A Semantical Framework To Engineering WSBPEL Processes

Mohsen Rouached

Information Systems and e-Business Management manuscript No.
(will be inserted by the editor)

A Semantical Framework To Engineering WSBPEL Processes

Mohsen Rouached, Walid Fdhila, Claude Godart

LORIA-INRIA-UMR 7503
BP 239, F-54506 Vandœuvre-les-Nancy Cedex, France

Received: date / Revised version: date

Abstract Web services promise the interoperability of various applications running on heterogeneous platforms over the Internet, and are gaining more and more attention. Web service composition refers to the process of combining Web services to provide value-added services, which has received much interest in supporting enterprise application integration. Industry standards for Web Service composition, such as WSBPEL, provide the notation and additional control mechanisms for the execution of business processes in Web Service collaborations. However, these standards do not provide support for checking interesting properties related to Web Service and process behaviour. In an attempt to fill this gap, we describe a formalization of WSBPEL business processes, that adds communications semantics to the specifications of interacting Web Services, and uses a formal logic to model their dynamic behaviour thus enabling their formal analysis and the inference of relevant properties of the systems being built.

1 Introduction

There is an increasing acceptance of Service-Oriented Architectures (SOA) as a paradigm for integrating software applications within and across organisational boundaries. In this paradigm, independently developed and operated applications are exposed as (Web) services that communicate with each other using XML-based standards, most notably SOAP and associated specifications [11]. While the technology for developing basic services and interconnecting them on a point-to-point basis has attained a certain level of maturity, there remain open challenges when it comes to engineering services that engage in complex interactions with multiple other services.

A number of approaches have been proposed to address these challenges. One such approach, known as (process-oriented) service composition [4] has its roots in workflow and business process management. The idea of service composition is to capture the business logic and behavioural interface of services in terms of process models. These models may be expressed at different levels of abstraction, down to the executable level. A number of domain-specific languages for service composition have been proposed, with consensus gathering around the Business Process Execution Language for Web Services, which is known as BPEL4WS and recently WS-BPEL (or BPEL for short).

WSBPEL [2] is quickly emerging as the language of choice for Web service composition. It provides a core of process description concepts that allow for the definition of business processes interactions. This core of concepts is used both for defining the internal business processes of a participant to a business interaction and for describing and publishing the external business protocol that defines the interaction behavior of a participant without revealing its internal behavior.

WSBPEL opens up the possibility of applying a range of formal techniques to the verification of the behavior of Web services. For instance, it is possible to check the internal business process of a participant against the external business protocol that the participant is committed to provide; or, it is possible to verify whether the composition of two or more processes satisfies general properties (such as deadlock freedom) or application-specific constraints (e.g., temporal sequences,limitations on resources). These kinds of verifications are particularly relevant in the distributed and highly dynamic world of Web services, where each partner can autonomously redefine business processes and interaction protocols. In the long term, we envision an environment where an agent executing one or more business processes can autonomously discover new types of services and extends its own processes accordingly. Before being integrated in the actor's processes, discovered resources must be verified against the agent's own requirements and constraints.

Different techniques have been already applied to the verification of business processes (see, e.g.[10, 8,19,18,21]). However, current approaches do not address the issues of how to model the requirements that the WSBPEL processes are supposed to satisfy, and of how to manage the evolution of processes and requirements.

We are interested in particular in those techniques that are applied to the verification of BPEL compositions: in this case, we have to verify the behaviors generated by the interactions of a set of BPEL processes, each specifying the workflow and the protocol of one of the services participating to the composition.

A key aspect for this kind of verification is the model adopted for representing the communications among the Web services. Indeed, the actual mechanism implemented in the existing BPEL execution engines is both very complex and implementation dependent. More precisely, BPEL processes exchange messages in an asynchronous way; incoming messages go through different layers of software, and hence through multiple queues, before they are actually consumed in the BPEL activity; and overpasses are possible among the exchanged messages. On the other hand, most of the approaches proposed for a formal verification of BPEL compositions are based on a synchronous model of communications, which does not require message queues and hence allows for a better performance in verification. This synchronous mechanism relies on some strong hypotheses on the interactions allowed in the composition: at a given moment in time, only one of the components can emit a message, and the receiver of that message is ready to accept it (see e.g., [8]). However these hypotheses are not satisfied by many Web service composition scenarios of practical relevance, where critical runs can happen among messages emitted by different Web services. This is the case, for instance, when a Web service can receive inputs concurrently from two different sources, or when a service which is executing a time consuming task can receive a cancellation message before the task is completed.

Finally, another key aspect is the ability to use the verification results (deviations specifications) to ameliorate the process models is an important key to obtain a reliable compositions. For instance, how to dynamically discover suitable web services to support Web services composition has become a challenge. However, little care has been taken towards the need of execution environments that support the discovery of replacement services that become unavailable or fail to meet certain requirements at run-time.

To adress these shortcomings, we propose in this paper a rigorous approach to specifying, modelling, verifying and validating the behaviour of Web service compositions with the goal of simplifying the task of designing coordinated distributed services and their interaction requirements. More precisely, we describe a semantic framework that provides a foundation for addressing the above limitations by supporting the following functionnalities:

1. to check of requirements for WSBPEL processes. The requirements specify behavioural properties of the composition process, or assumptions about the behaviour of the composition as a whole, its constituent services and external agents who interact with it.
2. to extend the approach to include models of service choreography with multiple interacting Web services compositions, from the perspective of a collaborative distributed composition development environment. The process of behaviour analysis moves from a single local process to that of modelling and analysing the behaviour of multiple processes across composition domains.
3. to use the specifications of the violated requirements to generate queries for discovering services that could substitute for malfunctioning services or services that may become unavailable or fail to meet certain requirements.

The remainder of this paper is structured as follows. Section 2 gives an overview of WSBPEL, introduces our event driven specification, and shows how WSBPEL activities can be easily mapped to Event Calculus semantics. Semantics of WSBPEL communications are discussed and translated to the same formalism in Section 3. Section 4 is dedicated to the analysis and the verification processes. It presents the requirements specification and verification. In Section 5, we explain how the specifications of the requirements violations can be used to generate queries for discovering services that could substitute for malfunctioning services. Finally, before some discussions and conclusions in Section 7, Section 6 summarises the implementation of the approach.

2 Modelling Web Service Composition

2.1 Overview of WSBPEL

The Business Process Execution Language for Web Services (BPEL) is a Web services orchestration language that introduces a stateful interaction model providing the means for Web services to exchange sequences of messages between business partners. A BPEL process and its partners are defined as abstract WSDL services using abstract messages as defined by the WSDL model for message interaction. The major parts of a BPEL process definition consist of (1) *partners* of the business process (Web services that this 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 by a business process are categorized into *basic* activities, *structured* activities 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. Scope-related activities enable defining logical units of work and delineating the reversible behaviour of each unit.

Below, we describe the main activities and show how the Event Calulus (EC) ontology [13] is close enough to the WSBPEL specification to allow it to be mapped automatically into the logical representation.

2.2 Mapping WSBPEL Processes to EC

To analyse these processes, we firstly use a formal notation to build a model of the semantics of a WSBPEL process. The language that we use in our transfomation scheme is based on Event Calculus (EC). A summary of the semantics for EC are listed below.

EC is a first-order temporal formal language that can be used to specify properties of dynamic systems using events and fluents. An event in EC is something that occurs at a specific instance of time (e.g., invocation of an operation) and may change the state of a system. Fluents are conditions regarding the state of a system. A fluent, for example, can indicate that system variable has a specific value at given time point. Fluents in EC are initiated and terminated by events. The occurrence of an event in EC is represented by the predicate $Happens(e, t)$. The meaning of this predicate is that an instantaneous event e occurs at some time t. The initiation or termination of a fluent f due to the occurrence of an event e at time t is represented in EC by the predicates $Initiates(e, f, t)$ and $Terminates(e, f, t)$, respectively. An EC formula may also use the predicates $Initially(f)$ and $HoldsAt(f, t)$ to denote that a fluent f holds at the start of the execution of a system and that f holds at time t, respectively. The auxiliary predicate $Clipped(t1, f, t2)$ expresses whether a fluent f was terminated during a time interval $[t1, t2]$. Similarly, the auxiliary predicate $Declipped(t1, f, t2)$ expresses if a fluent f was initiated during a time interval $[t1, t2]$. In our EC-based language, events represent exchanges of messages between the services that constitute the composition process. We distinguish 5 different types of events signifying:

1. The invocation of an operation by the composition process in one of its partner services. The occurrence of these events is represented by the predicate:

   Happens(invoke_ic(PartnerService,Operation(oId,inVar)),t)

   The term $invoke\_ic(PartnerService, Operation(oId, inVar))$ represents the invocation event. In this term, $Operation$ is the name of the invoked operation, $PartnerService$ is the name of the service that provides $Operation$, oId is a variable whose value determines the exact instance of the invocation of $Operation$ within a specific instance of the execution of the composition process, and $inVar$ is a variable whose value is the value of the input parameters of $Operation$ at the time of its invocation.
2. The return from the execution of an operation invoked by the composition process in a partner service. The occurrence of these events is represented by the predicate:

Happens(invoke_ir(PartnerService,Operation(oId)),t)

The term $invoke\_ir(PartnerService, Operation(oId))$ in this predicate represents the return event. $PartnerService$, $Operation$ and oId in this term are as defined in (1). In cases where $Operation$ has an output variable $outVar$, the value of this variable at the return of the operation is represented by the predicate:

Initiates(invoke_ir(PartnerService,Operation(oId)),equalTo(outVar1,outVar),t)

This predicate expresses the initialization of a fluent variable ($outVar1$) with the value of $outVar$.The fluent $equalTo(VarName, val)$ signifies that value of $VarName$ is equal to val.

3. The invocation of an operation in the composition process by a partner service. The occurrence of these events is represented by the predicate:

   Happens(invoke_rc(PartnerService,Operation(oId)),t)

   The term $invoke\_rc(PartnerService, Operation(oId))$ in this predicate represents the invocation event. $Operation$ and oId are as defined in (1) and $PartnerService$ is the name of the service that invokes the operation. In cases where $Operation$ has an input variable $inVar$, the value of this variable at the time of its invocation is represented by the predicate

   Initiates(invoke_rc(PartnerService,Operation(oId)),equalTo(inVar1,inVar),t)

   This predicate expresses the initialization of a fluent variable $inVar1$ with the value of $inVar$.
4. The reply following the execution of an operation that was invoked by a partner service in the composition process. The occurrence of these events is represented by the predicate:

   Happens(reply(PartnerService,Operation(oId,outVar)),t)

   The term $reply(PartnerService, Operation(oId, outVar))$ in this predicate represents the reply event. In this term, $Operation$ and oId are as defined in (1), $PartnerService$ is the name of the service that invoked $Operation$, and $outVar$ is a variable whose value is the value of the output parameter of the operation at the time of the reply.
5. The assignment of a value to a variable. The occurrence of these events is represented by the predicate:

   Happens(assign(aId),t)

   The term $assign(aId)$ in this predicate represents the assignment event. aId is a variable whose value identifies the exact instance of the assignment within a specific instance of the execution of the process.

The restriction of the events that may be used in the specification of behavioural properties and assumptions to the above types guarantees that the specified properties will be monitorable without the need to instrument the services involved in the composition process.

2.3 WSBPEL specification as EC formulas

The specification of a service composition process in WSBPEL defines: (a) the *partners* of the process (i.e. the Web services that invoke the process or are invoked by it); and (b) *variables* storing the contents

of messages which are exchanged between the process and its constituent Web services. Variables can be accessed at different stages in the execution of the composition process in order to direct appropriately the subsequent flow of control. A WSBPEL service composition process is specified using two kinds of activities, namely basic and structured activities. To ensure the mapping from WSBPEL activities into EC formulas, we base our work on the transformations patterns given in [15,16].

2.3.1 Basic activities Basic activities in BPEL express primitive functions such as the invocation of operations and assignments of variable values. More specifically, a basic activity can be:

- an *invoke* activity - this activity is used to call an operation in one of the partner services of the composition process;
- a *receive* activity - this activity makes the composition process to wait for the receipt of an invocation of one of its operations by some of its partner services;
- a *reply* activity - this activity allows the composition process to respond to a request for the execution of an operation previously accepted through a receive activity;
- an *assign* activity - this activity is used to copy the value of one composition process variable to another variable;
- a *throw* activity - this activity is used to signal an internal fault; or
- a *wait* activity - this activity forces the composition process to remain *idle* for a certain period of time

Basic WSBPEL activities are transformed into EC formulas according to the transformations shown in Figure 1. As shown in this figure, an *invoke* activity that calls an operation O in a partner service P is represented in EC as a conjunction of a predicate that signifies the event of calling O at some time $t1$ ($Happens(invoke\_ic(P, O(vID, vX)), t1)$), a predicate that signifies the event of the notification of the completion of O to the composition process at some time $t2$ after $t1(Happens(invoke\_ir(P, O(vID)), t2))$, and a predicate that signifies the initiation of a fluent representing the value of the output variable vY of O upon its return ($Initiates(invoke\_ir(P, O(vID)), equalTo(Y, vY), t2)$). It should be noted that in the EC formula for *invoke*, the variable vID takes as value a unique identifier that represents the exact instance of the operation invocation in the composition process, and the variable vX takes the value that the input variable X of O has at the time of the invocation.

Similarly, a *receive* activity is represented by a predicate signifying the receipt of the invocation of an operation O of the composition process by a partner service P $(Happens(invoke\_rc(P, O(vID)), t))$, and a predicate that initiates a fluent representing the value of the input variable of O at the time of the call $(Initiates(invoke\_rc(P, O(vID)), equalTo(X, vX), t)$).

The use of the $Initiates$ predicate in both the case of *invoke* and *receive* activities is based on the principle that value bindings of variables which are visible to the WSBPEL process should be represented by fluents in order to be accessible to the reasoning process that checks the satisfiability of formulas.

The same principle underpins the representation of assignment activities in EC which, as shown in Figure 1, is a conjunction of a predicate that signifies the assignment event ($Happens(assign(vID), t1)$), a predicate that signifies the value of the source variable of the assignment $(HoldsAt(equalTo(X.a, vX.a), t1))$

and a predicate that signifies the assignment of this value to the target variable of the assignment ($Initiates(assign(vID), equalTo(Y.b, vX.a), t2)$).

A *reply* activity responding to the invocation of an operation O in the composition process is represented by a $Happens$ predicate signifying the occurrence of an event which notifies the completion of the execution of O and returns its results as the value of the output variable X of O.

$Throw$ activities are represented by the predicate $Happens(th(fN(vID, vX)), t)$. In this predicate, the term $th(fN(vID, vX))$ signifies the generation of a fault signal (this is indicated by the type th of the term) whose name is fN at the time point t. The variable vID in this term takes as value the unique identifier that is generated for the specific fault during the execution of the WSBPEL process and the variable vX takes as value the data that are attached to the fault to allow fault handlers deal with it. In cases where a *throw* activity does not specify any such data, the term representing the fault is simplified to $th(fN(ID))$.

Finally, *wait* activities are represented by a time constraint requiring that the value of the time variable of the latest predicate in the EC formula representing the activity before it (i.e., $max_t(A)$) should be less or equal to the value of the time variable of the earliest predicate in the formula representing the activity after the wait (i.e., $min_t(B)$) (the terms $EC(A, [])$ and $EC(B, [])$ in the pattern for a *wait* activity are explained below).

Sample BPEL Code	**EC Specification**

`<invoke partnerLink="P"`
`portType= "a:Pport operation= "O"`
`inputVariable= "X" outputVariable= "Y"/>`
	Happens$(invoke\_ic(P, O(vID, vX)), t1) \wedge$
	$(\exists t2)$**Happens**$(invoke\_ir(P, O(vID)), t2) \wedge$ (t1$\leq$t2)$\wedge$
	Initiates$(invoke\_ir(P, O(vID)), equalTo(Y, vID), t2)$

`<receive partnerLink="P"` `portType= "a:Pport" operation="O" variable="X"/>`	**Happens**$(invoke\_rc(P, O(vID)), t) \wedge$ **Initiates**$(invoke\_rc(P, O(vID)), equalTo(X, vXc), t)$
`<reply partner="P"` `portType = "a:Pport" operation= "O" variable="X"/>`	**Happens**$(reply(P, O(vID, vX)), t) \wedge$ **Happens**$(invoke\_rc(P, O(vID, vX)), t1) \wedge (t1 < t)$

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

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

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

Fig. 1 Mapping of Basic activities.

2.3.2 Structured activities Structured activities in WSBPEL provide the control and data flow structures that enable the coordination of basic activities into a composition process. A structured activity in WSBPEL may be

- A *sequence* activitys pecifies an ordered list of other activities that must be performed sequentially.
- A *switch* activity specifies 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.
- A *flow* activity specifies a set of two or more other activities that should be executed concurrently. A flow activity completes when all of the activities in it have completed. Synchronization dependencies

between activities inside a flow can be specified using links. Each link defines a target activity that cannot start before the completion of a source activity which is also defined by the link.
- A *pick* activity forces the composition process to wait for different events and perform different activities associated with each of these events as soon as it occurs.
- A *while* activity is used to specify the iterative execution of one or more activities for as long as a condition is true.

Figure 2 presents examples of transformation patterns which are applied to transform the previous activities into EC formulas. As in [15], in these patterns,

- actType can be any type of a basic or structured WSBPEL activity,
- $EC(A, [t1, ..., tn])$ represents the EC formulas that an activity A is transformed to after replacing the quantifiers of all universally quantified time variables in $[t1, ..., tn]$ with the existential quantifier[1],
- $min_t(A)$ 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
- $max_t(B)$ represents the time variable 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).

2.4 Illustrative example

Consider, for instance, a car rental scenario (CRS) implemented as a composition process and involves five atomic 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). Finally, a Car Payment service (CPS) is used by the CBS to take electronic payments for car rentals.

A complete WSBPEL specification of this case study and the equivalent EC representation is given in `http://www.loria.fr/~rouached/crs.zip`. In Figure **??**, we just consider a fragment of this specification in order to show how the mapping scheme can be applied. This fragment refers to the part of process that receives a request for a car and checks for available cars.

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

[1] $EC(A, [])$ indicates that there should be no changes to the quantifiers of universally quantified time variables in A and $EC(A, [*])$ indicates that all the universally quantified time variables in A should be existentially quantified in the formula resulting from the transformation.

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>	$EC(A,[])\wedge$**Happens**$(om(O(vID,vX)),t2)\wedge$ $(max_t(A)\leq$t2$\leq(max_t(A)+$T$))\wedge$ **Initiates**$(om(O(vID,vX)),equalTo(X,vX),t2)$ $\Longrightarrow EC(B,[min_t(B)])\wedge$(t1<min$_t(B)$) EC(A,[]) $\wedge\neg$**Happens**$(om(O(vID,vX)),t2)\wedge$ $(max_t(A)\leq$t2$\leq(max_t(A)+$T$))\Longrightarrow$ EC$(C,[min_t(C)])\wedge(max_t(A)+T) < min_t(C)$
<switch> <case condition=" P=v1"> <actType name="A">...</actType></case> <otherwise> <actType name="C">...</actType></otherwise></switch>	**HoldsAt**$(equalTo(P,v1),t1)\Longrightarrow$EC$(A,[min_t(A)])$ $\wedge$(t1<min$_t(A)$) $\neg$**HoldsAt**$(equalTo(P,v1),t1)\Longrightarrow$ EC$(C,[min_t(C)])\wedge$(t2$< min_t(C)$)
<sequence> <actType name="A">...</actType> <actType name="B"> ...</actType></sequence>	$EC(A,[])\Longrightarrow$EC$(B,[*])\wedge max_t(A)< min_t(B)$
<while condition="P=v1"> <actType name="A">... </actType></while>	**HoldsAt**$(equalTo(P,v1),t1)\Longrightarrow$ 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>	$EC(A,[])\wedge$**HoldsAt**$(equalTo(P,v1),t1)\wedge$ $max_t(A)<$t2$\Longrightarrow$EC$(B,[min_t(B)])\wedge$t2<min$_t(B)$ $EC(A,[])\Longrightarrow$EC$(C,[min_t(c)])\wedge max_t(A) < min_t(C)$

Fig. 2 Mapping of structured activities

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 $findCar$.

3 Semantics of WSBPEL communication

A detailed translation of WSBPEL to EC models is given in previous sections, however, we add to this the semantics for how to translate the connectivity and communication between activities of the partner processes rather than from a single process focus. To commence this we require a process to analyze which activities are partnered in the compositions. For example, invoke from the UIS service (a rental request) will be received by the CRS process (receive a rental request). Equally the CRS invokes activity, to check the availability of cars by contacting CIS, will be aligned with receive in the CRS process. In WSBPEL, the communication is based upon a protocol of behavior for a local service. However, the partner communication can concisely be modeled using the synchronous event passing model, described in [14]. The Sender-Receiver example discussed uses *Channels* to facilitate message/event passing between such a sender and receiver model. The representation of a channel in WSBPEL is known as a *port*. The significant element of this discussion used in our process is that of synchronization of the invoking and

Part of WSBPEL composition process for CRS

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

EC formulas

$Happens(invoke\_rc(UIS, CarRequest(oID1)), t1) \wedge$
$Initiates(invoke\_rc(UIS, CarRequest(oID1)), equalTo(Req.Loc, vReq.Loc), t1) \wedge$
$Initiates(invoke\_rc(UIS, CarRequest(oID1)),$
$equalTo(Req.CId, vReq.CId), t1) \Longrightarrow$
$(\exists t2)(\text{t1}<\text{t2}) \wedge Happens(assign(aID), t2) \wedge (\exists t3)(\text{t2}<\text{t3}) \wedge$
$Initiates(assign(aID), equalTo(Q.Loc, vReq.loc), t3) \Longrightarrow$
$(\exists t4) Happens(invoke\_ic(CIS, FindAvailable(oID2, vQ)), t4) \wedge (\text{t3} \leq \text{t4}) \wedge$
$(\exists t5) Happens(invoke\_ir(CIS, FindAvailable(oID2, vQ)), t5) \wedge (\text{t4} \leq \text{t5}) \wedge$
$Initiates(invoke\_rc(CIS, FindAvailable(oID2, vQ)), equalTo(Res, vRes), t5)$

Fig. 3 Example of EC formulas extracted from the WSBPEL process for CRS.

receiving events within compositions between ports and whether this has been constructed concurrently (*flow* construct in WSBPEL) or as a sequence (*sequence* construct in WSBPEL) of activities.

In the following, we seek to further our modelling of WSBPEL interactions through two viewpoints. Firstly, we examine the interactions within the choreography layer of Web service compositions collaborating in a global goal. Secondly, through further behaviour analysis, we model the interaction sequences built to support multiple-partner conversations across enterprise domains and with a view of wider goals.

3.1 Modelling interactions

To model interacting Web service compositions there is clearly a need to elaborate our analysis of implementations by linking compositional interactions based upon:

- activities within the process
  - identifying invocation style (rendezvous or request only)
  - identifying and recording the points at which interaction occurs
- the abstract interface
  - linking between the private process activities and the public communication interface declared in the abstract WSDL service description.

To model the semantics of linking interactions between processes requires a mapping between activities in each of the processes translated (using the translation rules described in Section 2.2) and building an event port connector for each of the interaction activities linking *invoke* (input) with receives, and replies (output) and with the returned message to an *invoke*.

The physical linking of partnerlinks, partners and process models is undertaken as follows. For each invocation in a process, a messaging port is created. WSBPEL defines communication in a synchronous messaging model. WSBPEL process instance support in the specification specifies that in order to keep consistency between process activities, a synchronous request mechanism must be governed. The synchronous model can be formed by the following process.

Figure 1 Interactions Modelling Algorithm

```
For each composition process
    For each process invoke service activity
        Get invoke activity local partner
        Lookup partnerlink using local partner
        Get porttype using partnerlinktype
        For each process interface definition
            Lookup porttype using activity portype
            Store matching partner
            Lookup partner operation
        End For
        If invoke activity is in rendezvous style
            Add invokeoutput action to activity model
            Build reply-invokeoutput port
        End If
        Build invoke-receive connector partner labelling
    End For
End For
```

For every composition process selected for modelling we extract all the interaction activities in this process. As mentioned previously, interaction activities are service operation invocations (requests), receiving operation requests and replying to operation requests. In addition to an invocation request, we also add an invocation reply to synchronise the reply from a partner process with that of the requesting client process. The list is then analysed for invocation requests, and for each one found a partner/port lookup is undertaken to gather the actual partner that is specified in a partnerlink declaration. To achieve this, a partner list is used and the partner referenced in the invocation request is linked back to a partnerlink reference. The partnerlink specifies the porttype to link operation and partner with an actual interface definition. To complete the partner match, all interface definitions used in composition analysis are searched and matched on porttype and operation of requesting client process. This concludes the partner match. A port connector bridge is then built to support either a simple request invocation (with no reply expected) or in "rendezvous" style, building both invoke/receive and reply-invokeoutput models. This supports the model mapping. The sequence is then repeated for all other invocations in the selected composition process, and then looped again for any other composition processes to analyse. We therefore specify an algorithm that will enable mechanical linking between activities, partners and process compositions. The algorithm supports a mechanical implementation of linking composition processes together

based upon their interaction behaviour. Two build phases are required as part of the algorithm, being that of building a reply-invokeoutput port and invoke-receive connector between partnered processes.

In summary, the algorithm described provides a port connector based implementation of the communication between two partner processes. Where multiple partner communication is undertaken in a composition, a port connector is built between each instance of a message(and optionally a reply if used in rendezvous interaction style).

3.2 Event Invocations Connectors

To build connected composition interactions, port connector channels are used for each of the invocation styles between two or more partnered compositions. The algorithm is used from the viewpoint of a process composition at the "centre of focus", that is, the one in which initial process analysis is being considered. The interface of subsequent partner interactions is used in the algorithm to obtain a link between two partners and an actual operation. For example in Figure 2, two WSBPEL process interact using both a request only invocation (Channel A) and a Rendez-vous style (Channel A and B).

Figure 2 Channels and Interaction Activities of Web Service Compositions

Our model of interactions using channels is based upon the interaction state and not on the messaging architecture used for transport. In this way, we do not consider synchronous against asynchronous messaging models for modelling the communication flow between compositions. The model produced from analysis of the compositions is from the viewpoint of the composition performing as part of a role in choreography. This makes the model an abstract view of interactions for the purpose of linking invocations and not on the actual order of messages received by the process host architecture (synchronous and asynchronous messaging models for Web services can be referred to in [10]).

Figure 3 Event Invocation Connecters: example

3.2.1 Request only invocation (Channel A) Web Service compositions specified with the *invoke* construct and only an *input* container attribute declare an interaction on a request only basis (there is no immediate reply expected). More generally this requirement is for a reliable message invocation without any output response from the service host (other than status of receiving the request). The model for this is illustrated in definition **??**.

$\forall$t1: Happens (invoke_$ic(PartnerService, Operation(oId, inVar)), t1) \Longrightarrow ((\exists$t2)Happens (invoke_$rc(PartnerService, Operation(oId)), t2)) \wedge$Initiates (invoke_$rc(PartnerService, Operation(oId)), equalTo(inVar1, inVar), t2)) \wedge$(t1 ¡ t2).

$\forall$t2: Happens (invoke_$rc(PartnerService, Operation(oId)), t2)) \wedge$Initiates (invoke_$rc(PartnerService, Operation(oId)), equalTo(inVar1, inVar), t2)) \Longrightarrow ((\exists$t1)Happens (invoke_$ic(PartnerService, Operation(oId, inVar)), t1) \wedge$(t1 ¡ t2).

3.2.2 Rendezvous style invocation (Channels A and B) "Rendezvous" (Request and Reply) invocations are specified in WSBPEL with the *invoke* construct, with both *input* and *output* container attributes. To model these types of interactions, we use a generic port model for each process port. A synchronous event model in Web services compositions (such as WSBPEL) requires an additional activity of an "input_output" to link a reply in a partnered process to that of the caller receiving the output of the invoke, however, this is necessary only if the invocation style is that of rendezvous. The event synchronisation for this port model is shown in definition**??**.

$\forall$t1:time) Happens (invoke_$ic(PartnerService, Operation(oId1, inVar)), t1) \Longrightarrow ((\exists$t2)Happens (invoke_$rc(PartnerService, Operation(oId1)), t2)) \wedge$Initiates (invoke_$rc(PartnerService, Operation(oId1)), equalTo(inVar1, inVar), t2)) \wedge$(t1 ¡ t2).

$\forall$t2: Happens (invoke_$rc(PartnerService, Operation(oId)), t2)) \wedge$ Initiates (invoke_$rc(PartnerService, Operation(oId)), equalTo(inVar1, inVar), t2)) \Longrightarrow ((\exists$t1)Happens (invoke_$ic(PartnerService, Operation(oId, inVar)), t1) \wedge$(t1 ¡ t2).

$\forall$t3:Happens (reply (PartnerService, Operation (oId2, outVar)), t3)$\Longrightarrow$ ((\exists$t4)Happens (invoke_$ir(PartnerService, Operation(oId2)), t4)) \wedge$Initiates (invoke_$ir(PartnerService, Operation(oId2)), equalTo(outVar1, outVar), t4)) \wedge$(t3 ¡ t4).

$\forall$t4:Happens (invoke_$ir(PartnerService, Operation(oId2)), t4)) \wedge$Initiates (invoke_$ir(PartnerService, Operation(oId2)), equalTo(outVar1, outVar), t4)) \Longrightarrow ((\exists$t3)Happens (reply (PartnerService, Operation (oId2, outVar)), t3)$\wedge$(t3 ¡ t4).

3.2.3 Mapping Process Activities to Port Connectors The next step in the port connector modelling process is to map the activities of the WSBPEL process to the port connector activities. This is achieved using the semantics of WSBPEL for the interaction activities discussed earlier and replacing the port connector activities appropriately. The *invoke* activity in BPEl4WS is mapped from the client process to the *invoke_input* action of the port connector this represents the initial step of a request between web service partners. The associated receiving action of the WSBPEL partner process is mapped to the receive activity in the port connector. The reply from the partner process to the client process is mapped to the reply in the partnered process. Both *receive* and *reply* activities in the WSBPEL are discovered as part of the interface analysis described in Section 3.1. Figure 1 lists the mapping explained here.

WS Interaction	Port Action	BPEL4WS Actions (Example)
Invoke (client)	Invoke_input	invoke_client_CRS_CarRequest
Receive(Partner)	Receive	Receive_client_CRS_CarRequest
Reply(Partner to client)	Reply Invoke_output	reply_CRS_client_CarRequest output_CRS_client_CarRequest

Table 1 Mapping Process Activities to Port Connectors

4 Analysis and Verification

We come back now to our example introduced in Section 2.4. In a typical situation, CRS receives a car rental request from a UIS service and checks for the availability of cars by contacting CIS services. If an available car can be found at the requested location, CBS books the car rental through a CIS service, and takes payment through the CPS service. When cars move in and out of car parks, respective CSS services inform CBS, which subsequently invokes operations in CIS services to update the availability status of the moved car. However, 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.

4.1 Requirements specification

In this section, we present how to use the event specification and the WSBPEL mapping scheme to specify the requirements to be monitored. The monitorable properties may include behavioural properties of the composition process and/or assumptions that service providers can specify in terms of events extracted from this specification. The behavioural properties are specified in terms of: (i) events which signify the invocation of operations in different services or the composition process and responses generated at the completion of these executions, (ii) the effects that these events may have on state variables of the composition (e.g., assignment of values), and (iii) conditions about the values of state variables at different time instances. The events, effects and state variable conditions are restricted to those which can be observed during the execution of the composition process. Assumptions are additional constraints about the behaviour of individual services in the execution environment. These constraints are specified by system providers and must be expressed in terms of events, effects and state variable conditions which are used in the behavioural properties directly or indirectly.

The behavioural properties of individual Web services are extracted automatically from their WSDL descriptions and the WSBPEL specification of their composition process. Following the extraction of such properties, assumptions are specified by system providers in terms of event and state condition literals that have been extracted from the WSBPEL specification and, therefore, their truth-value can be established during the execution of the composition process. This specification can be amended by service providers, who can also use the atomic formulas of the extracted specification to additional assumptions about the composition requirements if appropriate.

Several requirements concerning the CRS case study were discussed in [23]. Here we just focus on the bahaviour of the CPS service to clarify our ideas throughout the rest of the paper.

An example of a requirement for the behaviour of the payment service of CRS specified using the specification introduced so far is illustrated in Figure 2.

$(R_{CPS}) : Happens(invoke\_ic(CPS, capture(oID1, cID, a)), t1) \wedge$
Happens(invoke_ir(CPS,capture(oID1)),t2)$\wedge(t1 < t2 < t1 + 5 * t_m)^2 \wedge$
Initiates(invoke_ir(CPS,capture(oID1)),equalTo(CRes,"OK"),t2)$\wedge$
Happens(invoke_ic(CPS,capture_reverse(oID2,cID,a)),t3)$\wedge(t2, t3 < t2 + 50 * t_m)$
$\Longrightarrow (\exists t4)(t3 <= t4) \wedge (t4 \leq t3 + 5 * t_m) \wedge$
Happens(invoke_ir(CPS,capture_reverse(oID2)),t4) $\wedge$
Initiates(invoke_ir(CPS,capture_reverse(oID2)),equalTo(RRes,"OK"),t4)

Table 2 Example of CRS Requirement

The requirement (R_{CPS}) indicates that if following a request for getting a payment of an amount a from a customer card cID sent from CBS to CPS at time $t1$ (see literal $Happens(invoke\_ic(CPS, capture(oID1, cID, a)), t1)$) and the acceptance of this request at time $t2$ ($Happens(invoke\_ir(CPS, capture(oID1)), t2) \wedge (t1 < t2 < t1 + 5 * t_m) \wedge Initiates(invoke\_ir(CPS, capture(oID1)), equalTo(CRes, "OK"), t2)$). CBS sends another request to CPS for reversing the payment within 50 time units (see literal $Happens(invoke\_ic(CPS, capture\_reverse(oID2, cID, a)), t3) \wedge (t2, t3 < t2 + 50 * t_m)$), CPS should reverse the payment and confirm this to CBS within 5 time units (see literals $(\exists t4)(t3 \leq t4) \wedge (t4 \leq t3 + 5 * t_m) \wedge Happens(invoke\_ir(CPS, capture\_reverse(oID2)), t4) \wedge Initiates(invoke\_ir(CPS, capture\_reverse(oID2)), equalTo(RRes, "OK"), t4)$). In this formula, the variable t_m refers to the minimum time between the occurrence of two events.

4.2 Requirements driven verification

Once both behavioural properties and additional assumptions are formalized, we move to check the satisfiability of a requirement against the recorded behaviour of the composition process. More specifically, the check that is carried out is whether the set of the recorded events that have been generated by the execution of the composition process entail the negation of a requirement[3]. To do this we propose to annotate the execution log with semantical information to enable reasoning on recorded events for checking the consistency of the above properties and gathering reasons about deviations that may arise. This means that given an event log and an EC property (requirement), we want to check whether the observed behaviour matches the (un)expected/(un)desirable behaviour.

To illustrate this, we come back to our example introduced in Section 2.4. As mentioned in [26], to offer higher flexibility to their customers, the providers of CBS decide to allow customers to cancel completed rental transactions if they are not happy with the cars that they have rented up to 40 minutes

[3] Formally, this is equivalent to proving that $\neg(\forall s.Spec(s) \Longrightarrow \neg R(s))$ where $Spec(s)$ is the specification of a service s and $R(s)$ is the requirement about s.

following the completion of a car rental. To support this feature, CBS providers update the UIS services deployed by CBS in order to be able to handle the relevant requests. They also check the specification of the CPS service that CBS deploys and, as they find that the specification does not always entail the negation of the requirement, they decide to continue using the service. However, they also start observing the behaviour of CPS to check if it always satisfies the requirement during the execution of the composition. For a certain period of time, CPS behaves in line with the requirement but at some point a violation (i.e., a denial to return a payment) is observed for a car rental cancellation request. For example, the negation of R_{CPS}, is entailed by the set of the recorded events of the composition process shown in Figure 4.

Figure 4 The CRS Event Log

L1 : Happens(invoke_rc(UIS,carRequest(op1)),49)
L2 : Initiates(invoke_rc(UIS:carRequest(op1)),equalTo(Park,p2)49)
L3 : Happens(invoke_ic(CIS,findAvailable(op2,p2)),50)
L4 : Happens(invoke_ir(CIS,findAvailable(op2)),51)
L5 : Initiates(invoke_ir(CIS,findAvailable(op2)),equalTo(Car,veh2),51)
L6 : Initiates(invoke_ir(CIS,findAvailable(op2)),equalTo(Price,10000),51)
L7 : Happens(invoke_ic(UIS,confirm(op3,veh2,10000)),52)
L8 : Happens(invoke_ir(UIS,confirm(op3)),55)
L9 : Initiates(invoke_ir(UIS,confirm(op3)),equalTo(CardNo,5555),55)
L10: Happens(invoke_ic(CPS,authorise(op4,5555)),52)
L11: Happens(invoke_ir(CPS,authorise(op4)),58)
L12: Initiates(invoke_ir(CPS,authorise(op4)),equalTo(Res,“OK”),58)
L13: Happens(invoke_ic(CPS,capture(op5,5555,10000)),59)
L14: Happens(invoke_ir(CPS,capture(op5)),61)
L15: Initiates(invoke_ir(CPS,capture(op5)),equalTo(CRes,“OK”),61)
L16: Happens(reply(UIS,carHire(op6,abc1)),62)
L17: Happens(invoke_rc(UIS,cancelRequest(op7)),92)
L18: Initiates(invoke_rc(UIS,cancelRequest(op7)),equalTo(Id,abc1),92)
L19: Happens(invoke_ic(CPS,capture_reverse(op8,5555,10000)),93)
L20: Happens(invoke_ir(CPS,capture_reverse(op8)),98)
L21: Initiates(invoke_ir(CPS,capture_reverse(op8)),equalTo(RRes,“AgedOff”),98)

This event log fragment shows exchanges of messages related to the execution of operations that realise a car renting transaction. More specifically, following the receipt of request for a car rental at car park $p2$ (events L1-L2), CBS contacts the CIS service to find an available car (event L3). CIS confirms the availability of $veh2$ at $p2$ and reports that $veh2$ would cost 10000 (events L4-L6). Subsequently, CBS contacts the UIS service to get a confirmation of the transaction by the customer and a credit card number to charge the transaction on (event L7). Following the confirmation of the transaction and the provision of a credit card number (events L8-L9), CBS contacts the CPS service to authorize the payment for the transaction (events L10-L12) and get this payment (events L13-L15). Following this, it confirms

the car rental with a reference number "abc1" to the customer through the UIS service (event L16). Subsequently, CBS receives a cancellation request for the transaction (events L17-L18) and CPS to reverse the payment that took earlier (event L19). CPS, however, refuses the reversal (events L20-L21). Formally, the violation of R_{CPS} is detected as the entailment of R_{CPS} by the recorded events. More specifically, the negation of R_{CPS} is entailed by the events L13, L14, L15 and L19 and the literal: $(\forall t4)(93 \leq t4) \wedge (t4 \leq 98) \neg Initiates(invoke\_ir(CPS, capture\_reverse(op8)), equalTo(RRes, "OK"), t4)$. The latter literal is established at $T = 98$ when the execution of the operation *capture_reverse* of the CPS service returned the value AgedOff (see the event L21) and, due to the principle of the negation as failure, the checker can establish that no $Initiates(invoke\_ir(CPS, capture\_reverse(op8)), equalTo(RRes, "OK"), t4)$ event occurred between times 93 and 98.

For the same case study, several others requirements verifications are discussed in [23]. Our focus now is on how the specifications of these vialations can be used to support the discovery of replacement services that become unavailable or fail to meet certain requirements at run-time.

5 The Service discovery Process

During the execution of CRS, a composed service used by CRS may become unavailable or fail to meet certain requirements as described in the scenario below: A customer requests CRS to find if there are cars available for renting at a specific car park. After confirming the availability of cars at the specific location and their prices, and checking with the customer whether he/she wants to proceed with the rental, CRS contacts payment service CPS to get a payment for the rental. CPS, however, is not available. To continue its operation in this scenario, CRS needs to find another payment service to replace CPS. A valid candidate service should : (i) provide all the operations that CPS provides and are used by CRS, and (ii) satisfy any other behavioural requirement that CPS meets.

One other scenario can occur. Suppose that the payment process must be handeled under a given composition requirement. For a certain period of time, CPS behaves in line with the requirement but at some point a violation (i.e., a denial to return a payment) is observed for a car rental cancellation request. In this case, to ensure uninterrupted availability to its customers, CRS should continue using CPS while trying to find another payment service to replace it at run-time. The new service should : (i) guarantee the satisfiability of the violated requirement, and (ii) offer the behaviour of CPS that is used by CRS.

Dealing effectively with the above scenarios requires a run-time service discovery framework to be able to:

1. keep the composition process alive when a constituant service fails and until a replacement service is found;
2. monitor the compliance of the run-time behaviour of the services involved with specific requirements;
3. create queries based on violated requirements to find alternative services that can satisfy these requirements; and
4. create queries based on the operations and quality-of-service requirements of unavailable services to find services to replace them.

This section discusses these elements and shows the ability to combine the components for monitoring the compliance of Web services compositions with specified requirements, and the components for discovering services at run-time. It presents the use of the specifications of the violated requirements to generate queries for discovering services that could substitute for malfunctioning services.

5.1 Query specification

As mentioned so far, our framework uses the specifications of the violated requirements to generate queries for discovering services that could substitute for malfunctioning services. These queries incorporate both structural and behavioural aspects of the required services.

The structural part of a query specifies the interface of the required service and possibly a categorisation of it. The service interface is specified in WSDL. This information is taken from the local registry of the components services maintained by the composition manager. The behavioural part of a query is specified as a conjunction of paths. A path is a list of typed elements with a variable number of arguments. An element may be of type:

- *Send* representing a message that is dispatched by the composition process
- *Receive* representing a message that is received by the composition process
- *State* representing unknown states in the state machine of a service to be located, or
- *Predicate test* representing a condition that must be true at a specific point within a path.

The ordering of elements within a path indicates the temporal order in which the elements must occur. A path element can be negated when it is necessary to specify that the element should not be present at the specific point within the path.

Figure 5 presents an example of a query for finding a service to replace the payment service that becomes unavailable in Scenario 1 (see Section **??**).

Figure 5 Example of queries for finding services required in Scenario 1 and 2

Query 1	**Query 2**
Structural-part (financial transaction processing,CPS.wsdl) **Behavioural-part**([[“state”, Service — Initial] , [“send”,CRS, Service,“authorise”,“cID”,“a”], [“receive”,CRS, Service,“authorise_response”,“OK”], [“send”,CRS,Service,“capture”,“cID”,“a”], [“receive”,CRS,Service,“capture_response”,“OK”], [“state”,Service—Final1]] ∧ [[“$state''$, $Service|Initial$], [“$send''$, CRS, $Service$, “$authorise''$, “cID'', “a''], [“$receive''$, CRS, $Service$, “$authorise\_response''$, “OK''], [“$send''$, CRS, $Service$, “$authorise\_reverse$, “cID'', “a''], [“$state''$, $Service|Final2$]] ∧ ...)	**Structural-part**(financial transaction processing,CPS.wsdl) **Behavioural-part**(... ∧ [[“$state''$, $Service|Initial$]], [“$send''$, CRS, $Service$, “$capture''$, “cID'', “a''], [“$receive''$, CRS, $Service$, “$capture\_response''$, “OK''], [“$after''$, $Service$, “$50''$], [“$send''$, CRS, $Service$, “$capture\_reverse''$, “cID'', “a''], [“$receive''$, CRS, $Service$, “$capture\_reverse\_response''$, “$OK''$], [“$state''$, $Service|Final3$], ...)

The paths in the query correspond to the messages exchanged between CRS and the payment service in the different execution paths of the WSBPEL composition process of CRS. The first path in the query represents an execution path in which the composition process requests the execution of the operation

authorise in the payment service to authorise a payment ([“send”,CRS,Service,“authorise”,“cID”,“a”]), receives a response that indicates the successful authorisation of the payment ([“receive”,CRS, Service,“authorise_response”,“OK”]), requests the execution of the operation *capture* in the payment service to get the payment ([“send”,CRS,Service,“capture”,“cID”,“a”]), and receives a response that indicates the successful completion of the payment ([“receive”,CRS,Class,“capture_response”,“OK”]).

The behavioural part of the query is formed by paths constructed from the WSBPEL process and from the specification of the requirement that has been violated. The construction from the WSBPEL composition process is ensured by using special transformation rules. As an illustration of how queries can be constructed from WSBPEL processes, consider the fragment of the CRS process shown in Figure 6.

Figure 6 A BPEL fragment of the CRS process

```
<invoke xmlns="http://schemas.xmlsoap.org/
   ws/2003/03/business-process/"
         partnerLink="PaymentSystem"
         portType="ps:PaymentSystem"
         operation="authorise"
         inputVariable="authData"/>
```

To transform the invocation of the operation *authorise* in the above WSBPEL code fragment into a *send* element in the query paths shown in Figure 5, the XML element $< invoke >$ is replaced by the element type *send*, and the value of the attributes *operation* and $inputVariable$ of this element are represented as arguments of this element (i.e., *authorise* and the actual parts of the input variable *authData*, namely cID and a). Note that some of the element arguments are variables. Variables are signified by strings not enclosed in quote marks (“”) within a query path element. In Figure 5, for example, CRS is a variable representing a client of the payment service, $Service$ is the unknown service whose state machine should be matched with the query, and $Initial$, $Final1$ and $Final2$ are unknown states in this machine that represent the boundary states of the transition path in the state machine that will be matched with the query (if any).

An example of a query for finding the service required in Scenario 2 is presented in Figure 5.

In this scenario, we would like to discover a service that supports the cancellation of payment captures within 50 minutes after the completion of the capture (sends the respond $capture\_reverse\_response(OK)$) following the invocation of the operation $capture\_reverse$ in it provided the operation was invoked up to 50 minutes after the payment. The structural part of this query is the same as that of Query 1 in Figure 5. The behavioural part of the query is formed by paths constructed from the WSBPEL process (as in Query 1) and from the requirement R_{CPS} that has been violated in this scenario as we discussed in Section 4.2. In Figure 6, we only present the path corresponding to the violated requirement.

The transformation of the violated requirements into query paths is also based on rules. A rule transforms EC predicates to *send* or *receive* path elements based on the type of the event calculus predicate. For example, the first condition of R_{CPS} in Figure 2 is transformed into a path element

signifying the invocation of the *capture* operation in the beginning of the path of Query 2. Time conditions in EC formulas are transformed into predicate test elements. For example, the range of the variable $t3$ in (R_{CPS}) is transformed into the path element ["after",Service,50] in Query 2. The predicate test $after$ in this case specifies that between the element that precedes the $after$ element in the query path and the element that follows it there shouldn't be a delay of more than 50 time units.

5.2 Matching

The matching between a service requested by a query and the services described in a service registry is based on matching the data types and signatures of service operations and the behavioural models of services. The matching between the data types is referred to as *structural matching*. The matching between behavioural models of services and the workflow process of a composition is referred to as *behavioural matching*.

Both the structural and behavioural matching use internal representations of the data types and behavioural models of the services. More specifically, data types are internally represented as type graphs and behavioural service models are internally represented as state machines. These internal representations are generated automatically from the original descriptions of services expressed in WSDL (service data types) and WSBPEL (service behavioural models). Our choice to use these internal representations of service structural and behavioural descriptions has been motivated by the genericity of type graphs and state machines that we deploy and their ability to provide common canonical representations for a wide range of data and behavioural service representations.

5.2.1 Example To illustrate the discovery process, consider the example of a state machine representing the behavioural specification for a payment service (CPS) shown in Figure 7 [26]. According to this state machine, a payment can be taken through a number of interactions with a client (in our case, the CRS system). In the simplest scenario, the CPS service authorises the payment (see the transition from the state *Authorisation* to the state *Init* in the state machine) and after the client confirms the customer's identification and, possibly, a signature is obtained, it captures (*deposits*) the funds (transition to the state *Deposited* in Figure 7). In the case of Query 2, consider first the situation when a candidate

Figure 7 State machine of a payment service

payment service CPS1 that behaves exactly as specified by the state machine in Figure 7 is discovered using service categorisation information. The matching of the path in Query 2 with the state machine in Figure 7, however, would fail. This happens because in accordance to the UML state machine semantics, given the delay of 50 minutes specified in Query 2, the transition $after(30)$ from the state *Deposited* to the state $AgedOff$ will be taken, thereby, preventing the transition from the state *Deposited* to the state *Reverse* (i.e., the execution of the operation $capture\_reverse$). Thus a path cannot be constructed to match the path in Query 2 and service CPS1 will be rejected.

Suppose, however, that there is a payment service CPS2 with a state machine like the one shown in Figure 7, but where transition after from *Deposited* to $AgedOff$ is specified to occur 60 minutes after the entrance to the former state (i.e., after the completion of the execution of the operation *capture*). In this case, the path shown in Figure 8 can be constructed to match the path in Query 2 and, therefore, the overall query will succeed. Note that in Figure 8, element of type $after$ in Query 2 is deleted because the elapsed time in transition [$after(60)$] is greater than 50 minutes. Thus, the transition from *Deposited* to *Reverse* is taken, and a valid path (paths) is constructed. Service CPS2 would be selected as a candidate substitution for CPS. In this case, CPS could be subsequently replaced by CPS2 in the composition process of CRS using the instrumentation techniques discussed in [26]. In the case when the state machine of a service does not have an $after$ transition, the path in Figure 8 is also valid and such service also meets the requirements.

Figure 8 A path transformation for Query 2

$Path_s$ =[
[“state”,“Payment”,“Approved”,“Ready for capture”,“Ready”],
[“send”,“M”,“Payment”,“capture”,“cID”,“a”],
[“guard”,“Payment”,“OK”],
[“receive”,“M”,“Payment”,“capture_response”,“OK”],
[“send”,“M”,“Payment”,“capture_reverse”,“cID”,“a”],
[“receive”,“M”,“Payment”,“capture_reverse_response”,“OK”],
[“state”,“Payment”,“Approved,“Ready for capture”,“Reversed”]]

6 Implementation

6.1 Verification process

Our approach has been implemented in Java and has used the engine bpws4j[4] and log4j[5] to generate logging events. It incorporates a requirements (behavioural properties and assumptions) editor, an event collector, a BPEL2EC tool, and a deviation viewer. The BPEL2EC tool is built as a parser that can automatically transform a given WSBPEL process into EC formulas according to the transformation scheme. 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 behavioural specification of this composition in Event Calculus. The description of this implementation is beyond the scope of this paper and may be found in [23]. We just focus here on the verification process.

As a verification back-end, we have used an automated induction-based theorem prover SPIKE [28]. SPIKE was chosen for the following reasons: (i) its high automation degree, (ii) its ability on case

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

analysis, (iii) its *refutational completeness* (to find counter-examples), (iv) its incorporation of *decision procedures*(to automatically eliminate arithmetic tautologies produced during the proof attempt[6]).

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.

The specification of the ingredients of our encoding, the behavioural properties, the log, and the EC axiomatisation can not be detailed here due to lack of space and is discussed in [25].

Following this EC specification, we build an algebraic specification from it. Once building this specification, we can check all behavioural properties by means the powerful deductive techniques (rewriting and induction) provided by SPIKE .

Given a conjecture (requirement) 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. Then, 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. Due to lack of space, verification results cannot be presented here but can be found in `http://www.loria.fr/~rouached/crs.zip`. Finally, details about the matching algorithms and the implementation of the discovery tool are presented in [24].

7 Related Work

One of the key aspects of Service-Oriented Architectures (SOAs) is the ability to rapidly build new applications and services by assembling the existing ones. Developing techniques and approaches to facilitate such an assembly process automatically has been widely researched in both academia and industry. However, most of the existing approaches ignore or oversimplify multiple aspects and characteristics that are specific to Web services and SOA, and are vital for the success of service-oriented computing paradigm, in general. Some of the important aspects include representation of functional and behavioral properties of services, ability to handle failure of composition, service adaptation during composition, analysis of service substitution, and handling semantic heterogeneity in service specifications. Without addressing these issues in an uniform manner, the present techniques and tools can only operate in a restricted setting, and hence cannot be applied to a wide-range of realistic problems and application domains.

Several attempts have been made to capture the behavior of BPEL [1] in some formal way. Some advocate the use of finite state machines [7], others process algebras [6], and yet others abstract state

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

machines [5] or Petri nets [20,17,27]. Another branch of work concerning the area of "adapting Golog for composition of semantic web services" is carried out by Sheila McIlraith and others [**?**]. 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 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.

With respect to Web service analysis approaches, in particular BPEL processes, several works were described. The closest to our approach are the tools presented in [8] and [10]. The first one, namely LTSA-BPEL4WS, is based on the process algebra formalisms and allows for the analysis of basic properties of WSBPEL specifications, such as safety and progress checks. The tool currently does not support the analysis of composition of several WSBPEL specifications and was unable to handle complex specifications as those of the CRS case study. Moreover, it is based on the synchronous communications model thus being restrictive with respect to the set of systems it is able to correctly analyze. On the contrary, the WSAT tool [10] is equipped with the synchronizability analysis techniques that allow to check whether the behavior of the system is valid under synchronous communications semantics. However, the techniques currently provided allow only for partial analysis. That is, if the analyzed system does not pass the check it is not necessarily the case that the system is not synchronizable. The reason is that the synchronizability analysis is based on sufficient but not necessary conditions and that it ignores the information appearing in transitions conditions thus leading to spurious violations of the synchronizability. Also the provided techniques do not exceed the limits of the synchronizability analysis, and therefore do not allow for the reasoning about more sophisticated communication models.

In [22,12], the analysis is performed basing on Timed Automata and inspired by process algebra notations. All these approaches exploit only the synchronous communication semantics, thus ruling out a certain class of composition scenarios, which are important in practice and can be managed in the proposed framework. On the contrary the aim of our approach is to attempt to find an appropriate specification for the given composition, under which it behaves correctly.

In [8], process algebras are exploited to verify BPEL processes. More precisely, that approach allows for the analysis of basic properties of BPEL specifications, such as safety and progress checks. The approach is based on the synchronous communications model and therefore is very restrictive with respect to the set of systems it is able to analyze correctly.

Several other attempts to formalize WSBPEL specification and a detailed comparaison between them can be found in [30,29]. [29] is a tutorial that provides an overview of the different models of BPEL that have been proposed. Furthermore, the authors discuss the verification techniques for BPEL that have been put forward and the verification tools for BPEL that have been developed.

In terms of choreography and Web service conversations, work on asynchronous Web service communication has been described in [10,9], with an example focus on the BPEL4WS specification reported in [10]. A formal specification framework is described to analyse the conversations proposed by the asynchronous communication channels utilized on the internet. Interestingly, we have showed later in this thesis, that BPEL4WS provides a pseudo-asynchronous interaction model (whereby an invocation is sent, and then a separate receive activity formulates the link of call and reply). The technique proposed

appears more useful for modelling general Web service communication, rather than that of compositional specifics. Both the work on asynchronous and BPEL4WS interaction modelling is achieved through the use of Guarded Finite State Automata (GFSA) which enables data dependencies to be modeled alongside process transitions. In [3] the authors describe an approach to formalizing conversations, by way of mapping the WSCI standard (to CCS for Web service choreography descriptions. The technique is similar to that of formalizing compositions by way of mapping each of the actions and data parameters between two or more partnered services in choreography. The conversation is traced by modelling the Web service invocations with that of the receive and reply actions of the partnered service. The authors call for a common view of representing both composition and choreography models, such that fluid design and maintenance of individual specifications is not detrimental to the development effort.

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. Another important element is that the composition and the choreography are not usually expressed within one single environment and therefore the verification techniques must be modified before using them. Instead, in our approach we provided a guide on how to translate the semantics of the BPEL4WS specification to EC and map implementation abstractions which preserve the interaction behaviour between services, yet also disposing of process characteristics which are not required in the analysis. Then, we elaborated these models to analyse the conversations of compositions across choreography scenarios, providing both interface and behavioural compatibility verification processes.

8 Conclusion

In this paper we presented a unified framework for the analysis and verification of Web service compositions provided as BPEL specifications. This framework enables the checking of requirements for WSBPEL processes. The requirements specify behavioural properties of the composition process, or assumptions about the behaviour of the composition as a whole, its constituent services and external agents who interact with it. The behavioural properties are initially extracted from the specification of the composition process that is expressed in WSBPEL. This ensures that the properties to be checked are expressed in terms of events occurring during the interaction between the composition process and the constituent services. The assumptions to be monitored are subsequently defined in terms of these detectable events. The specification of assumptions is supported by a graphical editor which allows users to select detectable events and define formulas over them. Both the behavioural properties and assumptions are specified in event calculus. Then, the approach is extended to include models of service choreography with multiple interacting Web services compositions, from the perspective of a collaborative distributed composition development environment. The process of behaviour analysis moves from a single local process to that of modelling and analysing the behaviour of multiple processes across composition domains. Finally, the specifications of the violated requirements are used to generate queries for discovering services that could

substitute for malfunctioning services or services that may become unavailable or fail to meet certain requirements.

References

1. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, 2003.
2. A. Arkin, S. Askary, B. Bloch, and F.Curbera. Web services business process execution language version 2.0. Technical report, OASIS, December 2004.
3. A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web service choreographies. *Electr. Notes Theor. Comput. Sci.*, 105:73–94, 2004.
4. F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Inf. Syst.*, 26(3):143–163, 2001.
5. D. Fahland and W. Reisig. ASM-based semantics for BPEL: The negative control flow. In D. Beauquier and E. Börger and A. Slissenko, editor, *Proc. 12th International Workshop on Abstract State Machines*, pages 131–151, Paris, France, March 2005.
6. A. Ferrara. Web services: A process algebra approach. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM Press.
7. J. Fisteus, L. Fernández, and C. Kloos. Formal verification of BPEL4WS business collaborations. In K. Bauknecht, M. Bichler, and B. Proll, editors, *Proceedings of the 5th International Conference on Electronic Commerce and Web Technologies (EC-Web '04)*, volume 3182 of *Lecture Notes in Computer Science*, pages 79–94, Zaragoza, Spain, Aug. 2004. Springer-Verlag, Berlin.
8. H. Foster, J. Kramer, J. Magee, and S. Uchitel. Model-based verification of web service compositions. In *18th IEEE International Conference on Automated Software Engineering (ASE)*, 2003.
9. X. Fu. Formal Specification and Verification of Asynchronously Communicating Web Services. Phd Thesis, Santa Barbara, CA, USA,University of California, 2004.
10. 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.
11. H. K. V. M. Gustavo Alonso, Fabio Casati. *Web Services: Concepts, Architectures and Applications*. Springer, 2004.
12. M. Koshkina and F. van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.
13. R. Kowalski and M. J. Sergot. A logic-based calculus of events. *New generation Computing 4(1)*, pages 67–95, 1986.
14. J. Magee and J. Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.
15. K. Mahbub and G. Spanoudakis. A framework for requirents monitoring of service based systems. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 84–93, New York, NY, USA, 2004. ACM Press.
16. K. Mahbub and G. Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 257–265, Washington, DC, USA, 2005. IEEE Computer Society.

17. A. Martens. Analyzing Web Service Based Business Processes. In M. Cerioli, editor, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *Lecture Notes in Computer Science*, pages 19–33. Springer-Verlag, Berlin, 2005.
18. S. Nakajima. Verification of web service flows with model-checking techniques. In *CW*, pages 378–385, 2002.
19. 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.
20. C. Ouyang, W. Aalst, S. Breutel, M. Dumas, , and H. Verbeek. Formal Semantics and Analysis of Control Flow in WS-BPEL. BPM Center Report BPM-05-15, BPMcenter.org, 2005.
21. M. Pistore, M. Roveri, and P. Busetta. Requirements-driven verification of web services. *Electr. Notes Theor. Comput. Sci.*, 105:95–108, 2004.
22. G. Pu, X. Zhao, S. Wang, and Z. Qiu. Towards the semantics and verification of bpel4ws. *Electr. Notes Theor. Comput. Sci.*, 151(2):33–52, 2006.
23. 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.
24. M. Rouached and C. godart. A Dynamic Query based Discovery for Web Services Composition. `http://www.loria.fr/~rouached/CWSDiscovery.pdf`.
25. M. Rouached and C. Godart. Requirements-driven verification of wsbpel processes. In *Proceedings of the IEEE International Conference on Web Services (ICWS'07)*. Salt Lake City, Utah, USA, July 9-13 2007.
26. G. Spanoudakis, A. Zisman, and A. Kozlenkov. A service discovery framework for service centric systems. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, pages 251–259, Washington, DC, USA, 2005. IEEE Computer Society.
27. C. Stahl. Transformation von BPEL4WS in Petrinetze (In German). Master's thesis, Humboldt University, Berlin, Germany, 2004.
28. S. Stratulat. A general framework to build contextual cover set induction provers. *Journal of Symbolic Computation*, 32(4):403–445, 2001.
29. F. van Breugel and M. Koshkina. Models and verification of bpel. Available at http://www.cse.yorku.ca/franck/research/drafts/tutorial.pdf, 2006.
30. Y. Yang, Q. Tan, and Y. Xiao. Verifying web services composition based on hierarchical colored petri nets. In *IHIS '05: Proceedings of the first international workshop on Interoperability of heterogeneous information systems*, pages 47–54, New York, NY, USA, 2005. ACM Press.