Towards Formal Verification of Web Service Composition

Mohsen Rouached, Olivier Perrin, and Claude Godart

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

Abstract. Web services composition is an emerging paradigm for enabling application integration within and across organizational boundaries. Current Web services composition proposals, such as BPML, WSBPEL, WSCI, and OWL-S, provide solutions for describing the control and data flows in Web service composition. However, such proposals remain at the descriptive level, without providing any kind of mechanisms or tool support for analysis and verification. Therefore, there is a growing interest for the verification techniques which enable designers to test and repair design errors even before actual running of the service, or allow designers to detect erroneous properties and formally verify whether the service process design does have certain desired properties.

In this paper, we propose to verify Web services composition using an event driven approach. We assume Web services that are coordinated by a composition process expressed in WSBPEL and we use Event Calculus to specify the properties and requirements to be monitored.

1 Introduction

In 2001, Gartner defined Business Process Management as *a general term describing a set of services and tools that provide for explicit process management (e.g. process modeling, analysis, simulation, execution, monitoring and administration), ideally including support for human and application-level interaction.* Five years later, Service Oriented Architectures (SOA) seems to be a key architecture to support BPM. With SOA, an application can be now considered as a composition of services, Workflow Management Systems (WfMSs), or legacy applications. Thus, a business process becomes a set of composed services that are shared across business units, organizations, or outsourced to partners.

Currently many products that offer modeling, analysis, and simulation facilities for business processes exist. However, one of the great advantages offered by the coupling of BPM and SOA is that designers can not only model, analyze, simulate, but they can also use the result directly for deployment, using WSBPEL for instance. Functions at the modeling layer can be linked to required services at the architecture level, and engines can now manage the resulting business process. This is a great improvement, and it clearly shows that BPM over SOA can add value over traditional WfMSs for instance. However, there are many challenges for trully realizing BPM over SOA.

S. Dustdar, J.L. Fiadeiro, and A. Sheth (Eds.): BPM 2006, LNCS 4102, pp. 257–273, 2006.

A first challenge deals with the ability to offer self-management of the designed processes [18]. This is an important topic since these processes are quite complex and dynamic, and deviations from the expected behavior may be highly desirable. In fact, one may want to adapt the process due to changes in the way the process is actually used, as it sometimes exists a gap between the designed process and the observed behavior. Then, once a deviation is found, it is important to dynamically adapt either the process, either the services that render the functions of the process. For that, it is important to collect information about business process activities, and to modify the process (at the design layer) and/or the services used by the process (at the execution layer). A second challenge is the need for checking the consistency of the process. This can be done either statically, i.e. at design time, or dynamically, i.e. at runtime. For the static part of the work, we should be able to express the business process using a formalism on which we can reason on. As business processes are quite huge and complex, proving the correctness of the composed business process is not an easy task, and it is hard to find their potential bottlenecks: livelocks, deadlocks, unused activities, inaccurate activities, inaccurate flows, inaccurate wiring between functions in the model and services in the SOA, etc. For the dynamic part of the verification, the business process should be auditable. For that, we can use process mining techniques, because processes (and their associated services) leave many traces of their behavior in the underlying systems they used to be executed. In our approach, we use mining techniques not for discovery but for dynamic verification of the execution of the process, i.e. requirements associated with the process. The verification deals with two kind of requirements: the functional requirements, and the non-functional requirements, such as security for instance.

In this paper, we propose an event-based approach for checking consistency of a business process, for mining the business process events, and for analyzing the process execution. It appears that using events is very attractive when compared to other approaches, as stated in [18]. Main advantages are: (i) business processes leave their business events in so-called event logs,(ii) it exists various works for checking event-based specifications consistency. Our proposition provides a formal framework for modeling and checking the consistency of WSBPEL compositions. We use the Event Calculus (EC) of Kowalski and Sergot [7], and an extension proposed by Mueller on Discrete EC [12]. Compared to other works, the choice of EC is motivated by both practical and formal needs, and it gives three major advantages. First, in contrast to pure state-transition representations, the EC ontology includes an explicit time structure that is independent of any (sequence of) events under consideration. This helps for managing event-based systems where a number of input events may occur simultaneously (risk of non-deterministic behavior [11]). Second, the EC ontology is close enough to the WSBPEL specification to allow it to be mapped automatically into the logical representation. Thus, we use the same logical foundation for verification at both design time (static analysis) and runtime (dynamic analysis). Third, the semantics of non-functional requirements can be represented in EC, so that verification is once again straightforward.

The paper is structured as follows. Section 2 introduces a scenario used to illustrate our approach. Section 3 rapidly presents WSBPEL and EC, and describes how to transform WSBPEL into EC. Section 4 studies the EC checking and indicates how the proposed formalism can verify and detect some examples of inconsistencies that may arise in the running scenario. The related work is discussed in Section 5, and Section 6 concludes and outlines future directions.

2 Case Study

Throughout this article, we will illustrate our ideas using a running example of Web services composition. We consider a car rental scenario that involves four services. A Car Broker Service (CBS) acts as a broker offering its customers the ability to rent cars provided by different car rental companies directly from car parks at different locations. CBS is implemented as a service composition process which interacts with Car Information Services (CIS), and Customer Management Service (CMS). CIS services are provided by different car rental companies and maintain databases of cars, check their availability and allocate cars to customers as requested by CBS. CMS maintains the database of the customers and authenticates customers as requested by CBS. Each Car Park (CP) also provides a Car Sensor Service (CSS) that senses cars as they are driven in or out of car parks and inform CBS accordingly. The end users can access CBS through a User Interaction Service (UIS). Typically, CBS receives car rental requests from UIS services, authorizes customers contacting CMS and checks for the availability of cars by contacting CIS services, and gets car movement information from CSS services. However, due to the autonomous nature of services and the run-time monitoring of requirements, many complications may arise. For example, CBS can accept a car rental request and allocate a specific car to it if, due to the malfunctioning of a CSS service, the departure of the relevant car from a car park has not been reported and, as a consequence, the car is considered to be available by the UIS service. Through this example, we aim to demonstrate how Web services interactions can be specified and formalized using events, and how this specification could facilitate their monitoring at run-time.

3 Transforming BPEL into Event Calculus

3.1 Overview of BPEL

WSBPEL [1] introduces a stateful model of Web services interacting by exchanging sequences of messages between business partners. The major parts of a BPEL process definition consist of (1) partners of the business process (Web services that the process interacts with), (2) a set of variables that keep the state of the process, and (3) an activity defining the logic behind the interactions between the process and its partners. Activities that can be performed are categorized into *basic*, *structured*, and *scope-related* activities. Basic activities

perform simple operations like receive, reply, invoke and others. Structured activities impose an execution order on a collection of activities and can be nested. Then, scope-related activities enable defining logical units of work and delineating the reversible behavior of each unit. Below, we describe the main activities (basic and structured).

Basic Activities. Basic activities in a WSBPEL process support primitive functions (e.g. invocation of operations and assignments of variable values):

(i) the *invoke* activity calls an operation in one of the partner services of the composition process.
(ii) the *receive* activity makes the composition process to wait for the receipt of an invocation of its operations by some of its partner services.
(iii) the *reply* activity makes the composition process to respond to a request for the execution of an operation previously accepted through a receive activity.
(iv) the *assign* activity is used to copy the value from a variable to another one.
(v) the *throw* activity is used to signal an internal fault.
(vi) the *wait* activity is used to specify a delay in the process that must last for a certain period of time.

Structured Activities. Structured activities provide the control and data flow structures that enable the composition of basic activities into a business process:

(i) the *sequence* activity includes an ordered list of other activities that must be performed sequentially in the exact order of their listing.
(ii) the *switch* activity includes an ordered list of one or more conditional branches that include other activities and may be executed subject to the satisfiability of the conditions associated with them.
(iii) the *flow* activity includes a set of two or more activities that should be executed concurrently. A flow activity completes when all these activities have completed.
(iv) the *pick* activity makes a composition process to wait for different events (expressed by onMessage elements) and perform activities associated with each of these events as soon as it occurs.
(v) the *while* activity is used to specify iterative occurrence of one or more activities as long some condition holds true.

3.2 Event Calculus

The Event Calculus [7] is a temporal formalism designed to model and reason about scenarios characterized by a set of events, whose occurrences have the effect of starting or terminating the validity of determined properties. Given a (possibly incomplete) description of when events take place and a description of the properties they affect, EC is able to determine the maximal validity intervals over which a property holds uninterruptedly. The reasoning is based upon the hypothesis that all changes must be due to a cause, and properties of the world can only change at particular time points when events happen.

Language. The ontology of the event calculus comprises *fluents*, *events* (or actions) and *timepoints*. Events are the fundamental concept that brings about changes to the world. Any property of the world that can change over time is known as a fluent. A fluent is a function of the timepoint. The Event Calculus uses predicates to specify actions and their effects. Then, the following predicates define fluents' initiation, state, and termination, and events happening:

- $HoldsAt(f, t)$ is true iff fluent f holds at timepoint t.
- $Happens(a, t)$ is true iff action a happens at timepoint t.
- $Initiates(a, f, t)$ expresses that fluent f holds after timepoint t if action a happens at t.
- $Terminates(a, f, t)$ expresses that fluent f does not hold after timepoint t if action a happens at t.
- $InitiallyTrue(f)|InitiallyFalse(f)$ define if f holds or not at timepoint 0.

Axiomatics. The four axioms below capture the behavior of fluents once initiated or terminated by an action.

- $Clipped(t1, f, t2) \longleftarrow Happens(a, t1) \wedge (t1 \leq t < t2) \wedge Terminates(f, t2)$
- $Declipped(t1, f, t2) \longleftarrow Happens(a, t1) \wedge (t1 \leq t < t2) \wedge Initiates(f, t2)$
- $HoldsAt(f, t2) \longleftarrow Happens(a, t1) \wedge Initiates(a, f, t1) \wedge (t1 < t2) \wedge \neg Clipped(t1, f, t2)$
- $\neg HoldsAt(f, t2) \longleftarrow Happens(a, t1) \wedge Terminates(a, f, t1) \wedge (t1 < t2) \wedge \neg Declipped(t1, f, t2)$

Clipped expresses if fluent f was terminated during time interval $[t1, t2[$. Similarly, *Declipped* expresses if fluent f was initiated during time interval $[t1, t2[$. Fluents which have been initiated by event continue to hold until it occurs an event which terminates them ($HoldsAt$). Similarly, fluents which have been terminated by an event continue not to hold until an event which initiates them.

Then, we need to describe fluents' behavior before the occurrence of any actions which affect them:

- $HoldsAt(f, t) \longleftarrow InitiallyTrue(f) \wedge \neg Clipped(0, f, t)$
- $\neg HoldsAt(f, t) \longleftarrow InitiallyFalse(f) \wedge \neg Declipped(0, f, t)$
- $InitiallyTrue(f)|InitiallyFalse(f)$

Using these predicates, a fragment of the event log of the car rental scenario introduced in Section 2 is shown in Figure 1. Variables loc_i, veh_i, and car_i represent respectively the park number, the car number, and the customer identifier.

3.3 Our Approach: BPEL2EC

We now focus on how to transform WSBPEL activities into EC formulas in order to formally specify services behavior and therefore facilitate their analysis and verification.

L1 : $Happens(CSS.Enter(op1), 1)$
L2 : $InitiatesCSS.Enter(op1), equalTo(v1, veh1), 1)$
L3 : $Initiates(CSS.Enter(op1), equalTo(p1, loc1), 1)$
L4 : $Happens(CSS.Enter(op2), 27)$
L5 : $Initiates(CSS.Enter(op2), equalTo(v1, veh1), 27)$
L6 : $Initiates(CSS.Enter(op2), equalTo(p1, loc3), 27)$
L7 : $Happens(UIS.RelKey(op3, veh2), 28)$
L8 : $Happens(UIS.RelKey(op3), 29)$
L9 : $Happens(UIS.CarRequest(op4), 49)$
L10: $Initiates(UIS.CarRequest(op4), equalTo(p, loc2), 49)$
L11: $Happens(CIS.FindAvailable(op5, loc2), 50)$
L12: $Happens(CIS.FindAvailable(op5), 51)$
L13: $Initiates(CIS.FindAvailable(op5), equalTo(Res, veh2), 51)$
L14: $Happens(UIS.CarHire(op6, veh2, loc2), 52)$
L15: $Happens(CSS.Enter(op7), 53)$
L16: $Initiates(CSS.Enter(op7), equalTo(v1, veh2), 53)$
L17: $Initiates(CSS.Enter(op7), equalTo(p1, loc4), 53)$
L18: $Happens(UIS.RetKey(op8), 54)$
L19: $Initiates(UIS.RetKey(op8), equalTo(v, veh2), 54)$
L20: $Happens(UIS.CarRequest(op9), 69)$

Fig. 1. The CRS Event Log

Mapping of Basic Activities. Basic WSBPEL activities are transformed into their EC counterparts according to the transformations shown in Figure 2.

The EC representation of an *invoke* activity that calls an operation O in a service P consists of a literal such that it exists an event of calling O (i.e., inv:$P.O(vID, vX)$) and an event notifying the reception of the execution of O by the service composition (i.e., rec:$P.O(vID)$). The variable vID takes as value a unique identifier that represents the exact instance of the operation invocation and the variable vX takes the value that the input variable X of O has at the time of the invocation. The value of the output variable Y of O is represented by the fluent $equalTo$[1](Y, vY) initiated by the $Initiates$ predicate.

The EC representation of a *receive* activity in a service P that receives an invocation of its operation O by other partner service consists of a literal such that it exists an event of receipt of an invocation of O (i.e. rec:$P.O(vID)$), where the variable vID represents the exact instance of the operation invocation by other partner. The value of the variable X of O on message receipt is represented by the fluent $equalTo(X, vX)$ initiated by the $Initiates$ predicate.

A *reply* activity in a service P that respond to a previously accepted request for the execution of the operation O is represented in EC using a literal such that it exists the completion of the execution of O (i.e. rep:$P.O(vID, vX)$), where the variable vID represents the exact instance of the operation invocation and the variable vX represents the value of the output variable X of O.

[1] The fluent $equalTo(VarName, val)$ signifies that value of $VarName$ is equal to val.

Sample BPEL Code	Sample EC Specification

`<invoke partnerLink="P"`
`portType= "a:Pport"`
`operation= "O"`
`inputVariable= "X"`
`outputVariable= "Y"/>`
	$Happens(\text{inv}{:}P.O(vID, vX), t1) \wedge (\exists t2)$
	$Happens(\text{rec}{:}P.O(vID), t2) \wedge (t1 \leq t2) \wedge$
	$Initiates(\text{rec}{:}P.O(vID), equalTo(Y,), t2)$

`<receive partnerLink="P"`
`portType= "a:Pport"`
`operation="O"`
`variable="X"/>`
	$Happens(\text{rec}{:}P.O(vID), t) \wedge Initiates(\text{rec}$
	$P.O(vID), equalTo(X, vXc), t)$

`<reply partner="P"`
`portType = "a:Pport"`
`operation= "O"`
`variable="X"/>`
	$Happens(\text{rep}{:}P.O(vID, vX), t) \wedge Happens$
	$(\text{rec}{:}P.O(vID, vX), t1) \wedge (t1 < t)$

`<assign name ="A">`
`<copy><from variable ="X"`
`part="a"/>`
`<to variable="Y" part="b"/>`
`</copy>`
`</assign>`
	$Happens(\text{as}{:}A(vID), t1) \wedge (\exists t2)(t1 < t2) \wedge$
	$Initiates(\text{as}{:}A(vID), equalTo(Y.b, vX.a),$
	$t2)$

`<actType name="A">...</actType>`
`<wait for = "T"/>`
`<actType name="B">...</actType>`
	$EC(A, \mathcal{T}) \wedge EC(B, \mathcal{T}) \wedge max_t(A) <$
	$(min_t(A) - T)$

`<throw faultName="faultname"` `faultVariable="X"/>`	$Happens(th : faultname(vID, vX), t)$

Fig. 2. Mapping of Basic activities

Then, the EC representation of a *throw* activity that signals internal fault $faultName$ in a service P consists of a literal such that it exists a throwing event, (i.e. $th{:}faultName(vID, vX)$), where the variable vID represents the exact instance of the *throw* activity and the variable vX represents the value of the $faultVariable$ being thrown.

Mapping of Structured Activities. After transforming the basic activities, it is also important to specify their temporal relationships. That is the case for the *sequence* and the $flow$ constructs. The translation scheme of the EC formulas for the *squence* and *switch* activities is given in Figure 3.

In these patterns, (i) actType can be any type of a basic or structured WS-BPEL activity; (ii) EC(A,$\mathcal{T}$) represents the EC formulas where A is an activity and $\mathcal{T}$ a temporal domain (we use an ordered set ($\mathbb{T}$,$\prec$), and the natural numbers $\mathbb{N}$ with their usual ordering); (iii) $min_t(A)$ represents the time variable of the earliest predicate in the formulas of activity A (i.e., the predicate that is expected to occur the first given the constraints between the time variables of the predicates representing A), and (iv) $max_t(B)$ represents the time variable

Sample BPEL Code	Sample EC Specification
<sequence> <actType name="A"> ... </actType> <pick> <onMessage partner="P" portType= "a:Pport" operation="O" variable="X"> <actType name="B">...</actType> </onMessage> <onAlarm for="T"> <actType name="C">...</actType> </onAlarm> </pick> </sequence>	$EC(A,\mathcal{T})\wedge Happens(\text{om:}O(vID,vX),t2)\wedge$ $(max_t(A)\leq \text{t2}\leq(max_t(A)+\text{T}))\wedge Initiates($ $\text{om:}O(vID,vX),equalTo(X,vX),t2) \Longrightarrow$ $EC(B,[min_t(B)])\wedge t1 < min_t(B)$ $EC(A,\mathcal{T})\wedge\neg Happens(\text{om:}O(vID,vX),t2)$ $\wedge(max_t(A)\leq \text{t2}\leq(max_t(A)+\text{T}))\Longrightarrow \text{EC}(C$ $,[min_t(C)])\wedge max_t(A)+\text{T}< min_t(C)$
<switch> <case condition=" P=v1"> <actType name="A">...</actType> </case> <otherwise> <actType name="C">...</actType> </otherwise> </switch>	$HoldsAt(equalTo(P,v1),t1)\Longrightarrow \text{EC}(A,$ $[min_t(A)]) \wedge (t1 < min_t(A))$ $\neg HoldsAt(equalTo(P,v1),t1)\Longrightarrow \text{EC}(C,$ $[min_t(C)])\wedge \text{t2}< min_t(C)$

Fig. 3. Mapping of *pick* and *switch* activities

of the latest predicate in the formulas of activity B (i.e., the predicate that is expected to occur the lastest given the constraints between the time variables of the predicates representing B). The rest of the transformations are analogous to the transformation of *switch* and *pick* activities, and are presented in Figure 4.

Let us now give an example of EC formulas extracted according to the above patterns. We show an extract of the WSBPEL specification of the car rental scenario introduced in section 2, and the EC formula extracted from it. This fragment refers to the part of process that receives a request for a car and checks for available cars. It is presented in Figure 5.

The first implication in the EC formula represents the link $rec{-}to{-}auth$ in the flow activity of the process. Conditions of this implication represent the receive activity $receiveRequest$, and its consequence represents the sequence activity in the process. The second implication represents the ordering of the constituent activities of the *sequence* activity: its conditions represent the *assign* activity $a1$ and its consequence represents the *invoke* of activity $findCar$.

4 EC Checking

In the previous section, we showed how to translate the WSBPEL constructs into their Event Calculus predicates counterparts. The objective of this section is to show how we offer reasoning about a WSBPEL process represented as a set

Sample BPEL Code	Sample EC Specification

`<sequence>`
`  <actType name="A">...</actType>`
`  <actType name="B"> ...</actType>`
`</sequence>`
	$EC(A,\mathcal{T}) \Longrightarrow \mathrm{EC}(B,\mathcal{T}) \wedge max_t(A) < min_t(B)$

`<while condition="P=v1">`
`  <actType name="A">... </actType>`
`</while>`
	$HoldsAt(equalTo(P,v1),t1) \Longrightarrow \mathrm{EC}(A, [min_t(A)]) \wedge t1 < min_t(A)$

`<flow>`
` <links>`
` <link name="AtoB"/>`
` <link name="AtoC"/> ... </links>`
` <actType name="A">`
` <source linkName="AtoB"`
`  transitionCondition="P=v1"/>`
` <source linkName="AtoC" /> ...`
` <actType name="B">`
` <target linkName="AtoB" /> ...`
` <actType name="C">`
` <target linkName="AtoC" /> ...`
`</flow>`
	$EC(A,\mathcal{T}) \wedge HoldsAt(equalTo(P,v1),t1) \wedge max_t(A) < \mathrm{t2} \Longrightarrow \mathrm{EC}(B,[min_t(B)]) \wedge \mathrm{t2} < \mathrm{min}_t(B)$
	$EC(A,\mathcal{T}) \Longrightarrow \mathrm{EC}(C,[min_t(c)]) \wedge max_t(A) < min_t(C)$

Fig. 4. Mapping of Structured activities

of EC predicates in order to check its consistency in three cases: the first case is a static check, before running the process, the second case is at runtime, and the third case is the ability to check the process execution against non-functional requirements.

4.1 Static Verification

The need for static verification is important for composite processes which coordinate a set of autonomous Web services because these processes can be very complex processes, and that we need to check if a WSBPEL process is consistent, which is not a trivial task as soon as a WSBPEL process manages concurrency, distribution and long-duration activities. Transforming WSBPEL constructs into EC predicates gives the opportunity to model-check such a process, with respect to temporal constraints, and to verify that processes satisfy certain properties.

For instance, let us suppose a process including a flow construct. This construct allows to specify one or more activities to be performed concurrently. The EC specification on this construct is given in Figure 4. Once it is rewritten using EC predicates, we propose a solution for verifying a WSBPEL process instance against its temporal constraints. For instance, we can express that, given a sequence of two services, the second service will be executed only once the first one is completed (see Figure 3 for the EC specification of WSBPEL sequence). This very basic example shows that it is possible to formally check the control flow of

Part of WSBPEL composition process for CRS

```
<process name="CRS">                    <target linkName="rec-to-auth"/>
<partners> ... </partners>              <assign name="a1">
 <flow>                                 <copy><from variable="Req" part="Loc"/>
 <links>                                <to variable="Q" part="Loc"/>
 <link name="rec-to-auth"/>             </copy>
 </links>                               </assign>
 <receive name="receiveRequest"         <invoke name="findCar"
  partner="UIS"                          partner="CIS"
  portType="sns:UISPT"                   portType="crns:CISPT"
  operation="CarRequest"                 operation="FindAvailable"
  variable="Req"                         inputVariable="Q"
  createInstance="yes">                  outputVariable="Res"/>
<sourcelinkName="rec-to-auth"/>         </sequence>
<sequence>                              </flow>
                                        </process>
```

EC formulas

$Happens(\text{rec}{:}UIS.CarRequest(oID1), t1) \wedge Initiates(\text{rec}{:}UIS.CarRequest(oID1),$ equalTo $(Req.Loc, vReq.Loc), t1) \wedge Initiates(\text{rec}{:}UIS.CarRequest(oID1),$ $equalTo(Req.CId, vReq.CId), t1) \Longrightarrow$ $(\exists t2)(t1<t2) \wedge Happens(\text{as}{:}a1(aID), t2) \wedge (\exists t3)(t2<t3) \wedge Initiates(\text{as}{:}a1(aID),$ $equalTo(Q.Loc, vReq.loc), t3) \Longrightarrow$ $(\exists t4) Happens(\text{inv}{:}CIS.FindAvailable(oID2, vQ), t4) \wedge (t3 \leq t4) \wedge (\exists t5)$ $Happens(\text{rec}{:}CIS.FindAvailable(oID2, vQ), t5) \wedge (t4 \leq t5) \wedge Initiates(\text{rec}{:}CIS.$ $FindAvailable(oID2, vQ), equalTo(Res, vRes), t5))$

Fig. 5. Example of EC formulas extracted from the WSBPEL process for CRS

a WSBPEL process (and the interactions between the Web services) using the EC predicates, and this offers the ability to discover the potential flaws of such a process such as livelocks, deadlocks, or unused branches in the control flow.

4.2 Dynamic Verification

A second aspect of verification is the runtime verification. This kind of verification is welcome since some interactions between Web services that constitute a process may be dynamically specified at runtime, causing unpredictable interactions with other services, and making the previous verification method (static) unusable. This dynamic behavior can not be model-checked, but it remains important to be sure that the execution of the process remains consistent. This is the reason why we offer the possibility to verify a process at runtime. As the verification occurs in real-time, it becomes possible to handle deviations wrt. the observed behavior of the process. To provide this verification, we use logical predicates (as in the previous method), but we compare these predicates with the events that occur and are recorded during the process execution. When one or several predicates are unsatisfied, this means that something wrong occurs in the execution, and it is possible to exactly point out what happens.

For instance, in our example, the CSS and the CIS services won't be owned by the owner of CBS. Moreover, new instances of the CSS and CIS services may be deployed when new car rental companies and car parks make their offerings available to the CBS, and existing instances may be withdrawn when companies and car parks stop their collaboration with CBS. When such conditions occur, services monitoring has to be based on events and state information that can be reasonably assumed to be in the ownership of the service provider and is fixed at runtime. Requirements for individual services are still to be specified and monitored but only if this is possible through events that are known to the composition process, or events that can be derived from them.

4.3 Non-functional Requirements Verification

Another interest is the ability to model non-functional requirements using the EC, and to check a process against these properties. Let us consider an example on policies (security policies for instance). We consider a WSBPEL process that expect to enforce some high-level authorization policies. The specifications of these authorization policies are separated from the process code, and they should be carefully audited. Using the EC, we are able to formalize these policies by embedding logical predicates, and to check if a process complies with the policies.

4.4 Example

Let us suppose the following CRS requirements, represented as rules.

R1) The rule R1 expresses an assumption about the behavior of the CSS sensoring services: $(\forall t1, t2)Happens(inv : CSS.Enter(oID1), t1) \wedge Initiates(inv : CSS.Enter(oID1), equalTo(v1, vID), t1) \wedge Initiates(inv : CSS.Enter(oID1), equalTo(p1, pID1), t1) \wedge Happens(inv : CSS.Enter(oID2), t2) \wedge (t1 + {t_m}^2 \leq t2) \wedge Initiates(inv : CSS.Enter(oID2), equalTo(v2, vID), t2) \wedge Initiates(inv : CSS.Enter(oID2), equalTo(p2, pID2), t2) \Longrightarrow (\exists t3)$ $Happens(inv : CSS.Depart(oID3), t3) \wedge (t1 + t_m \leq t3 \leq t2 - t_m) \wedge Initiates(inv : CSS.Depart(oID3), equalTo(v3, vID), t3) \wedge Initiates(inv : CSS.Depart(oID3), equalTo(p3, pID1), t3)$.

According to this rule, if a car vID is sensed to enter a car park $pID1$ at time $t1$ and later, at time $t2$, the same car is sensed to enter the same or a different car park, then a *Depart* event (signifying the departure of vID from $pID1$) must have also occurred between the two *enter* events. The *Happens* predicates in R1 represent the invocation of the operations *Enter* and *Depart* in CBS by CSS following the entrance and departure of cars in car parks. The *Initiates* predicates initiate fluents that represent the specific value bindings of the input parameters vi and pi (i=1,2,3) of the operations *Enter* and *Depart*. R1 represents a composite requirement whose satisfiability depends on the availability of CSS services and their

[2] t_m refers to the minimum time between the occurrence of two events.

ability to correctly execute. This requirement is an example of requirement that cannot be statically verified and that must be monitored at runtime.

R2) The rule R2 defines the behavior of CIS services:
$(\forall t1, t2) Happens(inv : CIS.FindAvailable(oID, pID), t1) \wedge Happens(rec : CIS.FindAvailable(oID), t2) \wedge (t1 \leq t2) \wedge HoldsAt(equalTo(availability(vID1), "not\ avail"), t2 - t_m) \implies \neg Initiates(rec : CIS.FindAvailable(oID), equalTo(vID2, vID1), t2)$.

According to this rule, the operation *FindAvailable*, which is provided by the CIS service and searches for available cars at specific car parks, should not return the identifier of a car to CBS unless this car is available.

R3) The rule R3 states that whilst a customer has the key of a car, this car cannot be available for rental:
$(\forall t1, t2, t3) Happens(inv : UIS.RelKey(oID1, vID), t1) \wedge Happens(rec : UIS.RelKey(oID1), t2) \wedge (t1 \leq t2) \wedge Happens(inv : UIS.RetKey(oID2), t3) \wedge (t2 \leq t3) \wedge Initiates(inv : UIS.RetKey(oID2), equalTo(v, vID), t3) \implies (\forall t4)(t1 < t4) \wedge (t4 < t3) HoldsAt(equalTo(available(vID), "not\ avail"), t4)$.

Detecting Some Deviations. Assuming the log of events of the car rental scenario (see Figure 1), we now show how we can detect some deviations:

D1) The behavior of CBS violates the requirement R1. This occurs because there are two *enter* events that signify the entrance of *veh*1 first to car park *loc*1 at T=1 (see literals L1-L3 in Figure 2) and, subsequently, to car park *loc*3 at T=27 (see literals L4- L6 in Figure 2) but no *depart* event to signify the departure of *veh*1 from *loc*1 between these *enter* events.

D2) The requirement R2 is violated by the behavior of CBS. The violation of R2 in this case occurs since we can derive from the requirement R3 that *veh*2 could not be available from T=30 when its key was released (see literals L7 and L8 in Figure 1) until T=53 (that is one time unit before its key was returned back). Nevertheless, the execution of the operation *FindAvailable* of the CIS service at T=51 reports that vehicle *veh*2 is available (see literal L13 in Figure 1).

5 Architecture and Implementation

To support the verification and the consistency checking of the behavior of a Web service composition, we propose the framework shown in Figure 6.

The EC checker processes the events which are recorded in the event log by the event extractor in the order of their occurrence, identifies other expected events that should have happened but have not been recorded (these events can be derived from the composition requirements by deduction), and checks if these events are compliant with the behavioral properties and assumptions of the composition. When events are not consistent with specified requirements, the EC checker records the deviation in a deviations log.

Non-functional requirements are additional constraints about the behavior of partners, or their individual services. These constraints are specified by service

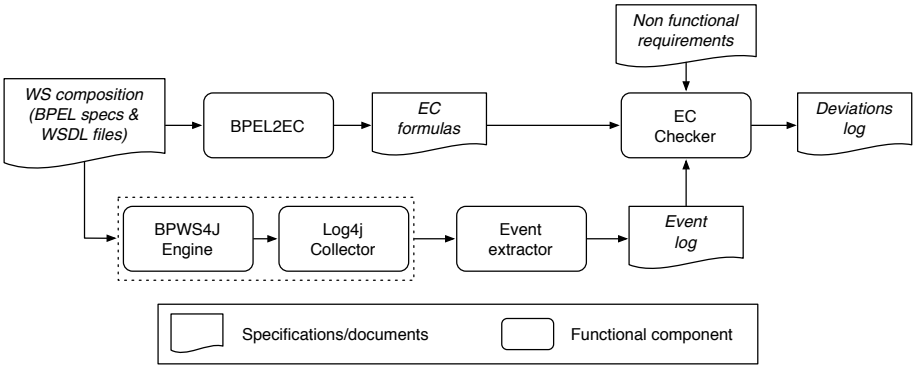

Fig. 6. Monitoring framework

providers and must be expressed in terms of events, effects and state variable conditions which are used in the behavioral properties directly or indirectly, and are formalized in terms of EC predicates. They may include, for example, security and control access policies.

The BPEL2EC tool is built as a parser that can automatically transform a given BPEL process into EC formulas according to the transformation scheme detailed in Section 3.3. It takes as input the specification of the Web service composition as a set of coordinated web services in WSBPEL and produces as output the behavioral specification of this composition in Event Calculus. This specification can be amended by the service providers, who can also use the atomic formulas of the extracted specification to specify additional assumptions about the operations if appropriate.

While executing the Web service composition, the process execution engine generates events which are sent as string streams to the event extractor of our framework. In our implementation, we have used the engine bpws4j[3] and log4j[4] to generate logging events. The event extractor (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.). The logging events from bpws4j that corresponds respectively to the invocation of an operation in some external service and its *receive* activity look as follows:

```
2006-03-13 11:41:59,714[Thread-34]
     DEBUG  bpws.runtime.bus Invoking external service with[WSIFRequest:se
     rviceID='{http://tempuri.org/services/CarReg}CarRegServicefb0b0-fbc59
     65758-8000'operationName='isAvailable'incomingMessage='org.apache.wsi
     f.base.WSIFDefaultMessage@155423name:nullparts[0]:[JROMString:loc:One
     ]'contextMessage='null']
2006-03-13 11:42:00,724[http8080-Processor25]
      INFO   bpws.runtime- Incoming request:[WSIFRequest:serviceID='{http:
     //carservice.org/wsdl/OnlineRenter}carServiceBP'operationName='receiv
```

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

```
e Request'incomingMessage='org.apache.wsif.base.WSIFDefaultMessage@25
491dname:null parts[0]:[JROMString:loc:One]parts[1]:[JROMString:custI
d:km r]'Context-Message='org.apache.wsif.base.WSIFDefaultMessage@1e32
382 name:null parts[0]:http://xml.apache.org/soap/v1parts[1]:{http://
carservice.org/wsdl/OnlineRenter}CarRenter parts[2]:CRS']
```

The [Thread-34] is the unique ID assigned by the bpws4j engine to the invocation of the external service of this instance of the invoke activity and the corresponding response from the external service. The [http8080-Processor25] is the unique ID assigned by the bpws4j engine to this instance of the *receive* activity and its corresponding *reply* activity. These events are then converted to EC events to be checked by the EC checker, which uses the Discrete Event Calculus Reasoner[5].

6 Related Work

It exists various research activities to formally define, analyze, and verify Web services orchestration languages. A group at Humboldt University is working on formalizations of BPEL for analysis, graphics and semantics [9], using Petri-nets and ASMs to formalize the semantics of BPEL. However, the pattern-based Petri-Net semantics of BPEL [16] does not capture fault handling, compensation handling, and timing aspects. Moreover, the feasibility of verifying more complex business processes is not clear and still subject to future work.

Additionally, there are some attempts based on finite state machines [5], and process algebras [3]. Although all of them are successful in unraveling weaknesses in the informal specification, they are of different significance for formal verification. Like abstract state machines, these approaches typically do not support some of BPEL's most interesting features such as fault and event handling.

Work concerning the area of adapting Golog for composition of semantic web services is carried out by McIlraith and others [10]. They have shown that Golog might be a suitable candidate to solve the planning problems occurring when services are to be combined dynamically at run-time. Additionally they related their work [8] to WSBPEL explicitly by stating that the semantic web efforts in the research area are disconnected from the seamless interaction efforts of industry and thus propose to take a bottom-up approach to integrating Semantic Web technology into Web services. But they mainly focus on introducing a semantic discovery service and facilitating semantic translations.

Formal verification of Web Services is addressed in several papers. The SPIN model-checker is used for verification [13] by translating Web Services Flow Language (WSFL) descriptions into Promela. [6] uses a process algebra to derive a structural operational semantics of BPEL as a formal basis for verifying properties of the specification. In [4], BPEL processes are translated to Finite State Process (FSP) models and compiled into a Labeled Transition System (LTS)

[5] http://decreasoner.sourceforge.net

in inferring the correctness of the Web service compositions which are specified using message sequence charts. In [14], Web services are verified using a Petri Net model generated from a DAML-S description of a service.

One common pattern of the above attempts is that they adapt static verification techniques and therefore violations of requirements may not be detectable. This is because Web services that constitute a composition process may not be specified at a level of completeness that would allow the application of static verification, and some of these services may change dynamically at run-time causing unpredictable interactions with other services.

Unlike these earlier verification efforts, we consider the correctness of the individual peer implementations as well as the verification of the global properties of the composite Web services. Verification of the communication flow does not guarantee that the composition behaves according to the specification unless we ensure that each individual service obeys its published contract.

The Event Calculus has been theoretically studied. Denecker et al. [2] use the Event Calculus for specifying process protocols using domain propositions to denote the meanings of actions. In [17] the Event Calculus has been used in planning. Planning in the Event Calculus is an abductive reasoning process through resolution theorem prover. [19] develops an approach for formally representing and reasoning about business interactions in the Event Calculus. The approach was applied and evaluated in the context of protocols, which represent the interactions allowed among communicating agents. Our previous work [15] is close enough to the current work. It presents an event-based framework associated with a semantic definition of the commitments expressed in the Event Calculus, to model and monitor multi-party contracts. This framework permits to coordinate and regulate Web services in business collaborations.

7 Conclusion and Future Directions

In this paper, we have presented a formal framework for checking both functional and non-functional requirements of Web service composition. The properties to be monitored are specified using the Event Calculus formalism. Functional requirements are initially extracted from the specification of the composition process that is expressed in WSBPEL. This ensures that they can be expressed in terms of events occurring during the interaction between the composition process and the constituent services that can be detected from the execution log. Non-functional requirements to be checked are subsequently defined in terms of the identified detectable events by service providers.

The framework is still under development. Ongoing work on it is concerned with: (1) the implementation of the EC checker since until now we have used the Mueller's Discrete EC Reasoner, (2) the study of the correctness requirements in Web service coordination protocols, and their specification in terms of events expressed in the Event Calculus in order to facilitate their integration in our framework, (3) the study of alternatives to establish links with other process

algebra in order to import process algebra specific verification techniques such as axiomatizations of behavioral equivalences.

References

1. A. Arkin, S. Askary, B. Bloch, and F.Curbera. Web services business process execution language version 2.0. Technical report, OASIS, December 2004.
2. M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In *Proceedings of the 10th European Conference and Symposium on Logic Programming (ECAI)*, pages 384–388, 1992.
3. A. Ferrara. Web services: a process algebra approach. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM Press.
4. 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.
5. 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.
6. M. Koshina and F. van Breugel. Verification of business processes for web services. Technical report, New York University, SFUCMPT-TR-2003-06.
7. R. Kowalski and M. J. Sergot. A logic-based calculus of events. *New generation Computing 4(1)*, pages 67–95, 1986.
8. M. S. Mandell, D.J. Adapting bpel4ws for the semantic web: The bottom-up approach to web service interoperation. In *Proc of the 2nd Int. Semantic Web Conf. (ISWC)*, 2003.
9. A. Martens. Analysis and re-engineering of web services. In *ICEIS (3)*, pages 419–426, 2004.
10. S. McIlraith and T. Son. Adapting golog for composition of semantic web services. In *Proc of the 8th International Conference on Principles of Knowledge Representation and Reasoning*, 2002.
11. R. Miller and M. Shanahan. The event calculus in classical logic - alternative axiomatisations, 1999.
12. E. T. Mueller. Event calculus reasoning through satisfiability. *J. Log. and Comput.*, 14(5):703–730, 2004.
13. S. Nakajima. Verification of web service flows with model-checking techniques. In *CW*, pages 378–385, 2002.
14. S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 77–88, New York, NY, USA, 2002. ACM Press.
15. M. Rouached, O. Perrin, and C. Godart. A contract-based approach for monitoring collaborative web services using commitments in the event calculus. In *Sixth International Conference on Web Information Engineering System (WISE05)*, pages 426–434, 2005.
16. K. Schmidt and C. Stahl. A petri net semantic for BPEL4WS validation and application. In *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN 04) / Ekkart Kindler (Ed.)*, pages 1–6. Bericht tr-ri-04-251, Universitt Paderborn, Sept. 2004.

17. M. Shanahan and M. Witkowski. Event calculus planning through satisfiability. *J. Log. and Comput.*, 14(5):731–745, 2004.
18. W. M. P. van der Aalst, H. T. de Beer, and B. F. van Dongen. Process mining and verification of properties: An approach based on temporal logic. In *OTM Conferences (1)*, pages 130–147, 2005.
19. P. Yolum and M. P. Singh. Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. *Annals of Mathematics and Artificial Intelligence*, 42(1-3):227–253, 2004.