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

Javier Cubo, Gwen Salaün,
Carlos Canal, Ernesto Pimentel
Dept. of Computer Science, University of Málaga
Campus de Teatinos, 29071, Málaga, Spain
Emails: {cubo,salaun,canal,ernesto}@lcc.uma.es

Pascal Poizat
INRIA/ARLES Project-Team, France, and
IBISC FRE 2873 CNRS – Université d'Évry, France
Email: pascal.poizat@inria.fr

***Abstract*—In this paper, we propose to relate model-based adaptation approaches with the Windows Workflow Foundation (WF) implementation platform, through a simple case study. 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 some conclusions and a list of open issues.**

I. Introduction

Software Adaptation [1] is a promising research area which aims at supporting the building of component systems [2] 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. Many approaches dedicated to model-based adaptation [3], [4], [5], [6], [7] focus on the behavioural interoperability level, and aim at generating new components 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 [5] and BPEL [8].

In this paper, we propose to relate adaptor generation proposals with existing implementation platforms. BPEL [9] and Windows Workflow Foundation (WF) [10] 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 with BPEL platforms.

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

II. WF Overview

In this section we present the WF constructs that we use in this work: `Code`, `Terminate`, `InvokeWebService`, `WebServiceInput`, `WebServiceOutput`, `Sequence`, `IfElse`, `Listen` with `EventDriven` activities, and `While`. The reader interested in more details may refer to [10].

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.

The `Code` activity is meant to execute user code provided for execution within the workflow. The `Terminate` activity is used to finalise the execution of a workflow. 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 `Sequence` construct executes a group of activities in a precise order. The WF `IfElse` activity corresponds to an *if-then-else* conditional expression. Depending on the condition evaluation, the `IfElse` activity launches the execution of one of its branches. If none of the conditions is true, the *else* branch is executed.

The `Listen` activity defines a set of `EventDriven` activities that wait for a specific event. One of the `EventDriven` activities is fired when the expected message is received. Last, the `While` construct defines a set of activities that are fired as many times as its condition is true.

III. 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 *Supplier* and a *Buyer*. These components have been implemented using WF/.NET, and their workflows are presented in Figures 1 and 2 respectively.

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 Figures 1 and 2, 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 `request_1_code`, correspond to the execution of *C#* code. Last, some `WebServiceInput` and `WebServiceOutput` activities may be meaningless with respect to the component functionality, and appear in the WF

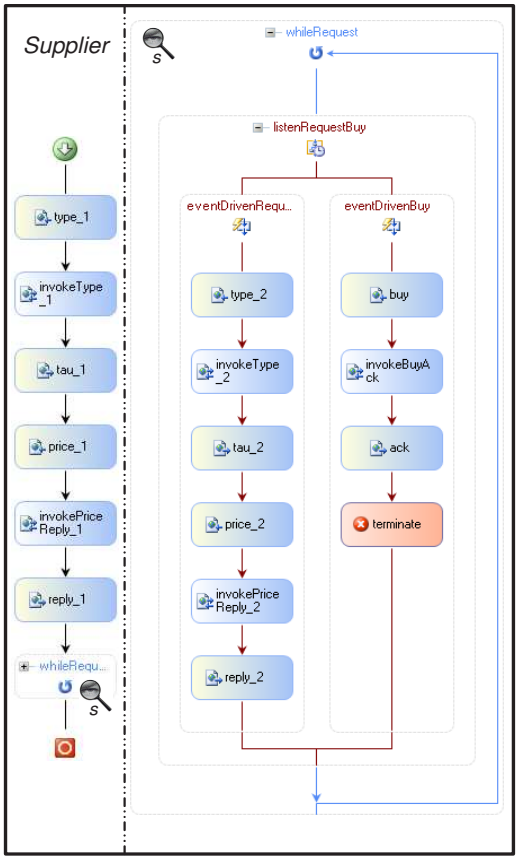

Fig. 1. WF workflow for the *Supplier* component

workflow only because WF obliges their presence before and after `InvokeWebService` activities. In Figures 1 and 2 these activities are identified with `tau` identifiers.

IV. Composition and Adaptation of WF Components

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

A. Extraction of the Behavioural Interfaces

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

- `Code` is interpreted as τ transition (internal);
- `Terminate` corresponds to a final state in LTS;
- `InvokeWebService` is split into two messages, one emission followed by a reception;
- `WebServiceInput` and `WebServiceOutput` messages are translated similarly in LTS;
- `Sequence` is translated so that it preserves the order of the involved activities in the resulting LTS;
- `IfElse` corresponds to a choice, that is two transitions outgoing from the same state, which encodes both parts of the conditional construct;

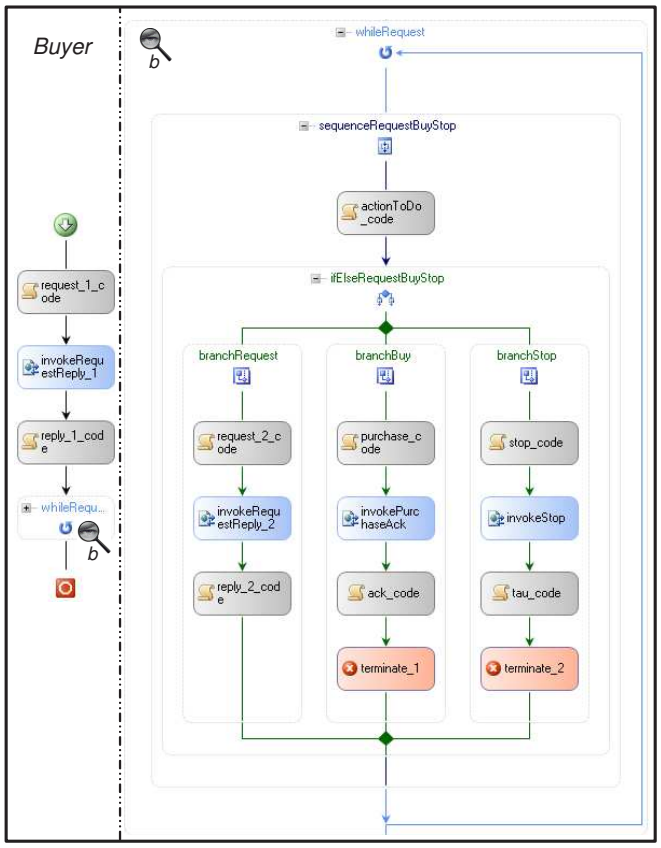

Fig. 2. WF workflow for the *Buyer* component

- `Listen` corresponds to a state with as many outgoing transitions as there are branches in the WF contruct; each transition holds a message that may be received;
- `While` is translated as a looping behaviour in the LTS.

LTS does not support the description of data expressions, consequently conditions appearing in `While` and `IfElse` constructs are abstracted during the LTS extraction stage. Likewise, `WebServiceInput` and `WebServiceOutput` activities identified with `tau` identifiers (see Figs. 1 and 2) are translated as τ transitions in the corresponding LTS.

Initial and final states in the LTS come respectively from the explicit initial and final states that appear in the workflow. There is a single initial state that corresponds to the beginning of the workflow. Final states correspond either to a `Terminate` activity or to the end of the whole workflow. Accordingly, several final states may appear in the LTS because several branches in the workflow may lead to the final state. Initial and final states are respectively depicted in LTSs using bullet arrows and darkened states.

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 interactions with external components (in charge of the material database), and are not of interest for the composition at hand. Therefore, the observable messages in this case are coming from the input and output messages surrounding the invoke activities. All the τ transitions in both LTSs corresponding to $C\#$ code in the $Buyer$ workflow, and to `tau WebServiceOutput` activities in the $Supplier$ one have been removed (by $\tau*.a$ reduction [11]) to favour readability. To identify unambiguously component messages in the adaptation process, their names are prefixed by the component identifier, respectively b for $Buyer$, and s for $Supplier$.

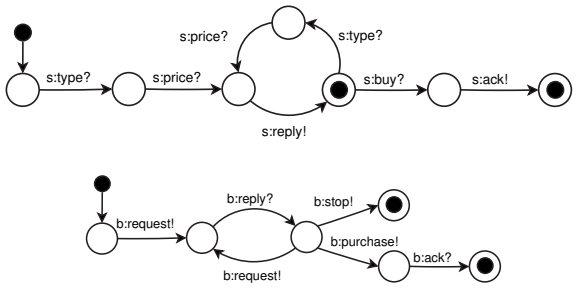

Fig. 3. LTS interfaces of *Supplier* (top) and *Buyer* (bottom) components

B. 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 $Supplier$.

C. 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 [4], but with respect to the example at hand, vectors are enough to automatically retrieve a solution adaptor.

$\mathtt{V_{req}} = \langle \mathtt{b{:}request!}, \mathtt{s{:}type?} \rangle$
$\mathtt{V_{price}} = \langle \mathtt{b{:}}\varepsilon, \mathtt{s{:}price?} \rangle$
$\mathtt{V_{reply}} = \langle \mathtt{b{:}reply?}, \mathtt{s{:}reply!} \rangle$
$\mathtt{V_{stop}} = \langle \mathtt{b{:}stop!}, \mathtt{s{:}}\varepsilon \rangle$
$\mathtt{V_{buy}} = \langle \mathtt{b{:}purchase!}, \mathtt{s{:}buy?} \rangle$
$\mathtt{V_{ack}} = \langle \mathtt{b{:}ack?}, \mathtt{s{:}ack!} \rangle$

The name mismatch can be solved by vector $\mathtt{V_{buy}}$. The correspondence between `request!` and messages `type?` and `price?` can be achieved using two vectors, $\mathtt{V_{req}}$ and $\mathtt{V_{price}}$, where the second contains an independent evolution of component $Supplier$. The last mismatch is solved using $\mathtt{V_{stop}}$ in which the message `stop!` is associated to nothing.

D. Generation of the Adaptor Protocol

Given a set of component LTSs (Section IV-A) and a mapping (Section IV-C), we can use existing approaches (here we rely on [4]) 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 4 presents the *Adaptor* LTS. Note first that the adaptor receives the request coming from the *Buyer*, and splits the message into messages carrying the type and price information. This LTS also shows how the termination is possible along the `stop?` message, and how the adaptor may interact on different names (`purchase?` and `buy!`) to make the interaction possible.

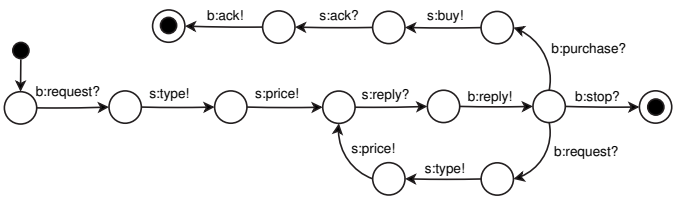

Fig. 4. *Adaptor* protocol for the case study

E. 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 IV-A, *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 5 the *Adaptor* workflow that has been encoded in WF.

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* component.

V. 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. To make this work, we had to face and work out specificities of the WF platform such as the use of `tau WebServiceOutput` activities, or of several `InvokeWebService` activities in one session. 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 stage as automated as possible:

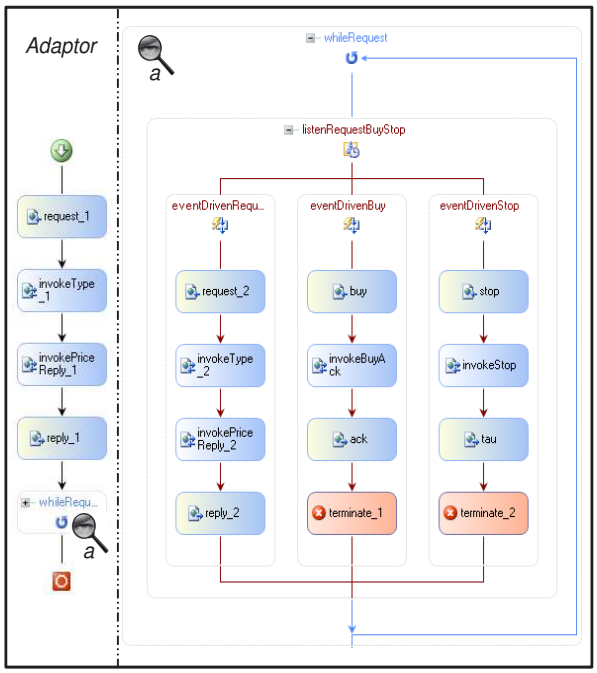

Fig. 5. WF workflow for the *Adaptor* component

- automating the LTS extraction from WF workflows;
- 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 workflows 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.

Acknowledgment

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.

References

[1] C. Canal, J. Murillo, and P. Poizat, "Software Adaptation," *L'Objet*, vol. 12, no. 1, 2006, special Issue on Coordination and Adaptation Techniques for Software Entities.

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Adisson-Wesley, 2003.

[3] A. Bracciali, A. Brogi, and C. Canal, "A Formal Approach to Component Adaptation," *Journal of Systems and Software*, vol. 74, no. 1, 2005.

[4] C. Canal, P. Poizat, and G. Salaün, "Synchronizing Behavioural Mismatch in Software Composition," in *Proc. of FMOODS'06*, ser. LNCS, vol. 4037, 2006.

[5] P. Inverardi and M. Tivoli, "Deadlock-Free Software Architectures for COM/DCOM Applications," *Journal of Systems and Software*, vol. 65, no. 3, 2003.

[6] H. W. Schmidt and R. H. Reussner, "Generating Adapters for Concurrent Component Protocol Synchronization," in *Proc. of FMOODS'02*. Kluwer, 2002.
[7] D. M. Yellin and R. E. Strom, "Protocol Specifications and Components Adaptors," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 2, 1997.
[8] A. Brogi and R. Popescu, "Automated Generation of BPEL Adapters," in *Proc. of ICSOC'06*, ser. LNCS, vol. 4294, 2006.
[9] T. Andrews *et al.*, *Business Process Execution Language for Web Services (WSBPEL)*, BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems, Feb. 2005.
[10] K. Scribner, *Microsoft Windows Workflow Foundation: Step by Step*. Microsoft Press, 2007.
[11] H. Garavel, R. Mateescu, F. Lang, and W. Serwe, "CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes," in *Proc. of CAV'07*, ser. Lecture Notes in Computer Science, vol. 4590, 2007.