An Operational Semantics for DFM, a Formal Notation for Modelling Asynchronous Web Services Coordination

Jingtao Yang, Corina Cîrstea, Peter Henderson
School of Electronics and Computer Science
University of Southampton, U.K.
{jy02r, cc2, ph}@ecs.soton.ac.uk

Abstract

This paper presents the DFM notation and its operational semantics. DFM (Document Flow Model) is a message-based workflow notation for modelling asynchronous web services communication, which uses coordination mechanisms to support dynamic configurations and long-running business processes. The operational semantics of a DFM specification describes the possible behaviours of a system of inter-related web services, in terms of the messages that can be exchanged during the execution of one or more business processes, and the effect each message execution has on the business process state.

1. Introduction

Web Services technology enables modern applications implemented over heterogeneous web environments. A web service publishes its interfaces in an XML document, WSDL, and is invoked through an XML-based messaging protocol, SOAP [1]. Using platform independent and standard XML documents, a service consumer can invoke a service following this shared understanding, without being concerned with how the service is implemented.

A service-oriented application is composed of dynamic web services orchestrated using asynchronous messages. The web services are owned and managed by several business partners. This architecture provides benefits over traditional applications on interoperability, flexibility and dynamic configuration. However, it also adds considerable complexity to the implementation and verification [2].

2. Motivation

The aim of our work is to investigate service-oriented system environments, and use formal modelling to capture a business process as a workflow specification, in order to facilitate service-oriented system automation and validation [3, 4].

Business Process Execution Language for Web Services (BPEL4WS) has been positioned as the standard for specifying web service workflows. Other workflow definition languages [5, 6] leverage the concepts from BPEL4WS, and focus on workflow patterns. BPEL4WS separates a workflow from the web services involved in its execution, by using a pre-defined workflow instance with full interaction states communicating with stateless services [7]. However, BPEL4WS provides limited support for long-running interactions, where failures are more likely to happen, and where services might require reconfiguration while some of their interactions are still active: on the one hand, failure of the workflow instance can result in the interaction state being lost; on the other hand, business process workflows can not be updated while processes are still running [8]. A mechanism for coordinating complex services and for managing long-running transactions is therefore required.

The requirements for dynamic web services that use asynchronous messaging were investigated in [9]. A formal notation for describing the flow of documents in such systems, DFM, was also introduced in loc. cit. (see also [10]). The most important features of DFM are summarized below:

- DFM uses an XML-like tree data structure to specify documents in a concise notation.
- DFM models asynchronous communication, and supports two kinds of communication patterns: 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 [11] conversations are modelled in DFM as a one-way plus a notification communication.
- DFM uses a coordination mechanism to support long-running interactions and dynamic

configurations. This includes the use of contexts to identify process states, of a decentralized context propagation mechanism to structure process-related data, and of a persistent component, a ContextStore, to maintain process states.

The syntax and informal semantics of DFM are summarized in Section 3. (Further details can be found in [10].) An operational semantics for DFM is then presented in Section 4.

3. The DFM notation

DFM is a message-based workflow notation, used to specify systems of independent web services, coordinated via asynchronous messages. Since messages are essentially XML documents, we call this notation Document Flow Model.

The systems specified in DFM are concurrent: multiple process sessions are carried out at the same time. Unique identities are used to distinguish between different processes or sub-processes. These identities are generated by a service following some request (incoming message), and are passed as parameters to outgoing messages. To ensure process integrity, the actual process state is maintained in a persistent component, ContextStore, and the process identity is used to access and update this state.

Web services interact with each other by messages. Since we are not interested in the detailed functionality of each service, we model a service as a collection of outgoing messages generated in response to an incoming message. The outgoing messages are independent, and can be sent out in any order. Prior to sending these messages, a service may perform an update of relevant process states, as maintained in the ContextStore. Also, in determining the messages to be sent, the service may rely on an evaluation of certain process states (subsequent to any state updates).

A web service is described in DFM using a set of message definitions. Each message definition, messagedef, specifies the actions taken by the service when executing a particular message. This includes actions for generating new process identities, actions for storing process data into the ContextStore, and actions for sending messages, either unconditionally or upon certain conditions being satisfied. A condition is a ContextStore evaluation expression, possibly containing logical operators. Simple conditions evaluate to true when certain entries, containing process data, are present in the ContextStore under a specified process identity, id. The three types of actions, i.e. generating process identities, storing process data and sending messages, are performed in this particular order. The storebody specifies the store actions to be carried out concurrently, after the process identities described in idaction have been generated, and before the message sending actions described in sendbody are carried out (also concurrently).

A document record data structure is used to specify the properties of an object, in an XML-like fashion. A document record consists of a list of property name/value pairs, or simply of a list of values, enclosed within square brackets. The values can themselves be document records, thus allowing for an arbitrary degree of nesting in a document record. For example, a message is a document record with property names **to:**, **query:** and **function:**, and with the corresponding property values describing the message receiver, the message data (or parameters), and the requested operation.

We use a travel booking example to illustrate the DFM notation.

```
messagedefs ::= messagedef | messagedef messagedefs
messagedef  ::= onMessage message  idaction  storebody  sendbody
storebody   ::= _ | storeaction  storebody
sendbody    ::= _ | sendaction  sendbody  | csendaction sendbody
idaction    ::= _ | generate new ids                    ids ::= id | id, ids
storeaction ::= store id -> entry in ContextStore       id  ::= string
sendaction  ::= send message
csendaction ::= if condition then { sendactions }       sendactions   ::= sendaction | sendaction sendactions
condition   ::= ContextStore [id] contains entries | condition and condition | condition or condition | not condition
entries     ::= entry | entry, entries
entry       ::= [from:from, query:query ] | [from:from , query:query, result:query]             from ::= string
message     ::= [ to:to, query:query, function:function]        to ::= string       function ::= string
query       ::= element | [from:from,query:query,context:id] | [from:from,query:query,result:query,context:id]
element     ::= string | [elements]                              elements ::= element | element, elements
```

Figure 1. Formal syntax for DFM

```
onMessage[to:a,query:[from:u,query:[f,h],context:c],
                                    function:bookTravel]
generate new id
store id->[from:u, query:[f,h], context:c] in ContextStore
send[to:fs,query:[from:a,query:f,context:id],
                                    function:bookFlight]
send[to:hs,query:[from:a,query:h,context:id],
                                    function:bookHotel]
```

Figure 2. Travel agent spec-I

Upon receiving a user request, the TravelAgent, a, creates a new process identity, id. It then stores the user query into the ContextStore. In this example, the TravelAgent decomposes the query into two new queries, to be sent to the FlightShop, fs, and the HotelShop, hs, using then new process identity. The two messages are parallel messages, i.e. they could be sent out in any order.

```
onMessage[to:a,query:[from:s,query:f,result:r,
                 context:id], function:shopReply]
store id -> [from:s, query:f, result:r] in ContextStore
if  ContextStore[id]  contains
      [from:a, query:[from:u,query:[f,h],context:c]],
      [from:fs,query:f,result:r1],[from:hs,query:h,result:r2]
then { send[to:u,
         query:[from:a,query:[from:u,query:[f,h],context:c],
               result:[r1,r2], context:id],
         function:bookReply] }
```

Figure 3. Travel agent spec-II

When the TravelAgent receives a reply from a shop service, it reads the context of the query and saves the other parts of the query into the ContextStore under that process identity. Since the systems specified in DFM are asynchronous, the two shop replies can arrive in any order. Therefore, each time the agent receives a reply, it will check the process state, and if the process has been completed a reply message will be sent to the user, otherwise no action will be taken.

4. Operational Semantics

We now present an operational semantics for DFM. The semantics describes the possible behaviours of a system consisting of a number of related services, in terms of the messages that can be executed in particular states of the system, and the effect such message executions have on the system state. The execution of a message is handled by the web service to which the message was sent. Since communication is asynchronous, the sequence of message executions is undetermined.

The environment within which the communications are taking place is modelled using a virtual daemon. The daemon uses a message pool to manage the sending and receiving of messages by the services, and at the same time maintains service configurations.

The execution of a message is carried out in the following key steps: the message is matched to a message pattern; the actions which need to be carried out are determined, based on information generated from the system specification and on the current process state; and finally, the resulting message patterns are evaluated, and the result is added to the message pool.

Our operational semantics supports dynamic configurations, by allowing the service names used in system specifications to be mapped to actual services at runtime. This is achieved using the configuration tables dynamically provided by the environment.

Formally, the operational semantics associates, to each DFM specification, a labelled transition system whose states correspond to possible states of the system (defined by the messages awaiting execution and by the current state of the ContextStore), and whose labels correspond to message executions. Defining this transition system requires several auxiliary functions, which we now describe.

4.1. Specification functions

The following information can be extracted from a given DFM specification.

4.1.1. ServiceId. The set ServiceId contains all the service identifiers present in the specification:

ServiceId = {s | to:s appears inside a message}

4.1.2. Message. The set Message contains all message patterns present in the specification:

Message = {m |
onMessage m appears inside a messagedef}

4.1.3. The ids, context and vars functions. As part of executing a message, new process identities can be created, in order to identify specific sub-processes. The ids function gives, for each message m, the process identities generated upon the execution of m:

ids: Message $\rightarrow \mathbb{P}$ (String)
ids(m) = {id | id appears in the idaction of **onMessage** m}

Similarly, the context and vars function give, for each message m, the process identities passed as parameters to m using the **context:** property, and the names of all the other variables used as parameters in definition of m, respectively:

context, vars : Message → ℙ (String)
context(m) = { id | **context:**id appears inside m}
vars(m) = {var | var appears as an element inside m}

4.1.4. Store action function. The function storeactions gives, for each message m, the set of storeactions to be carried out upon the execution of m.

StoreAction = {storeA | storeA is generated by storeaction}

storeactions: Message → ℙ (StoreAction)
storeactions(m) = {storeA∈StoreAction |
storeA appears inside **onMessage** m}

4.1.5. Send action functions. The function sendactions gives, for each message m, the set of sendactions to be carried out unconditionally upon the execution of m:

SendAction = {sendA | sendA is generated by sendaction}
sendactions: Message → ℙ (SendAction)
sendactions(m) = {sendA∈SendAction |
sendA appears inside **onMessage** m}

For each message m, the set conds(m) gives the conditions which must be evaluated as part of the execution of m:

Condition = {c | c is generated by condition}
conds(m) = {c∈Condition| c appears inside **onMessage** m}

while the function csendactions(m) gives the actions associated to each such condition:

csendactions(m) : conds(m) → ℙ (SendAction)
csendations(m)(c) = {sendA∈SendAction | sendA appears
inside **if** c **then {…}**, inside **onMessage** m }

4.2. Semantic Functions

A number of semantic functions will be used to describe the effect of message executions, both on the message pool (where new messages will typically be added), and on the current state of the ContexStore (where some process states will be updated).

4.2.1. Context Store. The ContextStore stores business process states using entries, and organizes them using process identities.

Entry = {e | e is generated by entry}
ContextStore = string → ℙ(Entry)

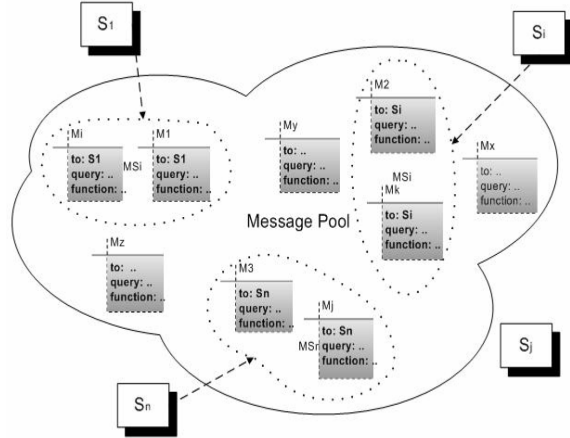

Figure 4. A message pool

4.2.2. Service. The set Service contains the names of all services relevant to a particular specification. In the following, we assume Service = $\{S_1, S_2, \dots S_n\}$. For each service, S_i, a message-set, MS_i, gives the pending messages of S_i., as found in the message pool (see also Figure 4). In addition, for each service S_i, a configuration table links the service identifiers used in the definition of S_i to actual services (elements of Service). This is captured by the function:

$config_i$: ServiceId → Service

Table 1. A TravelAgent service configuration

Service Id	Service
a	S_1
fs	S_3
…	…

4.2.3. MessageVal. The message pool is the container for messages awaiting execution. The set MessageVal contains all possible such messages.

MessageVal = {**[to:**t, **query:**q, **function:**f**]** | t, q, f ∈ string}

4.2.4. Matching function. For each service S_i, the $matches_i$ function gives the message pattern m that corresponds to a message M from the message pool.

$matches_i$: MessageVal → Message
$matches_i$(**[to:**t, **query:**q, **function:**f**]**) =
[to:t', **query:**q', **function:**f**]** ∈ Message if $config_i$(t') = t

4.2.5. Evaluation of a document. The evaluation function $eval_{M,i}$ defines how to evaluate document expressions appearing inside a message definition m, based on the values provided by a corresponding actual message M waiting to be executed by S_i, and on the values generated from the environment upon the execution of M. The function $eval_{M,i}$ is defined inductively on the structure of document expressions. The base cases correspond to service names, message parameters/contexts and process identities:

$eval_{M,i}(t)$ = $config_i(t)$ if m=$matches_i(M)$
and **to:**t or **from:**t inside m
$eval_{M,i}(v)$ = value obtained from m=$matches_i(M)$,
if v ∈ vars(m) ∪ context(m)
$eval_{M,i}$ (id) = new value generated from the environment,
if m=$matches_i(M)$ and id ∈ ids(m)

while the induction cases correspond to messages, queries and entries:

$eval_{M,i}$ ([**to:**t,**query:**q,**function:**f])
= [**to:**$eval_{M,i}(t)$, **query:**$eval_{M,i}(q)$, **function:**f]
$eval_{M,i}$([var1,..,varn]) = [$eval_{M,i}$(var1),..,$eval_{M,i}$(var2)]
$eval_{M,i}$([**from:**f, **query:**q, **context:**c])
= [**from:**$eval_{M,i}(f)$,**query:**$eval_{M,i}(q)$,**context:**$eval_{M,i}(c)$]
$eval_{M,i}$([**from:**f, **query:**q, **result:**r, **context:**c])
= [**from:**$eval_{M,i}(f)$, **query:**$eval_{M,i}(q)$,**result:**$eval_{M,i}(r)$,
context:$eval_{M,i}(c)$]
$eval_{M,i}$([**from:**f, **query:**q])
= [**from:**$eval_{M,i}(f)$, **query:**$eval_{M,i}(q)$]
$eval_{M,i}$([**from:**f, **query:**q, **result:**r])
= [**from:**$eval_{M,i}(f)$, **query:**$eval_{M,i}(q)$, **result:**$eval_{M,i}(r)$]

4.2.6. Evaluation of a condition. An additional function needs to be defined for evaluating the conditions appearing inside message definitions, given an actual message M waiting to be executed by service S_i, and a particular state of the ContextStore, CS:

$eval_{M,i}$: conds(m) × ContextStore → {true, false}
$eval_{M,i}$ (**ContextStore**[id] **contains** e1,…,en , CS)
= true if each $eval_{M,i}$(ej) ∈ CS[$eval_{M,i}$(id)]
= false otherwise

The boolean operators are evaluated in the usual way.

4.2.7. Send function. For each message M waiting to be executed by a service S_i, and each state CS of the ContextStore, the function $send_fun_{M,CS,i}$ gives, for each service, S_j, the messages to be sent to S_j as a result of executing M:

$send_fun_{M,CS,i}$: Service → ℙ(MessageVal)
$send_fun_{M,CS,i}$ (S_j) = {M' ∈MessageVal | m=$matches_i(M)$
and send m' ∈ sendactions(m) ∪
(csendactions(m)(c) where $eval_{M,i}$(c,CS)=true)
and **to:**t in m' and $eval_{M,i}(t)=S_j$
and M'=$eval_{M,i}$(m')}

4.3. System configurations and transitions

The operational semantics of a DFM specification is defined in terms of transitions between system configurations, where a configuration describes the messages waiting to be executed by each service, together with the current state of the ContextStore. Formally, a configuration is tuple:

$$(MS_1,...,MS_n,CS) \in \mathbb{P}(MessageVal)\times\ldots\times \mathbb{P}(MessageVal) \times ContextStore$$

and transitions between configurations have the form:

$$(MS_1,...,MS_n,CS) \xrightarrow{execute(M)} (MS_1',\ldots,MS_n',CS')$$

where the latter configuration is completely determined by executing M in the former configuration. A single operational rule, given in Figure 5, describes when such a transition is possible, and what the outcome of the transition is. Specifically, the message M being executed must belong to some message-set MS_i; its execution (by service S_i) results in M being taken out from the message-set, in some process states being updated, and in new messages being added to some of the message-sets. The $matches_i$

$$(\mathbf{MS_1,\ldots,MS_n,\ CS}) \xrightarrow{\mathbf{execute(M)}} (\mathbf{MS_1',\ \ldots,MS_n',\ CS'})$$

if (M ∈ MS_i)
where:
$matches_i(M)$ = m
for id ∈ ids(m)∪context(m), CS'[$eval_{M,i}$(id)] = CS[$eval_{M,i}$(id)] ∪ {$eval_{M,i}$(e) | **store** id → e ∈ storeactions(m)}
$MS_i' = MS_i \setminus \{M\} \cup send_fun_{M,CS',i}(N_i)$, and for j ≠ i, $MS_j' = MS_j \cup send_fun_{M,CS',i}(N_j)$

Figure 5. A transition rule

function (of service S_i) is used to determine the message pattern m that matches the message M, and subsequently the $eval_{M,i}$ and $send_fun_{M,CS,i}$ functions are used to compute the required state updates (as specified in the storeactions of m), and the new messages to be added to the message-sets (as specified in the sendactions of m).

This operational rule can be used to generate a transition system, whose states are configurations, and whose transitions are all possible instances of the given rule. Any configuration containing at least one message in one of the message-sets can, in principle, be chosen as an initial state. Unfolding the transition system starting from this state yields all the behaviours which can be exhibited by the services $S_1,...,S_n$, while they cooperate towards the execution of the messages in the initial configuration.

For the travel booking example of Section 3, letting the initial state consist of a single message bookTravel(A) yields the transition system in Figure 6 (where only the message part of configurations is shown).

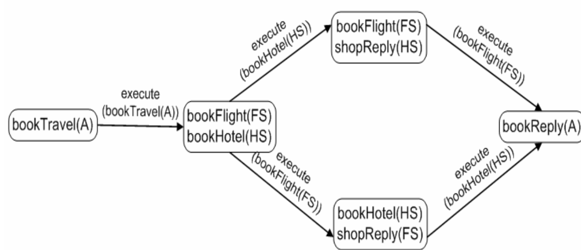

Figure 6. A TravelAgent process flow

4.4. Discussion

Web services are independent entities in a service-oriented system. As a result, the sending of a message should not rely on the availability of other web services. A web service should be able to pick up a message after it has recovered from a failure, or immediately after its addition to a web service system. The operational semantics presented here conforms to this asynchronous communication behaviour which also supports dynamic configurations.

Earlier in this section we discussed the use of configuration tables in mapping system prototypes to actual services. Transition rules for changing system configuration (e.g. by adding or removing a service) have not been discussed here due to lack of space, but it is relatively straightforward to extend our operational semantics to also account for the dynamic aspect of web service systems.

5. Conclusion

We have described a message-based workflow notation for modelling asynchronous web services communication, with the capability to support long-running interactions and dynamic configurations. A formal operational semantics for this notation has also been presented. Future work includes the use of this operational semantics to develop a simulation tool for asynchronous web services coordination.

6. References

[1] "Web Services Architecture", http://www.w3.org, W3C Working Group Note 11 February 2004.

[2] C. Peltz, "Web services orchestration and choreography", *Computer, vol. 36*, 2003, pp. 46-52.

[3] D. Hollingsworth, "The Workflow Reference Model Workflow Management Coalition", WFMC org, 1995.

[4] M. H. D. Georgakopoulos, A. Sheth, "An Overview of workflow management: From process modeling to workflow automation infrastructure", *Distributed and Parallel Databases, vol*. 3, 1995, pp. 119-153.

[5] D. Cybok, "A Grid Workflow Infrastructure", *presented* at *GGF 10*, Berlin, March 2004.

[6] T. Fahringer, J. Qin and S. Hainzer, "Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow language", *IEEE International Symposium on Cluster Computing and the Grid 2005,* Cardiff, UK, 2005.

[7] "Business Process Execution Language for Web Service", BEA, IBM, Microsoft, 2002. ftp://www6.software.ibm.com/software/developer/library/

[8] "BPWS4J - A platform for creating and executing BPEL4WS processes", IBM alphaWorks, http://www.alphaworks.ibm.com/tech/bpws4j, 2004.

[9] P. Henderson. and J. Yang, "Reusable Web Services", *Proceedings of 8th International Conference, ICSR 2004*, Madrid, Spain, 2004, pp. 185-194.

[10] J. Yang, C. Cîrstea and P. Henderson, "Document Flow Model: A Formal Notation for Modelling Asynchronous Web Services Composition", submitted to *AWeSOMe'05,* 2005.

[11] E. Christensen et al, "Web Services Description Language (WSDL) 1.1", http://www.w3.org/TR/wsdl ,W3C, 2001.