Relating Model-Based Adaptation and Implementation Platforms: 
A Case Study with WF/.NET 3.0 *

Javier Cubo1, Gwen Salaün1, Carlos Canal1, 
Ernesto Pimentel1, Pascal Poizat2,3

1 Department of Computer Science, University of Málaga 
Campus de Teatinos, 29071, Málaga, Spain 
{cubo,salaun,canal,ernesto}@lcc.uma.es

2 IBISC FRE 2873 CNRS – Université d’Évry Val d’Essonne, France
3 ARLES project, INRIA Rocquencourt, France 
pascal.poizat@inria.fr

Abstract. Our goal is to relate model-based adaptation approaches 
with implementation platforms such as BPEL or Windows Workflow 
Foundation (WF). In this paper, we present results of experiments on 
this latter one. We successively introduce a client/server system with 
mismatching components implemented in WF, our formal approach to 
work mismatch cases out, and the resulting WF adaptor. We end with 
conclusions and a list of open issues.

1 Introduction

Software Adaptation [4] is a promising research area which aims at supporting 
the building of component systems [9] by reusing software entities. These can 
be adapted in order to fit specific needs within different systems. In such a way, 
application development is mainly concerned with the selection, adaptation and 
composition of different pieces of software rather than with the programming 
of applications from scratch. Most of the approaches dedicated to model-based 
adaptation [2, 5–7, 10] have focused on the generation of pieces of software called 
adaptors which are used to solve mismatch in a non-intrusive way. This process 
is completely automated being given an adaptation mapping which is an abstract 
description of how mismatch can be solved with respect to behavioural interfaces 
of components. However, very few of these approaches relate their results with 
existing programming languages and platforms. To the best of our knowledge, the 
only attempts in this direction have been carried out using COM/DCOM [6] 
and BPEL [3].

In this paper, we propose to relate adaptor generation proposals with ex- 
isting implementation platforms. BPEL [1] and Windows Workflow Foundation

* This work has been partially supported by the project TIN2004-07943-C04-01 funded 
by the Spanish Ministry of Education and Science (MEC), and the project P06-TIC- 
02250 funded by Junta de Andalucía.
(WF) [8] are very relevant platforms because they support the behavioural descriptions of components/services. Implementing BPEL services is possible with the Java Application Server included in Netbeans Enterprise. On the other hand, WF belongs to the .NET Framework 3.0 developed by Microsoft®. Here, we have chosen WF to achieve our goal because the .NET Framework is widely used in private companies whereas BPEL is a language that recently emerged and for which tool support is being released. In addition, WF can be used to implement Web services, as it is the case for BPEL, but also any kind of software component. WF makes the implementation of services easier thanks to its workflow-based graphical support. Last, by using with WF, most of the code is automatically generated, which is not the case of BPEL.

The remainder of the paper is organised as follows. We give a quick overview on WF in Section 2. We present in Section 3 a simple example of on-line computer sale, and the WF components on which it will rely on. In Section 4, we apply successively the main steps that are necessary to compose and adapt these WF components: extraction of behavioural interfaces from components code, mismatch detection, writing of the mapping, generation of adaptor protocol, and implementation of the adaptor component from its abstract description. In Section 5, we draw up some conclusions, and discuss issues that we will tackle in future work.

2 WF Overview

In this section we present the WF constructs that we are going to use in this paper: InvokeWebService, WebServiceInput, WebServiceOutput, IfElse, While, and Listen. The reader interested in more details may refer to [8].

WF belongs to the .NET Framework 3.0, and is supported by Visual Studio 2005. The available programming languages to implement the workflows in Visual Studio 2005 are Visual Basic and C#. In this work, C# has been chosen as the implementation language.

A WF InvokeWebService activity calls a Web service and receives the requested service result back. If such an invoke has to be accessed by another component C, it has to be preceded by a WebServiceInput activity, and followed by a WebServiceOutput activity. Hence, C will interact with this new service using these two input/output activities that enable and disable the data reception and sending, respectively, with respect to the invoked Web service. WF-based XML Web services require at least one WebServiceInput and one or more WebServiceOutput activities. The input and output activities are related, thus each output activity must be associated with an input activity. It is not possible to have an instance of WebServiceInput without associated outputs, as well as having outputs without at least one WebServiceInput.

The WF IfElse activity corresponds to an if-then-else conditional expression. It requires a condition which is actually implemented as an event handler. Depending on the result value, the IfElse activity launches the execution of one
of its branches. The **while** construct defines a set of activities that are fired as many times as its condition is true.

Last, the **listen** activity defines a set of **event-driven** activities that wait for a specific event. One of the **event-driven** activities is fired when the expected message is received.

### 3 Case Study: On-line Computer Sale

In this section we introduce a simple case study of on-line computer sale. The example consists of a system whose purpose is to sell computer material such as PCs, laptops, or PDAs to clients. As a starting point we reuse two components: a **buyer** and a **supplier**. These components have been implemented using WF, and their workflows are presented in Figure 1.

![WF workflows for the Supplier and Buyer components](image)

**Fig. 1.** WF workflows for the *Supplier* and *Buyer* components

First, the **supplier** receives a request under the form of two messages that indicate the type of the requested material, and the max price to pay (**type?** and **price?**). Then, it sends a response indicating if the request can be replied positively (**reply!**). Next, the **supplier** can terminate the session, receive and reply other requests, or receive an order of purchase (**buy?**). In the latter case,
a confirmation is sent (ack!) pointing out if the purchase has been realised correctly or not.

The Buyer can submit a request (request!), in which it indicates the type of material he wants to purchase and the max price to pay for that material. Next, once he/she has received a response (reply?), the Buyer may realise another request, buy the requested product (purchase!), or end the session (stop!).

In both Supplier and Buyer we have split the workflows of Figure 1, presenting them into two parts. On the left-hand side, we show the initial execution belonging to the first request, and on the right-hand side we present the loop offering the possibility of executing other requests, performing a purchase or finalising. We identify the names of certain activities, whose functionality is the same, with an index (such as type_1 and type_2, or invokeType_1 and invokeType_2 in Supplier), because WF does not accept activities identified using the same name. Note that in the Buyer component, the messages with the code suffix, such as req_1_code, correspond to the execution of C# code.

4 Composition and Adaptation of WF Components

In this section, we focus on the composition and adaptation of the Buyer and Supplier components.

4.1 Extraction of the Behavioural Interfaces

First of all, we present in Figure 2 the LTS (Labelled Transition Systems) extracted from the workflow-based components presented in Section 3. The main ideas of the obtaining of LTS from workflow constructs are the following:

- InvokeWebService is split into two messages, one emission followed by a reception;
- WebServiceInput and WebServiceOutput messages are translated similarly in LTS;
- IfElse corresponds to a choice, that is two transitions outgoing from the same state, which encodes both parts of the conditional construct;
- While is translated into a looping behaviour in the LTS;
- Listen corresponds to a state with as many outgoing transitions as there are branches in the WF construct; each transition holds a message that may be received.

We emphasise that in the Buyer component, invoke activities are surrounded by code that have no counterpart in the corresponding LTS. The messages that appear in the Buyer LTS come from the output and input parameters that appear in its invoke activities. As far as the Supplier component is concerned, the invoke activities are made abstract because they correspond to internal behaviour. Therefore, the observable messages in this case are coming from the input and output messages surrounding the invoke activities.
In addition, initial and final states in the LTS correspond respectively to the explicit initial and final states that appear in the workflow. Note that one final state in the workflow may correspond to several final states in the LTS because several branches in the workflow may lead to the final state, and these different situations are made explicit in the LTS. To identify unambiguously component messages in the adaptation process, their names are prefixed by the component identifier, respectively \( b \) for \( Buyer \), and \( s \) for \( Supplier \). Initial and final states are respectively depicted in LTSs using bullet arrows and darkened states.

**Fig. 2.** LTS interfaces of components: \( Supplier \) (top) and \( Buyer \) (bottom)

### 4.2 Mismatch Cases

In this simple example, we can emphasise three cases of mismatch:

1. name mismatch: the \( Buyer \) may buy the computer using \( purchase! \) whereas the \( Supplier \) may interact on \( buy? \);  
2. mismatching number of messages: the \( Buyer \) sends one message for each request (\( request! \)) while the \( Supplier \) expects two messages, one indicating the type (\( type? \)), and one indicating the max price (\( price? \));  
3. independent evolution: the \( Buyer \) may terminate with \( stop! \) but this message has no counterpart in the \( Supplier \).

### 4.3 Adaptation Mapping

Now a mapping should be given to work the aforementioned cases of mismatch out. We use vectors that define some correspondences between messages. More expressive mapping notation exist in the literature, such as regular expressions of vectors [5], but with respect to the example at hand, vectors are enough to automatically retrieve a solution adaptor.
The name mismatch can be solved by vector \( V_{\text{buy}} \). The correspondence between \text{request!} and messages \text{type?} and \text{price?} can be achieved using two vectors, \( V_{\text{req}} \) and \( V_{\text{price}} \), where the second contains an independent evolution of component \text{Supplier}. Last, an independent evolution can be supported using a vector similar to \( V_{\text{price}} \) in which the message \text{stop!} is associated to nothing.

### 4.4 Generation of the Adaptor Protocol

Being given a set of component LTSs (Section 4.1) and a mapping (Section 4.3), we can use existing approaches (here we rely on [5]) to generate the adaptor protocol automatically. This is a strength of this proposal because in some cases, the adaptor protocol may be very hard to derive manually. Since the adaptor is an additional component through which all the messages transit, all the messages appearing in the adaptor protocol are reversed.

Figure 3 presents the Adaptor LTS. Note first that the adaptor receives the request coming from the \text{Buyer}, and splits the message into messages carrying the type and price information. This LTS also shows how the termination is possible along the \text{stop?} message, and how the adaptor may interact on different names (\text{purchase?} and \text{buy!}) to make the interaction possible.

#### 4.5 Implementation of the WF Adaptor

From the adaptor LTS presented above, a corresponding WF component is obtained following the reversed process that we have sketched in Section 4.1, \textit{i.e.}, by generating a workflow from an LTS. Therefore, every emission followed by a
reply is encoded as an `InvokeWebService` construct. Other input/output events are translated using `WebServiceInput/WebServiceOutput` activities. The decision of the `Buyer` is translated as a `Listen` construct, and the looping behaviour as a `While` activity. We present in Figure 4 the `Adaptor` workflow that has been encoded in WF.

![WF workflow for the Adaptor](image)

Finally, we point out that the system presented in this section has been completely implemented using WF, and the Buyer and Supplier components works as required thanks to the use of the WF `Adaptor`.

5 Conclusion

This paper has presented on a simple yet realistic example how existing model-based adaptation approaches can be related to implementation platforms such as WF in the .NET Framework 3.0. This work is very promising because it shows that software adaptation is of real use, and can help the developer in building software applications by reusing software components or services.

We end with a list of future tasks we will tackle to make the adaptation task as automated as possible:
automating the LTS extraction from WF components;
automating the mismatch detection, and generating the list of mismatch situations from a set of component LTSs;
beyond mismatch detection, tackling verification of WF components;
supporting techniques to help the designer to write the mapping out, and to generate automatically part of it;
generating WF components from the adaptor LTS.

We would also like to carry out experiments on the implementation of adaptors using BPEL and the Netbeans Enterprise platform to compare on precise criteria the adequacy of both platforms to apply adaptation in practice.

References