Dependable Workflow Reconfiguration in WS-BPEL

Manuel Mazzara[1], Nicola Dragoni[2], and Mu Zhou[2]

[1] Newcastle University, Newcastle upon Tyne, UK
Manuel.Mazzara@ncl.ac.uk
[2] Technical University of Denmark (DTU), Copenhagen
ndra@imm.dtu.dk, mu.zhou31@gmail.com

Abstract. This paper describes a workflow reconfiguration and how to implement it in WS-BPEL, a language that would not naturally support reconfiguration. We state the requirements on a system implementing the workflow and its reconfiguration, and we describe the system's design in BPMN. The WS-BPEL recovery framework is then exploited to implement the reconfiguration.

1 Introduction

Modern dependable systems are required to be flexible, 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 $Web\pi_\infty$ [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.

In Section 2, we state the requirements on a system implementing the workflow and its reconfiguration. In Section 3, we describe the system's design in BPMN. Finally, the WS-BPEL implementation is discussed in Section 4, 5,6 and 7. Further details on the case study can be also found in [12]. 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 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. 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 organization 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 synchronisation 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.

3 BPMN Design of the Office Workflow

In this section, we present a design of the office workflow by using notation and concepts of BPMN. The case study is depicted in Figure 1 as composed of several *Participants* working together to process orders from customers and it employs several *Pools* to represent different functional entities (*Order Generator, Credit Check, Inventory Check, Bill&Ship, Archive*). It is worth noting the requirements state that only the *Credit Check* service is supposed to be external. Thus, every other service might be included in a single *pool*, representing an activity within the organization. However, we have decided to design a more generic situation where the different services are offered by external entities by adopting a *pool* for each service. We are now describing the diagram in more detail.

The *Office Workflow* pool is the coordinating entity. It starts by receiving a request message from a customer. Then an order is created by calling the *Order Generator* entity. The process within the *Order Generator pool* starts by receiving the order request message from *Office Workflow*; it then executes the "Make Order" *Task* and finally it sends an order back to *Office Workflow* again. This order is then sent to both the credit check handler (*Credit Check*) to check the customer credit's availability and the inventory check handler (*Inventory Check*) to verify the availability of the product. The latter is performed only in case the credit check is successful. After receiving the evaluation result, an *Exclusive Data-Based Gateway* is used. 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. Otherwise, the order is sent to *Inventory Check*. The same happens with the result from Inventory Check: in case of a negative reply the customer is notified and the order is rejected. In case of a positive reply, the order is processed.

The *Bill&Ship* represents the entity responsible for both the billing and shipping activities by using two *lanes* (*Bill* and *Ship*) in a *pool*. For the sake of simplicity and readability, we assume that neither billing nor shipping provide a negative result. When *Bill&Ship* receives the order, the two activities are called concurrently by a *Parallel Gateway*. The same gateway is used to merge the result from *Bill* and *Ship*. A message containing the bill and ship details is sent to *Office Workflow* to call *Archive* to store the order. *Office Workflow* terminates by receiving a response from *Archive* and sending a confirmation notification to the customer.

3.1 Change in Configuration

Now the company decides to reconfigure the order of billing and shipping: the billing activity will now take place before the shipping. Both the old and the new

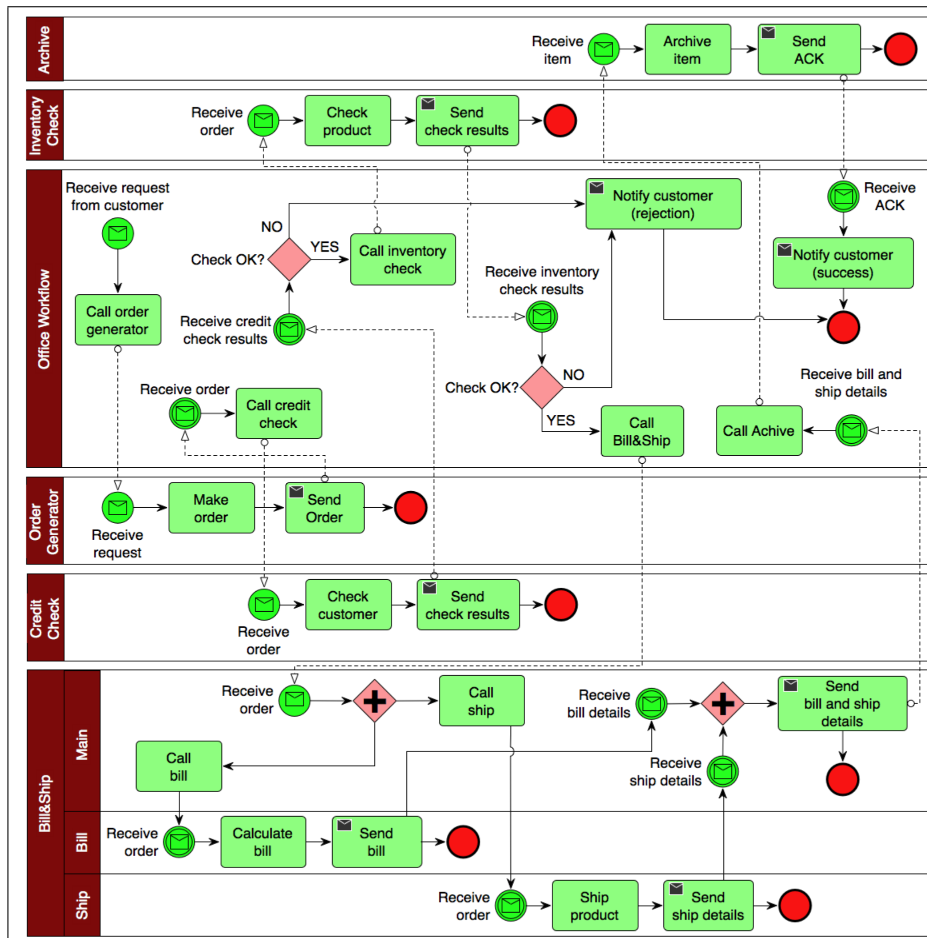

Fig. 1. The Case Study Workflow - Old Configuration BPMN Model

configuration processes will have to be simultaneously available while processing the already active orders. Thus, our concern is performing the structural change safely without flushing the system. Looking at the design we have presented so far, it should be quite easy to realize that this reconfiguration requires a change in the main lane of *Bill&Ship* only, where the actual billing and shipping activities are called. The rest of the workflow has to remain the same. The new configuration diagram is shown in Figure 2. When compared to the old configuration diagram of Figure 1, the two *Parallel Gateways* in the main lane of *Bill&Ship* have been now removed and the two activities are synchronously called.

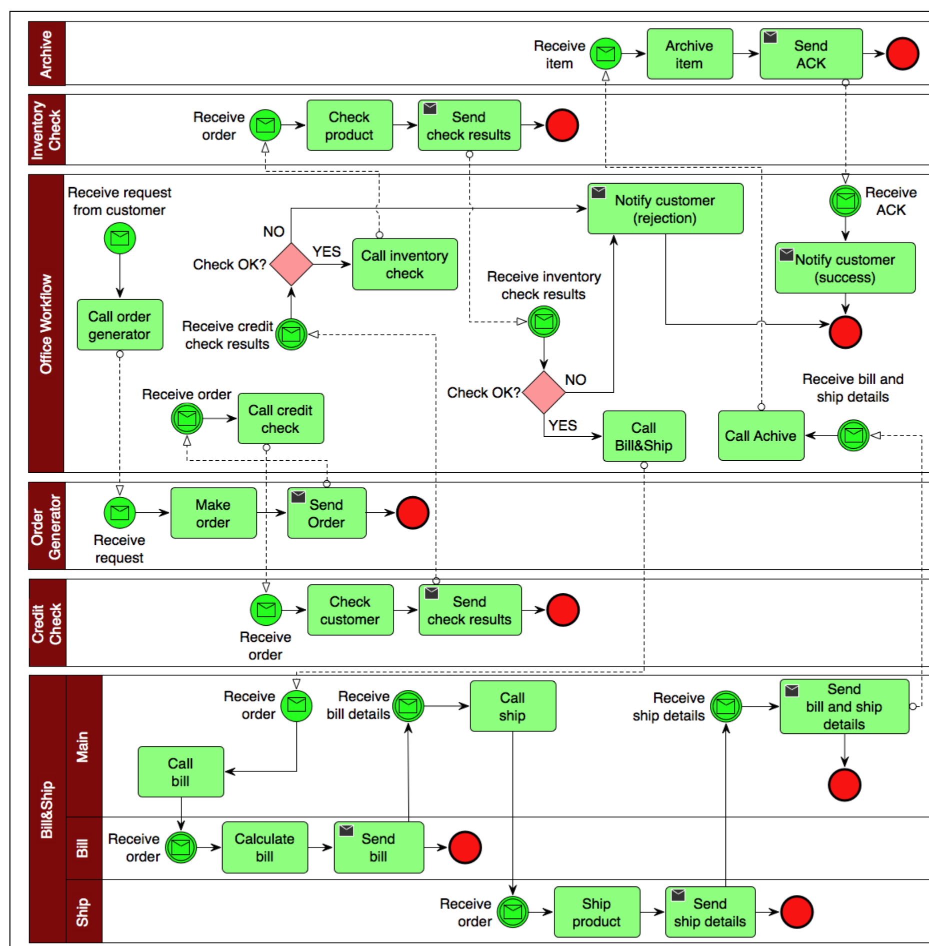

Fig. 2. The Case Study Workflow - New Configuration BPMN Model

How the transition from the old configuration (Figure 1) to the new configuration (Figure 2) can be performed? The BPMN design for the entire workflow, during its reconfiguration, is shown in Figure 3. In order to keep both the old configuration process and the new one simultaneously available, we define a default flow that is identical to the old configuration. This default flow can be altered through an interrupting *Message Event* contained in the "Determine configuration" activity included in a separate *Reconfig.region pool*. This activity determines which configuration should be used when *Bill&Ship* is called. In this way, we highlight that an authority has to be in charge of triggering the reconfiguration. Thus, when the interrupt event happens, it will affect the flow and it will activate the new configuration.

4 WS-BPEL Implementation of the Office Workflow

In [11] the mapping from WS-BPEL to π-calculus has been investigated. The idea was to design the system at the WS-BPEL level and then verifying it at the π-calculus level. In [2], the opposite direction has been instead explored. That work supports the idea that building the π-calculus model, check it and only then map it into WS-BPEL seems to be a more effective way to tackle the problem of verification for WS-BPEL systems. In this paper, we have instead decided to follow an approach based on the BPMN design because we think it is a powerful design tool with widespread use these days and it is easy to be understood by designers and by non formalists (while the other two approaches actually require a specific technical knowledge). In this section, we will present a BPMN derived WS-BPEL implementation of the case study and the basic ideas behind it.

Our intuition was that, although WS-BPEL itself has not been designed to cope with dynamic reconfiguration, it presents some features which could be used for 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 mechanisms of the WS-BPEL recovery framework would have offered a support to dynamic reconfiguration, in the same way as $Web\pi_\infty$ did. This was just an intuition, but we have then focused on the details to make it work and the results are 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

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, when using event handlers for reconfiguration, the

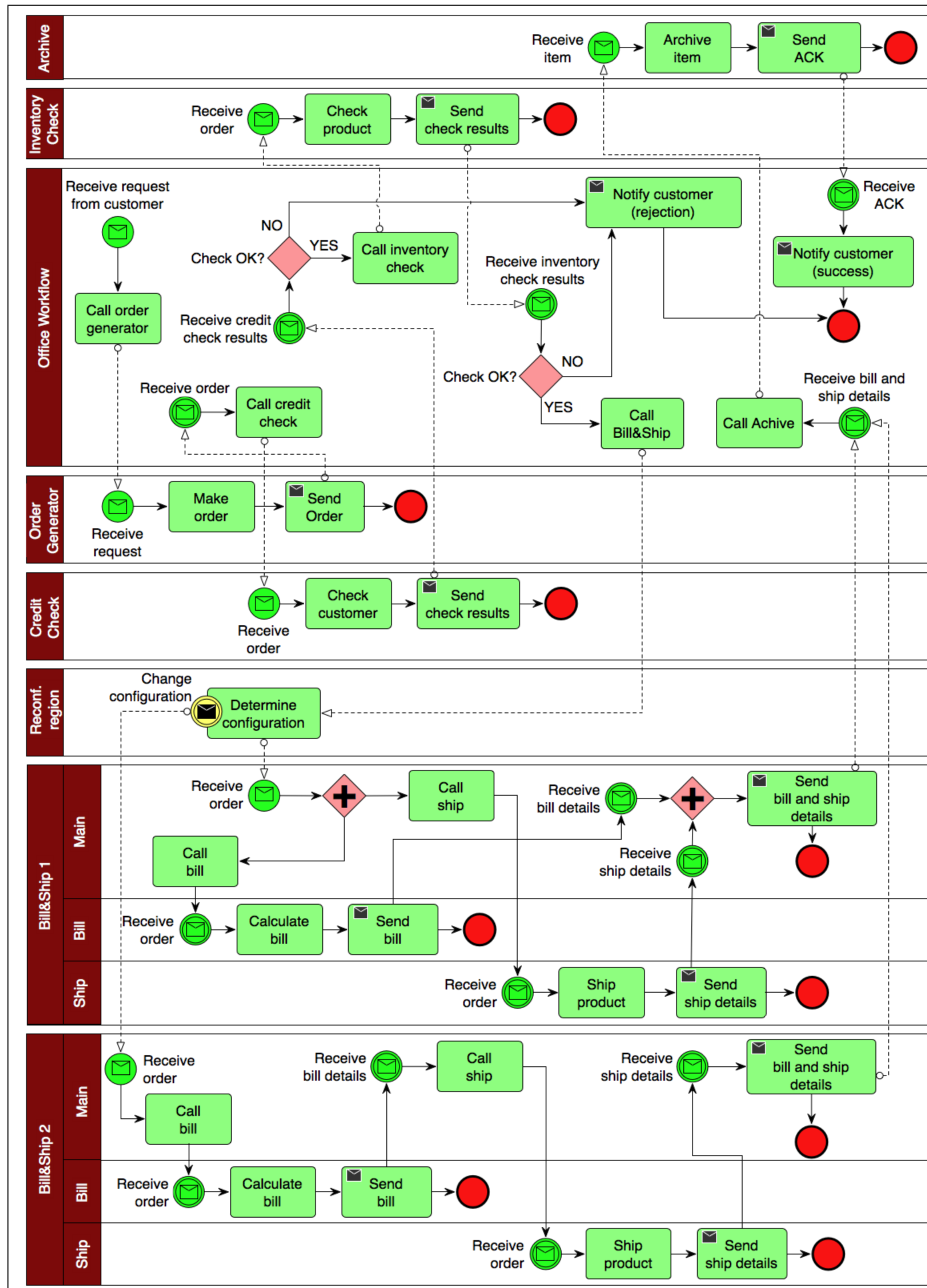

Fig. 3. The Case Study Workflow - Configuration Transition BPMN Model

new configuration has to be triggered by an event handler while the old one will have to be terminated by a termination handler. As said, the scope body runs separately (in parallel) from the event handler, so the old configuration can be terminated while the event handler brings the new one into play.In this way we can implement 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 WS-BPEL and who are interested in more details can find them in the following sections. Readers who are not interested in the details of the implementation can just skip the following sections and go straight to the conclusions without missing to grasp the general concept of our research.

5 Manual Mapping BPMN to WS-BPEL

The first problem we have encountered when mapping the BPMN design into a WS-BPEL implementation comes from the evident observation that BPMN and WS-BPEL are representative of two different classes of languages. BPMN is graph oriented while WS-BPEL is mainly block-structured [4], at least in its commonly used XLANG [15] derived subset (WS-BPEL has been also influenced by the graph oriented WSFL [10]). A consequence of this divergence is that the mapping from BPMN to WS-BPEL is hard and it has a number of limitations since BPMN is able to express process patterns which cannot be expressed in WS-BPEL. As a general comment, we could say that the block structured nature of a WS-BPEL process is too limited for modeling purposes. However, we believe that WS-BPEL cannot be ignored when it comes to workflow modeling 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 WS-BPEL to define the composite structure of a business process. In WS-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 [4] consisting of a complete translation based on the identification of patterns of BPMN fragments which can be directly mapped onto WS-BPEL code. Due to space constraints, in this paper we will show only a part of the implementation and we cannot show the details of the mapping. The full implementation and mapping can be found in [16]. Figure 3 shows a BPMN *Activity* named "Determine Configuration" with a *Non-Interrupting Intermediate Message Event*, which can be mapped to a BPEL *scope* with an *event handler* activity as follows:

```
<scope name="OldConfigScope">
  <terminationHandler>
```

```
      <scope name="NewConfigScope">
        <sequence>
          <!--perform the new configuration activities>
        </sequence>
      </scope>
    </terminationHandler>
    <eventHandlers>
      <onEvent partnerLink="Reconf.Region"
               operation="DetermineConfig"
               portType="Reconf.region:ChangePortType"
               variable="Rec"
               messageType="Reconf.region:Rec">
                <scope name="Scope">
                 <exit name="terminate"/>
                </scope>
      </onEvent>
    <eventHandlers>
  </scope>
  <scope name="BillAndShip1">
    <!-- perform bill and ship activities in parallel>
  </scope>
```

This WS-BPEL segment of code does not show the details but the underlying idea is that, if the process receives the *Rec* change message event once the *OldConfigScope* scope has been entered, it will terminate the current process and execute the new process defined within the scope *NewConfigScope* in the *termination handler*. This other process is precisely the new configuration. Otherwise, the order will enter the *BillAndShip1* scope and it will be processed accordingly to the original procedure. The full implementation of the system proves how the two basics principles presented in Section 4 are actually effective in implementing (planned) dynamic reconfiguration.

6 Discussion of the Requirements

In this section, we discuss the requirements given in section 2.1 and how they hold during the reconfiguration interval.

- **The result of the Evaluation activity for any given order should not be affected by the change in procedure.** The acceptability of an order (**Evaluation** activity) is computed outside the region to be reconfigured, and there is no interaction between **Evaluation** and the region. It means that the **Evaluation** in the old procedure *workflow* is exactly the same as in the new procedure *workflow*.
- **All accepted orders must be billed and shipped exactly once, then archived, then confirmed.** This process describes billing and shipping happening in any order but both before archiving and confirming. We declare individual variables for *BillShip1* and *NewConfigScope*. These are the

request messages used to invoke the *Bill* and *Ship* 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, *NewConfigScope* will be enabled. However, if the event is received after the order leaving *OldConfigScope*, *BillShip1* will run because the request message has been initialized. If the event is received while *OldConfigScope* is running, *OldConfigScope* will be terminated and *NewConfigScope* will start redoing *order receipt, order evaluation*, and executing *BillShip2*. *BillShip1* will not be run because no request message has been initialized and *NewConfigScope* only calls *BillShip2*.
– **All orders accepted after the change in procedure must be proceed according to the new 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 procedures: *OldConfigScope* represents the procedure running before billing and shipping, *BillShip1* represents the concurrent billing and shipping and *NewConfigScope* represents the new configuration procedure which includes sequential billing and shipping. When a management decision is made, the *event handler* for *OldConfigScope* will be invoked and it will terminate *OldConfigScope*, which contains the procedure for *order receipt, order evaluation* activities. We use a *termination handler* to replace *OldConfigScope* with *NewConfigScope* representing the new procedure. In this way, after its termination, the process will restart calling the new procedure.

In the real world, 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 continuosly receiving the "*change reconfiguration*" event.

7 Tool-based Mapping BPMN to WS-BPEL

The BPMN to BPEL mapping presented so far has been obtained by following the approach given in [4]. This allowed us to have some flexibility but the process had to be entirely manually generated. Another option, although more restrictive, is to use some automatic tool for the translation. In this section we will discuss this option using the *Intalio BPMS Designer* version 6.0 [1].

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 modeled by introducing *Pools* containing the operations of the WSDL. The process interacts with this external participants through message flows. After the process has been modeled 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. Once the project has been created, we can then create a BPMN diagram with the embedded BPMN modeler. The palette provides an immediate access to all the existing BPMN shapes. After the BPMN modeling 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.

Once the Office Workflow process is ready to be executed we can easily deploy it. There are several artifacts 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 of 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 to use a sub-process to include the two configurations and to add an *Non-interrupt Message Event* to perform the choice. If the process receives the change message, then the *configuration2* sub-process will execute (the new configuration) otherwise the process will automatically execute the old configuration sub-process *configuraiton1*. The generated BPEL code, partner links can be found in [16]. As we can see from the generated BPEL code, the interaction between the *Reconf.Region* web service and BPEL process is mapped into an *event handler* and *fault hander* activity.

```
<bpel:scope bpmn:label="Reconfiguration" name="Reconfiguration"
      bpmn:id="_EPcN4ClKEeCRVpI5R3SUgw">
     <bpel:scope bpmn:label="configuration1" name="configuration1"
          bpmn:id="_Kw694ClKEeCRVpI5R3SUgw">
       <bpel:variables>
         <!---define variables>
         ....
         <bpel:variable
              name="BillShipReply"
              messageType="BillShip1:BillingAndShipping"/>
       </bpel:variables>
       <bpel:faultHandlers>
```

```
      <bpel:catch faultName="bpmn:_YPUcsClKEeCRVpI5R3SUgw"
            faultVariable="thisChange_ConfigurationRequestMsg"
            faultMessageType="this:Change_ConfigurationRequest">
         <bpel:scope bpmn:label="configuration2"
                     name="configuration2"
                     bpmn:id="_bJFfIClKEeCRVpI5R3SUgw">
            <bpel:variables>
                ....
              <bpel:variable name="billShip2ShipReplyRequestMsg"
                    messageType="BillShip2:Shipping"/>
            </bpel:varables>
            <bpel:sequence>
            <!--perform all the new configuration actvities>
            </bpel:sequence>
         </bpel:scope>
      </bpel:catch>
   </bpel:faultHandlers>
   <bpel:eventHandlers>
     <bpel:onEvent
           partnerLink="reconfig.RegionAndOffice_WorkflowPlkVar"
           portType="this:ForReconfig.Region"
           operation="Change_Configuration"
           messageType="this:Change_ConfigurationRequest"
           variable="thisChange_ConfigurationRequestMsg"
           bpmn:label="Change Configuration"
           name="Change_Configuration"
           bpmn:id="_YPUcsClKEeCRVpI5R3SUgw">
        <bpel:scope bpmn:label="Change ConfigurationScope"
              name="Change_ConfigurationScope"
              bpmn:id="_YPUcsClKEeCRVpI5R3SUgw_scope">
           <bpel:throw faultName="bpmn:_YPUcsClKEeCRVpI5R3SUgw"
                 faultVariable="thisChange_ConfigurationRequestMsg"/>
      </bpel:scope>
     </bpel:onEvent>
   </bpel:eventHandlers>
   <bpel:sequence>
   <!--perform the activities happens before calling BillAndShip>
   </bpel:sequence>
</bpel:scope>
<bpel:scope bpmn:label="BillShip1Scope" name="BillShip1Scope"
       bpmn:id="_S_Dc8ClKEeCRVpI5R3SUgw">
   <bpel:variables>
   <!--define variables for bill and ship>
   </bpel:variables>
   <bpel:sequence>
   <bpel:variables>
   <!--define variables only visible in this scope>
      ....
     <bpel:variable name="billShip1BillShipReplyRequestMsg"
            messageType="BillShip1:BillingAndShipping"/>
```

```
        </bpel:variables>
        <!--call bill and ship in parallel>
        </bpel:sequence>
      </bpel:scope>
  </bpel:scope>
      <!--call Archive>
      ...
```

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.

The main difference between this implementation and the manual mapping is the way in which the dynamic reconfiguration is handled. The manual mapping uses an event handler to react to the change event. The event handler starts when *OldConfigScope* scope starts. This scope defines the procedure running before *bill* and *ship*. We use a termination handler activity to define a new scope *NewConfigScope* which will start the new procedure for the *order generator, order evaluation* and *BillShip2* activities. So when the *Change Configuration* event occurs, all the activities before *bill ship* will be terminated and the process will restart calling the new configuration procedure.

In the *Intalio BPMS* implementation, instead, the *fault handler* activity is used to define the new configuration procedure used to recover from the fault. The generated code is also using the *event handler* activity to react to the *Change Configuration* event throwing an *error* once the event occurs. In the BPMN diagram, it is not allowed to have two message flows entering the same *task* object: *intermediate message event* can only receive one message (bill and ship details) either from *Bill&Ship1* or from *Bill&Ship2*. So we have to add a new *Data Object* (BillShipReply) to represent a global *BillShip* reply message for the whole process. If the *Bill&Ship1* service is invoked, the reply message of *Bill&Ship1* is copied into this global variable while if the *Bill&Ship2* service is invoked the reply of *Bill&Ship2* is copied. The request message for the *Archive* Web Service is constructed by this defined global variable.

By comparing these two implementation options, we can see that even though this automatic tool can succeed in generating BPEL code, additional procedures have still to be manually provided. For example, the variables have to be initialized and correlation set has to be established. The BPMN diagram has too many limitations, the generated BPEL process maybe not suitable and able to satisfy all the requirements. In general, we cannot completely rely on the code generated by the automatic tools. We have to keep in mind that tool-based transformation has its weaknesses and should always be used with caution.

8 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 $Web\pi_\infty$. We are planning to consider more complex case studies to validate our statements. We are also working on a more complete comparisons of formalisms than the one presented in [13]: this will include the modeling of this workflow case study in several different formalisms (including π-calculus, $Web\pi_\infty$, VDM, etc...) with the consequent verification of the desired requirements. This work will also include the full WS-BPEL implementation.

Acknowledgments

The paper has been improved by conversations with Anirban Bhattacharyya, John Fitzgerald and Cliff Jones. We also want to thank members of the Reconfiguration Interest Group (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. *Intalio BPMS Designer 6.0.* Tutorials avaliable at `http://community.intalio.com/tutorials-6.0.html` (checked February 3, 2011).
2. F. Abouzaid. A mapping from pi-calculus into bpel. In *Proceeding of the 2006 conference on Leading the Web in Concurrent Engineering: Next Generation Concurrent Engineering.* IOS Press, 2006.
3. A. Carter. Using dynamically reconfigurable hardware in real-time communications systems: Literature survey. Technical report, Computer Laboratory, University of Cambridge, November 2001.
4. O. Chun, M. Dumas, and A. H. ter Hofstede. From bpmn process models to bpel web services. In *Proceedings of the IEEE International Conference on Web Services*, pages 285–293. IEEE Computer Society, Los Alamitos, 2006.
5. N. Dragoni and M. Mazzara. A formal semantics for the ws-bpel recovery framework - the pi-calculus way. In *WS-FM'09, Springer Verlag*, 2009.
6. C. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *Proceedings of the Conference on Organizational Computing Systems (COOCS 1995).* ACM, 1995.
7. P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP J. Embedded Syst.*, 2006, January 2006.
8. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *European Conference on Object-Oriented Programming (ECOOP)*, page 133147. Lecture Notes in Computer Science 512, 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. N. D. A. B. M. Mazzara, F. Abouzaid. Design, modelling and analysis of a workflow reconfiguration. Technical report, School of Computing Science, University of Newcastle, February 2011.
13. M. Mazzara and A. Bhattacharyya. On modelling and analysis of dynamic reconfiguration of dependable real-time systems. In *DEPEND, International Conference on Dependability*, 2010.
14. M. Mazzara and I. Lanese. Towards a unifying theory for web services composition. In *WS-FM*, pages 257–272, 2006.
15. S. Thatte. Xlang: Web services for business process design. Microsoft Corporation, 2001.
16. M. Zhou. A case study of workflow reconfiguration: Design and implementation. Technical report, Master Thesis.Informatics and Mathematical Modelling Department, Technical University of Denmark, 2011.