An Event Based Model for Web Service Coordination

Mohsen Rouached

AN EVENT-BASED MODEL FOR WEB SERVICES COORDINATION

Mohsen Rouached
LORIA-INRIA
BP 239, F-54506 Vandoeuvre-les-Nancy Cedex, France
rouached@loria.fr

Claude Godart
LORIA-INRIA
BP 239, F-54506 Vandoeuvre-les-Nancy Cedex, France
godart@loria.fr

Keywords: Web services, Web service composition, composite Event, Event Calculus.

Abstract: The promise of Web services is centered around standard and interoperable means for integrating loosely coupled Web based components that expose well-defined interfaces, while abstracting the implementation and platform specific details. The current and more mature core Web services standards SOAP, WSDL and UDDI provide a solid foundation to accomplish this. However, these specifications primarily enable development of simple Web services whereas the ultimate goal of Web services is to facilitate and automate business process collaborations both inside and outside enterprize boundaries. Useful business applications of Web services in B2B, B2C, and enterprize application integration environments will require the ability to compose complex and distributed Web services and the ability to formally describe the relationships between the constituent low-level services.
This paper advocates an event-based approach for Web services coordination. We focused on reasoning about events to capture the semantics of complex Web service combinations. Then we present a formal language to specifying composite events for managing complex interactions amongst services, and detecting inconsistencies that may arise at run-time.

1 INTRODUCTION

Web services have been gathering an increasing amount of attention lately. They have emerged as a distributed computing paradigm for applications, or business processes, that interact over the open Internet through the use of standard protocols. Current Web services standards such as SOAP, and WSDL provide rudimentary mechanisms for defining interactions amongst services that may be located in different organizations. While WSDL provides the definitions for the entry-points of a service, in many cases, interactions between services have more structure than can be described by just the definition of entry points. Indeed, to form real B2B or B2C interactions, a set of services need to work together and be executed in specified order. Such execution is termed service composition or choreography. Several standards have been proposed and are being introduced into practice, especially BPML, BPEL4WS, and WSCI. However, these approaches are conceptually no more than simple extensions of traditional workflow technologies wherein Web services, which have to cope with a highly dynamic environment, are used to represent tasks instead of ad hoc legacy processes. Indeed, distributed enterprize environments assume that Web services are not limited to peer-to-peer interactions between business partners, but that it should be possible to coordinate interactions of a group of business partners. However, the above approaches are largely limited to rather simple request/response services and the Web choreography standards rely on the XML based RPC which is essentially a one-to-one mechanism, and is not suitable for coordinating Web services in complex environments. Additionally, Web service specifications mostly concentrate on lower levels and do not offer high-level abstractions to accommodate variations in service invocations and run-time interactions.

On the other hand, the event-based architectural style has become prevalent for large-scale distributed applications due to the inherent loose coupling of the participants. It facilitates the clear separation of communication from computation and carries the potential for easy integration of autonomous, heterogeneous components into complex systems that are easy to evolve and scale (Pietzuch et al., 2003). However, the event-driven technology is not well exploited in

the context of Web services. Indeed, many existing initiatives such as WS-Eventing and WS-Notification restrict subscriptions to single events only and thus luck the ability to express interest in the occurrence of patterns of events. However, especially in large-scale Web applications, event sinks may be overwhelmed by the vast number of primitive, low-level events, and would benefit from a higher-level view. Such a higher-level view is given by composite events that are published when an event pattern occurs. Therefore, Web services need to support composite event detection, in order to quickly and efficiently notify their clients of new, relevant information in the network. This can improve efficiency and robustness in the case of widely distributed Web services where bandwidth is limited and services are loosely coupled. Nevertheless, without composite event detection, many messages would still be sent unnecessarily, because specific event combinations or patterns could not be expressed by recipients.

In this paper, we address the problem of coordinating large-scale Web services by adopting a purely event-based approach. We propose a distributed architecture and a general framework to handle the event composition in Web services. This framework includes a formal language with well formal semantics to specifying and reasoning about composite events, which facilitates the detection of several inconsistencies that may arise at run-time. The remainder of the paper is structured as follows. Section 2 describes a car rental scenario used as a running example. In section 3 we present our event-based architecture. We start by defining events in Web services context. Then, we present the proposed model. Section 4 studies the event composition in distributed Web service environments. It details a formal language for specifying composite events and precises how this formalism can verify and detect some examples of inconsistencies that may arise in the running scenario. In section 5, the related work is discussed. Finally, section 6 concludes the paper and discusses some future directions.

2 RUNNING EXAMPLE

Throughout this article, we will illustrate our ideas with an interesting running example. We consider a car rental scenario that involves four complementary services. A car broker service (CBS) which 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 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 requirements monitoring, 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 precise how Web services interactions can be specified and formalized using events, and how this specification could facilitate their monitoring at run-time.

3 EVENT BASED WEB SERVICE COORDINATION

The services will interact by responding to communal events. We are not interested in the intra-service interaction between the different business objects that make out a single service, but we focused on providing a formal framework for coordinating distributed and autonomous Web services . Then, if a such framework could be applied to inter-services interaction, the local business objects of each service can be considered as small atomic services themselves and therefore it can be applied to intra-service interaction recursively. Accordingly, this approach can be easily generalized into an n-level system since the number of parties that participate in an event can be easily increased by just adding another service that subscribes to the event, without having to redesign the entire chain of one-to-one message exchanges. Let us now introduce the notion of events and describe the role that can play in Web services interactions.

3.1 Events in Web Services

Events can be simply perceived as occurrences that happen over time, and can be independent or not from each other. They are represented by structured messages which are exchanged between the event genera-

tor and any number of event receivers, and carry contextual information in which the event is described, plus the circumstances in which it occurred (event context). Such contextual information may be an explicit part of the event representation, or can be implicitly deduced from said circumstances. The significance of business events is ensured by assigning types to these events. The type of an event is modeled by its body, which incorporates a data structure representing the context in which the event was generated. In our framework, three event types could be distinguished:

- The invocation of an operation by a partner service. These events are represented by terms of the form *Q.ServiceName.OperationName(parameters)*. For example, the event *Q.CBS.ReleaseKey(car,customer,Park)* precises that CBS invokes an operation in UIS that signifies the release of a car key to a customer.
- The reply following the execution of an operation that was invoked by a partner service. These events are represented by terms of the form *R.ServiceName.OperationName(parameters)*. For example, the event *R.CSS.Depart(car,park)* is a reply event that signifies the exit of the car from the car park. If the previous event did not occur, the event *Q.CBS.Available(car,park)* must be generated to CIS in order to check the availability of the requested car.
- The assignment of a value to a variable. These events are represented by terms of the form *AS.AssignmentName(assignmentId)*. For example, if the URL of a service (for example CSS or CIS) has to be changed, an assignment event *AS.AssignementURL(NewURL)* is generated.

In next section, we show how to use these events to manage and monitor interactions between Web services.

3.2 Web Services Interactions

Web services interactions are based on the simultaneous participation in shared events. Event notifications are not propagated one-to-one but are broadcast in parallel to all services that have an interest in relevant event. Then, the updates that result from a given business event are to be coordinated throughout the entirety of all interested services that participate in that event. This broadcasting paradigm is fully compatible with current Web service standards such as SOAP, WSDL and UDDI.

Since each Web service operates as an autonomous and separate entity and is not subject to a form of centralized monitoring, we associate to each service a local event history, and to each group of services an event space. The event history represents events that occurred or they are assumed to be occur whereas the event space represents a group of collaborating services which interact by exchanging events from their local event histories, and changes over time as services join or leave the collaboration. Each event space is represented by a monitor, which operates as an UDDI-like registry for advertising and discovering events that may be published or subscribed to by the participating services. As soon as a relevant event is published, the monitor will dispatch it to subscribed participants, after which they may consume the event once it conforms to prescribed policies. Service policies such as security policies and access controls are beyond the scope of this paper, but are planned as very important topics for future research.

The monitor may be allocated to one of the participants according to some criteria. For example, in the rent car scenario, we can specify an event space formed by the CBS, CIS, CSS and UIS. The monitor is delegated to the CBS since it is responsible for the instantiation of the interaction and therefore it has a directed relationship with customers. In some other cases, in order to stress the independent and trusted nature of the monitor, it can be implemented on an individual trusted party chosen by all service partners.

In complex environment where numerous business partners interact in shared business processes, the response to a given request usually needs to handle several event spaces at the same time and therefore needs a mean to ensure interactions between them at runtime. In our approach, this is illustrated by allowing communication between monitors of all the involved event spaces. Indeed, in an event space, if an event occurs in a given Web service, this event is routed to the local monitor which broadcast it to all subscribed services. If the monitor detects that a subscribed service is situated in an external event space, it propagates the event to its representing monitor that will carry out the same treatment. The event routing is carried out according to the URL of services and monitors, that are introduced as parameters of the generated event. These URL are then stored as attribute values to each event space's monitor and can be changed dynamically by triggering an event of assignment type. Figure 1 shows how communications between three different event spaces can be established.

After introducing the notion of events and presenting their role in coordinating and composing Web services, we focus in next section on providing a formal specification to composite events, and then using this specification to handle some inconsistencies that may arise in the car rental scenario described in section 2.

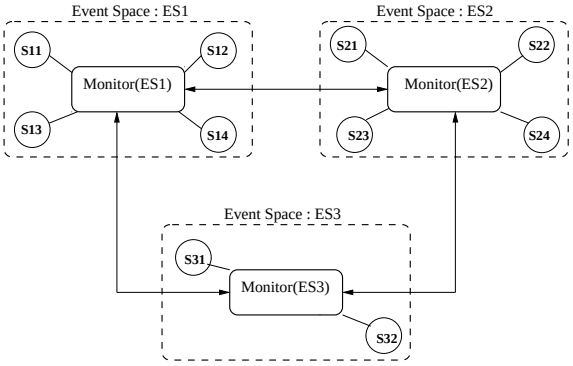

Figure 1: Web services Coordination

4 EVENT COMPOSITION

In Web services, the current event based approaches such as (Lemahieu et al., 2003a; Lemahieu et al., 2003b) only provide parameterized primitive events and leave the task of composing events to the application programmer. That means that they filter event notifications trying to deliver events of interest to consumers without considering any correlation with other event occurrences. Instead, we are interested in specifying and handling event composition. We start by proposing a core composite event language, then we use this formalism to check some examples of deviations derived from the car rental scenario.

4.1 Composite Event Language

In order to provide a computational model for events, we use the event calculus (Kowalski and Sergot, 1986) which is a calculus that allows for specifying some state at particular time-points for time-varying properties (called fluents). The occurrence of an event is represented by the predicate $Happens(e, t, \Re(t1, t2))$ which signifies that an event e occurs at some time t within the time range $\Re(t1, t2)$. The boundaries of $\Re(t1, t2)$ can be specified by using either time constants, or arithmetic expressions over the time variables of other $Happens$ predicates of the same formula. An event may initiate or terminate a fluent. A fluent is specified as a condition over the value of a specific variable of the composition process of a system. The fluent $equalTo(x, y)$, for example, signifies that the value of the variable x is equal to y. The effects of events on fluents are represented by the predicates $Initiates(e, f, t)$ and $Terminates(e, f, t)$. $Initiates(e, f, t)$ signifies that a fluent f starts to hold after the event e at time t. $Terminates(e, f, t)$ signifies that a fluent f ceases to hold after the event e occurs at time t. An EC formula may also use the predicate $HoldsAt(f, t)$ which signifies that the fluent f holds at time t. Using these predicates, some literals included in the event log of the car rental scenario are shown in figure 2. Variables l_i, v_i, and c_i represent respectively the park number, the car number, and the customer identifier. Given this

L1 : $Happens(R.CSS.enter(v1, l1), 1, \Re(1, 1))$

L2 : $Happens(Q.UIC.RelKey(v1, c1, l1), 5, \Re(5, 5))$

L3 : $Happens(Q.CIS.Available(v1, l1), 9, \Re(9, 9))$

L4 : $Happens(R.UIS.RetKey(v1, l1), 15, \Re(15, 15))$

L5 : $Happens(RC.CSS.Enter(v2, l2), 18, \Re(18, 18))$

L6 : $Happens(R.UIS.RetKey(v2, l2), 23, \Re(23, 23))$

L7 : $Happens(Q.CIS.Available(v2, l2), 26, \Re(26, 26))$

L8 : $Happens(Q.CSS.Enter(v1, l1), 27, \Re(27, 27))$

L9 : $Happens(Q.UIS.RelKey(v2, c2, l2), 28, \Re(28, 28))$

L10 : $Happens(Q.CIS.Available(v2, l2), 34, \Re(34, 34)$

L11 : $Happens(R.UIS.CarRequest(c1, l2), 49, \Re(49, 49))$

L12 : $Happens(Q.CIS.FindAvailable(l2, v), 50, \Re(50, 50))$

L13 : $Happens(Q.CIS.FindAvailable(l2, v), 51, \Re(51, 51))$

L14 : $Happens(R.UIS.CarHire(c1, l2, v2), 52, \Re(52, 52))$

L15 : $Initiates(Q.CIS.FindAvailable(l2, v), equalTo(v, v2, 51))$

L16 : $Happens(R.UIS.RetKey(v2, l2), 54, \Re(54, 54))$

Figure 2: The CRS Event Log

specification, both behavioral properties and assumptions of each event space associated to a Web service composition can be formally expressed, which permits to monitor different types of deviations and inconsistencies. A monitoring scheme and a detailed algorithm were discussed in (Spanoudakis and Mahbub, 2004). However, in that work, only simple business events were considered and the event composition was not taken into account.

4.1.1 Temporal Orders

Based on the event calculus, it is easy to treat event interrelationships, as different $Happens$ can refer to the same timepoint. Given this property, we can deduce that two events (with their corresponding timepoints) can be totally ordered based on the ordering of their timepoints and the event calculus standard order relation for time. Indeed, let $e1$ and $e2$ be any two primitive events then the temporal order of these two events is defined as follows:

- $Happens(e1, t1)$ is said to be happen *before* $Happens(e2, t2)$ if $t1 < t2$.
- $Happens(e1, t1)$ is said to be *concurrent* with $Happens(e2, t2)$ if $t1 = t2$.

- $Happens(e1, t1)$ is said to be happen *after* $Happens(e2, t2)$ if $t1 > t2$.

4.1.2 Disjunction

The meaning of the disjunction is that as soon as either event occurs, the disjunctive event occurs. The disjunction of two events $e1$ and $e2$ is denoted $disj(e1, e2)$. Formally:
$disj(e1, e2)(t) = Initiates(e_2, raised, t) \longleftarrow Happens(e_1, t) \vee Initiates(e_1, raised, t) \longleftarrow Happens(e_2, t)$

4.1.3 Conjunction

The meaning of the conjunction is that two events must both occur before the conjunctive event $e3$ occurs, but that the order of occurrence, and any overlap of occurrence, is immaterial. The conjunction of two events $e1$ and $e2$ is denoted $conj(e1, e2)$. Formally:
$conj(e1, e2)(t) = Initiates(e_3, raised, t) \longleftarrow Happens(e_1, t_1) \wedge Happens(e_2, t_2) \wedge (t \geq sup\{t_1, t_2\})$

4.1.4 Sequence

Sequence is said to be strict, i.e. one event must have occurred before the next event in the sequence. Sequence of two events $e1$ and $e2$ is denoted $seq(e1, e2)$, and is defined as follows:
$seq(e1, e2)(t) = Happens(e_2, t_2) \longleftarrow Happens(e_1, t_1) \wedge (t_2 > t_1)$

It is possible that after the occurrence of $e1$, $e2$ does not occur at all. To avoid this situation, we must appropriately use definite events, such as absolute temporal event or the end of each operation's invocation.

4.1.5 Negation

The meaning of a negation is that an event $e1$ does not occur in a closed interval formed by two events $e2$ and $e3$. It is denoted by $neg(e1, e2, e3)$. Formally:
$neg(e1, e2, e3)(t) = \forall t_2 \in [t_1, t_3], Happens(e_1, t_1) \wedge Happens(e_3, t_3) \wedge \neg Happens(e_2, t_2)$

4.1.6 Temporal Iteration

A periodic event is a temporal event that occurs periodically. It is denoted by $P(e1, d, e2)$ where $e1$ and $e2$ are two arbitrary events and d is a time slot. Event $e1$ occurs for every d in the interval $[e1, e2]$. Formally:
$P(e1, d, e2)(t) = (\exists t_1)(\forall t_2 \in [t_1, t], t = t_1 + i * d \; for \; i \in N)(Happens(e1, t_1) \wedge \neg Happens(e2, t_2))$
If we have the constraint that $e1$ occurs only once $e2$ occurs, the previous definition becomes:
$P(e1, d, e2)(t) = (\exists t_1)(t > t_1)(Happens(e1, t_1) \wedge Happens(e2, t))$

Additionally, we have the possibility to combine different operators in the same time. For example, $disj(e1, seq(e2, e3))$ represents a composite event which occurs as a result of the disjunction of $e1$ and the sequence of $e2$ and $e3$. After specifying composite events and defining semantics of composition operators, it is necessary to study their relationships in order to ensure the synchronization of the exchanges and enable the communication among services either in the same event space or in different event spaces.

4.2 Event-driven Causality

From an abstract point of view, a Web service composition can be described by the types and relative order of events occuring in each atomic service. Let E_i denote the set of events occurring in service S_i, and let $E = \cup_{i=1,...,N} E_i$ denote the set of all events (either primitive or composite) in the N compound services. These event sets are evolving dynamically during exchanges between services. Events in E_i are totally ordered by the sequence of their occurrence. Thus, it is convenient to index the events of a service S_i in the order in which they occur: $E_i = \{e_{i1}, e_{i2}, e_{i3}, ...\}$. We will refer to this occurrence to specify an enumeration of E_i.

Definition 1 *Given the enumeration of E_i, the causality relation $\prec$ onto $E \times E$ is the smallest transitive relation satisfying:*
(1) If e_{ij}, $e_{ik} \in E_i$ occur in the same service S_i and $j < k$, then $e_{ij} \prec e_{ik}$.
(2) If $s \in E_i$ is a sent event and $r \in E_j$ is the corresponding received event, then $s \prec r$.

If for two events $e1$ and $e2$, neither $e1 \prec e2$, nor $e2 \prec e1$ holds, then neither of them causally affects the other. There is no way to decide which of the events $e1$ and $e2$ took place first, i.e, we do not know their absolute order. This motivates the following definition of concurrency:

Definition 2 *The concurrency relation $\parallel$ onto $E \times E$ is defined as*

$$e1 \parallel e2 \equiv \neg((e1 \prec e2) \vee (e2 \prec e1))$$

If $e1 \parallel e2$, $e1$ and $e2$ are said to be concurrent.

To illustrate the concept of causality between events, we suggest the example illustrated in figure 3. In this example, a service composition involves three atomic services: S_1, S_2 and S_3. $\{a1, a2, a3, a4, a5\}$, $\{b1, b2, b3, b4\}$ and $\{c1, c2, c3, c4\}$ are sets of operations invoked respectively by S_1, S_2 and S_3. $\{m1, m2, m3, m4, m5\}$ is the set of messages (queries) exchanged between services during the response to a customer request.
In this example, we have the following relations: $a1 \prec c1$, $a3 \prec b2$, $c2 \prec a4$, $b3 \prec a5$, $b4 \prec c4$. Certain events couples are independant or concurrent:

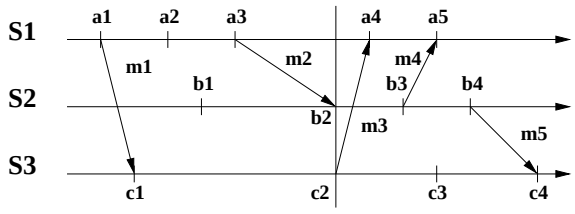

Figure 3: Event-driven Causality

$a2 \parallel b1$, $b1 \parallel c3$, $a2 \parallel c3$. The concurrency relation $\parallel$ is not transitive. For example, in Figure 3 $a3 \parallel c1$ and $c1 \parallel b2$ hold, but obviously $a3 \prec b2$. In general, an unspecified pair of events always satisfies one and only one of the following relations:

$$\forall e1, e2 : e1 \prec e2 \bigoplus e2 \prec e1 \bigoplus e1 \parallel e2$$

The symbol $\bigoplus$ is the "exclusive or" operator.

In principle, we can determine causal relationships by assigning to each event e the set of events causally related with it, we denote this set $C(e)$. $C(e)$ is defined as follows:

Definition 3 *Let $E = \cup_{i=1,...,N} E_i$ denote the set of exchanged events, and let $e \in E$ denote an event occurring during an operation invocation. $C(e)$ is defined as*

$$C(e) = \{e1 \in E \mid e1 \prec e\} \cup \{e\}$$

The projection of $C(e)$ on E_i, denoted $C_i(e) = C(e) \cap E_i$. For example, event $c4$ in Figure 3 is reachable by $c3, c2, c1, a1, b4, b3, b2, b1, a3$ and $a2$; hence, $C(c4) = \{c3, c2, c1, a1, b4, b3, b2, b1, a3, a2\}$. After defining $C(e)$, we can characterize the causality relationship between two different events $e1$ and $e2$ as follows:

1. $e1 \prec e2$ iff $e1 \in C(e2)$.
2. $e1 \parallel e2$ iff $e1 \notin C(e2) \wedge e2 \notin C(e1)$.

In order to facilitate the handling of $C(e)$, we have represented it by a vector time ($C(e) = \cup_{i=1,...,N} C_i(e)$). If $E_k = \{e_{k1}, e_{k2}, e_{k3}, ..., e_{km}\}$, then $e_{kj} \in C_k(e)$ implies that $\{e_{k1}, ...e_{kj-1}\} \subset C_k(e)$. Therefore, for each k the set $C_k(e)$ is sufficiently characterized by the largest index among its members, i.e, its cardinality. Thus, $C(e)$ can be uniquely represented by an N-dimensional vector $V(e)$ of cardinal numbers, where $V_k(e) = \mid C_k(e) \mid$ holds for the k-th component $(k = 1, ..., N)$ of vector $V(e)$. However, in some service composition scenarios, the number of atomic services is not fixed. To handle this case, we suggest to represent $V(e)$ by the set of all those pairs $(k, \mid C_k(e) \mid)$ for which the second component is different from 0. As an example, the set of causality relationships of event $c4$ in 3 can be represented by $V(c4) = [3, 4, 4]$ because the cardinality of $C_1(c4)$,$C_2(c4)$ and $C_3(c4)$ is 3,4, and 4 respectively. For notational convenience, let the supremum $sup\{v1, v2, ..., vm\}$ of a set $\{v1, v2, ..., vm\}$ of n-dimensional vectors denote the vector v defined as $v[i] = max\{v1[i], ..., vm[i]\}$ for $i = 1, ..., n$. This leads to the following definition:

Definition 4 *Let $e1$ and $e2$ denote two events, let $C(e1)$, $C(e2)$ the sets of causality relationships of these events, and let $V(e1)$, $V(e2)$ denote the corresponding vector representations, respectively. The vector representation of the union $C(e1) \cup C(e2)$ is*

$$V(C(e1) \cup C(e2)) = sup\{V(e1), V(e2)\}$$

Let $V(e)$ denote the vector time V_i which results from the occurrence of event e in service S_i. We consider $V(e)$ the vector timestamp of event e. Since a simple one-to-one correspondence between vector timestamp $V(e)$ and $C(e)$ exists for all $e \in E$, we can determine causal relationships solely by analysing the vector timestamps of the events in question.

Definition 5 *For two events $e1$ and $e2$, we have*

1. *$e1 \prec e2$ iff $V(e1) < V(e2)$.*
2. *$e1 \parallel e2$ iff $V(e1) \parallel V(e2)$.*

For two vectors $v1$ and $v2$ of dimension m, we have:

1. *$v1 \leq v2$ iff $v1[k] \leq v2[k]$ for $k = 1, ..., m$.*
2. *$v1 < v2$ iff $v1 \leq v2$ and $v1 \neq v2$.*
3. *$v1 \parallel v2$ iff $\neg(v1 < v2) \wedge \neg(v2 < v1)$.*

Furthermore, we can restrict the comparison to just two vector components in order to determine the precise causal relationship between two events if their origins services S_i and S_j are known. For two events $e1 \in E_i$ and $e2 \in E_j, e1 \neq e2$, we have

1. $e1 \prec e2$ iff $V_i(e1) \leq V_i(e2)$.
2. $e1 \parallel e2$ iff $V_i(e1) > V_i(e2) \wedge V_j(e2) > V_j(e1)$.

The meaning of this characterization is that if the "knowledge" of event $e2$ in service S_j about the events in service S_i (i.e,$V_i(e2)$) is at least as accurate as the corresponding "knowledge" $V_i(e1)$ of $e1$ in S_i, then there must exist a chain of events which propagated this knowledge from $e1$ at S_i to $e2$ at S_j, hence $e1 \prec e2$ must hold. If, on the other hand, event $e2$ is not aware of as many events in S_i as is event $e1$, and $e1$ is not aware of as many events in S_j as is $e2$, then both events have no knowledge about each other, and thus they are concurrent.

4.3 Event Based Verification

Some monitoring requirements of the car rental scenario, introduced in section 2, are specified as follows:

(A1) $Happens(R.UIS.RelKey(v, c, l), t1, \Re(t1, t1)) \wedge \neg(\exists t2) Happens(R.CSS.Depart(v, l), t2, \Re(t1, t1 + 6 * tu) \Longrightarrow (\exists t3) Happens(R.CIS.Available(v, l), t3, \Re(t1 + 6 * tu, t1 + 6 * tu))$

(A2) $Happens(R.UIS.CarRequest(c,l),t1,\Re(t1,t1)) \wedge Happens(R.CIS.FindAvailable(l,v3),t2,\Re(t1,t1+tu)) \wedge Initiates(R.CIS.FindAvailable(l,v)$,equalTo (v,v3),t2)$\Longrightarrow (\exists t3) Happens(Q.UIS.CarHire(c,l,v),t3,\Re(t2,t2+tu))$

(A3) $Happens(R.CIS.FindAvailable(l,v),t1,\Re(t1,t1)) \wedge HoldsAt(equalTo(availability(v3),not\ avail),t1-tu) \Longrightarrow \neg Initiates(R.CIS.FindAvailable(l,v), equalTo(v,v3))$

(A4) $Happens(Q.UIS.RelKey(v3,c,l),t1,\Re(t1,t1)) \wedge Happens(Q.UIS.RetKey(v3,l),t2,\Re(t2,t2) \wedge (t1 \leq t3) \wedge (t3 \leq t2) \Longrightarrow HoldsAt(equalTo(availability(v3),not\ avail),t3)$

These requirements are considered as events and are expressed in terms of the EC predicates to facilitate the detection of inconsistencies that may arise. For example, the formula A1 specifies that when CBS invokes an operation in UIS ($R.UIS.RelKey(v,c,l)$)that signifies the release of a car key to a customer, it waits for an event signifying the exit of the car from the car park for 6 time units (this message is to be sent by CSS). If the latter event does not occur, CBS invokes the operation $Available(v,l)$ in CIS to mark the relevant car as available.

We precise that in the above formulas, tu refers to the minimum time between the occurrence of two events. tu can be set by the service provider.

Given this specification and the event log of figure 2, the assumption A3 is found to be inconsistent with the expected behavior of CBS at t=54. A3 is an assumption about the behavior of the CIS service stating that when CIS executes the operation $FindAvailable$ it should not report a car as available unless this is indeed the case. The inconsistency arises because the literals L13 and L14 in Figure 2 and the literal $HoldsAt(equalTo(availability(v2),not\ avail),50)$, which is derived from the literals L9 and L16 and the assumption A4 entail the negation of A3. In this example, the inconsistency is caused by the failure of the CSS service to send an $R.CSS.Depart(v2,l2)$ event to CBS following the event $Happens(Q.UIS.RelKey(v2,c2,l2),28,\Re(28,28))$. Thus, according to A1, CBS invoked the operation $Available$ to mark the vehicle v2 as available (see the literal L10 in figure 2). Subsequently, when the operation $Q.CIS.FindAvailable(l2,v)$ was invoked in CIS (see literal L12), CIS reported v2 as an available vehicle. Note, however, that this inconsistency could only be spotted after the event signified by the literal L16 and by virtue of A4 (according to A4, a car whose key is released should not be considered as available until the return of its key).

One other case is that at t=54, the event L15 which was generated due to A2 can be detected as unjustified behavior. This is because this event can only have been generated by A2. Note that, although in this case CBS has functioned according to A2, one of the conditions of this property is violated by the literal $Initiates(R.CIS.FindAvailable(l2,v),equalTo(v,v2),51)$. This literal can be deduced from A3, the literal L13, and the literal $HoldsAt(equalTo(availability(v2),not\ avail),50)$. The latter literal is deduced from L9 and L16 and assumption A4.

5 RELATED WORK

Specifying and managing Web services interactions can be challenging due to the autonomous and heterogenous nature of services providers . Below we discuss some leading approaches that are most related to our work.

Recent emerging workflow projects such as eFlow (Casati et al., 2000), and CrossFlow (Hoffner et al., 2001) focus on loosely coupled processes. However, they lack a formal model for specifying and verifying Web services. They also do not address the semantics of composite events in Web services interactions.

Web service integration requires more complex functionality than SOAP, WSDL, and UDDI can provide. The functionality includes transactions, workflow, negotiation, management, and security. There are several efforts that aim at providing such functionality, for example, the recently released Business Process Execution Language for Web Services (BPEL4WS), which represents the merging of IBM's Web Services Flow Language (WSFL) and Microsof's XLANG, is positioned to become the basis of a standard for Web service composition. These languages are based on both SOAP, WSDL, and UDDI basic stack, are complex procedural languages, and very hard to implement and deploy. There are also some proposals such as DAML-S, which is a part of DARPA Agent Markup Language project that aims at realizing the Semantic Web concept. However, DAML-S is a complex procedural language for Web service composition.

A part from this, lots of proposals originally aim at: i) specifying web services at an abstract level using formal description techniques and reasoning on them, ii) using jointly abstract descriptions and executable languages (BPEL), iii) developing web services from abstract specifications. However, due to lack of space, we just point the reader to the related work section of paper (Ferrara, 2004).

On the other hand, to monitor the existence of specific patterns of events that indicate the violation of requirements, the existing techniques use either special purpose monitoring architectures such AMOS (Cohen et al., 1997) and FLEA (Feather et al., 1998)

that maintain event logs and offer proprietary event pattern specification languages, or store events in relational databases and deploy standard SQL querying for detecting requirement violations (Robinson, 2003). Typically, existing approaches assume that the events to be monitored are generated by special statements, which must be inserted in the code of a system for this purpose . These statements may be inserted in the source or compiled code of a system. The main drawback of instrumentation is that it has to be done manually. To alleviate this problem, Dingwall-Smith and Finkelstein (Smith and Finkelstin, 2002) have developed an aspect oriented approach, in which system providers specify instrumentation code in separate classes, and define composition rules that determine how this code is to be merged with application code.

A different approach has been developed by Robinson (Robinson, 2003). In this approach, requirements are expressed in KAOS and analyzed to identify obstacles for them. If an obstacle is observable (i.e., it corresponds to a pattern of events that can be observed at run-time), it is assigned to an agent for monitoring. At run-time, an event adaptor translates web service requests and replies expressed as SOAP messages into events and a broadcaster forwards these events to the obstacle monitoring agents which are registered as event listeners to the broadcaster.

A common pattern of the related works discussed above is that all of them suppose that all interacting partners "know" one another in advance: together they form an extended enterprize. However, our work concerns situations where partners have to dynamically find one another, after which they participate in short lived, ad hoc partnerships.

6 CONCLUSION

In this paper, we have presented a distributed event-based architecture, that is grounded on a formal meta-model stating its structural and dynamic characteristics, and proving its reliability and efficiency for coordinating Web services. The formal semantics of the event composition operators is expressed in terms of event calculus predicates. We have made use of event calculus, especially because it is one of the best known formalism for reasoning about events. The use of this formal model allows the verification of properties and the detection of inconsistencies both within and between services. Current work aims at refining and extending the approach in various directions. Indeed, we are working on the implementation of a tool to operationalize the architecture that we have developed, and we intend to establish a strong link with other process algebra, such as CSS (Milner 1989) and ACP (Bergstra & Klop 1985) in order to import process algebra specific verification techniques such as axiomatizations of behavioral equivalences.

REFERENCES

Casati, F., Ilnicki, S., Jin, L.-J., and Shan, M.-C. (2000). An open, flexible, and configurable system for service composition. In *WECWIS '00: Proceedings of the Second International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems (WECWIS 2000)*, page 125, Washington, DC, USA. IEEE Computer Society.

Cohen, D., Feather, M. S., Narayanaswamy, K., and Fickas, S. S. (1997). Automatic monitoring of software requirements. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 602–603, New York, NY, USA. ACM Press.

Feather, M. S., Fickas, S., Lamsweerde, A. V., and Ponsard, C. (1998). Reconciling system requirements and runtime behavior. In *IWSSD '98: Proceedings of the 9th international workshop on Software specification and design*, page 50, Washington, DC, USA. IEEE Computer Society.

Ferrara, A. (2004). 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. ACM Press.

Hoffner, Y., Ludwig, H., Grefen, P., and Aberer, K. (2001). Crossflow: integrating workflow management and electronic commerce. *SIGecom Exch.*, 2(1):1–10.

Kowalski, R. and Sergot, M. J. (1986). A logic-based calculus of events. *New generation Computing 4(1)*, pages 67–95.

Lemahieu, W., Snoeck, M., Michiels, C., and Goethals, F. G. (2003a). An event based approach to web service design and interaction. In *APWeb*, pages 333–340.

Lemahieu, W., Snoeck, M., Michiels, C., Goethals, F. G., Dedene, G., and Vandenbulcke, J. (2003b). Event based web service description and coordination. In *WES*, pages 120–133.

Pietzuch, P. R., Shand, B., and Bacon, J. (2003). A framework for event composition in distributed systems. In *Middleware*, pages 62–82.

Robinson, W. N. (2003). Monitoring web service requirements. In *RE '03: Proceedings of the 11th IEEE International Conference on Requirements Engineering*, page 65, Washington, DC, USA. IEEE Computer Society.

Smith, A. D. and Finkelstin, A. (2002). From requirements to monitors by way of aspects. In *Proceedings of the 1th International Conference on Aspect Oriented Software Development*.

Spanoudakis, G. and Mahbub, K. (2004). Requirements monitoring for service-based systems: Towards a framework based on event calculus. In *ASE '04: Proceedings of the Automated Software Engineering, 19th International Conference on (ASE'04)*, pages 379–384, Washington, DC, USA. IEEE Computer Society.