School of Computing Science,
University of Newcastle upon Tyne

Notations for the Specification and Verification of Composite Web Services

Woodman, S.J., Palmer, D.J., Shrivastava, S.K.,
and Wheater, S.M.

Technical Report Series

CS-TR-849

June 2004

Published by the University of Newcastle upon Tyne,
School of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK.

Notations for the Specification and Verification of Composite Web Services

S.J.Woodman, D.J.Palmer, S.K.Shrivastava
School of Computing Science
University of Newcastle upon Tyne,
Newcastle upon Tyne, UK
{s.j.woodman, d.j.palmer,
santosh.shrivastava}@ncl.ac.uk

S.M.Wheater
Arjuna Technologies
Nanotechnology Centre,
Newcastle upon Tyne, UK
stuart.wheater@arjuna.com

Abstract

Availability of a wide variety of Web services over the Internet offers opportunities of providing new value added services built by composing them out of existing ones. Service composition poses a number of challenges. A composite service can be very complex in structure, containing many temporal and data-flow dependencies between their constituent services. Furthermore, each individual service is likely to have its own sequencing constraints over its operations. It is highly desirable therefore to be able to validate that a given composite service is well formed: proving that it will not deadlock or livelock and that it respects the sequencing constraints of the constituent services. With this aim in mind, the paper proposes simple extensions to web service definition language (WSDL) enabling the order in which the exposed operations should be invoked to be specified. In addition, the paper proposes a composition language for defining the structure of a composite service. Both languages have an XML notation and a formal basis in the pi-calculus (a calculus for concurrent systems). The paper presents the main features of these languages, and shows how it is possible to validate a composite service by applying the pi-calculus reaction rules.

1. Introduction

Creating new services by combining a number of existing ones is becoming an attractive way of developing value added web services. This pattern is not new but it does pose some new challenges which have yet to be addressed by current technologies and tools for web service composition. Ideally, it is desirable to automatically compose a service capable of achieving a goal specified by a client request. However, in the near future this is unlikely to be possible due to the lack of semantic information provided by current web services. The first step in this direction is to provide more semantic information about each web service in order to be able to reason about a composition which has been created manually.

There are two perspectives that can be taken when considering composite services: that of the provider of the web services, and that of the service composer who wishes to create a value added service by utilising existing services. Using current technology, the web service provider will deploy a service and expose the interface to the service using Web Service Definition Language (WSDL). The WSDL description of a service contains a specification of the operations which a service exposes and binding information detailing how to invoke the operations in terms of protocols and addressing. Although this level of detail is sufficient for constructing simple web services applications it is insufficient when it comes to creating complex services and reasoning about their composition [15].

To a service composer, it is desirable to be able to verify that the composition is well formed: for example that it does not contain any deadlocks or livelocks which would cause the composition to not terminate under certain conditions; and that the composition uses each web service “correctly”. It is possible to verify the former using formal notations and model checkers but for the latter it is necessary to describe what is meant by “correctly”. One aspect of using a web service correctly is invoking the operations in the order in which the provider intended. However, the WSDL description of a web service does not specify any ordering information for the operations which are exposed by the service. To allow a service composer to verify this aspect of correctness of the composition, the ordering information must be provided by the web service in addition to the WSDL description.

1.1. Motivating Example

It is useful at this point to present an example to further explain the motivations and clarify the role played

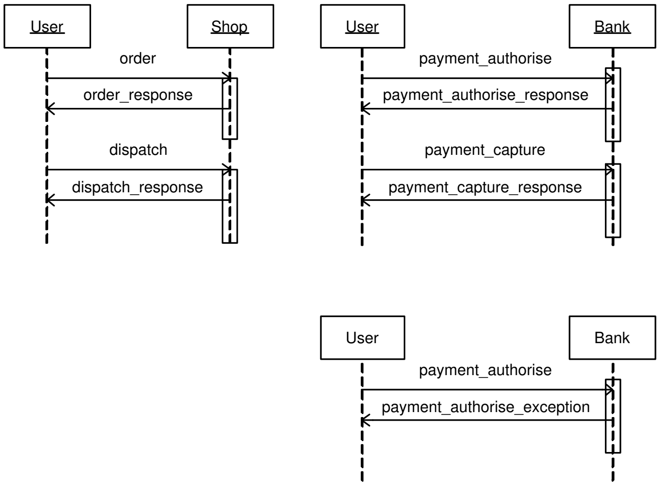

Figure 1. Sequence Diagrams for the shop and bank web services

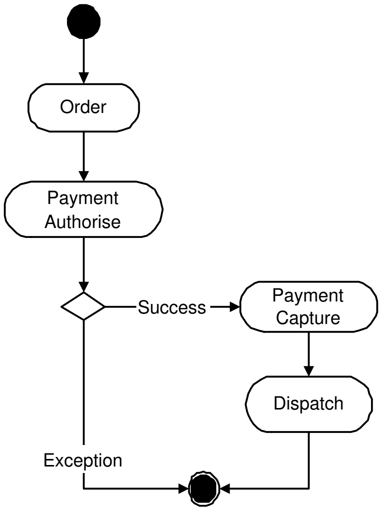

Figure 2. UML Activity diagram for a composite order service

by each party. The example contains two web services provided by third parties, one by a shop and one by a bank. Both are simplified for brevity and ease of understanding. The shop web service exposes two operations, `order` and `dispatch` both of which are RPC (Remote Procedure Call) style services accepting requests and generating responses. To use the service correctly, the `dispatch` operation must be invoked after the `order` operation. The bank web service also exposes two operations, `payment_authorise` and `payment_capture`. These operations must be invoked in the above order, but in addition, `payment_authorise` must return a response message rather than an exception before `payment_capture` can be invoked. Should `payment_authorise` return an fault, it is invalid to invoke the `payment_capture` operation. UML Sequence Diagrams showing the legal sequences of operations for each service are shown in Fig. 1.

A service composer wishes to utilise the shop and bank web services to provide a single point of access to customers who wish to purchase items from the shop. The composer wishes to ensure that despite the different responses from each of the services, the composition always uses the services according to the specifications defined above and does not contain any deadlocks or livelocks which would prevent termination.

The composition of these services could be as follows. Invoke the `order` operation, followed by the `payment_authorise` operation. If `payment_authorise` succeeds then `payment_capture` and `dispatch` may be invoked. If `payment_authorise` fails then the composition also fails. This composition is illustrated by the UML Activity diagram in Fig. 2.

In this simplified example it is possible to see that there are two “execution traces” which are possible through the composition dependant on whether the `payment_authorise` operation returns a success response or an exception. Clearly neither of these contain any livelocks or deadlocks. However, in the general case this may not be easy to infer as the size and complexity of a composition increases. It is also trivial to see that this composition utilises the shop and the bank web services correctly as `dispatch` is always invoked after `order` and `payment_capture` is correctly dependant on the output of `payment_authorise`. Again however, as the number of tasks in the composition increases and as complex inter task dependencies are introduced this will become harder to state by studying the composition. It is highly desirable to be able to automatically verify that an arbitrary composition correctly uses all of its component services.

In this paper we present three aspects of service specification and verification: Firstly, we present a simple language for capturing the order in which the operations of a web service should be invoked to achieve a goal; Secondly we describe a language for the specification of composite web services as a business process; the language permits orchestration of the process using workflow management systems in either a centralised or distributed, peer-to-peer fashion. Both languages have a formal basis in the π-calculus, enabling us to prove, using reduction semantics of the π-calculus, that a given composite service is free from deadlocks, livelocks and it invokes the operations of the third

party web services in the correct order.
The remainder of this paper is structured as follows: Section 2 gives an overview of the current state of the art; Section 3 describes a language for defining composite services; the sequencing constraints which can be exposed by a web service are described in Section 4 and Section 5 shows how the two languages complement each other and uses the previous example to formally show that the composition is well formed. Finally, further work is presented and conclusions are drawn in Sections 6 and 7.

2. Related Work

Web Services technology is evolving rapidly. In the following section notable, recent or ongoing efforts will be discussed with emphasis on those aspects that are relevant to service composition and validation.

The Business Process Execution Language for Web Services (BPEL4WS) [1] provides a standard for specifying both business process behaviour (service composition) and business process interactions (sequencing constraints). BPEL4WS attempts to describe business process interactions using the mutually visible message exchange of each of the parties involved in the protocol, such descriptions are called *business protocols*. Another facet of BPEL4WS is the specification of *executable processes* which describe the structure of a composition in sufficient detail to be executed by an enactment engine. Both of the aspects of BPEL4WS are encoded in XML using a rich set of structured programming style constructs. However, BPEL4WS is lacking a formal, well understood basis and due to this and the rich set of constructs, specifications written in BPEL4WS are not readily susceptable to automatic verification. When considering a subset of BPEL4WS, it has been shown in [13] that verification of safety and liveness conditions can be achieved.

The purpose of Web Service Conversation Language (WSCL) [4] is to provide a standard for specifying business level conversations. WSCL provides an XML schema for specifying business level conversations that take place at a single Web service. The WSCL notion of a conversation is a series of messages exchanged between a service-consumer and a service-provider. The WSCL specification models a conversation as a finite state machine where state changes are triggered by interactions. An interaction is the exchange of one or two documents between a service-consumer and a service-provider. WSCL is simple, and analysable, but does have some limitations, such as only being capable of modeling two party conversations and does not define how to specify an executable process. There are no signs that WSCL has been widely adopted or that an updated version will be published.

The Web Services Choreography Working Group [5] is an initiative by the World Wide Web Coalition (W3C) and was started in January 2003. The Working Group is chartered to create the definition of a choreography, language(s) for describing a choreography, as well as the rules for composition of, and interaction among, such choreographed Web services. At this time the Working Groups First Working Draft Specification is still in preparation.

In [11] a technique is presented to allow automatic composition of web services to achieve a goal. This approach is based on Mealy Finite State Machines (MFSMs), a finite state machine with input and output queues. Each service which can form part of the composition must be described by a MFSM and the goal of the desired composition must also be described by a MFSM. The former part can be considered similar to exposing sequencing constraints but with a different formal background. The algorithm provided for automatically creating the composition is an effective one but relies on the specifying the desired composition as a MFSM, a requirement which may not always be desirable. In many respects, this approach is similar to the DAML-S Coalition [9] which is defining an ontology and related language for describing web services with the aim of being able to compose them automatically [24, 20]. This technology will undoubtedly play an important role in the future but at present is in its infancy with a lack of tools support and rapidly changing specifications.

The results based on Mealy machines presented in [12], suggest that there is a lack of understanding of the relationship between local properties of web services, and the global properties of a composition created from them. It is shown that unexpected behaviour can occur when messages are queued and distributed decisions taken. It is possible for a service to use an interceptor to ensure that the operations it exposes are invoked in the correct order [25]. This work relies on a language based on CSP to describe the legal sequences of operations but has the disadvantage that it is only able to model two party interactions rather than the multi-party interactions presented here.

Our work makes complementary contributions to those outlined above. As we discuss in the next section, our language notations represent an advance over the current industrial practice as represented by BPEL4WS. We draw upon our earlier work on business processes specification languages and enactment (orchestration) environments [23, 26]. We allow the service composer to make use of a graphical notation for defining the composition as a business process which we believe to be more intuitive and expressive than an FSM notation. A clear separation is drawn between the specification of sequencing constraints for individual web services and that of the composition of those services. We also allow the verification, albeit not automatic composition, that a composition respects those constraints placed on the constituent services. Although simple, our languages are expressive enough to be able to model complex interaction patterns within a composition and capture elaborate sequencing constraints [23, 14]. Message queueing is not considered in this paper but we believe that our π-calculus based approach to service composition can aid understanding of the global properties of a service, when those properties are concerned with the order of invocation of operations.

3. Specifying Composition

3.1. Language Features

In addition to being able to verify that a composition is well formed and uses the constituent services correctly, it is also desirable to be able to enact a composite service in a distributed, peer-to-peer manner [10]. Centralised coordination is sufficient for some classes of applications. There are others which benefit from peer-to-peer style enactment. Value added services provided by Virtual Organisations (VOs) are gaining in popularity and fall into this category. This is due to trust and organisational issues which may prevent the service being enacted from one location.

Industry led efforts aimed at specifying composition languages for web services detailed earlier, take a centralised view of composition and subsequent execution. For example, the use of shared variables in BPEL4WS makes it very difficult to coordinate the execution in a distributed manner. Also, many of these languages specify complicated control flow mechanisms, making it difficult to analyse such compositions. The composition language that we propose has been developed with both of these drawbacks in mind: it contains elements to allow distributed enactment of the composition and the simple data flow sequencing model is based on the π-calculus to allow analysis of compositions.

Fault tolerance is necessary to maintain application specific consistency in the face of failures such as processor crashes, network related failures and application exceptions. The fault tolerance requirements of composite services have been split into the requirements at the application level and at the system level (execution environment). The composition language provides notations and structures for meeting application level fault-tolerance requirements through exceptions, alternative tasks and compensating tasks, whereas the execution environment is responsible for meeting system level fault tolerance. The execution environment is described in [27] and is based on the OPENflow workflow engine [23].

The composition language has two core concepts: a task and a process. A task in a composition is the basic unit of work and corresponds to an invocation of a web service operation. When tasks are composed together they are said to form a process. However, processes can be composed recursively, that is a process can contain other processes as well as tasks. A graphical representation of a task is given in Fig. 3. It depicts a task (called prepare) that has one input message (I) with two data parts (i_1 and i_2), corresponding to the `messages` and `parts` defined in the WSDL description of the operation. This task represents one invocation of a web service called prepare. The input message must have all of its input parts available before the task can start (invoking the web service). A task terminates in one of the named output states (called outcomes) when the web service returns a response. One of these outcomes is considered normal and all others are considered fault outcomes following the convention of WSDL. In Fig. 3, O_1 represents an output message and F_1 represents a fault message. Each outcome of a task has a distinct set of parts, which can be used as input by subsequent tasks or output by a composing processes. The output message O_1 in Fig. 3 has two named parts o_1 and o_2. The fault message F_1 has one fault part f_1. If the format of the inputs and outputs does not match precisely, it is possible to perform simple transformations on the data to overcome this. It is possible for an input or output message to be “empty”, i.e. contain no parts, which models methods which take no parameters and void return types respectively.

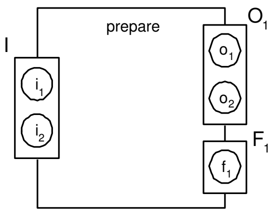

Figure 3. A task showing the input and output parts and messages

The control structure of a process is described in terms of inter-task dependencies linking tasks together to form a process. Two types of inter-task dependency can be used to control the execution of a composition: temporal dependencies and data dependencies. Temporal dependencies are used to control the execution of a task based on other tasks or processes being in particular states. Such a state could be “started” or “completed” with particular outcome. Temporal dependencies are represented by dotted arrows in the graphical representation of the composition. Data dependencies describe where a task acquires its input from, such as the output of another task or the input into the composing process. Data dependencies are represented by solid arrows in the graphical representation. A task can have an arbitrary mix of data and temporal dependencies describing when it can be executed (grouped as “input dependencies”). A process can have a similar mix of dependencies controlling its

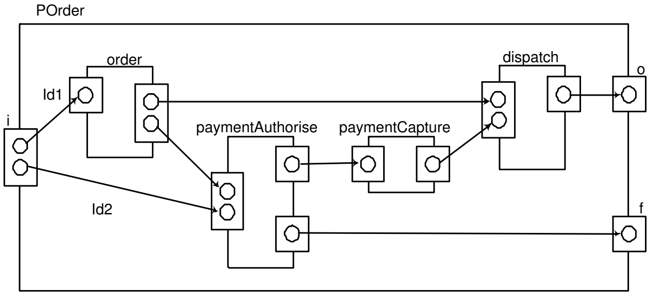

Figure 4. Composite Order service utilising third party bank and shop services

start but also controlling its completion. Such dependencies which control completion are called “output dependencies” and specify how the output of the process is constructed from the data used by its constituent tasks. It is possible to provide an element of fault tolerance through redundancy by using multiple alternate data dependencies. Combinations of such alternate data and temporal dependencies can be used to describe complex process structures as discussed in [23, 26].

The graphical representation of the example presented in Section 1, comprising of four tasks linked by data dependencies is shown in Fig. 4. The order task and the payment_authorise task in the composition both have input dependencies on the input to the composite service (labelled id1 and id2). This means that when the input message, i is received from the client, these input dependencies will be available, and if all of a tasks input dependencies are fulfilled it will begin execution. In this case, the order task can begin execution as its only input dependency is fulfilled. When the order task completes, the results will be propagated to payment_authorise and dispatch along the dependencies shown. This action completes the input dependencies for payment_authorise so it is able to execute. Such behaviour continues until the output dependencies of the outer process (POrder) are fulfilled. In this scenario, the normal output message, o will be complete after the dispatch task has completed (fulfilling the output dependency between dispatch and POrder), or the fault message, f will be complete after the payment_authorise task has terminated with a fault message (fulfilling the output dependency between payment_authorise and POrder). A segment of the XML notation of the POrder composite service is shown below, consisting of the tasks payment_authorise and payment_capture. It is possible to see the task definition and dependency structure. The payment_authorise task (P_A_Task) has an two input dependencies, one from the composing process POrder and one from the output of the orderTask (not shown). The P_C_Task representing payment_capture has just one dependency, on the output of the P_A_Task. When the input dependencies for each task are fulfilled they will execute, invoking the web service operation detailed in the operation attribute of the taskDefinition element. When each service is invoked, the parameters are renamed according to the sinkPartName attributes. For brevity namespaces and address data for the services has been omitted.

```
<processDefinition name="POrder" ... >
  <subProcesses>
    ...
    <taskDefinition name="P_A_Task"
     operation="paymentAuthorise" ... >
      <inputDependencies>
        <dataDependency sourceProcess="POrder"
         sourcePartName="accountNum"
          sinkPartName=”accountToDebit”... />
        <dataDependency sourceProcess="orderTask"
         sourcePartName="amount"
          sinkPartName=”debitAmountInSterling” ... />
      </inputDependencies>
    </taskDefinition>
    <taskDefinition name="P_C_Task"
     operation="paymentCapture" ... >
      <inputDependencies>
       <dataDependency sourceProcess="P_A_Task"
        sourcePartName="authNum"
         sourceMessageType=”output”
         sinkPartName=”authorisationCode” ... />
      </inputDependencies>
    </taskDefinition>
    ...
  </subProcesses>
</processDefinition>
```

It is possible and likely in some application domains that composite services could be very large, involving many tasks and complex inter-task dependencies. To make such compositions easier to create, maintain and understand it is desirable to be able to modularise them and reuse the definitions where possible. To achieve this, it is possible to reference other, external process definitions from within a process. Such referenced process definitions may include frequently used modules of compositions, which are defined separately and referenced by a number of different compositions. Examples include a fragment to log into a frequently used web service or interact with a transaction manager.

Resource availability may be at a premium on the node which is enacting the composite service. In order to allow efficient resource management there are two stages at which the composition designer can choose to instantiate the tasks and sub processes within it: Early or late. When Early (traditional) instantiation is used, all of the tasks, sub processes and externally referenced processes are loaded into the execution environment and initialised when the composite service is instantiated following a client request. This leads to a static system which is easier to reason about but more difficult to modify. Late instantiation results in the tasks, sub

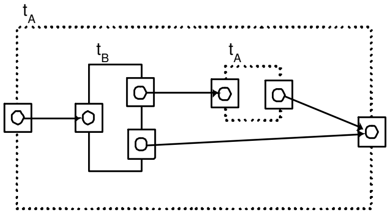

Figure 5. Using a late instantiating task to perform recursion

processes and externally referenced processes not being instantiated until they are able to run, i.e. when all of their input dependencies are satisfied. Late instantiation implies that only those parts of large process definitions that are needed will be instantiated, giving more efficient resource usage.

Many structures within a composite service will require a form of recursion to perform a task a number of times, often not known until runtime. Using late instantiated processes, depicted as a process with a dotted border, allows the designer to achieve this. It is possible for a late instantiated service to refer to itself and instantiate another instance of itself under certain conditions giving the desired recursive behaviour [23]. For instance, t$_A$ in Fig. 5 refers to itself, causing repeated execution of t$_B$ until t$_B$ completes with the lower outcome.

3.2. Orchestration

Orchestration of composite services defined in the composition language can be carried out using a workflow management system. Our current execution environment is DECS [27], a workflow enactment engine, built on top of the J2EE architecture [17] which allows flexible coordination of composite services. That is, the orchestration can either be centralised or can be distributed where each engine communicates with each other in a peer-to-peer manner. When decentralised orchestration is employed, each engine is responsible for part of the execution of the composite service. Each engine will invoke the constituent services for its part of the composition and send notification messages to other engines when certain events occur. Such notifications only contain the minimal amount of data necessary for the other engines to continue enacting their part of the composition (see Fig. 6.). This gives rise to increased security and organisational autonomy as each engine is only aware of the data necessary for it to continue execution. The composition language can be mapped onto other execution environments. We currently provide such a mapping to JOpera, a centralised workflow management system [22].

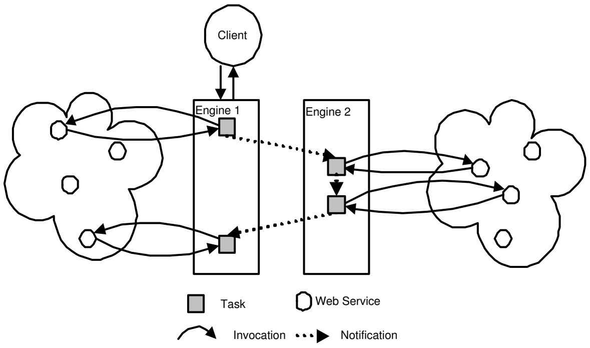

Figure 6. Distributed orchestration of a composite service

3.3. Semantics

To allow reasoning about a composition with respect to deadlocks, livelocks and respecting sequencing constraints of the constituent services, the composition language has a formal basis in the π-calculus [19]. It is possible to translate from the XML format of the language to the π-calculus format. In the π-calculus format, tasks are represented as π-calculus processes, and dependencies linking the tasks, represented by π-calculus channels. (An overview of π-calculus is given in the Appendix.) Channels represent data dependencies, as temporal dependencies are represented implicitly using the operators of π-calculus directly. As each task in the composition language is analogous to an invocation of an operation of a web service, this invocation is also modelled as the sending of a message along a channel to the web service. The receipt of a response or exception from the web service is modelled as the receipt of a message along a channel from the web service. The composite service as a whole is modelled as a parallel composition of all of these processes. For readability, a notational convention has been adopted whereby the channels are named as the processes which they connect, for example, papc is a channel between the `payment_authorise` and the `payment_capture` tasks. The channels which represent a connection to a web service are written as an abbreviation of the operation name such as o for order, appended with an abbreviation of the type of message it is (input - i, output - o, exception - e). The names that are sent down each channel represent either wsdl:messages or wsdl:parts and where necessary, an internal action (τ) can perform transformation on these messages to extract/combine them. The full range of pi-calculus constructs: sequence (.), parallel composition (|), choice (+) and replication(!) are used to define the flow control within the composition. In [19] it is shown that these operators are sufficient to model the communication in any system, or in this case, composition.

The pi-calculus format of the composition from Fig. 4 is shown below. It consists of 5 pi-calculus processes composed in parallel to form the *system*: PO (the outer composite service), Order (O), payment_authorise (PA), payment_capture (PC) and dispatch (D). The names which are sent down the channels represent the input/output dependencies between the tasks, for example order# (on), account# (an), amount (am), delivery_day (dd), exception_code (ec), invoice# (in), reference# (rn) and authorisation# (ac).

As PO represents a process in the composition language, its structure is different from that of a π-calculus process which represents a task in the composition. PO begins by sending two messages along different channels in parallel: on (order#) is sent along the $p\bar{o}o$ channel from PO to order (o); an account# (an) is sent along the $p\bar{o}pa$ channel from PO to payment_authorise (pa). The PO process then waits to receive messages which will form its output. There is a choice of messages which can form the output, either receiving a delivery_day (dd) from the dispatch (D) process along channel dpo, or receiving an exception_code (ec) along the papo channel which connects payment_authorise (pa) to PO. The final 0 in the process signals that the process is complete and in this case, also that the composite service is complete.

$$PO \quad = \quad (p\bar{o}o{<}on{>} \mid p\bar{o}pa{<}an{>}).(dpo(dd){+}papo(ec)).0$$

π-calculus processes which represent tasks in the composition language all follow the same structure: they wait to receive their input, send a message to the web service that they are invoking, receive the response from the web service and finally send messages to other ”downstream” processes which have dependencies on them. For instance, the Order process (O) waits to receive an order# (on) along the channel from PO named poo. The process then performs an internal action to signify that the input data is transformed into a request (req) for the web service. This request is sent along the input channel for the web service (oi) and then the response gathered from the output channel of the web service (receiving rsp along oo). Again, an internal action denotes the deserialisation of the response and parts of the response are propagated to downstream tasks. In this case, the propagation involves sending an amount (am) to payment_authorise along the $o\bar{p}a$ channel and, in parallel, an invoice number to dispatch along the $\bar{o}d$ channel. The terminating 0 shows that the process is complete, but in this case does not signify that the composite service is complete.

$$O \quad = \quad poo(on).\tau.\bar{o}i{<}req{>}.oo(rsp).\tau.(o\bar{p}a{<}am{>} \mid \bar{o}d{<}in{>}).0$$

$$\begin{aligned} PA \quad = \quad & (popa(an) \mid opa(am)).\tau.p\bar{a}i{<}req{>}.(pao(rsp).\tau.p\bar{a}pc{<}rn{>} \\ & {+}pae(flt).\tau.p\bar{a}po{<}ec{>}).0 \end{aligned}$$

$$PC \quad = \quad papc(rn).\tau.p\bar{c}i{<}req{>}.pco(rsp).\tau.p\bar{c}d{<}ac{>}.0$$

$$D \quad = \quad (od(in) \mid pcd(ac)).\tau.\bar{d}i{<}req{>}.do(rsp).\tau.d\bar{p}o{<}dd{>}.0$$

$$COMP \quad = \quad (PO|O|PA|PC|D)$$

Section 5 discusses how to verify that such a composition is free of deadlocks and cyclic dependencies whilst utilising the constituent web services in the correct manner.

4. Sequencing Constraints

4.1. Language Features

In order to be able to verify that a composition described by the composition language uses the third party services in the correct way, it is necessary for these services to expose additional semantic information describing what “the correct way” is. The language described in this section intends to define the order in which the operations of a web service should be invoked, or the *sequencing constraints* which are placed on a service. Such constraints should be: flexible - to be able to model any possible sequence of operations; complete - so that all legal sequences are represented; concise - to avoid ambiguities which might be introduced by a complex language.

It is possible to think of the sequencing constraints placed on a web service as the “protocol” that the web service supports. Descriptions of protocols are not new and there are many common descriptions that are used [21], however these tend to be intended for human readability and not machine interpretation.

The sequencing constraints are defined by the web service provider and exposed in the WSDL definition of the service by utilising the extensibility elements in WSDL. There are only five language constructs necessary to describe any possible sequence of messages:

- Sequence: perform all child elements in sequence with one starting only when the preceding one has completed
- Choice: perform exactly one of the child elements
- Parallel: perform all of the child elements in parallel and complete when all parallel executions have completed

- Multiple: perform the child elements an arbitrary number of times
- Nothing: do nothing.

The language constructs are used to describe the order in which the service is expecting events to happen. The events are described in terms of four communication primitives:

- Send: The service will send a message.
- Receive: The service will receive a message.
- Service: the server side view of a call. There are three elements associated with a Service: serviceInput, serviceOutput and serviceFault. A ServiceInput receives the input to a call. ServiceOutput and ServiceFault correspond to replying to the client with either the output or fault message defined in the WSDL description
- Invoke: A client side view of a call. InvokeOutput is analogous to sending the call request and InvokeInput/InvokeFault are used to model receiving the result or fault from a call

Initially the Service and Invoke primitives may seem a little unintuitive. However, they correspond to the Client (Invoke) and Server (Service) ends of a Remote Procedure Call (RPC). It is possible to model an RPC simply in terms of send and receive but ambiguities can occur when using this method. For example, it becomes difficult to associate receive operations with the corresponding send operation if multiple send operations occur in parallel. Explicitly describing RPCs using the invoke and service primitives removes these ambiguities and reduces the complexity of the verification process.

The sequencing constraints for the bank web service described before are shown below in the XML format. They consist of one "protocol" called pay which begins by a client invoking the payment_authorise operation. This is described by the element serviceInput as it is an RPC style service exposed by the bank. Following this invocation the sequencing constraints allow a choice of activities: a serviceFault can occur which equates to an exception being emitted from the payment_authorise operation. Should a serviceFault occur, the protocol implicitly terminates as there are no activities left (all other activities are ruled out by the choice). The alternative to the serviceFault in the choice element is a sequence of activities occurring. These are initiated by a serviceOutput activity, in this case the "normal" output from payment_authorise being returned. Following this, the protocol expects the payment_capture operation to be invoked and will then return a response from this operation via the serviceOutput element. The protocol is then in a terminating state as no more actions are expected.

```
<protocolType name="pay">
  ...
  <serviceInput operation="payment_authorise" ...>
    <choice>
      <serviceFault/>
      <sequence>
        <serviceOutput/>
        <serviceInput operation="payment_capture" ...>
          <serviceOutput/>
        </serviceInput>
      </sequence>
    </choice>
  </serviceInput>
</protocolType>
```

When considering asynchronous services it is possible that the web service designer has specified full WSDL for their service, i.e. the WSDL describes the messages which will be produced as well as consumed. If this is the case, it is possible to define the sequencing constraints in terms of that single WSDL document. However, most services are not defined in this manner so it is necessary to provide an alternative method for specifying the sequencing constraints. To achieve this the language allows one participant to be defined as the "inverse" of another. For instance, the send operation which is not defined in one WSDL document is the inverse of a receive defined in another WSDL document. Whether or not this other document exists is not relevant to the interaction constraints. This simply allows the language to deal with incomplete but legal WSDL.

It is an issue for the author of the sequencing constraints to decide what level of detail they wish to provide. Some may wish to simply model the client and server interaction, preserving the encapsulation offered by the web service. Other designers may wish to expose the sequencing which happens behind the scenes in communicating with other services. The latter offers advantages in scenarios such as asynchronous multi-party interactions. It allows the client of a service to fully reason about the service that they are using and gives a form of causality where asynchronous messages are received from other services than that invoked. The language provides constructs for both options to a service designer and does not constrain them to model only simple interaction involving two parties [14] .

When conversations take place, the participants involved could be known before the protocol starts, this is referred to as having statically bound participants. Alternatively the participants may be discovered as the protocol progresses, this is referred to as having dynamically bound participants, this discovery being deduced from the content of messages within the conversation. Naturally, conversation may have a mixture of both statically and dynamically bound participants. Dynamically bound participants is a common occurrence in more complicated protocols, such as in Web Services Coordination and Web Services Transaction (WS-C and WS-T) [6, 7]. In WS-C, an application may be passed a context containing the address of the coordinator to use. The language allows the specification of such scenarios containing late binding of services using optional attributes on the communication primitives. Providers of sequencing constraints should ensure that the participants which are dynamically bound play no part in the conversation before they are bound to a concrete service.

4.2. Semantics

The sequencing constraints language has a formal basis in the π-calculus and there is a π-calculus representation which can be derived from the XML format. This representation uses similar constructs to those described at the beginning of this section for the language constructs (sequence, parallel, replication and choice). Each of the participants in the protocol is connected by multiple channels (one channel per operation exposed by the service). The communication primitives described above are modelled as sending the parts which comprise a wsdl:message along a channel to an operation. The naming convention is the same that was described for the Composition Language π-calculus representation, ie. an abbreviation for the operation name appended with the message type (input - i, output - o, exception - e).

$$\begin{aligned} SHOP &= oi(req).\tau.\bar{o}o{<}rsp{>}.(0{+}di(req).\tau \\ &\quad .\bar{d}o{<}rsp{>}.0) \end{aligned}$$

$$\begin{aligned} BANK &= pai(req).\tau.(p\bar{a}o{<}rsp{>}. \\ &\quad pci(req).\tau.p\bar{c}o{<}resp{>}.0 \\ &\quad +p\bar{a}e{<}flt{>}.0) \end{aligned}$$

The π-calculus above corresponds to the UML sequence diagrams shown in Fig. 1. The shop service is expecting to receive a request over the `order` operation channel. It will then return a response over the `orderResponse` channel. Following this, the user is not obliged to call any other operations as indicated by the terminating 0 in the choice element (+). However, to confirm the order, the user must invoke the `dispatch` operation by sending a request over the `dispatch` channel. The shop will then return a response over the `dispatchResponse` channel. The bank service can be described in a similar way, except that should a fault message be sent along the `paymentAuthoriseException` channel it is not legal to invoke any other operations. However, if a response is returned over the `paymentAuthoriseResponse` channel, `paymentCapture` may be invoked by sending a request along the input channel and a response will be returned along the response channel. This is again modelled as a choice between performing more operations if a message is received along the response channel and doing nothing (O) if a fault is received along the exception channel.

5. Verification of the Composition

As described in Section 1 it is desirable to be able to verify that the composition meets certain correctness requirements such as:

- Is free of deadlocks
- Is free of livelocks
- Respects the sequencing constraints placed on constituent services

To achieve this, it is possible to apply the reaction rules defined by pi-calculus. These rules prescribe how a system denoted in pi-calculus can react and change depending on the messages which are sent and received. A pair of actions are said to be complimentary when they perform a send and a receive over the same channel. If they are both unguarded and not in the same summation (and so alternatives to each other) they are termed a redex. The firing of such a redex constitutes a reaction in the system causing the system to move from one state to another, i.e. $s \rightarrow s'$. The new state is equivalent to the old state with the actions that formed the redex removed.

To analyse the system it is necessary to create a "global view" of the system, containing both the composition and the sequencing constraints for the services which are being used. To achieve this, a parallel composition is created which is a union of the composition and a replicated version of the sequencing constraints which were defined earlier. It is necessary to replicate the sequencing constraints as multiple instances of the same services may be consumed by the same composition.

$$SYSTEM = (COMP \,|\, !SHOP \,|\, !BANK)$$

In order to show that the composition meets the requirements identified above we apply the pi-calculus reaction rules to this system. Whilst doing this it is necessary to show the following:

1. Following any reaction, either another reaction can occur or the system is in a completion state
2. From every state, it is eventually possible to reach the completion state

Where the completion state is defined as: the composition has been reduced to an empty expression, i.e. no terms are left, and the sequencing constraints have been reduced to either an empty expression or are still in the starting state. Point 1 above shows a lack of deadlocks and point 2 indicates a lack of livelocks within the global picture.

Informally, the global picture models three aspects: Firstly, the third party web services offered by the service provider are modelled by the sequencing constraints placed on them; secondly, the structure of the composition is modelled by the channels connecting different π-calculus processes in the composition language; thirdly, the interaction between the composition and the third party services are captured by the "external channels" in the composition language. Showing that following any reaction, there is another reaction possible proves that there are no deadlocks in the system. Such deadlocks could be because of poorly formed structure within the composition, or could be because of a mismatch between the composition and the sequencing constraints (the composition does not respect the sequencing constraints). In order to show that the system is free of livelocks it is necessary to show that no cycles exist which would prevent eventual termination.

To formally illustrate the reaction rules, below is a partial view of the system after the first reaction has occurred. The first reaction which was possible involved the outer process (PO) performing two operations in parallel, sending the order# (on) to O along the $p\bar{o}o$ channel and sending the account# (an) to PA along the $p\bar{o}pa$ channel. The O and PA processes performed the complementary receives to the sends performed by PO and thus the terms formed a redex. This redex caused the transformation $SYSTEM \rightarrow SYSTEM'$where the redex is removed. (In the interest of brevity, not all processes are shown below.)

$$PO' \quad = \quad (dpo(dd) + papo(ec)).0$$

$$O' \quad = \quad \bar{o}i{<}req{>}.oo(rsp).\tau.(o\bar{p}a{<}am{>} \,|\, \bar{o}d{<}in{>}).0$$

$$\begin{aligned} PA' \quad = \quad & opa(am)).\tau.p\bar{a}i{<}req{>}.(pao(rsp).\tau.p\bar{a}pc{<}rn{>}+ \\ & pae(flt).\tau.p\bar{a}po{<}ec{>}).0 \end{aligned}$$

$$\begin{aligned} SHOP \quad = \quad & oi(req).\tau.o\bar{o}{<}rsp{>}.(0+di(req).\tau \\ & .\bar{d}o{<}rsp{>}.0) \end{aligned}$$

$$SYSTEM' \quad = \quad (PO' \,|\, O' \,|\, PA' \,|\, PC \,|\, D \,|\, !SHOP \,|\, !BANK)$$

Following this reaction, it is clear by inspection that another can occur - the sending of request message along the oi channel to the SHOP from O and causing the reaction $SYSTEM' \rightarrow SYSTEM''$.When continually applying the reaction rules to the system, there are often several states which can be reached from a given state. This happens, for example, when a task has alternative outcomes such as normal and an exception. The use of a model checker eases checking in these situations as the state space which must be checked can become larger than is easy to reason about by inspection.

6. Further Work

Many services are designed such that different logical meanings of messages are separated into multiple messages. However, some services are designed in ways that a message can convey multiple meanings. For instance, the LoginResponse message type used in the xCBL Order Management Use Case [8] can indicate both success and failure of the login. It would be desirable to be able to offer a different sequence of operations dependant on the content of a message, i.e. be able to inspect the message. We are investigating ways of achieving this and assessing the implications on the formal model of both the sequencing constraints and the composition. Initial results indicate that it may be possible to utilise the type system of pi-calculus to achieve this.

Current tools support for verification of pi-calculus are in their infancy. Most do not support the complete language and require a complex and error prone input syntax. We are investigating using various π-calculus model checkers [3, 2] to automatically validate composite services. It is also possible to map our languages onto Promella and then use the SPIN model checker [16]. In the future we hope to be able to integrate one of these into the tool used to create the composition and automatically validate the composition at creation time.

7. Concluding Remarks

A composite service can be very complex in structure, containing many temporal and data-flow dependencies between their constituent services. Furthermore, each individual service is likely to have its own sequencing constraints over its operations. It is highly desirable therefore to be able to validate that a given composite service is well formed: proving that it will not deadlock or livelock and that it respects the sequencing constraints of the constituent services. With this aim in mind, the paper has proposed simple extensions to web service definition language (WSDL) enabling the order in which the exposed operations should be invoked to be specified. In addition, the paper proposed a composition language for defining the structure of a composite service. Both languages have an XML notation and a formal basis in the pi-calculus (a calculus for concurrent systems). The formal verification procedure was demonstrated by applying the pi-calculus reaction rules to a system containing the composite service and sequencing constraints for each web service.

Acknowledgements

Discussions with Gustavo Alonso and Ricardo Jimenez-Peris clarified our ideas. This work is part-funded by the European Union under Project IST-2001-37126: ADAPT (Middleware Technologies for Adaptive and Composable Distributed Components) and by the UK EPSRC under grant GR/S63199: Trusted Coordination in Dynamic Virtual Organisations. The authors also wish to thank the anonymous reviewers for their valuable comments.

References

[1] Business Process Execution Language for Web Services (BPEL4WS) version 1.1. http://www-106.ibm.com/developerworks/library/ws-bpel/.

[2] Profundis: Proof tools for mobile distributed systems. http://www.it.uu.se/profundis/tools.shtml.

[3] The Mobility workbench. http://www.it.uu.se/research/group/mobility/mwb.

[4] Web Service Conversation Language (WSCL) 1.0. http://www.w3.org/TR/wscl10/.

[5] Web Services Choreography Working Group. http://www.w3.org/2002/ws/chor/.

[6] WS-Coordination (WS-C). http://www-106.ibm.com/developerworks/library/ws-coor/.

[7] WS-Transaction (WS-T). http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/.

[8] xCBL - XML Common Buisness Library. http://www.xcbl.org/.

[9] A. Ankolenkar, M. Burstein, J. Hobbs, and O. L. et al. DAML-S: Web Service Description for the Semantic Web. In *The First International Semantic Web Conference (ISWC)*, Sardinia (Italy), June 2002.

[10] B. Benatallah, M. Dumas, Q. Sheng, and A. Ngu. Declarative composition and peer-to-peer provisioning of dynamic web services. In *Proc. 18th International Conference on Data Engineering*, February 2002.

[11] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of E-services That Export Their Behavior. In *Proceedings of the 1st International Conference on Service-Oriented Computing (ICSOC'2003)*, Trento, Italy, 44-58 2003. Springer LNCS, Vol. 2910.

[12] T. Bultan, X. Fu, R. Hull, and S. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *Proc. 12th Int'l World Wide Web Conference (WWW03)*, May 2003.

[13] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatability Verification for Web Service Choreography. In *Proc. of IEEE International Conference on Web Services (ICWS 2004)*, June 2004.

[14] A. Group. Deliverable D6: Service Specification Language. Technical report, 2003.

[15] J. Hanson, P. Nandi, and S. Kumaran. Conversation Support for Business Process Integration. In *Proceedings of 6th Int'l Enterprise Distributed Object Computing (EDOC02)*, September 2002.

[16] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 2002.

[17] S. Microsystems. Java 2 Enterprise Edition. http://java.sun.com/j2ee/.

[18] R. Milner. The Polyadic pi-Calculus: A Tutorial. In *Logic and Algebra of Specification*. 1993.

[19] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[20] S. Narayanann and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc 11th Int'l World Wide Web Conference (WWW-11)*, May 2002.

[21] OMG. Unified Modelling Language. http://www.uml.org.

[22] C. Pautasso and G. Alonso. JOpera: a Toolkit for Efficient Visual Composition of Web Services. Technical Report TR-432-2003, ETH-Zurich, 2003.

[23] F. Ranno, S. Wheater, and S. K. Shrivastava. A System for Specifying and Co-ordinating the Execution of Reliable Distributed Applications. In *Proc of 1st Conf. on Distributed Applications and Interoperable Systems (DAIS'97)*, September 1997.

[24] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions. In *Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*, Angers, France, April 2003.

[25] M. Venzke. Automatic Validation of Web Services. In *Proc 8th CaberNet Radicals Workshop*, October 2003.

[26] S. M. Wheater, S. K. Shrivastava, and F. Ranno. A CORBA Compliant Transactional Workflow System for Internet Applications. In *Proc. Int'l Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, September 1998.

[27] S. J. Woodman, D. J. Palmer, S. K. Shrivastava, and S. M. Wheater. A System for the Distributed Enactment of Composite Web Services. Technical Report DIT-03-056, University of Trento, 2003.

Appendix: π-calculus

The π-calculus [19] is an algebra for describing and analysing the behaviour concurrent systems. A π-calculus system is described in terms of processes, channels and names. Processes are independent of each other and communicate using channels which connect them. Each channel is referred to by a name and the communication unit along a channel is a name. A name is the most primitive unit of addressing in π-calculus. Processes are built from the following action terms and operators:

- Send [$\bar{x}<a>.P$] - Send the name a along channel named x and then execute process P.

- Receive [$x(b).Q$] - Receive name b down the channel named x and then execute Q. This has the effect of binding all occurrences of x in process Q.

- Choice [$P1 + P2$] - Execute exactly one of the processes P1 and P2. The execution of one half of this expression precludes the other half from ever being executed. This operator is associative and commutative.

- Parallel Composition [$P1 \mid P2$] - Execute the processes P1 and P2 in parallel. These two processes may communicate with each other via named channels. This operator is associative and commutative.
- Sequence [$P1 \,.\, P2$] - Execute Process P1. When it completes execute process P2.
- Replication [$!P$] - Execute an infinite number of copies of P in parallel. It is possible to use replication to simulate recursion and therefore not necessary to include a separate operator.

There are two special actions that exist in the π-calculus which should be considered: τ and 0. Firstly, the τ action denotes an internal unobservable action. This action may perform transformations of data or other such actions which are not externally visible. Secondly, the 0 operator signifies explicit termination, for instance, $P.Q.0$ means execute process P, when it completes, execute process Q and then stop. The 0 is often omitted for brevity, simply writing $P.Q$ but where it adds clarity or cannot be implied from the context it is included.

Two forms of π-calculus exist: monadic and polyadic. In the monadic form of π-calculus, only one name may be sent along a channel in an execution step. For instance, $\bar{x}.{<}a{>}.P$ is allowed but $\bar{x}.{<}ab{>}.P$ is not, assuming that a and b are separate names. The polyadic form of π-calculus allows multiple names to be sent and received along a channel in one computation step. It can be shown that the polyadic form is necessary and that the natural monadic abbreviation $\bar{x}.{<}a{>}.\bar{x}{<}b{>}.P$ is not equivalent to the polyadic term $\bar{x}.{<}ab{>}.P$ [18]. This paper deals with the polyadic form of π-calculus.

Computation in π-calculus is defined by a set of reaction rules which describe how a system P can be transformed into P' in one computational step ($P{\rightarrow}P'$). Every computation step in the π-calculus consists of communication between two terms (which may be part of separate processes or the same process). Communication may only occur between two terms which are unguarded (that is they are not part of a sequence prefixed by an action yet to occur) and not alternatives to each other. Consider $P=(...+x(b).Q)\mid(...+\bar{x}{<}a{>}.R)$, when the system is in its initial state P, two parallel processes are executing, and the latter sends the name a along the channel x. The former process receives a along channel x as the sending and receiving terms are complementary and unguarded (said to form a redex). The action of receiving a has the effect of substituting a for b in the process Q and the transformation $P{\rightarrow}P'$ has occurred where $P'=\{a/b\}Q\mid R$. The substitution is denoted by {a/b} in the process P'. A side effect of this communication occurring is that the alternatives (denoted by ...) have been discarded and any communication that they would have performed has been pre-empted. We have now performed one computation step in the system and the system is in a new state.

In many cases there may be multiple states which a process can be transformed into. For example, following process $P=(\bar{x}{<}a{>}.Q)\mid(x(b).R)\mid(x(c).S)$ there are two transformations possible $P{\rightarrow}P'$or $P{\rightarrow}P''$. In the process P, name a is being sent along the channel x but can only be received by one of the other two parallel compositions. Therefore after state P, the following states are $P'=Q\mid\{a/b\}R\mid(x(c).S)$ which assumes that the name a is received by the middle composition causing a substitution of a for b in process R; or $P''=Q\mid(x(b).R)\mid\{a/c\}S$ where a has been received by the other composition and is substituted for c in process S.

It is possible to apply the reaction rules recursively, that is apply them to the state P' that process P has moved into following the previous computation step. If this is followed to its natural conclusion, it can be shown that the system is free of deadlocks and livelocks. This is achieved by reducing the system using the reaction rules and showing that one of the following always holds:

1. Following any reaction, another reaction can occur.
2. Every process in the system is either in its initial state or a termination state where no action terms remain.