Implementing Workflow Reconfiguration in WS-BPEL

Manuel Mazzara
UNU-IIST, Macau
and Newcastle University, UK
manuel.mazzara@ncl.ac.uk

Nicola Dragoni*
Technical University of Denmark (DTU)
Copenhagen, Denmark
ndra@imm.dtu.dk

Mu Zhou
Technical University of Denmark (DTU)
Copenhagen, Denmark
mu.zhou31@gmail.com

Abstract

This paper investigates the problem of dynamic reconfiguration by means of a workflow-based case study used for discussion. We state the requirements on a system implementing the workflow and its reconfiguration, and we describe the system's design in BPMN. WS-BPEL, a language that would not naturally support dynamic change, is used as a target for implementation. The WS-BPEL recovery framework is here exploited to implement the reconfiguration using principles derived from previous research in process algebra and two mappings from BPMN to WS-BPEL are presented, one automatic and only mostly manual. Differences between the two are finally detailed.

Keywords: Workflow Reconfiguration, BPMN, WS-BPEL

1 Introduction

Modern dependable systems are required to be fexible, available and dependable and dynamic reconfiguration is one way of achieving these requirements. While a significant amount of research has been performed on hardware reconfiguration (see [3], and [7]), reconfiguration of services — especially regarding computational models, formalisms and methods — has not been fully explored yet. In [13] and [12] these observations lead to the conclusion that further research is required on dynamic reconfiguration of dependable services, and especially on its formal foundations, modelling and verification. To bring our contribution to this research field, we first examined a number of well-known formalisms for their suitability for reconfigurable dependable systems [13] and then we approached a case study of workflow reconfiguration using an asynchronous π-calculus [8] and $\texttt{web}\pi_\infty$ [14] to model the design and to verify whether or not it meets the requirements [12]. In this paper, instead, we describe the same workflow reconfiguration and how to implement it in WS-BPEL [9], a language that would not natively support reconfiguration. The major contribution of this paper is showing how WS-BPEL can be exploited in implementing a workflow reconfiguration by means of its recovery framework [5]. This evaluation may be useful to system designers intending to design dynamically reconfigurable systems as well as WS-BPEL specialists who have to cope with workflow reconfigurations which are not natively supported in the language.

2 Office Workflow: Requirements and Design

This case study describes dynamic reconfiguration of an office workflow for order processing that is commonly found in large and medium-sized organizations [6]. These workflows typically handle large

Journal of Internet Services and Information Security (JISIS), volume: 2, number: 1/2, pp. 73-92

*Corresponding author: Department of Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Kongens Lyngby, Denmark, Tel: +45-45253731

numbers of orders. Furthermore, the organizational environment of a workflow can change in structure, procedures, policies and legal obligations in a manner unforseen by the original designers of the workflow. Therefore, it is necessary to support the unplanned change of these workflows. Furthermore, the state of an order in the old configuration may not correspond to any state of the order in the new configuration (as we shall see). These factors, taken in combination, imply that instantaneous reconfiguration of a workflow is not always possible; neither is it practical to delay or abort large numbers of orders because the workflow is being reconfigured. The only other possibility is to allow overlapping modes for the workflow during its reconfiguration.

2.1 Requirements

A given organisation handles its orders from existing customers using a number of activities arranged according to the following procedure:

(1) `Order Receipt`: an order for a product is received from a customer. The order includes customer identity and product identity information.

(2) `Evaluation`: the product identity is used to perform an inventory check on the availability of the product. The customer identity is used to perform a credit check on the customer using an external service. If both the checks are positive, the order is accepted for processing; otherwise the order is rejected.

(3) `Rejection`: if the order is rejected, a notification of rejection is sent to the customer and the workflow terminates.

(4) If the order is to be processed, the following two activities are performed concurrently:

   (a) `Billing`: the customer is billed for the total cost of the goods ordered plus shipping costs.

   (b) `Shipping`: the goods are shipped to the customer.

(5) `Archiving`: the order is archived for future reference.

(6) `Confirmation`: a notification of successful completion of the order is sent to the customer.

In addition, for any given order, `Order Receipt` must precede `Evaluation`, which must precede `Rejection` or `Billing` and `Shipping`.

After some time, managers notice that lack of synchronization between the Billing and Shipping activities is causing delays between the receipt of bills and the receipt of goods that are unacceptable to customers. Therefore, the managers decide to change the order processing procedure, so that `Billing` is performed before `Shipping` (instead of performing the two activities concurrently). During the transition interval from one procedure to the other, the following requirements must be met:

(1) The result of the `Evaluation` activity for any given order should not be affected by the change in procedure.

(2) All accepted orders must be billed and shipped exactly once, then archived, then confirmed.

(3) All orders accepted after the change in procedure must be processed according to the new procedure.

2.2 Design

In this section we present a design of the office workflow case study by means of the Business Process Modeling Notation [2]. The choice of using BPMN as a design tool is motivated by its wide adoption as graphical representation for specifying business processes. Indeed, BPMN is currently maintained by the Object Management Group (OMG) [15], representing a standard for business process modeling.

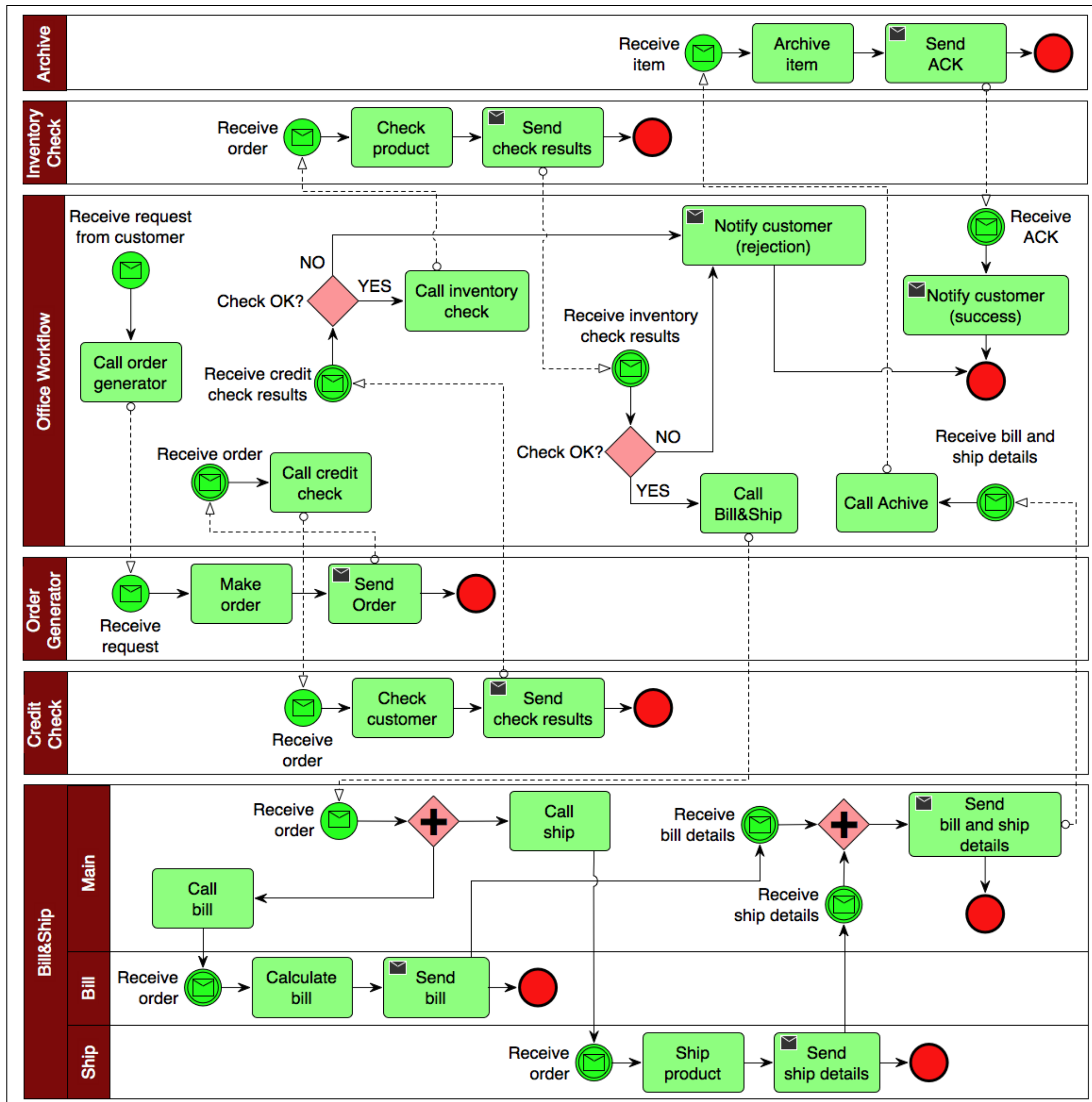

Figure 1: Office workflow - BPMN diagram of the original configuration

The BPMN diagram representing the original configuration of the office workflow is shown in Figure 1. The diagram is based on six pools representing different functional entities. It is worth noting that this is not strictly imposed by our requirements. Indeed, only the credit check service is supposed to be

external (Section 2.1). Thus, each other service might be included in a lane of a single pool, representing in this way a specific activity within the organization. However, we decided to adopt a pool for each service to design a more generic situation where the different services are offered by external parties.

The coordinating entity is represented by the pool Office Workflow. When a customer's request is received, an order is created by calling the Order Generator entity. This order is then sent to both the credit check handler (Credit Check pool) and the inventory check handler (Inventory Check pool). The former is used to check the customer credit's availability, the latter to verify the availability of the product. Note that the inventory check is performed only in case the credit check is successful. This has been expressed by means of an Exclusive Data-Based Gateway. In case of a negative reply from Credit Check, a notification is sent to the customer, the order is rejected and the overall workflow terminates. We do the same with the results from Inventory Check: by means of an Exclusive Data-Based Gateway we proceed only in case of a positive reply. We notify the user and reject the order in case of a negative reply. Thus, according to the requirements, if both the checks are positive the order is processed; otherwise the order is rejected and the customer notified. The Bill&Ship pool represents the entity responsible for both the billing and shipping activities, which are represented as two different lanes (Bill and Ship, respectively) of this pool. Note that when the order is received by Bill&Ship, the two activities are called concurrently by means of a Parallel gateway. The same gateway is used to merge the results from Bill and Ship. The bill and ship details are then sent to the caller (Office Workflow) which, according to the requirements, calls the Archive service for storing the order. Finally, a successful notification is sent to the customer and the workflow terminates.

It is worth noting that, for the sake of simplicity and readability of the overall workflow, we assume that neither the billing activity nor the shipping activity provides a negative result. This explains why we do not check the results of such activities, for instance notifying the user in case of a negative result. Adding these further checks would be straightforward, since it could be done by means of two Exclusive Data-Based Gateways (as in the Office Workflow pool).

Let us now focus on the reconfiguration problem. The key change concerns the order of billing and shipping activities: instead of calling the two activities concurrently, the organization now requires that billing is performed before shipping. Looking at the design we have presented so far, it should be not so difficult to realize this reconfiguration requires a change in the main lane of Bill&Ship only (that is, where the billing and shipping activities are called, and therefore their invocations ordered), while the rest of the workflow remains unaltered. The resulting BPMN diagram is shown in Figure 2. Technically speaking, the Parallel gateways have been removed and the two activities are now called synchronously.

The key issue which remains unanswered is how the transition from the original configuration (Figure 1) to the new one (Figure 2) can be done. In other words, how the reconfiguration that we have applied can be designed. Figure 3 shows exactly this, i.e., the overall workflow during its reconfiguration. The basic idea is that we have a default flow that is exactly the same of the one of the original configuration. This default flow can be altered through an interrupting message event contained in a ”Determine configuration” activity, an activity that determines which configurations should be used when Bill&Ship is called. This activity has been included in a separate pool (Reconfig. region) to highlight where the flow can take one the two different directions. Moreover, in this way we represent a possible authority in charge of deciding the reconfiguration. Thus, if the interrupting event in the ”Determine configuration” activity happens, this will affect the flow activating the new configuration instead of the original one.

3 WS-BPEL Implementation of the Office Workflow

In this section we will present a BPMN derived BPEL implementation of the case study and the basic ideas behind it. Our intuition was that, although BPEL itself has not been designed to cope with dynamic

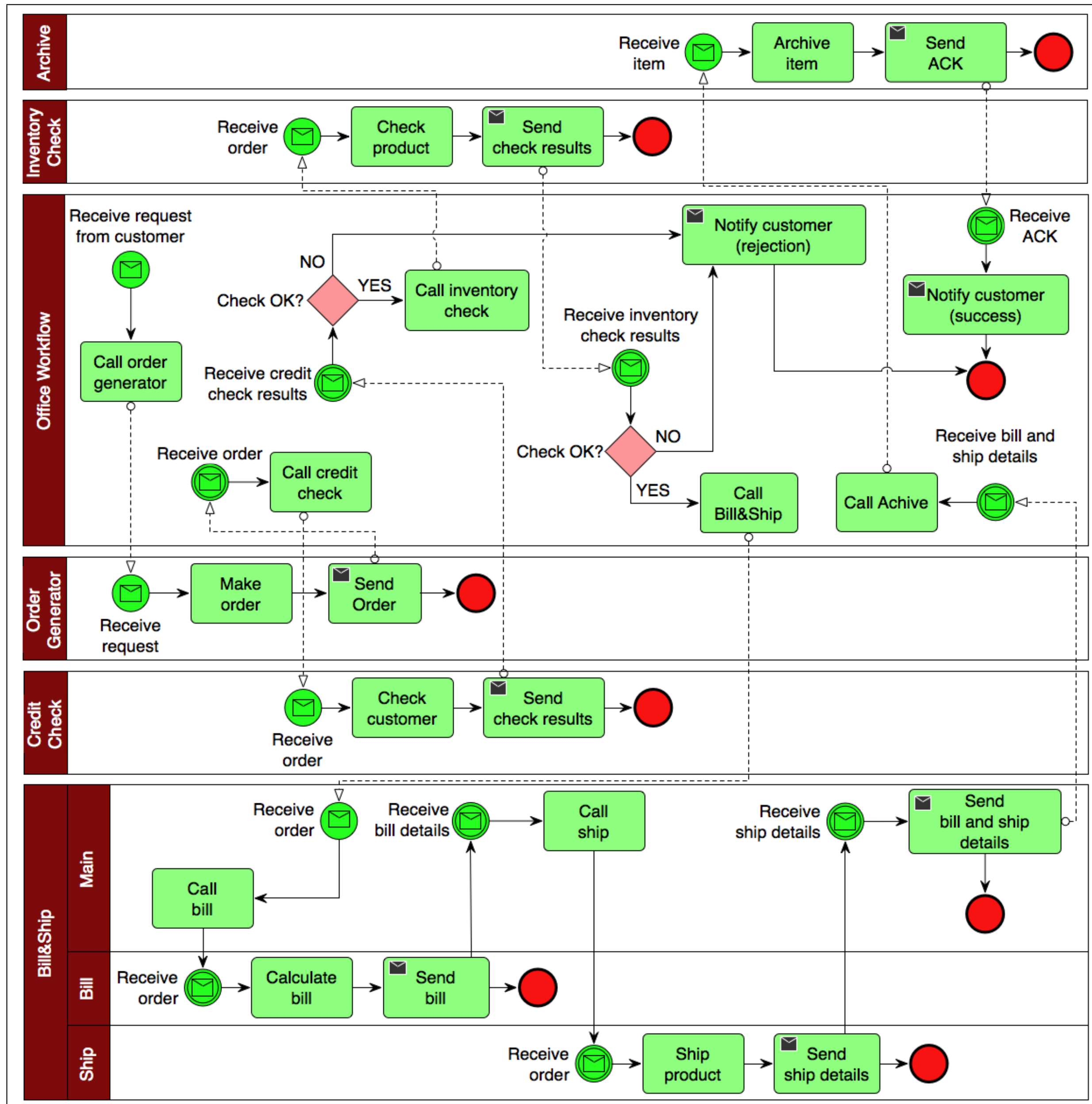

Figure 2: Office workflow - BPMN diagram of the new configuration

reconfiguration, it presents some features which can be used to this purpose. This idea has emerged because of similar considerations we have done about $Web\pi_\infty$ [14]. Since $Web\pi_\infty$ has been used to encode WS-BPEL [11], we have suspected that the basic mechanism of the BPEL recovery framework would work as the $Web\pi_\infty$ mechanism worked for this purposes. This was just an intuition but we worked to make it work and the results will be presented in this section. The basics principles, derived from the $Web\pi_\infty$ experience, on which our implementation is constructed are:

- The regions to be reconfigured have to be represented by BPEL scopes
- Each BPEL scope (i.e. region) will be associated with termination and event handlers

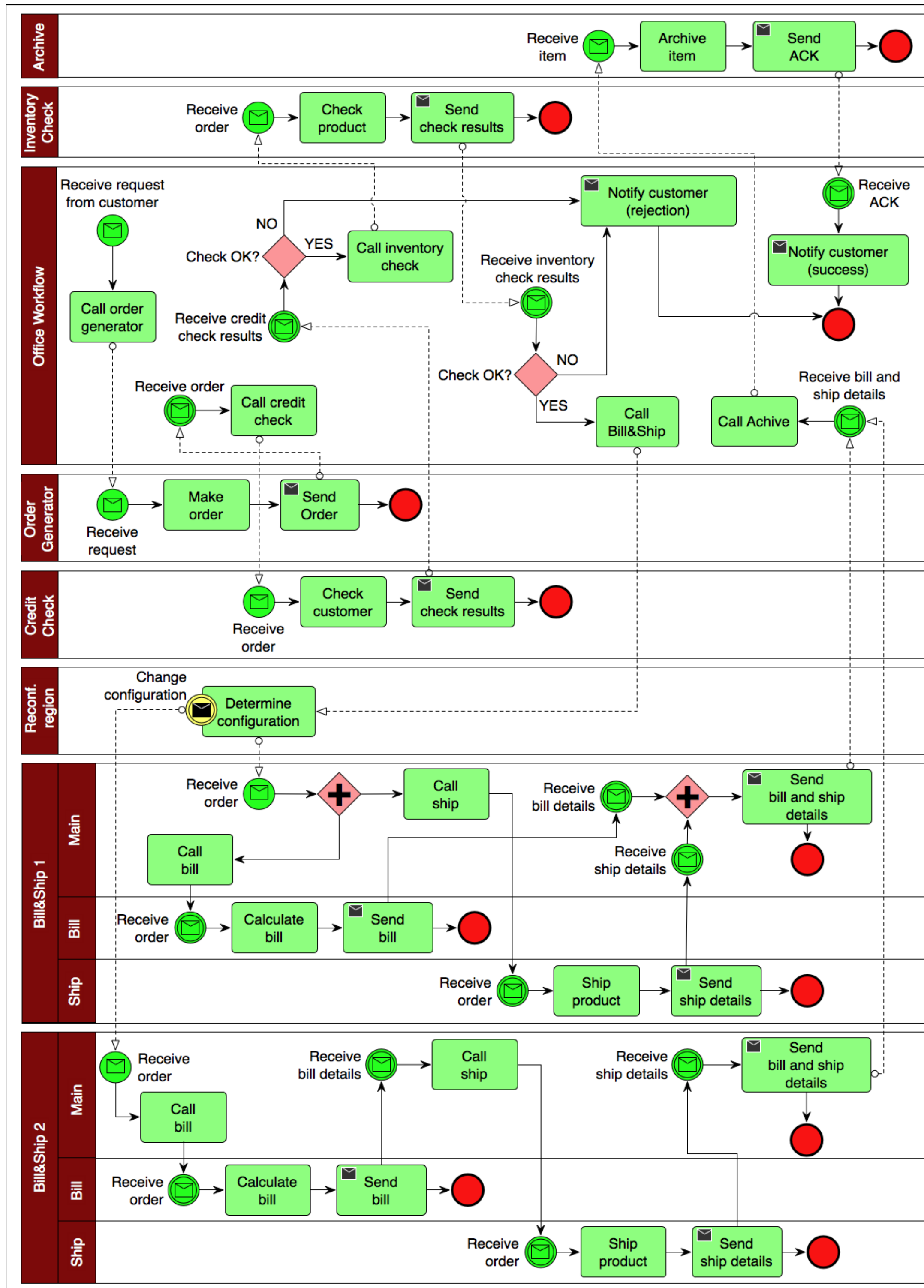

Figure 3: Office workflow - BPMN diagram of the reconfiguration

- Interference ("overlapping modes") will be implemented by the combined use of event and termination handlers

For a better understanding of how event handlers work please have a look at [5]. However, that paper does not investigate termination handlers (please see [9] for more details on this). Event handlers run in parallel with the scope body and are available more than once to be called (one single call does not suspend further availability). Thus, the new region has to be triggered by the event handler while the old region will be then terminated by the termination handler. As said, the body scopes run separately from the event handler so the old region can be terminated separately while the event handler brings the new region into play. This has not to be immediate. We think we can implement this way the synthetic cut-over change as defined in terms of Petri nets in [6].

While so far we have just presented the general principles on which the implementation is based, readers who are familiar with BPEL and who are interested in the details can find them in the following.

3.1 Mapping BPMN Models to BPEL

The first problem we have encountered when mapping the BPMN design into a BPEL implementation comes from the evident observation that BPMN and BPEL are representative of two different classes of languages. BPMN is indeed graph oriented while BPEL is mainly block-structured [16], at least in its commonly used XLANG [17] derived subset (however, BPEL has been also influenced by the graph oriented WSFL [10]). A consequence of this divergence is that the mapping from BPMN to BPEL is hard and it has a number of limitations since BPMN is able to express process patterns which cannot be expressed in BPEL. As a general comment we could say that the block structured nature of a BPEL process is too limited for modeling purposes.

However, we believe that BPEL cannot be ignored when it comes to workflow modelling because, although the business analysts more easily work with BPMN as modeling language and use its graphical notation to describe a business process (*Task, Activity, sequence flow, etc*), the system developers manage better to work with an executable language like BPEL to define the composite structure of a business process. In BPEL such a structure is defined in terms of a flow of structured activities *Sequence, Parallel, etc*) where each activity, in turn, can contain a nested list of other activities being those Web service invocations or other structured activities.

In this work the structure mismatch between BPMN and BPEL has been resolved following the approach presented in [16] consisting of a complete translation based on the identification of patterns of BPMN fragments which can be directly mapped onto BPEL code. The transformation approach will be described in the following of this section where we will give an overview of our mapping and how it is intended to work for the case study we are considering.

Basic Activities Translation BPMN basic activities (the ones based on messages, events and assignments) can be directly mapped to BPEL according to the following translation schema :

BPMN	BPEL
Send Task, Service Task, Message Event	Invoke
Receive Task, Message Event	Receive
Send Task, Message Event	Reply
Assignment	Assign
Termination end event	Exit

Structured Activities Translation We can classify different types of well-structured BPMN patterns resembling BPEL structured activities - *sequence, flow, switch, pick and while* - and translate them as follows:

BPMN	BPEL
Sequence Flow	Sequence
Parallel Fork-Join Gateway	Flow
Exclusive Data-based Gateway	Switch
Exclusive Event-based gateway, Message/Timer Event	Pick
Loops	While, RepeatUntil

3.2 WSDL Descriptions of the Involved Processes

As the reader can see in section 2.2, the BPMN design of the office workflow is made up of a set of independent components, which are shown as separate *pools* (using BPMN terminology) with separate sequence flows. It is interesting to note that this is not an abstract process since it includes the specific service calls involved, i.e. it includes the interactions points between the different participants. There are actually six participants involved (implemented as asynchronous Web services) as shown in the table below.

Participant	Interface	Operation	inMessageRef
Office Workflow	OrderReceiptPortType NotifyPortType	OrderReceipt Confirm Reject	OrderReceiptRequest ConfirmRequest RejectRequest
Order Generator	OrderGeneratorPortType OrderGeneratorReplyPortType	OrderGenerator OrderGeneratorReply	OrderGeneratorRequest OrderGeneratorReplyRequest
Credit Check	CreditCheckPortType CreditCheckReplyPortType	CreditCheck CreditCheckReply	Customer CheckrResult
Inventory Check	InventoryCheckPortType InventoryCheckReplyPortType	InventoryCheck InventoryCheckReply	Item CheckResult
Bill Ship	BillShipPortType BillShipReplyPortType	BillShip Bill Ship BillShipReply BillReply ShipReply	Order Order Order BillingAndShipping Billing Shipping
Archive	ArchivePortType ArchiveReplyPortType	Archive ArchiveReply	ArchiveItem ACK

WSDL [4] requires the specification of operations and port types the service is offering, the accepted messages and their types. Consequently, a precise WSDL descriptions of the involved services can be derived from the table above. We now present in detail the WSDL description for each of the six processes.

- *Office Workflow*: *OrderReceiptPortType* allows the order message to be received by means of the *OrderReceipt* operation. To return the result, the Web service specifies a second port type: *NotifyPortType*. This port type specifies *Confirm* and *Reject* operations to return notification messages back to the customer.

```
<wsdl:portType name="OrderReceiptPortType">
  <wsdl:operation name="OrderReceipt">
    <wsdl:input message="OrderRequest" name="Input"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:portType name="NotifyPortType">
  <wsdl:operation name="Confirm">
    <wsdl:input message="ConfirmNotify" name="Input"/>
  </wsdl:operation>
  <wsdl:operation name="Reject">
```

```
      <wsdl:input message="RejectNotify" name="input"/>
    </wsdl:operation>
  </wsdl:portType>
```

- *Order Generator:* *OrderGeneratorPortType* allows generating the order code by means of the *OrderGenerator* operation. The result is returned through the *OrderGeneratorReply* operation specified by *OrderGeneratorReplyPortType*.

```
<wsdl:portType name="OrderGeneratorPortType">
  <wsdl:operation name="OrderGenerator">
      <wsdl:input message="OrderRequest" name="Input"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:portType name="OrderGeneratorReplyPortType">
  <wsdl:operation name="OrderGeneratorReply">
    <wsdl:input message="GeneratorReply" name="Input"/>
  </wsdl:operation>
</wsdl:portType>
```

- *Credit Check:* *CreditCheckPortType* allows checking the identity of the customer with the operation *CreditCheck*. *CreditCheckReplyPortType* is instead used to return the check result through the operation *CreditCheckReply*.

```
<wsdl:portType name="CreditCheckPortType">
  <wsdl:operation name="CreditCheck">
    <wsdl:input message="Customer" name="Input"/>
    </wsdl:operation>
  </wsdl:portType>
<wsdl:portType name="CreditCheckReplyPortType">
  <wsdl:operation name="CreditCheckReply">
    <wsdl:input message="CheckResult" name="Input"/>
  </wsdl:operation>
</wsdl:portType>
```

- *Inventory Check:* Similarly to the Credit Check service *InventoryCheckPortType* is used to check the identity of the product by means of the operation *InventoryCheck*. *InventoryCheckReplyPortType* is instead used to return the result of the check through the operation *InventoryCheckReply*.

```
<wsdl:portType name="InventoryCheckPortType">
  <wsdl:operation name="InventoryCheck">
    <wsdl:input message="Item" name="Input"/>
    </wsdl:operation>
  </wsdl:portType>
<wsdl:portType name="InventoryCheckReplyPortType">
  <wsdl:operation name="InventoryCheckReply">
    <wsdl:input message="CheckResult" name="Input"/>
  </wsdl:operation>
</wsdl:portType>
```

- *B*ill and Ship: *BillShipPortType* is used to trigger the bill and ship activity through the operation *BillShip*. The bill activity is performed using the operation *Bill* and the ship activity is performed using the operation *Ship*. To return the result, the service specifies a second port type: *BillShipReplyPortType*. The bill details are returned through the operation *BillReply* while the ship details are returned through the operation *ShipReply*. The overall bill and ship details are returned through the operation *BillShipReply*.

```
   <wsdl:portType name="BillShipPortType">
     <wsdl:operation name="BillShip">
       <wsdl:input message="Order" name="Input"/>
     </wsdl:operation>
     <wsdl:operation name="Bill">
       <wsdl:input message="Order" name="input"/>
     </wsdl:operation>
     <wsdl:operation name="Ship">
       <wsdl:input message="Order" name="input"/>
     </wsdl:operation>
</wsdl:portType>

    <wsdl:portType name="BillShipReplyPortType">
      <wsdl:operation name="BillShipReply">
        <wsdl:input message="BillingAndShipping" name="Input"/>
      </wsdl:operation>
      <wsdl:operation name="BillReply">
        <wsdl:input message="Billing" name="input"/>
      </wsdl:operation>
      <wsdl:operation name="ShipReply">
        <wsdl:input message="Shipping" name="input"/>
      </wsdl:operation>
</wsdl:portType>
```

- *Archive: ArchivePortType* allows archiving the ordered product for further reference using the operation *Archive*. *ArchiveReplyPortType* is specified to return the result through the operation *ArchiveReply*.

```
    <wsdl:portType name="ArchivePortType">
      <wsdl:operation name="Archive">
         <wsdl:input message="ArchiveItem" name="Input"/>
</wsdl:operation>
    </wsdl:portType>

    <wsdl:portType name="ArchiveReplyPortType">
      <wsdl:operation name="ArchiveReply">
        <wsdl:input message="ACK" name="Input"/>
        </wsdl:operation>
    </wsdl:portType>
```

To make all the services working together BPEL requires the definition of a *partnerLink* section as follows:

```
<partnerLinks>
 <partnerLink myRole="OrderReceiptServiceProvider"
              name="OfficeWorkflow"
```

```
                partnerLinkType="OfficeWorkflow:OfficeworkflowPLT"
                partnerRole="NotifyServiceRequester"/>
  <partnerLink myRole="OrderGeneratorReplyServiceRequester"
                name="OrderGenerator"
                partnerLinkType="OrderGenerator:OrderGeneratorPLT"
                partnerRole="OrderGeneratorServiceProvider"/>
  <partnerLink myRole="CreditCheckReplyServiceRequester"
                name="CreditCheck"
                partnerLinkType="CreditCheck:CreditCheckPLT"
                partnerRole="CreditCheckServiceProvider"/>
  <partnerLink myRole="InventoryCheckReplyServiceRequester"
                name="InventoryCheck"
                partnerLinkType="InventoryCheck:InventoryCheckPLT"
                partnerRole="InventoryCheckServiceProvider"/>
  <partnerLink myRole="BillShipReplyServiceRequester"
                name="BillandShip1"
                partnerLinkType="BillShip1:BillShipPLT"
                partnerRole="BillShipSeriveProvider"/>
  <partnerLink myRole="ArchiveReplyServiceRequester"
                name="Archive"
                partnerLinkType="Archive:ArchivePLT"
                partnerRole="ArchiveServiceProvider"/>
</partnerLinks>
```

In this way, the interfaces of the other services which interacts with *Office Workflow* are linked to the main BPEL process of the workflow itself.

3.3 Office Workflow BPEL Main Body

The main process describing the workflow starts with the reception of an order request coming from a customer, then it asynchronously invokes *Order Generator* and, after having received a reply from it, *Credit Check* is asynchronously invoked. It continues asynchronously invoking different services one by one, according to the specification. Two structure patterns can be identified: the *sequence pattern* involving the whole process and the *If-else pattern* for handling both the credit check reply and the inventory check reply. The BPMN *Message Start Event* initiates the process receiving a message. This is mapped into BPEL using a *receive* activity.

```
<sequence>
    <receive createInstance="yes"
    name="Receiverequestfromcustomer"
    operation="OrderReceipt"
    partnerLink="OfficeWorkflow"
    portType="OfficeWorkflow:OrderReceiptPortType"
    variable="CustomerRequest"/>
```

Next, we have to prepare the request message for the *Order Generator* service. We have to send a message consisting of customer and item parts built through the corresponding BPEL *assignment* activity.

```
<assign name="Callordergenerator">
           <copy>
               <from>$CustomerRequest.Customer/Name</from>
```

```
            <to>$OrderRequest.part1/CustomerName</to>
        </copy>
        <copy>
            <from>$CustomerRequest.Customer/ID</from>
            <to>$OrderRequest.part1/CustomerID</to>
        </copy>
        <copy>
            <from>$CustomerRequest.Item/Name</from>
            <to>$OrderRequest.part2/ItemName</to>
        </copy>
        <copy>
            <from>$CustomerRequest.Item/Quantity</from>
            <to>$OrderRequest.part2/ItemQuantity</to>
        </copy>
    </assign>
```

Now, the *Order Generator* service will be invoked. Because it is an asynchronous service, the callback will be received using the BPEL *receive* activity. We have so to invoke the *OrderGeneratorReply* operation on the *OrderGeneratorReplyPortType*. The callback message contains a Order number (*OrderID*) which is used to initiate the correlation set.

```
<invoke partnerLink="OrderGenerator"
   operation="OrderGenerator"
   portType="OrderGenerator:OrderGeneratorPortType"
   inputVariable="OrderRequest">
 </invoke>
 <receive name="Receiveorder"
   partnerLink="OrderGenerator" operation="OrderGeneratorReply"
   portType="OrderGenerator:OrderGeneratorReplyPortType"
   variable="Order">
        <correlations>
            <correlation set="OrderId" initiate="yes"/>
        </correlations>
 </receive>
```

After having received the response message from the *Order Generator* service, the process will invoke the *Credit Check* service. This involves checking customer identity. Mapping the call of the *Credit Check* service is similar to mapping the *Order Generator* service. Again, we start with the preparation of the input message for the Credit Check service and then we invoke the service itself.

```
    <assign name="Callcreditcheck">
        <copy>
            <from>$Order.part/CustomerName</from>
            <to>$Customer.part/CustomerName</to>
        </copy>
        <copy>
            <from>$Order.part/OrderID</from>
            <to>$Customer.part/CustomerID</to>
        </copy>
    </assign>
    <invoke partnerLink="CreditCheck" operation="CreditCheck"
    portType="CreditCheck:CreditCheckPortType" inputVariable="Customer">
```

```
        <correlations>
            <correlation set="OrderId" pattern="out"/>
        </correlations>
    </invoke>

    <receive name="Receivecreditcheckresult" createInstance="no"
    partnerLink="CreditCheck" operation="CreditCheckReply"
    portType="CreditCheck:CreditCheckReplyPortType"
    variable="CreditCheckResult">
        <correlations>
            <correlation set="OrderId" initiate="no"/>
        </correlations>
    </receive>
```

The BPMN exclusive gateway following the ”Receive credit check result” message event is mapped into a BPEL *If-else* structured activity:

```
<if name="If1">
        <condition>$CreditCheckResult.Part</condition>
        <sequence name="Sequence1">
            ...

        </sequence>
        <else>
            ...
        </else>
</if>
```

The *Inventory Check* works exactly in the same way as the *Credit Check*. The BPMN process then moves to ”Receive item check result”, it goes through the ”Yes” condition and the *Bill&Ship* operation is invoked on the *BillShipPortType* of the *Bill And Ship* service. At this point, two operations *bill* and *ship* are invoked in parallel. Both *bill* and *ship* return their details by means of the operation *BillShipReply* and then the *Archive* service is invoked. After having received the return message *ACK* from *ArchiveReplyPortType* an *invoke* activity on *NotifyPortType* is performed to send a confirmation message back to the customer.

```
 <invoke name="Notifycustomerconfirm"
  partnerLink="OfficeWorkflow"
  operation="Confirm"
  portType="OfficeWorkflow:NotifyPortType"
  inputVariable="ConfirmNotify"/>
```

Change Configuration To implement in BPEL the BPMN model depicted in figure 2 we have just to replace *Bill Ship*. So we need to define a new interface for it and then map it onto a new partner link. We have to do the same as before and finally we get *BillandShip2*. This is also an asynchronous process, containing both the invocation of bill and ship operations and the invocation of a callback operation.

```
<partnerLink
myRole="BillShipReplyServiceRequester"
```

```
name="BillandShip2"
partnerLinkType="BillShip2:BillShipPLT" partnerRole="BillShipSeriveProvider"/>
```

The process is simpler than the former ”Bill Ship”, only one structure pattern is now involved: *Sequence*. After the *BillShip* operation is invoked on *BillShipPortType* of the *Bill And Ship* service, the *Bill* operation is invoked. Then, the return message from *BillReplyPortType* is received and the *Ship* operation is invoked. Ship details are returned by the operation *ShipReply* and the return message sent from *BillShipReplyPortType* is received. Finally, the *Archive* service is invoked.

```
<sequence name="Sequence3">
  <assign name="Callbill">
    <copy>
        ...
    </copy>
  </assign>
  <invoke inputVariable="BillShipOrder"
   operation="Bill"
   partnerLink="BillandShip2"
   portType="BillShip2:BillShipPortType"/>
  <receive createInstance="yes"
   partnerLink="BillandShip2"
   operation="BillReply"
   portType="BillShip2:BillShipReplyPortType" variable="Billing">
    <correlations>
      <correlation set="OrderId" initiate="yes"/>
    </correlations>
  </receive>
  <assign name="Receivebilldetails">
    <copy>
     ...
    </copy>
  </assign>
  <invoke inputVariable="BillShipOrder"
   operation="Ship"
   partnerLink="BillandShip2"
   portType="BillShip2:BillShipPortType"/>
  <receive createInstance="no"
   partnerLink="BillandShip2" operation="ShipReply"
   portType="BillShip2:BillShipReplyPortType" variable="Shipping">
    <correlations>
      <correlation set="OrderId" initiate="no"/>
    </correlations>
  </receive>
  <assign name="Receiveshipdetails">
    <copy>
      ...
    </copy>
  </assign>
</sequence>
```

Transition between Configurations The most interesting part of the BPMN design is the one depicted in figure 3. To map this to BPEL we have to define a new partner link for a new participant *Reconf.region* which will be then used to invoke the new configuration. We define a partner link with the role *provider* to change configuration:

```
<partnerLink myRole="provider"
name="Reconf.region"
partnerLinkType="Reconf.region:Reconf.RegionPLT" />
```

Within *Reconf.region* there is a BPMN *Activity* ”Determine configuration” with a *Non-Interrupting Intermediate Message Event*, which can be mapped to a BPEL *scope* with an *event handler* [2].

```
<scope name="BillAndShip1">
   <eventHandlers>
     <onEvent partnerLink="Reconf.region"
              operation="change"
              portType="Reconf.region:ProviderPortType"
              variable="Rec"
              messageType="Reconf.region:Rec">
               <scope name="BillAndShip2">
               ...
               </scope>
     </onEvent>
   <eventHandlers>
</scope>
```

Let us describe here in details how this works. If the process receive the *Rec* change message once the *BillAndShip1* scope has been entered, it will execute the new process defined within the scope *BillAndShip2*. This other process is exactly the new configuration. Otherwise, the order will be processed accordingly with the original procedure.

In order to distinguish between these two situations — receiving the event before billing and shipping activities have started or after — we use scopes to define different event handlers: *Scope1* represents the procedure running before billing and shipping, *BillShip1* represents the concurrent billing and shipping and *BillShip2* represents the sequential billing and shipping. When a management decision is made, the event handler for *Scope1* will be invoked and it will terminate *Scope1*, which contains the procedure for order receipt, order evaluation and *BillShip1* activities. We use termination handler to replace *Scope1* with a new scope representing the new procedure for order receipt, order evaluation and, this time, *BillShip2* activities. In this way, after its termination, the process will restart calling the new procedure.

We declare individual variables for *BillShip1* and *BillShip2*. These are the request messages used to invoke the billing and shipping services and they are only visible within their own scope. This means that, if the request message for billing and shipping has already been created, this activity can be invoked without any interrupt. Technically, the event handler is used to implement the management decision for change. When the event is received, *BillShip2* will be enabled. However, if the event is received after *Scope1* has been executed, *BillShip2* will not be run because no request message has been initialized and *Scope1* only calls*BillShip1*. If the event is received while *Scope1* is running, *Scope1* will be terminated and *Scope2* will start redoing order receipt, order evaluation. After that, *BillShip2* will start because the receiving event, and also the request message, have been initialized exactly for it.

In the real word, after the management decision is made to switch to *BillShip2*, *BillShip1* would be not available anymore. It is like ending to offer the *BillShip1* service. However, in BPEL, we cannot

model exactly this situation. All the services remain available. If we want to ensure all the instances of the workflow created after the change run *BillShip2* instead of *BillShip1*, the process needs to continue receiving the "change reconfiguration" event.

3.4 Tool-based Mapping BPMN Models to BPEL

The BPMN to BPEL mapping we have presented so far has been obtained by following the approach given in [16]. This allowed us to have some flexibility but the process had to be entirely manually executed. Another option, although more restrictive, is to use some automatic tool for the translation. In this section we will indeed discuss this option using the *Intalio BPMS Designer* version 6.0.

Intalio BPMS Designer is a set of Eclipse plugins allowing process designers to model processes with BPMN and to use several graphical tools to manage the data. It includes most of the BPMN elements which are relevant to executable business process models. External activities and message flows are mapped into specific interface operations and message definitions using WSDL. The message structures are indicated by XML Schema elements. Service calls are modelled by introducing *Pools* containing the operations of the WSDL. The process interacts with this external participants through message flows. After the process has been modelled and concrete services, messages and data have been defined, Intalio Designer will automatically generate a BPEL description.

To model the office workflow with the Intalio Designer the first thing we have to do is creating a 'Business Process Project' containing Business Process diagrams, XML Schemas, WSDL files, etc. Once the project has been created, we can then create a BPMN diagram with the embedded BPMN modeler. After the BPMN modelling for the office workflow will be completed, we can start implementing the process *Office Workflow* by integrating all the operations from the existing Web services, creating the interface to define how it will be exposed to the external users and defining the graphical mappings to invoke the services.

Integrating web services The tool integrates a full WSDL visual browser which allows to edit and introspect WSDL documents. To implement the case study we have to create WSDL documents for *Office Workflow Service, Order Generator Service, Credit Check Service, Inventory Check Service, Bill&Ship Service and Archive Service* respectively. Then we have to create pools representing all the external Web services and set them to 'Non-executable' as these pools represent the sequence of service operations that will be invoked from the main business process *Office Workflow*.

It is very important to make the distinction between operations invoked by a process and operations that will invoke a process. An operation in a non-executed pool, represented as a BPMN task, it either provides the operation or invokes the operation. The operations like *OrderGenerator, CreditCheck, InventoryCheck, etc* are operations that the *Office Workflow* process will invoke; whereas *OrderGeneratorReply, CreditCheckReply, InventoryCheckReply, etc* are operations that will invoke the process.

Finally, we have to connect the process tasks to the Web service operations. The order is defined by creating the links. For the operations of *Order Generator* we want the message received from the customer to go from the executable task ("Invoke order generator") to the corresponding *Order Generator* operation and the response message to go from the *Order Generator* operation to the message intermediate event ("Receive order"). In the same way, we have to integrate *Credit Check, Inventory Check, Bill&Ship, Archive*. All the data involved in the *Office Workflow* process are created automatically when integrating the WSDL files.

Generate BPEL code Once the *Office Workflow* process is ready to be executed we can easily deploy it. There are several artifact being generated at this point: the BPEL code corresponding to the

Office Workflow process, the WSDL files used by the process to represent its interactions with the other participants and the different WSDLs used to represent external services.

Change Configuration As before, we have to deal with the *Office Workflow* reconfiguration, i.e. the process will invoke *Bill* and *Ship* in sequence instead parallel. The remaining parts like partner links, external services, WSDLs are not altered by this but the BPEL is. We need indeed a new participant *Reconf.region* used to send a reconfiguration message and invoke the new procedure. We also have to create a WSDL for it.

We need two use *sub-process* to include the two configurations and to add an *Exclusive Event-based Gateway* to make the choice. If the process receives the change message then the *configuration2* sub-process will execute (the new configuration) otherwise the process will automatically executes the old configuration sub-process *configuraiton1*. The generated BPEL code, partner links and, in general, all the material related to this project could not be reported in this document due to its size but it is available upon request, so the interested reader can contact the authors.

As we can see from the generated BPEL code, the interaction between the *Reconf.region* Web service and BPEL process is mapped into a *pick* activity.

```
<pick ...>
       <onMessage partnerLink="..."
       operation="Change"
       portType="ReconfigService:ReconfigService"
       variable="...">
           <sequence>
             <scope ...>
               <variables>
               </variables>
                 <sequence>
                  <assign>
                 ...
                  </assign>
                   <invoke name="..."/>
                   <receive name="" .../>
                   <invoke name="..."/>
                   <receive name="" .../>
                  </sequence>
                 </scope>
        </onMessage>
                <sequence>
                 <scope name="...">
                   <variables>
                       ...
                   </variables>
                    <sequence>
                       ...

                    </sequence>
                  </scope>
                 </sequence>
   </pick>
```

Thus, if the process receives the change message before invoking the *BillAndShip* operation on *BillAndShipPortType*, the order will be processed according to the new procedure, otherwise it will be

processed according to the old one.

4 Conclusions

In this paper we investigated the issue of workflow reconfiguration in BPMN and WS-BPEL. We then proposed two implementations, one manually generated and one tool-based and we identified the weaknesses of the tool-based one. With this work we have shown how WS-BPEL, which is not originally intended to model dynamic reconfiguration, can be exploited for this purpose by the use of its very powerful recovery framework and, in particular, event and termination handlers. The idea on which the paper is based derives from some intuitions emerged during our previous work on process algebra. We are also working on a more complete comparisons of formalisms than the one presented in [13] which includes the modeling of this workflow case study in several different formalisms (including π-calculus, $\mathtt{web}\pi_\infty$ and VDM) with the consequent verification of the desired requirements. Preliminary results can be found at [1].

Acknowledgments

This work has been made possible by a number of very useful conversations with Anirban Bhattacharyya, Cliff Jones, John Fitzgerald, Jeremy Bryans, Alexander Romanovsky, Gudmund Grov, Mario Bravetti, Massimo Strano, Carlos molina-jimenez, Michele Mazzucco, Anjan Pakhira, Peter Andras, Paolo Missier and many others. We also want to thank all the members of the Reconfiguration Interest Group at Newcastle and, in particular, Kamarul Abdul Basit, Carl Gamble and Richard Payne, the Dependability Group (at Newcastle University) and the EU FP7 DEPLOY Project (Industrial deployment of system engineering methods providing high dependability and productivity).

References

[1] F. Abouzaid, A. Bhattacharyya, N. Dragoni, J. Fitzgerald, M. Mazzara, and M. Zhou. A case study of workflow reconfiguration: Design, modelling, analysis and implementation. Technical Report CS-TR No. 1265, School of Computing Science, University of Newcastle, November 2011.

[2] BPMN. Bpmn - business process modeling notation. `http://www.bpmn.org/`.

[3] A. Carter. Using dynamically reconfigurable hardware in real-time communications systems: Literature survey. Technical report, Computer Laboratory, University of Cambridge, November 2001.

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl 1.1), W3C, 2001.

[5] N. Dragoni and M. Mazzara. A formal semantics for the ws-bpel recovery framework: the π-calculus way. In *Proc. of the 6th international conference on Web services and formal methods (WS-FM'09), LNCS, Bologna, Italy*, volume 6194, pages 92–109. Springer-Verlag, September 2009.

[6] C. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *Proc. of International Conference on Organizational Computing Systems (COCS'95), Milpitas, California, United States*, pages 10–21. ACM Press, August 1995.

[7] P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP Journal of Embedded Systems*, 2006(1), January 2006.

[8] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proc. of the 1991 European Conference on Object-Oriented Programming (ECOOP'91), LNCS, Geneva, Switzerland*, volume 512, pages 133–147. Springer-Verlag, July 1991.

[9] D. Jordan and J. E. editors. Web services business process execution language version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf`.

[10] F. Leymann. Web services flow language (wsfl 1.0). '`http://www-01.ibm.com/software/solutions/soa/`.

[11] R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2007.

[12] M. Mazzara, F. Abouzaid, N. Dragoni, and A. Bhattacharyya. Toward design, modelling and analysis of dynamic workflow reconfigurations - a process algebra perspective. In *Proc. of 8th International Workshop on Web Services and Formal Methods, LNCS, Clermont-Ferrand, France*, volume 7176, pages 64–78. Springer-Verlag, September 2011.

[13] M. Mazzara and A. Bhattacharyya. On modelling and analysis of dynamic reconfiguration of dependable real-time systems. In *Proc. of the 2010 Third International Conference on Dependability (DEPEND'10), Venice, Italy*, pages 173–181. IEEE, July 2010.

[14] M. Mazzara and I. Lanese. Towards a unifying theory for web services composition. In *Proc. of the 3rd international conference on Web Services and Formal Methods (WS-FM'06), LNCS, Vienna, Austria*, volume 4184, pages 257–272. Springer-Verlag, September 2006.

[15] OMG. Omg - object management group. '`http://www.omg.org/`.

[16] C. Ouyang, M. Dumas, A. ter Hofstede, and W. van der Aalst. From bpmn process models to bpel web services. In *Proc. of the 2006 IEEE International Conference on Web Services (ICWS'06), Chicago, USA*, pages 285–292. IEEE, September 2006.

[17] S. Thatte. Xlang: Web services for business process design. Microsoft Corporation, 2001.

Manuel Mazzara achieved his Masters in 2002 and his Ph.D in 2006 at the University of Bologna. His thesis was based on Formal Methods for Web Services Composition. During 2000 he was a Technical Assistant at Computer Science Laboratories (Bologna, Italy). In 2003 he worked as Software Engineer at Microsoft (Redmond, USA). In 2004 and 2005 he worked as a free lance consultant and teacher in Italy. In 2006 he was an assistant professor at the University of Bolzano (Italy) and in 2007 a researcher and project manager at the Technical University of Vienna (Austria). Currently he is a Research Associate at the Newcastle University (UK) working on the DEPLOY project.

Nicola Dragoni obtained a M.Sc. Degree and a Ph.D. in computer science, respectively in 2002 and 2006, both at University of Bologna, Italy. He visited the Knowledge Media Institute at the Open University (UK) in 2004 and the MIT Center for Collective Intelligence (USA) in 2006. In 2007 and 2008 he was post-doctoral research fellow at University of Trento, working on the security for mobile systems. Between 2005 and 2008 he also worked as freelance IT consultant. In 2009 he joined Technical University of Denmark (DTU) as assistant professor in security and distributed systems. He was promoted to associate professor in 2011.

Mu Zhou obtained a M.Sc. Degree in Computer Science and Engineering at Technical University of Denmark. She is currently junior developer at A-Solutions, Copenhagen, Denmark.