A Model-Based Approach to the Verification and Adaptation of WF/.NET Components

Javier Cubo, Gwen Salaün, Carlos Canal, Ernesto Pimentel

Department of Computer Science, University of Málaga
Campus de Teatinos, 29071, Málaga, Spain
Email: {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

This paper presents an approach which supports verification and model-based adaptation of software components and services implemented using Windows Workflow Foundation (WF). First, we propose an abstract description of WF workflows, and we formalise the extraction of Labelled Transition Systems from these workflows. Next, verification and adaptation are applied using respectively model-checking techniques and existing model-based adaptation approaches. Last, we explain how a WF workflow can be generated from an adaptor protocol.

Keywords: Sofware Components, Services, Composition, Model-based Adaptation, WF Workflows, Model-checking

1 Introduction

Software Adaptation [4,7] is a promising research area which aims at supporting the building of component systems [25] 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 [5,8,16,19,23,26] 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 the behavioural interfaces of components. However, most of these approaches are independent of the implementation framework, and few of them relate with existing programming languages and platforms. To the best of our knowledge,

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the only attempts in this direction have been carried out using COM/DCOM [16], BPEL [6], and SCA components [19].

In this paper, we focus on Windows Workflow Foundation (WF) [24] which belongs to the .NET Framework 3.0 developed by Microsoft®. We have chosen WF because this platform supports the behavioural descriptions of components/services using workflows. In addition, the .NET Framework is widely used in private companies, and makes the implementation of services easier thanks to its workflow-based graphical support and the automation of the code generation. More than dealing only with adaptation of WF components, our approach also allows the verification of such components by extracting abstract descriptions from them and by using model-checking tools. This work extends and formalises the ideas sketched in [9].

Our approach is summarised in Figure 1. To make the verification and adaptation possible, in a first stage, abstract behavioural descriptions (Labelled Transition Systems, LTSs) have to be extracted from WF workflows. Next, being given a set of LTSs, mismatch detection is computed to check whether the involved components need adaptation or not. If a mismatch exists, we apply adaptation techniques that aim at generating an adaptor protocol/LTS from a mapping. Assessment techniques are then helpful to check that the adaptor is as expected. If not, another mapping may be proposed. We emphasise that formal verification of WF components takes place twice: when detecting mismatch, and when assessing the resulting system (components+adaptor). Last, once the designer is satisfied by the abstract adaptor, the corresponding WF workflow is generated.

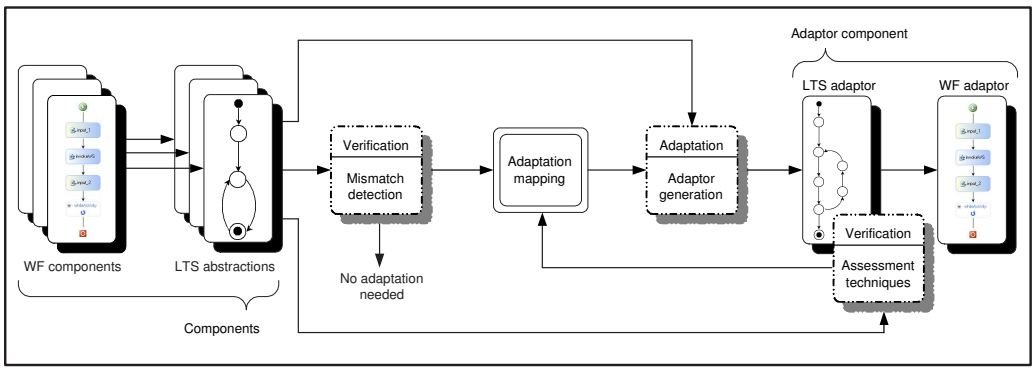

Fig. 1. Overview of our approach for the adaptation of WF components

The remainder of the paper is organised as follows. We give an overview of WF, and we define an abstract notation for WF workflows in Section 2. We present in Section 3 a simple on-line computer sale example, and the WF components it relies on. Section 4 formalises the extraction of LTSs from WF workflows. In Section 5, we focus on the verification and adaptation of WF components based respectively on model-checking techniques and model-based adaptation. Section 6 presents the encoding of an adaptor LTS into a WF workflow. In Section 7, the contributions of our approach are compared to related work. Finally, in Section 8, we conclude the paper and present future work.

2 WF Workflow Notation

In this paper, we present a representative kernel of the WF activities, namely `Code`, `Terminate`, `InvokeWebService`, `WebServiceInput`, `WebServiceOutput`, `Sequence`, `IfElse`, `Listen` with `EventDriven` activities, and `While`. The reader interested in more details may refer to [24]. We also introduce a textual and abstract notation for the aforementioned WF activities.

2.1 WF Overview

WF belongs to the .NET Framework 3.0, and is supported by Visual Studio 2005. The available programming languages to implement 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 an activity that is fired as many times as the `While` condition is true.

2.2 Abstract Notation for WF Workflows

Here, we define a textual and abstract notation for WF workflows. This notation makes abstract several implementation details. Our proposal considers as input textual workflows instead of their graphical description. Table 1 formalises the grammar for the textual notation of WF activities $\mathcal{A}$, where C, C_i are boolean conditions, and I, I_i (inputs), O, O_i (outputs) are parameters of activities.

$\mathcal{A}$	::=	`Code`	*executes a chunk of code*
	|	`Terminate`	*ends a workflow's execution*
	|	`InvokeWebService(`$O_1,\ldots,O_n,I$`)`	*calls a Web service (WS)*
	|	`WebServiceInput(`$I_1,\ldots,I_n$`)`	*receives data from a WS*
	|	`WebServiceOutput(`O`)`	*sends data to a WS*
	|	`Sequence(`$\mathcal{A}_1,\mathcal{A}_2$`)`	*executes first* $\mathcal{A}_1$ *and then* $\mathcal{A}_2$
	|	`IfElse(`$(C_1,\mathcal{A}_1),\ldots,(C_n,\mathcal{A}_n),\mathcal{A}_{n+1}$`)`	*executes* $\mathcal{A}_i$ *if* C_i *is true, or* $\mathcal{A}_{n+1}$ *otherwise*
	|	`Listen(`$E_1,\ldots,E_n$`)`	*fires one of the* E_i *branches*
	|	`While(`$C,\mathcal{A}$`)`	*executes* $\mathcal{A}$ *while* C *is true*
E	::=		
		`EventDriven(WebServiceInput(`I`),`$\mathcal{A}$`)`	*executes* $\mathcal{A}$ *when* I *is received*

Table 1
Grammar for the abstract notation of WF workflows

3 Running Example: On-line Computer Sale

This section introduces an on-line computer sale example. It 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./NET, and their workflows are presented in Figure 2.

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`, (1) and (2) respectively in Fig. 2). Then, it sends a response indicating if the request can be replied positively (`reply`, (3)). Next, the *Supplier* can terminate the session, receive and reply other requests ((4), (5) and (6)), or receive an order of purchase (`buy`, (7)). In the latter case, a confirmation is sent (`ack`, (8)) emphasising if the purchase has been realised correctly or not.

The *Buyer* can submit a request (`request`, (9) in Fig. 2), in which it indicates the type of material he/she wants to purchase and the price to pay. Next, once he/she has received a response (`reply`, (10)), the *Buyer* may realise another request ((11) and (12)), buy the requested product (`purchase` and `ack`, (13) and (14)), or end the session (`stop`, (15)).

In both *Supplier* and *Buyer* we have split the workflows of Figure 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. In the *Buyer*

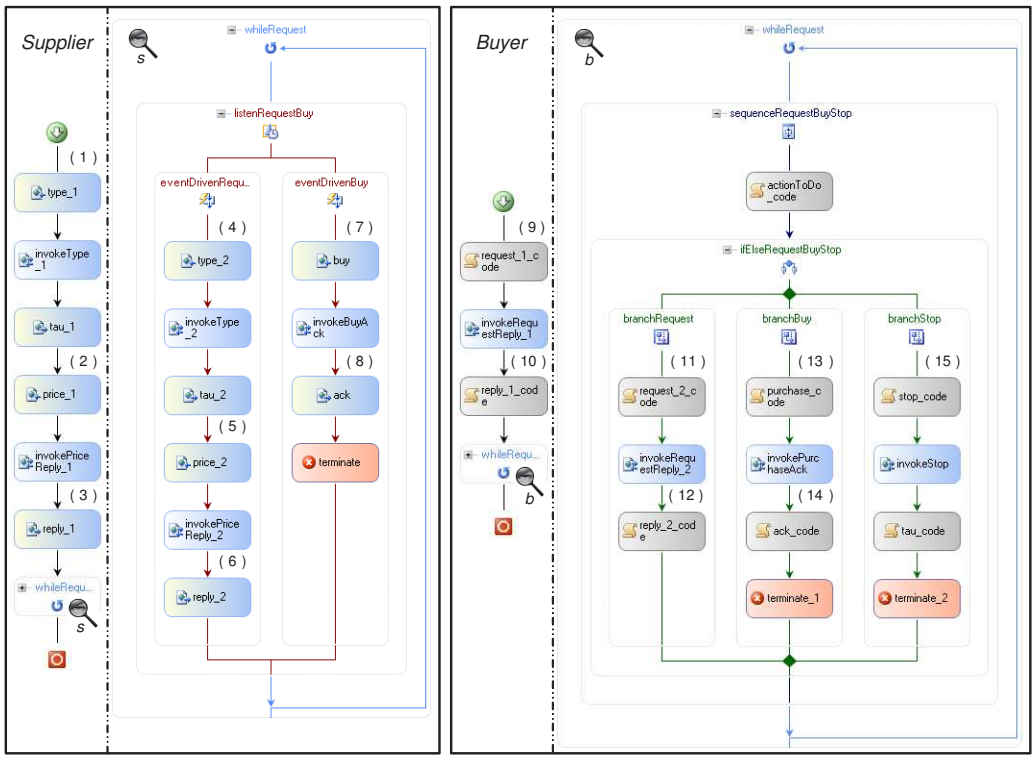

Fig. 2. WF workflows for the *Supplier* (left) and *Buyer* (right) components

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

To illustrate the textual notation defined in Section 2.2, we apply it on the *Supplier* WF workflow. We focus on the `While` construct, and present a part of the `Listen` activity it contains. The condition of the `While` construct is `true` because the component loops on requests until it receives an order of purchase, or until the system stops.

```
Sequence
  ( ...,
    While
      ( true,
        Listen
          ( EventDriven
              ( WebServiceInput(type),
                ...
              ),
            EventDriven
              ( WebServiceInput(buy),
                Sequence
                  ( InvokeWebService(buy, ack),
                    Sequence
                      ( WebServiceOutput(ack),
                        Terminate
                      )
                  )
              )
          )
      )
```

)
)

Note that we remove in the abstract notation all the suffixes used in the workflows to distinguish activity names. Last, we recall that in the following we consider such an abstract description of WF components, as an input (Section 4) and output (Section 6) to our verification and adaptation proposal.

4 Extracting LTSs from WF Workflows

Since we want to reuse existing techniques to make verification and adaptation of WF components, we have first to extract from the abstract WF notation the required model, namely Labelled Transition Systems. A LTS is a tuple (A, S, I, F, T) where: A is an alphabet (set of events or messages), S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ are final states, and $T \subseteq S \times A \times S$ is the transition function. The extracted LTSs must preserve the semantics of workflows as encoded in WF/.NET Framework 3.0. A formal proof of semantics preservation between both levels is not achieved yet since WF does not provide a formal semantics. Our encoding has been deduced from our experiments using the WF platform. The main ideas of the LTS obtaining from abstract description of workflow constructs are the following.

- `Code` is internal and hence interpreted as an internal transition, τ;
- `Terminate` corresponds to a final state;
- `InvokeWebService` corresponds to a sequence of emissions followed by a reception, `WebServiceInput` corresponds to a sequence of receptions, and `WebServiceOutput` corresponds to an emission;
- `Sequence` is translated so as to preserve the order of the involved activities. For this, the final states of the first activity are linked to the initial state of the second activity using ϵ transitions;
- `IfElse` corresponds to an internal choice. This corresponds to as many τ transitions as there are branches in the `IfElse` construct (including the else branch). Each of these τ transitions leads to the initial state of the corresponding activity;
- `Listen` corresponds to an external choice. This corresponds to as many outgoing transitions as there are branches in the `Listen` construct. These transitions are labelled with receptions corresponding to the messages that can be received and target the initial state of the related activity;
- `While` is translated as a looping behaviour, where the choice between termination or loop is encoded using internal non-determinism (τ transitions).

Formally, an LTS $\mathcal{L} = (A, S, I, F, T)$ can be obtained from an abstract workflow represented by activity $\mathcal{A}$ using function $awf2lts : WF \rightarrow LTS$. For an LTS $\mathcal{L} = (A, S, I, F, T)$, we define $X(\mathcal{L}) = X$ for every X in $\{A, S, I, F, T\}$. This notation is overloaded for activities: for some activity $\mathcal{A}$, we define $X(\mathcal{A}) = X(awf2lts(\mathcal{A}))$ for every X in $\{A, S, I, F, T\}$. Finally, we use $new\ s$ to denote the creation of s as a new (fresh) state in the LTSs we are building. $awf2lts$ can be defined inductively on the structure of WF activities as follows:

$\texttt{Code} \mapsto (\{\tau\}, \{new\ s_1, new\ s_2\}, s_1, \{s_2\}, \{s_1 \xrightarrow{\tau} s_2\})$

$\texttt{Terminate} \mapsto (\varnothing, \{new\ f\}, f, \{f\}, \varnothing)$

$\texttt{InvokeWebService}(O_1, \ldots, O_n, I) \mapsto (\{O_1!, \ldots, O_n!, I?\}, \bigcup_{i\in\{0,\ldots,n+1\}}\{new\ s_i\},$
$s_0, \{s_{n+1}\}, (\bigcup_{i\in\{1,\ldots,n\}} s_{i-1} \xrightarrow{O_i!} s_i) \cup \{s_n \xrightarrow{I?} s_{n+1}\})$

$\texttt{WebServiceInput}(I_1, \ldots, I_n) \mapsto (\{I_1?, \ldots, I_n?\}, \bigcup_{i\in\{0,\ldots,n\}}\{new\ s_i\}, s_0, \{s_n\},$
$\bigcup_{i\in\{1,\ldots,n\}} s_{i-1} \xrightarrow{I_i?} s_i)$

$\texttt{WebServiceOutput}(O) \mapsto (\{O!\}, \bigcup_{i\in\{0,1\}}\{new\ s_i\}, s_0, \{s_1\}, \{s_0 \xrightarrow{O!} s_1\})$

$\texttt{Sequence}(\mathcal{A}_1, \mathcal{A}_2) \mapsto (A(\mathcal{A}_1) \cup A(\mathcal{A}_2) \cup \{\epsilon\}, S(\mathcal{A}_1) \cup S(\mathcal{A}_2), I(\mathcal{A}_1), F(\mathcal{A}_2),$
$T(\mathcal{A}_1) \cup T(\mathcal{A}_2) \cup \{f \xrightarrow{\epsilon} I(\mathcal{A}_2) \mid f \in F(\mathcal{A}_1)\})$

$\texttt{IfElse}((C_1, \mathcal{A}_1), \ldots, (C_n, \mathcal{A}_n), \mathcal{A}_{n+1}) \mapsto ((\bigcup_{i\in\{1,\ldots,n+1\}} A(\mathcal{A}_i)) \cup \{\tau\},$
$(\bigcup_{i\in\{1,\ldots,n+1\}} S(\mathcal{A}_i)) \cup \{new\ s\}, s, \bigcup_{i\in\{1,\ldots,n+1\}} F(\mathcal{A}_i),$
$\bigcup_{i\in\{1,\ldots,n+1\}}(T(\mathcal{A}_i) \cup \{s \xrightarrow{\tau} I(\mathcal{A}_i)\}))$

$\texttt{Listen}(\texttt{EventDriven}(\texttt{WebServiceInput}(I_1), \mathcal{A}_1), \ldots,$
$\texttt{EventDriven}(\texttt{WebServiceInput}(I_n), \mathcal{A}_n)) \mapsto (\bigcup_{i\in\{1,\ldots,n\}}(A(\mathcal{A}_i) \cup \{I_i?\}),$
$(\bigcup_{i\in\{1,\ldots,n\}} S(\mathcal{A}_i)) \cup \{new\ s\}, s, \bigcup_{i\in\{1,\ldots,n\}} F(\mathcal{A}_i),$
$\bigcup_{i\in\{1,\ldots,n\}}(T(\mathcal{A}_i) \cup \{s \xrightarrow{I_i?} I(\mathcal{A}_i)\}))$

$\texttt{While}(C, \mathcal{A}) \mapsto (A(\mathcal{A}) \cup \{\epsilon\} \cup \{\tau\}, S(\mathcal{A}) \cup \{new\ s, new\ f\}, s, F(\mathcal{A}) \cup \{f\},$
$T(\mathcal{A}) \cup \{s \xrightarrow{\tau} I(\mathcal{A})\} \cup \{s \xrightarrow{\tau} f\} \cup \{f' \xrightarrow{\epsilon} I(\mathcal{A}) \mid f' \in F(\mathcal{A})\})$

Once the LTS is constructed, ϵ transitions are removed [15]. LTS does not support the description of data expressions, consequently conditions appearing in `While` and `IfElse` constructs are abstracted away while extracting LTS. Likewise, `WebServiceInput` and `WebServiceOutput` activities identified with `tau` identifiers (see Fig. 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 a final state. Initial and final states are respectively depicted in LTSs using bullet arrows and hollow states.

Let us illustrate the extraction of LTSs from abstract WF workflows on our running example (Figure 3). 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 surrouding the invoke activities. All the τ transitions in LTSs are removed using a $(\tau*.a)$ behavioural reduction [12] before the adaptation process to favour efficiency and 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*.

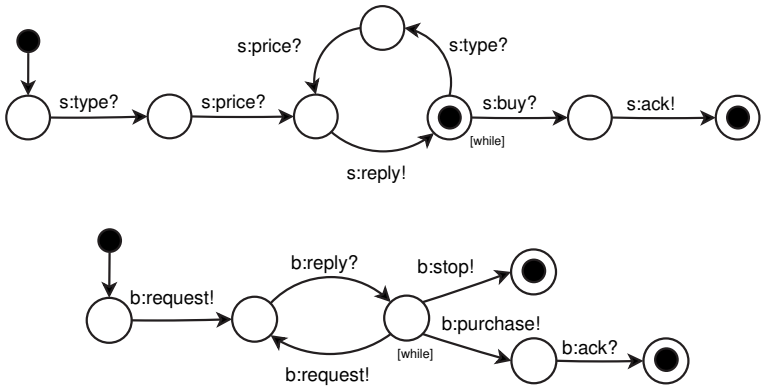

Fig. 3. LTS interfaces of *Supplier* (top) and *Buyer* (bottom) components

5 Verification and Model-Based Adaptation in WF

This section presents our approach to verify and compose/adapt WF components. Verification techniques are useful in two cases: first, they may help to identify mismatch situations, and, in a second step, they allow to check if the adaptor works correctly, since the designer writes the mapping by hand, therefore it may contain some errors that will be reflected in the adaptor protocol.

5.1 Detection of Mismatch Cases

First of all, let us introduce verification techniques that can be used to check component LTSs, and their composition with the adaptor LTS (see Section 5.4). All existing model-checking tools that accept automata-based format as input are good candidates to these checks, namely SPIN [14], CADP [12] or mCRL2 [13]. In this paper, we illustrate these ideas with CADP which is a verification toolbox for asynchronous concurrent systems. CADP allows to deal with very large state spaces, and implements various verification techniques such as model checking, compositional verification, equivalence checking, distributed model checking, etc.

The main idea is to generate the full system using parallel composition operators available in CADP (or similar tools), and then to reason on the resulting system using mainly visual checking and model-checking of temporal properties. Model-checking is an automatic technique that efficiently detects subtle architectural flaws. Classical properties such as liveness or safety properties can be easily formalised reusing patterns [17], and then checked against the system model (LTS) using model-checkers, *e.g.,* Evaluator [18] which belongs to CADP.

As regards our running example, we first compute the resulting LTS by composing components *Buyer* and *Supplier* and enforcing their interaction on all messages appearing in both components. The resulting LTS consists of a single state with no outgoing transitions. This is quite obvious because both components suffer of mismatch in their first transition (`request!` in the *Buyer* versus `type?` in the *Supplier*). Indeed, a study of these LTSs points out the three following cases of mismatch:

(i) name mismatch: the *Buyer* may buy the computer using `purchase!` whereas the *Supplier* may interact on `buy?`;

(ii) 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?`);

(iii) independent evolution: the *Buyer* may terminate with `stop!` but this message has no counterpart in the *Supplier*.

5.2 Adaptation Mapping

Now, a mapping should be given to work the aforementioned cases of mismatch out. ε is used in vectors when some message has no counterpart in a component (see e.g. $\mathtt{V}_{\mathtt{price}}$ and $\mathtt{V}_{\mathtt{stop}}$ bellow). We use vectors that define correspondences between messages. More expressive mapping notations exist in the literature, such as regular expressions of vectors [8], but with respect to the example at hand, vectors are enough to automatically retrieve a solution adaptor. A possible mapping for our example is as follows:

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

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

5.3 Generation of the Adaptor Protocol

Given a set of component LTSs (Section 4) and a mapping (Section 5.2), we can use existing approaches (here we rely on [8]) to generate the adaptor protocol automatically. This automation is crucial because in some cases, the adaptor protocol may be very hard to derive manually.

Figure 4 presents the *Adaptor* LTS. Since the adaptor is an additional component through which all the messages transit, all the messages appearing in the adaptor protocol are reversed with respect to the ones in the components. 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.

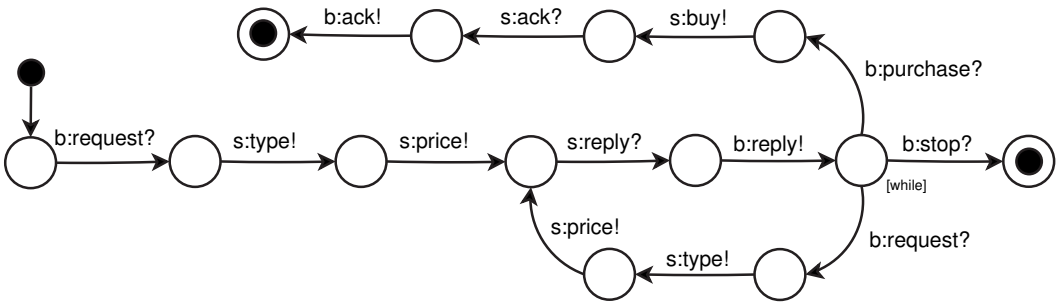

Fig. 4. *Adaptor* protocol for the case study

5.4 Assessment Techniques

In this last step, we use model-checking techniques to validate the adaptor LTS generated from the mapping proposed above. First, the LTS corresponding to the composition of both components and the adaptor is computed. Synchronisation is made explicit, and both components interact together through the adaptor. Next, the designer may write some properties to be verified by the final system. In the rest of this section, we show some examples of μ-calculus formulas we checked on this system using Evaluator:

(i) a supplier always replies a buyer request

```
[true* . "b_request"]
   mu X. (<true> true and [not ("s_reply")]X)
```

(ii) a buyer request is always followed by stop, purchase, or request

```
[true* . "b_request"]
   mu X. (<true> true and
      [not ("b_stop" or "b_purchase" or "b_request")]X)
```

(iii) a buyer request is always followed by stop, or purchase

```
[true* . "b_request"]
   mu X. (<true> true and [not ("b_stop" or "b_purchase")]X)
```

Properties (i) and (ii) are true, whereas the last one (iii) is false, but this is normal because the system can loop forever on exchanging request/reply messages. If some properties turn out to be false whereas a positive answer was expected, it means that the adaptation mapping contains errors that does not make the system behave as required. In this situation, the mapping must be corrected and assessment applies again.

6 Generating WF Workflows from LTSs

The last step in our proposal is to generate an abstract workflow from an adaptor protocol. Formalising the function *lts*2*awf* is quite tough, especially because cycles in the adaptor LTS have to be encoded with `While` activities which must preserve

the LTS behaviour. Therefore, as a first attempt, we give in this section some guidelines for this encoding.

First, the initial state of the LTS is encoded as the initial state of the workflow. Final states are encoded as `Terminate` activities. The adaptation process removes all the τ transitions. Then, all the needed pieces of $C\#$ code will be added by hand while refining the abstract workflow into a real WF workflow.

The translation process derives step by step parts of the abstract workflow by focusing on one state of the LTS after the other. We distinguish in the following the translation of transitions corresponding to message activities (`InvokeWebService`, `WebServiceInput`, `WebServiceOutput`), and the generation of structuring activities (`Sequence`, `IfElse`, `Listen`, `While`). Let us start with messages, and note that the three rules below have to be applied in this order to check if the sequence of messages corresponds to an `InvokeWebService` before translating it in separate `WebServiceOutput` or `WebServiceInput` activities:

- a sequence of transitions with labels holding one or several emissions followed by a reception is encoded as an `InvokeWebService` activity;
- one or several transitions with receptions as labels are translated into a `WebServiceInput` activity;
- a transition with one emission corresponds to a `WebServiceOutput` activity.

Now, we focus on the encoding of the LTS structuring into the abstract workflow:

- a `Sequence` activity is generated for a sequence of transitions in the LTS corresponding to two successive message activities, and for which no states involve more than one outgoing transition;
- if the state of the LTS to be translated involves two or more outgoing transitions:
  - if all the outgoing transitions hold input messages, a `Listen` activity is derived,
  - otherwise a conditional choice `IfElse` activity is generated;
- a cycle in an LTS is translated using a `While` activity. If several cycles loop on a same state, it corresponds to a single `While` activity. However if a cycle in the LTS contains another (local) cycle, this latter will also be translated as a `While` activity nested in the outmost one.

Following these guidelines, an abstract workflow has been derived for our running example that we do not show here for space reasons. Last, this abstract workflow has been refined into a WF workflow (Fig. 5). This refinement step requires the intervention of the designer, to (i) concretise conditions in `IfElse` and `While` activities, and (ii) add $C\#$ pieces of code to get the adaptor WF component works correctly. Moreover, WF requires addresses of components to be specified in invocations. Therefore, to deploy our adapted system, we have first to update the components workflows to change these addresses into the adaptor one. However, this can be done automatically.

Finally, we point out that the simple system presented in this paper has been completely implemented using WF, and the *Buyer* and *Supplier* components worked as required thanks to the use of the *Adaptor* component.

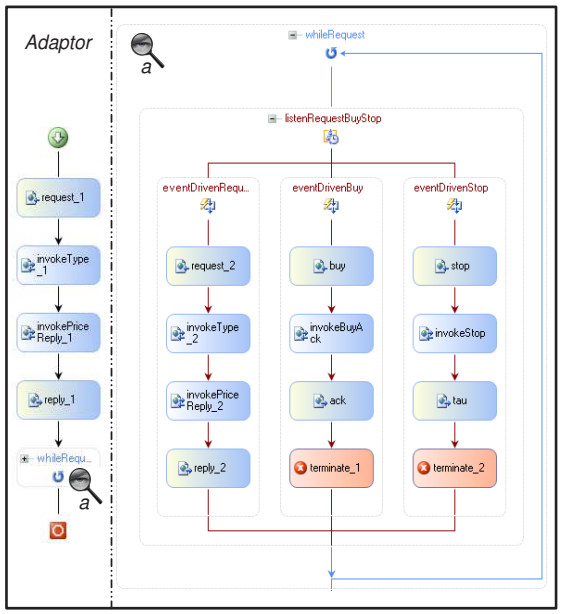

Fig. 5. WF workflow for the *Adaptor* component

7 Related Work

The first group of related work concerns proposals that aimed at applying adaptor generation approaches to existing implementation platforms. Brogi and Popescu [6] outline a methodology for the automated generation of adaptors capable of solving behavioural mismatches between BPEL processes [1]. In their adaptation methodology they use YAWL workflow as intermediate language. Once the adaptor workflow is generated, they use lock analysis techniques to check if a full adaptor has been generated or only a partial one (some interaction scenarios cannot be resolved).

In [6], the authors chose BPEL. Both BPEL and WF languages allow to design Web services, but WF can also be used to implement any kind of software component. Their respective platforms make the implementation easier thanks to their workflow-based graphical support, and the automated generation of most of the underlying code (XML+Java in BPEL, using Java Application Server included in Netbeans Enterprise, and XML+C# in WF). In this work, we have focused on WF because it is an interesting alternative to BPEL that has not been studied yet. In addition, as a long term purpose, we want our proposal to benefit to the wide number of people who use the .NET Framework in private companies around the world. Compared to [6] our adaptation approach is able to reorder messages in between components when required.

Inverardi and Tivoli [16] tackle the automatic synthesis of connectors in the COM/DCOM framework, by guaranteeing deadlock-free interactions among components. They may also define properties that the resulting system should verify using liveness and safety properties expressed as specific processes. Compared to this proposal, our approach does not only restrict the adaptor to possible non-deadlocking behaviours [16] but may also address behavioural adaptation. That comes from the notation and adaptation techniques we rely on that allows to deal

with possibly complex adaptation scenarios, whereas this approach does not use any mapping language for adaptor specification.

As regards verification of component-based systems, recent approaches have been dedicated to the verification of software components specified using LOTOS, LTSs and synchronisation networks [2,3]. These works present a method and a tool intended to application developers, to build behavioural models of Fractal components on which properties can be verified using CADP. In the Web Service area, different works have been dedicated to verifying Web service description to ensure some properties of systems [10,11,21,22]. Summarising these works, they use model-checking to verify some properties of cooperating Web services described using XML-based languages (DAML-S, WSFL, BPEL, WSCI). Accordingly, abstract representations are extracted from Web service implementations, and some properties may be ensured using *ad-hoc* or well-known tools (*e.g.*, SPIN, LTSA). Last, Mouakher *et al.* [20] start with a description of components using UML class and state diagrams that they encode into the B method to use its associated theorem prover, namely Atelier B or B4free, so as to perform compatibility checks. In a second step, they specify adaptors in B, and address their correctness.

Compared to these different proposals, ours focuses on both verification and adaptation of components. We prefer model checking (instead of theorem proving with B for instance) because it makes verification steps easier thanks to a full automation and its adequacy to automata-based models. In addition, adaptation techniques support the automatic generation of adaptors in case verification reveals that components cannot be directly reused (the adaptor is completely specified by hand in [20]).

8 Concluding Remarks

This paper has presented an approach to verify WF components, and in case they cannot be directly composed, we have sketched how an adaptor protocol can be generated, and encoded into a new WF component. We have illustrated the application of our proposal in practice on a simple yet realistic example. This work is promising because it demonstrates that software adaptation can be of real interest for widely used implementation platforms such as the .NET Framework 3.0, and can help the developer in building software applications by reusing software components or services.

As far as future work is concerned, here is a list of perspectives we will tackle to complement our approach:

- extending the set of WF activities considered in our proposal;
- extending our LTS model with respect to these new activities, and keeping data description at this level;
- formalising both functions $awf2lts$ and $lts2awf$ to support the automatic extraction and generation of abstract workflows;
- extending our verification and adaptation proposal to deal with this new model;
- implementing our translation functions between LTSs and abstract workflows in a prototype tool;

- implementing in this tool automatic translators between WF workflows (described in XML format) and abstract workflows;
- experimenting the proposal on more complex and realistic examples.

In parallel, 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.

Acknowledgements. This work has been partially supported by project TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), and project P06-TIC-02250 funded by Junta de Andalucía.

References

[1] T. Andrews et al. *Business Process Execution Language for Web Services (WSBPEL)*. BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems, 2005.

[2] T. Barros, A. Cansado, and E. Madelaine. Model-Checking Distributed Components: The Vercors Platform. In *Proc. of FACS'06*, volume 182 of *ENTCS*, pages 3–16. Elsevier, 2007.

[3] T. Barros, L. Henrio, and E. Madelaine. Verification of Distributed Hierarchical Components. In *Proc. of FACS'05*, volume 160 of *ENTCS*, pages 41–55, 2006.

[4] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an Engineering Approach to Component Adaptation. In *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 193–215.

[5] A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.

[6] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 27–39, 2006.

[7] C. Canal, J.M. Murillo, and P. Poizat. Software Adaptation. *L'Objet*, 12(1):9–31, 2006. Special Issue on Coordination and Adaptation Techniques for Software Entities.

[8] C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of FMOODS'06*, volume 4037 of *LNCS*, pages 63–77, 2006.

[9] J. Cubo, G. Salaün, C. Canal, E. Pimentel, and P. Poizat. Relating Model-Based Adaptation and Implementation Platforms: A Case Study with WF/.NET 3.0. In *Proc. of WCOP'07*, pages 9–13, 2007.

[10] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *Proc. of ASE'03*, pages 152–163, Canada, 2003. IEEE Computer Society Press.

[11] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*, pages 621–630. ACM Press, 2004.

[12] 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*, volume 4590 of *LNCS*, 2007.

[13] J. F. Groote, A. Mathijssen M. van Weerdenburg, and Y. S. Usenko. From μCRL to mCRL2: Motivation and Outline. In *Proc. of the Workshop "Essays on Algebraic Process Calculi" (APC 25)*, volume 162 of *ENTCS*, pages 191–196, 2006.

[14] G. Holzmann. *The SPIN Model-Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[15] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition, 2003.

[16] P. Inverardi and M. Tivoli. Deadlock-Free Software Architectures for COM/DCOM Applications. *Journal of Systems and Software*, 65(3):173–183, 2003.

[17] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[18] R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, 2003.

[19] H. Motahari, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-Automated Adaptation of Service Interactions. In *Proc. of WWW'07*, pages 993–1002. ACM Press, 2007.

[20] I. Mouakher, A. Lanoix, and J. Souquieres. Component Adaptation: Specification and Verification. In *Proc. of WCOP'06*, pages 23–30, 2006.

[21] S. Nakajima. Model-Checking Verification for Reliable Web Service. In *Proc. of OOWS'02, satellite event of OOPSLA'02*, 2002.

[22] S. Narayanan and S. McIlraith. Analysis and Simulation of Web Services. *Computer Networks*, 42(5):675–693, 2003.

[23] H. W. Schmidt and R. H. Reussner. Generating Adapters for Concurrent Component Protocol Synchronization. In *Proc. of FMOODS'02*, pages 213–229. Kluwer, 2002.

[24] K. Scribner. *Microsoft Windows Workflow Foundation: Step by Step*. Microsoft Press, 2007.

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

[26] D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.