Document Flow Model: A Formal Notation for Modelling Asynchronous Web Services Composition

Jingtao Yang, Corina Cîrstea, and Peter Henderson

School of Electronics and Computer Science, University of Southampton,
Southampton SO17 1BJ, UK
{jy02r, cc2, ph}@ecs.soton.ac.uk

Abstract. This paper presents a formal notation for modelling asynchronous web services composition, using context and coordination mechanisms. Our notation specifies the messages that can be handled by different web services, and describes a system of inter-related web services as the flow of documents between them. The notation allows the typical web services composition pattern, asynchronous messaging, and has the capability to deal with long-running service-to-service interactions and dynamic configuration behaviors.

1 Motivation

A service-oriented application is a composition of web services aimed to achieve certain business goals [1,2]. The goals are fulfilled by service-to-service interactions. Our work intends to capture the behavior of service level interactions in a formal notation for specifying web services composition. The language is general and can be applied to various business environments, in order to support web services composition, automation and validation.

First of all, a service-to-service interaction is not just a transaction. Web service transactions are a subset of service interactions [3]. A transaction is a group of logical operations that must all succeed or fail as a group. Currently most service composition specifications such as BEPL4WS only provide mechanisms to support long-running transactions by providing two-phase commit protocols and compensation activities [4]. However, an interaction may include multiple transactions, and can last longer than a transaction. For example, in a banking application, customers pay an estimate amount of money for their utility expense in advance. When the actual numbers arrive a month later, perhaps longer, the difference has to be paid to the customer's account or utility provider's bank account. This interaction is completed by two payment transactions. As long-running interactions are the basis of modern enterprise applications, they should also be considered when specifying services composition.

We believe that service composition should be more dynamic. Storing interaction states in short-lived instances at web containers, as BPEL4WS does, means that services can not be replaced in the middle of interactions. But to meet dynamically-changing business, web service applications are often required to be recomposable.

Context management is a more fundamental requirement than transactions in some business environments [3]. A context allows web services to share information such as message correlations, security tokens and so on. We use the context mechanism to model service interactions by giving each interaction a context. Using this context, a service can continue an interaction that was previously operated by another service. Our model allows interactions and contexts to be structured hierarchically. Thus an interaction could be coordinated by not only a certain service, but also distributed services.

We model services composition by describing the messages exchanged between web services. Each service specifies its contributions to an interaction by updating the interaction state or coordinating its context.

In a previous paper [5], we investigated the requirements for dynamic configurations for service-oriented systems, and argued that long-running interactions are necessarily asynchronous. A notation for describing such systems, called the Document Flow Model (DFM), was also introduced in loc. cit. In this paper, we give a complete formal syntax, and an informal semantics for this notation. We also discuss the use of a special coordination service, in conjunction with a global store, as a means to coordinate interactions over dynamical web services. A formal operational semantics for our notation has also been developed, and is described in [6]. The present paper complements this work by focusing on the use of context and coordination mechanisms in modelling complex web service interactions.

The paper is structured as follows: Section 2 gives the formal syntax, and an informal semantics for our notation. Section 3 uses a job submission example to illustrate the use of DFM in specifying web services composition, and to discuss the use of a special coordination service. Section 4 further discusses the capability of dealing with dynamic configurations, while Section 5 summarizes our approach.

2 Document Flow Model

Our notation describes a system of web services as a set of messages that can be sent from individual services, and the consequences for other services of receiving them. Because messages are basically XML documents, we call it Document Flow Model. A hierarchical (tree) data structure called a document record is used in DFM to abstractly model systems that are eventually realized using XML documents.

The DFM notation is intended to model systems composed by sets of independent web services, orchestrated by asynchronous messages. Since we are not interested in the functionality and performance for each service, we model a web service as a collection of outgoing messages sent in response to an incoming message. Two kinds of communication are supported: one-way communication, which amounts to a service receiving a message, and notification, which amounts to a service sending a message. Request-response and solicit-response [7] conversations are modeled as a one-way communication plus a notification communication.

DFM provides support for long-running interactions and dynamic configurations. One aspect of the ability to simply unplug something in the middle of an interaction and plug in a substitute, is whether or not the component has state. Replacing a stateful component with another is always more difficult than replacing a stateless component with another [8, 9]. This is one of the reasons why one of the main design criteria for web services is that they should be stateless [1]. Our notation models an interaction via stateful messages passed around stateless web services. A context is given to each message to identify the interaction it belongs to. A decentralized context propagation mechanism is used to structure interaction-related data. A persistent component, a ContextStore, is used to maintain the execution state. A stateless web service simply reacts to incoming messages, and updates the state within the ContextStore when necessary. By coordinating all state with the persistent component, an interaction can carry on even if the system configuration has changed.

The Basic Specification Structure. A DFM specification is built from message definitions, or messagedefs. A web service is described by a collection of messagedefs, specifying the messages which the web service receives and operates on.

```
messagedefs ::= messagedef | messagedef messagedefs
messagedef   ::= **OnMessage** message msgdefbody
```

A web service is accessed by XML messages. In DFM, each messagedef defines the service response to an incoming message: when an incoming message matches the message pattern in messagedef, the corresponding actions in msgdefbody are triggered.

The Message Definition Body. A message definition body, msgdefbody, defines the actions to be carried out when an incoming message matches a certain pattern. Possible actions include storing a document in a document store and sending a message.

```
msgdefbody ::= idaction  storebody  sendbody
storebody    ::= _ | storeaction storebody
sendbody     ::= _ | sendaction sendbody | csendaction sendbody
```

A msgdefbody may contain three pieces of information, idaction, storebody and sendbody, in this particular order; any of these can be absent. The idaction describes some new identities used to identify interactions started as a result of a message being acted upon. The storebody describes the set of store actions to be carried out, before the (possibly conditional) message sending actions described in sendbody are carried out in no particular order.

Actions. A message definition may contain essentially four kinks of actions: idaction, storeaction, sendaction and csendaction, as mentioned earlier.

```
idaction ::= _ | generate new ids
ids        ::= id | id, ids
```

When a service starts a new business interaction, it usually creates a new identity to identify that interaction. An idaction specifies the identities generated in this way. The newly generated identities are universally unique, that is, identities generated by the same / different services are different; this can, for instance, be ensured by embedding information such as service identity, date, time and message content in each newly generated identity.

```
storeaction ::= store id->entry in ContextStore
```

A storeaction describes the action of storing a piece of information, an entry, into the ContextStore, under a particular identity id.

```
sendaction   ::= send message
csendaction ::= if condition then { sendactions }
sendactions ::= sendaction | sendaction sendactions
```

A sendaction describes the action of sending out a message. A csendaction specifies one or more sendactions to be performed only when a certain condition (involving the current state of the ContextStore) holds. When the condition evaluates to true, the corresponding sendactions are taken. A simple control flow, a collection of non nested *if… then…*statements, is available in the DFM notation.

Conditions. A condition is a ContextStore evaluation expression, possibly containing logical operators. A simple condition evaluates to true when the specified entries are present in the ContextStore under the identity id, otherwise the condition evaluates to false. Conditions containing logical operators are evaluated in the standard way.

```
condition ::= ContextStore [id] contains entries
            | condition and condition | condition or condition | not condition
```

XML Document Data Structure. Web Services interact with each other by messages which are essentially XML files. To model an XML message, we introduce a new data structure, a document record. A document record allows us to specify the properties of a document. A document record literal consists of a comma-separated list of colon-separated property name / value pairs, all enclosed within square brackets. In the document record, a property name is a string identifier, while a property value is an atom or another document record. A simpler form of document record contains no property names, only property values. In relation to XML, a document record is an XML element. We ignore XML attributes, important though they are in practice, because at the modelling level it is unnecessary to distinguish between nested attributes and nested elements. In a document record, an XML attribute is modeled by a property of that element.

A message is modeled in DFM as a document record with properties **to:**, **query:** and **function:**. The property values to and function are simple strings which describe the message receiver and the requested operation.

```
message ::= [to:to,query:query,function:function]
```

The property value query is a document record that refers to the message data, or message parameters.

```
query    ::= element | [from:from,query:query,context:uid]
                     | [from:from,query:query,result:query,context:uid]

element  ::= string | [elements]
elements ::= element | element, elements
```

Three types of queries are defined in DFM. The first one, element, is a simple document record with no property names, and the property value given by either a string or a list of elements. The second is a document record with **from:**, **query:** and **context:** properties. It includes the query initiator, content and identity. It is used, for example, when a web service initiates a business process by passing a query to other web services. The third is a document record with **from:**, **query:**, **result:** and **context:** properties. When a query has been completed, the results are put into a message together with the original query. As in the message document record, the **query:** and **result:** property values are further document records.

ContextStore. The systems modeled using DFM are concurrent: multiple interaction sessions are carried out at the same time. To maintain the system state, a unique identity is created and assigned to each interaction. The state is structured into document records, entries, and stored under the process identity in the ContextStore.

```
entries  ::= entry | entry, entries
entry    ::= [from:from,query:query] | [from:from,query:query,result:result]
```

An interaction is represented in the ContextStore by a set of entrys which point out that a query has been started, or that the query / its sub-queries have been completed.

3 An Example

We use a job submission system to illustrate our notation. When an application involves a large number of tasks, instead of buying a supercomputer, a more effective way is to deliver subtasks to different computers, and subsequently combine their results. Web services are one of the technologies used to implement such systems.

We have described that our notation allows an interaction to be coordinated by one service or by distributed services. In the following example, we use a Coordination Service to maintain the state of an interaction over stateless web services.
In a previous example [5], all the services participating in an interaction were able to access the state-maintaining component, ContextStore. In this example, the Coordination Service is the only service accessing the ContextStore. The reasons for this are as follows: First, restricting the access largely releases the concurrent control workloads on persistent components, especially in applications involving huge computing tasks. Second, by maintaining the state solely through the Coordination Service, this service can monitor the overall interaction, so that any failure can be detected and recovered timely. Finally, replacing a service with access to the state component is much more complicated than replacing a service with no access to it. Thus, the use of the Coordination Service makes our system more amenable to dynamic reconfiguration.

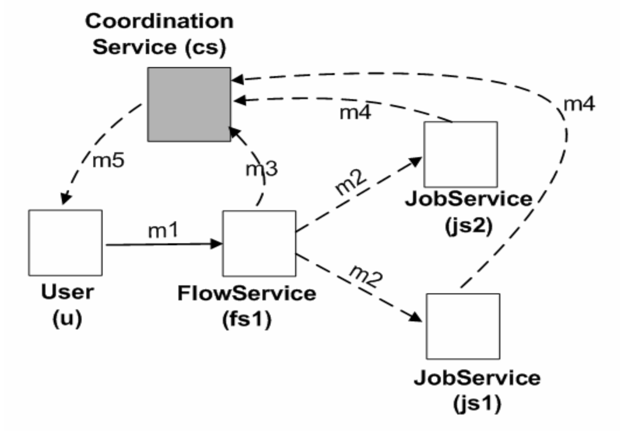

Fig. 1. A job submission system example

Our job submission specification defines three kinds of services. A FlowService, fs1, an orchestrating web service, makes use of JobServices. JobServices, js1 and js2, execute jobs and send results to the Coordination Service (m4). A Coordination Service, cs, coordinates job executions. In our example, when fs1 receives a message with a composed job (m1), it forwards two sub-jobs to js1 and js2 (m2). It also sends the job submission state to cs (m3). Once both sub-jobs have been completed, cs will return their combined result to the user who originally requested the job execution (m5). The different line formats in Fig. 1 are used to distinguish messages with different contexts.

```
OnMessage [ to:FS, query:[from:u,query:[ j1,j2],context:c], function:startjobs ]
        generate new uid
        send [ to:CS, query:[from:FS,query:[from:u,query:[ j1,j2],context:c],context:uid],
                                                                function:jobssubmitted]
        send [ to:JS1, query:[from:FS,query:j1,context:uid], function:jobexecute ]
        send [ to:JS2, query:[from:FS,query:j2,context:uid], function:jobexecute ]
```

Fig. 2. A FlowService Specification

A FlowService (referred to by FS in our specification) requires a number of services, identified by JS1, JS2 and CS, to define its workflow. In the semantics of DFM, such service identifiers are mapped to actual services (e.g. js1, js2 and cs from Fig. 1), thus allowing a system to be dynamically-configured. In particular, the services corresponding to JS1 and JS2 could not only be JobServices, but also FlowServices with the extended capabilities of a JobService (see Section 4).

The user's query contains three parts: a simple query containing job tasks; the user who initiates it; and a context to identify the query. According to the specification in Fig. 2, when a FlowService receives a query from a user, it creates a unique identity, uid, to identify a new interaction. In this case, the job results will be delivered by CS. Thus, the FlowService informs CS that a new interaction has been started, by forwarding the user's query. FS only forwards the actual jobs to the two JobServices, thus preventing a direct interaction between users and JobServices. The FlowService sends out the messages concurrently, and then continues servicing other interleaved queries and replies.

```
OnMessage [ to:JS, query:[from:fs,query:job,context:uid], function:jobexecute]
       send [ to:CS, query:[from:JS,query:job,result:result,context:uid], function:jobcomplete]
```

Fig. 3. A JobService Specification

The JobServices execute jobs and forward their results together with the original queries and contexts, to CS. The original query is required to indicate to the CoordinationService which part of the interaction has been completed. When a

JobService (e.g. js2) receives a query containing two jobs (e.g. in the form [j2,j3]), it executes them and forwards the result [r2,r3] to CS.

```
OnMessage [ to:CS, query:[from:fs,query:[from:u,query:[ j1,j2],context:c],context:uid],
                    function:jobssubmitted ]
  store uid->[ from:fs, query:[from:u,query:[ j1,j2],context:c] ]  in ContextStore
  if ContextStore[uid]  contains  [ from:fs, query:[ from:u,query:[j1,j2],context:c] ],
                                  [ from:js1, query:j1, result:r1 ], [ from:js2, query:j2, result:r2 ]
  then {send [ to:u, query:[from:CS,query:[from:u,query:[ j1,j2],context:c],result:[r1,r2],context:uid],
                    function:jobsreply ] }

OnMessage[ to:CS, query:[from:js,query:job,result:result,context:uid], function:jobcomplete]
  store uid->[from:js,query:job,result:result]  in ContextStore
  if ContextStore[uid]  contains  [ from:fs, query:[ from:u,query:[ j1,j2], context:c] ],
                                  [ from:js1, query:j1, result:r1], [ from:js2, query:j2, result:r2 ]
  then {send [ to:u, query:[from:CS,query:[from:u,query:[ j1,j2],context:c],result:[r1,r2],context:uid],
                   function:jobsreply ] }
```

Fig. 4. A Coordination Service Specification

Because we assume that communications between services are asynchronous, the messages received by the Coordination Service are in an undetermined order. In our solution, each time the Coordination Service receives a message, it takes all contents except the context of the query and stores them into the ContextStore under the interaction's unique identity. The Coordination Service then checks if the ContextStore contains all the queries and results of that interaction. When sufficient information has been gathered, the Coordination Service replies to the user.

We can therefore see that the combination of a ContextStore and stateful interactions is sufficient to solve the problem of asynchronous coordinated interactions.

4 Discussion

The previous example shows how to describe asynchronous interactions in DFM. The DFM notation also aims to support dynamic configurations. Dynamic configuration is a very complex issue, especially in distributed systems. Our work is only concerned with high level interactions: we only model and analyse the integrity of an interaction, assuming that all the messages are safe and reliable.

To improve performance, some new services are added to the job submission system, as in Fig. 5. Specifically, the JobService js2 is replaced by a FlowService fs2 that has access to JobServices js2 and js3. The FlowService fs1 behaves as in the previous example, except that it will now send a job execution request (m6) to fs2 instead of js2. (The actual specification of the FlowService remains unchanged as far as receiving messages from users is concerned. The only change is in how the service identifier JS2 known to fs1 is mapped to an actual service.) Upon receiving a request from fs1, the FlowService fs2 passes two sub-jobs to the JobServices js2 and js3. This way, the jobs received by fs1 can be executed simultaneously by three JobServices.

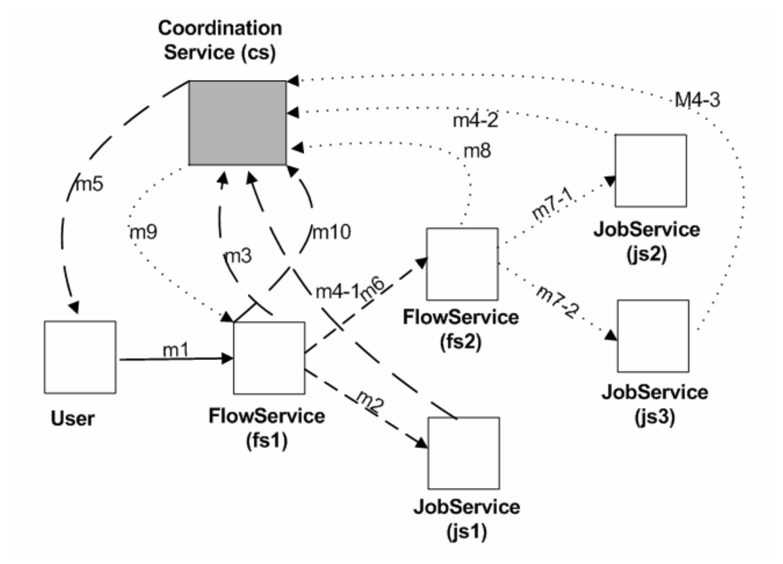

Fig. 5 A complex job submission example

Since all the interactions are coordinated by the Coordination Service and ContextStore, all interactions started before and after the re-configuration can be executed consistently and without interruption. Also, when replacing a service, an important requirement is that other services do not need to be aware of the change. Therefore, after the re-configuration, the FlowService fs2 needs to behave as a JobService when it receives jobs from fs1 (m6), and as a FlowService when it passes the sub-jobs to js2, js3 (m7) and forwards the job state to cs (m8).

Thus, two messagedefs need to be added to the FlowService specification: one describes how a FlowService (in our example, fs2) handles a jobexecute request, the other describes how a FlowService (fs1 in our case) handles a jobsreply message.

```
OnMessage [ to:FS, query:[ from:u,query:[ j1,j2],context:c], function:jobexecute ]
    generate new uid
    send [ to:CS, query:[ from:FS,query:[from:u,query:[ j1,j2],context:c],context:uid],
            function:jobssubmitted ]
    send [ to:JS1, query: [ from:FS,query:j1,context:uid], function:jobexecute ]
    send [ to:JS2, query: [ from:FS,query:j2,context:uid], function:jobexecute ]

OnMessage [ to:FS, query:[from:cs,query:[from:u,query:job,context:c],result:result,context:uid],
            function:jobsreply ]
    send [ to:CS, query:[ from:FS,query:job,result:result,context:c], function:jobcomplete ]
```

Fig. 6. Updated FlowService Specification

The first messagedef is similar to startjobs in Fig. 2, but with a different function name. This time, the user, u, of fs2 is a FlowService, fs1. When fs2 receives a jobexecute message, it will start a new interaction by creating a new identity. As a result, the Coordination Service will create two interaction records in the ContextStore, for fs1-submitted jobs and fs2-submitted jobs (Table 1). From the point of view of the Coordination Service, fs1 is the user of fs2-submitted jobs; thus, the Coordination Service will send replies to fs1 corresponding to those jobs.

The second messagedef specifies that, when fs1 receives a jobsreply message from the Coordination Service, it extracts the query and informs the Coordination Service that the jobs previously submitted to fs2 have been completed. A jobcomplete message received by the Coordination Service from a FlowService produces the same result as one received from a JobService.

Table 1. Interactions State at ContextStore

ContextStore[uid1]	**ContextStore[uid2]**
[**from:** fs1, **query:** [**from:** u, **query:** [j1,[j2,j3]], **context:** c]], [**from:** js1, **query:** j1, **result:** r1], [**from:** fs2, **query:** [j2,j3], **result:** [r2,r3]]	[**from:** fs2, **query:** [**from:** fs1, **query:** [j2,j3], **context:** uid1]], [**from:** js2, **query:** j2, **result:** r2], [**from:** js3, **query:** j3, **result:** r3]

This example demonstrates the capability of using DFM to model long-running interactions and dynamic configurations. By sharing interfaces, a service behaves multi-functionally. Assuming that the FlowService fs1 is able to handle a jobsreply message, this service doesn't have to be stopped and rewritten in order to allow the JobService js2 to be replaced by a FlowService fs2 (that is able to handle a jobexecute message). Also, using our context and coordination mechanisms, a query can be completed by two hierarchical interactions (Table 1). Therefore a service can not only be replaced by a peer service, but also by a service with more workflows.

We also note that our query structure and use of contexts is highly flexible in terms of the kinds of service-oriented applications it can model. An application coordinated by more than one service could, for instance, be modeled with a less hierarchical query structure than in our previous example. Applications involving service compositions that do not require either a CoordinationService or a ContextStore can also be described. In addition to modelling service compositions, our context and coordination mechanisms can also be used to model business processes in specific domains, or security sensitive applications.

5 Conclusion

A service-oriented application is composed by dynamic services orchestrated using asynchronous messages. We have introduced a formal modelling notation which uses context and coordination mechanisms to specify asynchronous web services composition, and has the additional capability to support long-running interactions and dynamic configurations. An operational semantics for this notation has already been developed, see [6] for details. Future work includes the use of this operational semantics to develop a simulation tool for asynchronous web services coordination.

References

1. Booth, D. et al.: Web Services Architecture, http://www.w3.org/, 2004.
2. Peltz, C.: Web services orchestration and choreography, IEEE Computer, vol. 36, 2003, pp. 46-52.
3. Bunting, D. et al.: Web Services Composition Application Framework (WS-CAF) V1.0, http://www.oasis-open.org, 2003.
4. Andrews, T. et al.: Business Process Execution Language for Web Services Version 1.1, http://www.ibm.com/developerworks/library/ws-bpel/, 2003.
5. Henderson, P. and Yang, J.: Reusable Web Services, Proceedings of 8th International Conference, ICSR 2004, Madrid, Spain, 2004, pp. 185-194.
6. Yang, J., Cîrstea, C. and Henderson, P.: An Operational Semantics for DFM, a Formal Notation for Modelling Asynchronous Web Services Coordination, accepted by First International Workshop on Services Engineering (SEIW 2005), Melbourne, Australia, 2005.
7. Christensen, E. et al.: Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl, 2001.
8. Henderson, P.: Laws for Dynamic Systems, Proceedings of International Conference on Software Re-Use (ICSR 98), Victoria, Canada, 1998.
9. Henderson, P.: Modelling Architectures for Dynamic Systems. In: McIver, A. and Morgan, C. (eds.), Programming Methodology, Monographs in Computer Science, Springer, 2003.