = e5 /
\\ u3 = e3 /\\ u4 = 3 => u6 < (u4 + (6)) = true ;")

while proving the following initial conjectures
[ 6584 ] Happens (p (isAvailable (u1, u2, u3), u4)) = true /\ Happens (p (inform
 (u2, u1, u3, u5), u6)) = true => u6 < (u4 + (6)) = true ;
Elapsed time: 0.186 s

We failed
```

7 Related Work

Up to now, few works have been conducted on logging and analyzing Web services usage information. Akkiraju et al. [1] proposed a framework blending logging facilities to private Web service registries. However, no details are provided about the log structure or how to implement it. Irani [7] proposed the use of intermediaries to collect information about authentication, auditing and management of services through the use of logs, but he also does not provide any detail on the log structure. Brittenham et al. [4],

from WS-I Test Tools Working Group, proposed an architecture that consists of a message monitor and an analyzer. The monitor is used to log the messages that were sent to and from a Web service, while the analyzer is used to validate that the Web service interactions contained in the message log conform to a WS-I profile. However, WS-I monitor captures in a single log file HTTP data and the whole SOAP message content. These data are captured in their raw format making it difficult to differentiate analytical information from disposable data. Capturing the whole SOAP message brings another problem: huge amount of data, many times larger than traditional HTTP logs.

Formal analysis and verification of Web Services in the aim of detecting anomalies are addressed in several papers. The SPIN model-checker is used for verification [10] by translating Web Services Flow Language (WSFL) descriptions into Promela. [8] uses a process algebra to derive a structural operational semantics of BPEL as a formal basis for verifying properties of the specification. In [5], BPEL processes are translated to Finite State Process (FSP) models and compiled into a Labeled Transition System (LTS) in inferring the correctness of the Web service compositions which are specified using message sequence charts. In [6], finite automata were augmented with (i) XML messages and (ii) XPath expressions as the basis for verifying temporal properties of the conversations of composite Web services. Rao et al. [12] introduces a method for automatic composition of semantic Web services using Linear Logic theorem proving. Services are presented by extralogical axioms and proofs in Linear Logic. For a complete overview of use of event calculus in the context of Web services we refer the reader to our previous work [13, 14].

8 Conclusions

This paper has outlined a technique to log composite Web services and a methodology, using the logging facilities, to analyze services' behaviours. More specifically, it permits to check whether the observed behaviour of each involved service matches the (un)expected/(un)desirable behaviour. The methodology is also supported by a formal representation of behavioural properties and execution logs considered as the basis for the automatic composition of Web services. The analysis process was supported by a novel specification of the $\mathcal{DEC}$ formalism. As verification back-end we used an automated induction-based theorem prover SPIKE that provide support to the proof of correctness properties, expressed as logical formulas.

The framework is still under development. Ongoing work on it is concerned with (1) to test our approach on larger and more complex specifications, for example of composition with more several involved services and complicated properties, and (3) to automatically capture and analyse the SPIKE execution traces in order to construct scenarios useful for ensuring re-engineering activities.

References

1. R. Akkiraju, D. Flaxer, H. Chang, T. Chao, L. Zhang, F. Wu, and J. Jeng. A framework for enabling dynamic e-business via web service. In *Proceedings of the OOPSLA*. Florida, USA, 2001.

2. M. Baglioni, U. Ferrara, A. Romei, S. Ruggieri, and F. Turini. Use soap-based intermediaries to build chains of web service functionality, 2002.
3. A. Bouhoula, E. Kounalis, and M. Rusinowitch. Spike: an automatic theorem prover. In *Proceedings of LPAR 92, number 624 in LNAI*. Springer-Verlag, July 1992.
4. P. Brittenham, J. Clune, J. Durand, L. Kleijkers, K. Sankar, S. Seely, K. Stobie, and G. Turrell. Ws-i analyzer tool functional specification.
5. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility verification for web service choreography. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 738, Washington, DC, USA, 2004. IEEE Computer Society.
6. X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM Press.
7. R. Irani. Web services intermediaries adding value to web services, November 2001.
8. M. Koshina and F. van Breugel. Verification of business processes for web services. Technical report, New York University, SFUCMPT-TR-2003-06.
9. R. Kowalski and M. J. Sergot. A logic-based calculus of events. *New generation Computing 4(1)*, pages 67–95, 1986.
10. S. Nakajima. Verification of web service flows with model-checking techniques. In *CW*, pages 378–385, 2002.
11. J. Punin, M. Krishnamoorthy, and M. Zaki. Web usage mining: Languages and algorithms. In *Studies in Classification, Data Analysis, and Knowledge Organization*. Springer-Verlag, 2001.
12. J. Rao, P. Küngas, and M. Matskin. Logic-based web services composition: From service description to process model. In *ICWS*, pages 446–453, 2004.
13. M. Rouached, W. Gaaloul, W. M. P. van der Aalst, S. Bhiri, and C. Godart. Web service mining and verification of properties: An approach based on event calculus. In *Proceedings 14th International Conference on Cooperative Information Systems (CoopIS 2006)*, November 2006. To appear.
14. M. Rouached, O. Perrin, and C. Godart. Towards formal verification of web service composition. In *Forth International Conference on Business Process Management (BPM06)*, 2006.