Automated Generation of BPEL Adapters*

Antonio Brogi and Razvan Popescu

Computer Science Department, University of Pisa, Italy

Abstract. The heterogeneous, dynamic, distributed, and evolving nature of Web services calls for adaptation techniques to overcome various types of mismatches that may occur among services developed by different parties. In this paper we present a methodology for the automated generation of (service) adapters capable of solving behavioural mismatches among BPEL processes. The adaptation process, given two communicating BPEL processes whose interaction may lock, builds (if possible) a BPEL process that allows the two processes to successfully interoperate. A key ingredient of the adaptation methodology is the transformation of BPEL processes into YAWL workflows.

1 Introduction

BPEL [2] is currently used to (manually) compose WSDL [15] services into complex business applications. A main problem to achieve automated service composition is that the composite application may lock due to interaction mismatches among the participant services. One possibility to overcome such mismatches is a disciplined use of *adapters*, as services "in-the-middle" capable of mediating the information exchanged by the involved parties.

Service adaptation may be tackled at various levels of the Web services stack [10]. For example, signature-based adaptation [9,12] addresses issues due to syntactic differences among the exchanged messages (e.g., different orderings of the message parts), ontology-based adaptation [8,11] mediates semantic mismatches among the exchanged messages (e.g., messages belonging to different ontology concepts), and behaviour-based adaptation [1,3] handles the integration of services into a lock-free aggregate due to mismatches in their communicating protocols (e.g., different orderings of message exchanges). However, Web service adaptation is in its early stages and current approaches feature only partial solutions to the issues of adaptation.

Our long term objective is to develop a general methodology for service adaptation capable of suitably overcoming signature, ontology and behaviour mismatches in view of business application integration within and across organisational boundaries. In this paper we present a methodology for the automated generation of (service) adapters capable of solving *behavioural mismatches* among BPEL processes. The adaptation process, given two communicating BPEL processes whose interaction may lock, builds (if possible) a BPEL process that allows the two processes to successfully interoperate. Three strong

* This work has been partially supported by the SMEPP project (EU-FP6-IST 0333563) and by the F.I.R.B. project TOCAI.IT.

motivations for adapting services are the need to develop adapters for service composition, for ensuring backwards compatibility of new service versions, as well as the need to develop adapters for each class of clients a service may have. A key ingredient of the adaptation methodology is the use of service contracts [6] including WSDL *signatures* and YAWL *behaviour*, where YAWL [13] is used as intermediate (formal) language to provide a (partial) description of the service behaviour. Immediate advantages of using such an abstract language are the possibility of adapting services written in different service description languages, multiple deployment of the adapter as a real-world service, as well as developing formal analyses and transformations independently of the different languages used by providers to describe the behaviour of their services. Moreover, integration with the YAWL-based service customisation [5] and aggregation of Web services [4,6] becomes straightforward.

Regrettably, space limitations do not allow us to introduce BPEL and YAWL. Detailed descriptions of the two languages are to be found in [2] and [13], respectively.

2 Motivating Example

Consider the following example consisting of two interacting BPEL processes: *Command Centre* (CC) and *Mars Explorer* (ME). The former provides a Web service interface for the assignment of exploration tasks. The latter is a Web service interface to the robot performing the tasks. Hereafter we present a simplification of the BPEL processes (e.g., in order to express the message exchanges we simply use service names instead of *partnerLink*s and *portType*s). Although fairly simple, the example illustrates various interactions among services. On the one hand, CC communicates with its client, as well as with the ME service. On the other hand, ME interacts with CC (viz., its client), as well as with the *Logger* and *Explorer* services.

```
<process name="CommandCentre"><sequence>
  <receive op="ExecTask" from Client var="taskInfo" createInst="yes"/>
  <invoke op="Login" of MarsExplorer var="loginInfo"/>
  <assign><copy> from="/taskInfo/coords" to="coords"></copy></assign>
  <invoke op="SetCoords" of MarsExplorer var="coords"/>
  <assign><copy> from="/taskInfo/job" to="jobDetails"></copy></assign>
  <invoke op="SetJob" of MarsExplorer var="jobDetails"/>
  <pick>
    <onMsg op="SubmitRep" from MarsExplorer var="report"><sequence>
      <receive op="JobID" from MarsExplorer var="id"/>
      <invoke op="Logout" of MarsExplorer/>
      <reply op="ExecTask" of Client var="report"/></sequence></onMsg>
    <onMsg op="SubmitErr" from MarsExplorer var="error"><sequence>
      <invoke op="Logout" of MarsExplorer/>
      <assign><copy> from="error" to="report"></copy></assign>
      <reply op="ExecTask" of Client var="report" faultName="Task_Error"/>
    </sequence></onMsg></pick></sequence></process>
```

The CC service[1] first receives the task information from its client. It then logs in with the ME, to which it forwards the location and the job details. It waits next either a report or an error message from the ME. In the former case, it first receives the job id from the ME, then it closes the connection with the ME, and finally, it forwards the report to the client. In the latter case, it first logs out from the ME, and then it replies to the client with the error message.

[1] We use "process" and "service" interchangeably to denote BPEL processes.

```
<process name="MarsExplorer"><sequence>
   <receive op="Login" from CommandCentre var="loginInfo" createInst="yes"/>
   <invoke op="JobID" of CommandCentre var="id"/>
   <receive op="SetJob" from CommandCentre var="jobDetails"/>
   <receive op="SetCoords" from CommandCentre var="coords"/>
   <invoke op="ValidateLocation" of LoggerService inVar="coords" outVar="rep1"/>
   <invoke op="Explore" of ExplorerService inVar="jobDetails" outVar="rep2"/>
   <assign><copy> from="concat(rep1,rep2)" to="report"></copy></assign>
   <invoke op="SubmitRep" of CommandCentre var="report"/>
   <receive op="Logout" from CommandCentre/></sequence>
<faultHandlers>
   <catch faultName="Task_Error" faultVar="error"><sequence>
      <invoke op="SubmitErr" of CommandCentre var="error"/>
      <receive op="Logout" from CommandCentre/>
   </sequence></catch></faultHandlers></process>
```

The ME service starts by waiting for the CC to log in, to which it sends immediately the job's id. It receives next from the CC the job description and the location of the exploration site. In order to carry out the task, the ME first validates the coordinates (e.g., by checking previous exploration logs) and moves the robot to the respective location by (synchronously) invoking the *Logger Service* (LS), and then, it delegates the *Explorer Service* (ES) for the actual execution of the job (again, through a synchronous invocation). If the latter two invocations return successfully, the ME generates the final report, sends it to the CC, and waits for the CC to log out. Note that, although not represented in the example, the invocations to the LS and to the ES may return a "Task_Error" fault. (This information has to be specified in the WSDL file(s) defining the respective operations). In that case, the ME service catches the fault, forwards to the CC the error, and finally, it waits for the CC to close the connection.

It is easy to see that the two services, CC and ME, cannot successfully interact because of mismatches between their behaviour. Immediately after the login information exchange, while the CC sends the location of the exploration site to the ME, the ME sends the job id to the CC. Furthermore, the CC first sends the location, and then the details of the job to the ME, which expects them in the reversed order. A further mismatch is the fact that, while the CC expects the job id only when the exploration is successful, the ME always sends it, and moreover, at a different moment.

The following Section 3 shows how we automatically generate BPEL adapters to cope with such behabioural mismatches.

3 Adaptation Methodology

The adaptation methodology inputs two communicating BPEL processes, C and S, whose interaction may lock, and it builds (if possible) a BPEL process adapter A, which allows the two processes to successfully interoperate. The four adaptation phases are: **(1.) Service Translation**. This phase is in charge of translating the BPEL descriptions of C and S into corresponding YAWL workflows [7]. **(2.) Adapter Generation**. This phase builds the YAWL workflow of A from the workflows of C and S. It first generates the *Service Execution Trees* (SETs) of C with respect to S ($SET(C_S)$), as well as of S with respect to C ($SET(S_C)$), followed by the generation of the SETs of their duals ($SET(\overline{C_S})$ and $SET(\overline{S_C})$). Informally, when a service X outputs a message m, a dual of

X is a service that inputs m, and vice-versa. Next, $SET(A)$ is obtained by suitably merging $SET(\overline{C_S})$ and $SET(\overline{S_C})$. Finally, the YAWL workflow of A is derived from $SET(A)$. **(3.) Lock Analysis**. This phase verifies whether the YAWL-based aggregation [4,6] of C, A, and S locks. If it does, we consider that the adaptation has failed. Otherwise, we consider that the adaptation is successful. **(4.) Adapter Deployment**. If the adaptation is successful, this phase deploys the YAWL workflow of A as a BPEL process, which can be used as a service-in-the-middle between C and S.

3.1 Service Translation

In [7] we present a methodology for translating BPEL processes into YAWL workflows. Its main strengths are that (1) it defines YAWL patterns for all BPEL activities, (2) it provides a compositional approach to construct structured patterns from suitably interconnecting other patterns, and (3) it handles events, faults and (explicit) compensation.

On the one hand, the pattern of each BPEL basic activity (with the exception of *assign* and *compensate*) is obtained by suitably instantiating the *Basic Pattern Template* (BPT). The BPT is a template of YAWL tasks, which serves both for identifying the translated activity (through an *Activity Specific Task*, or AST for short), as well as the control-logic of executing or skipping the activity. On the other hand, the pattern of each BPEL structured activity (together with *assign*, *compensate*, and *process*) is obtained from the *Structured Pattern Template* (SPT) template. The SPT consists of a *Begin* (logically marking the initiation of the structured activity) and of an *End* pattern (logically marking the termination of the structured activity), as well as a pattern template (BPT or SPT) for each child activity. Furthermore, the *Scope* and *Process* patterns add SPTs for handling exceptional behaviour. Each pattern inputs and outputs at most three types of control-flow links, called *green*, *blue*, and *red* lines. The green lines serve for translating the structural dependencies among BPEL activities. The blue lines are used for translating the BPEL synchronisation links, and the red lines are necessary for implementing the fault handling mechanism. As space limitations do not allow us to go into further details, please see [7] for more (in-depth) details on the BPEL2YAWL translator.

The YAWL workflows of the CC and ME services of our example can be seen in Figure 1.[2] In the workflow of ME, *Begin(Process)* and *End(Process)*, logically mark the initiation and the termination, respectively, of the BPEL process. The process activity, a *sequence* leads to generating the *Begin(Sequence)* as well as the *End(Sequence)* tasks. The first activity in the sequence is a *receive*, which gives the *Receive* task. Furthermore, the rest of the activities are translated correspondingly. (The numbers inside some of the task labels are used for disambiguation purposes only.) Note however the translation of the BPEL *pick*. The *Begin(Pick)* task contains the branch selection logic (basically a deferred

[2] The two workflows are represented in a slightly simplified form w.r.t. the description given in [7] (e.g., the default *faultHandlers* of the process, as well as redundant *green gate*s are not represented, the *assign* is represented in a compact form, etc.).

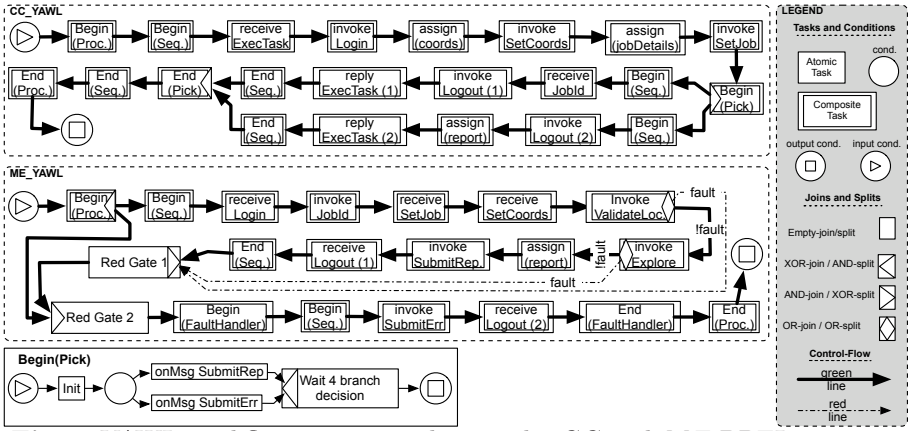

Fig. 1. YAWL workflows corresponding to the CC and ME BPEL processes.

choice construct [13]), and it outputs two tokens[3]. One leads to executing the chosen branch, while the second leads to skipping the other branch (so as to achieve the dead-path-elimination).

The workflow of CC is built in a similar manner. However, the composite tasks representing the *invoke ValidateLocation* and *invoke Explore* activities output *either* green tokens, if the invocations succeed, *or* red tokens, if the invocations fail (i.e., faults are being raised). In the former case, the execution of the workflow continues normally, and the green output of *End(Sequence)* leads to skipping the tasks inside the *Begin(FaultHandler)* $\rightarrow$ *End(FaultHandler)* zone (so as to achieve the dead-path-elimination). In the latter case, the execution of the faulty invocation is (immediately) followed by the execution of the tasks in the fault handling zone.

3.2 Adapter Generation

The Adapter Generation phase consists of the four steps discussed hereafter.

Service Execution Trees. This step automatically generates the Service Execution Trees (SETs) of the two services to be adapted. The SET of a BPEL process X is a tree describing all the possible scenarios of executing the basic activities (or activities, for short) of X. Informally, the root of the SET is given by the activity (or activities) that can be executed first, while the leaves correspond to activities executed last. Each intermediary node represents the execution of one or more activities. A node consisting of more than one activity denotes a concurrent execution of the respective activities. Given a node n, child nodes of n contain (distinct sets of) activities that can be executed immediately after executing the activities of n. Hence, one may think of each path in the tree as a service execution trace. We generate the SET of a BPEL process X through a reachability analysis [4] of its corresponding YAWL workflow obtained during the Service Translation phase. Note that the BPEL2YAWL translator [7] allows

[3] The semantics of executing YAWL workflows is quite similar to executing PNs.

us to cope – when adapting – both with synchronisation links and with the exceptional behaviour of BPEL. In order to cope with loops in the process, our reachability analysis uses the modified reachability trees defined in [14]. Furthermore, each node of the SET can be labelled with a condition constraining its execution. Such conditions are due to guards employed by *switch* activities and synchronisation links. In [4,5] we show how to generate the logical expression constraining the fulfilment of a service execution trace. However, in order to ease the description of the methodology, and due to space limitations, we do not detail this issue here. The SET one obtains for a service X contains all message exchanges of X with other services. We call this the *full-form* of the SET, and we denote it by $SET(X)$. $SET(ME)$ is given in Figure 2(a). For example, the execution of the (synchronous) *invoke ValidateLocation* can be followed *either* by the *invoke Explore*, *or* by the *invoke SubmitErr*. The former is due to a successful execution of the *invoke ValidateLocation*, while the latter is executed in the case of a fault being received by the *invoke ValidateLocation*. Furthermore, the successful termination of the sequence activity of the BPEL process leads to employing the dead-path-elimination inside the pattern implementing the *faultHandler* of the BPEL process [7]. This is indicated in $SET(ME)$ by the dark coloured *invoke SubmitErr* and *rcv Logout* nodes.

From (the full-form of) $SET(X)$ we derive next the (*compact-form* of) SET of X with respect to another service Y, with which X interacts. We denote it by $SET(X_Y)$. Informally, from the original $SET(X)$ we keep only message exchanges between X and Y. First, all message exchanges (viz., receive/reply/invoke) of X with services other than Y, as well as all other basic activities (e.g., assign), and all skipped activities are replaced by *empty* activities. We denote the resulting SET as $SET(X^*_Y)$. For example, the *invoke ValidateLocation* and the *invoke Explore*, which ME performs on the *Logger Service* and *Explorer Service*, respectively, are set to *empty* when computing $SET(ME_{CC})$. ($SET(ME^*_{CC})$ is given in Figure 2(b).) Second, each *empty* node in $SET(X^*_Y)$ (with the exception of the root) is removed from the tree, and its sub-trees (if any) are merged with its parent nodes. We denote the resulting tree by $SET(X_Y)$. Note that the merge process applied at a node n of $SET(X^*_Y)$ also removes duplicate subtrees of n. For instance, by removing the three empty nodes of $SET(ME^*_{CC})$, we get two identical subtrees (*invoke SubmitErr* → *receive Logout*) at the *receive SetCoords* node. The merge at *receive SetCoords* will then remove one duplicate. $SET(ME_{CC})$ is represented in Figure 2(c). Due to space limitations, we present only the $SET(CC_{ME})$ – which is built analogously – in Figure 2(d).

Dual SETs. This step generates for each service X (to be adapted), the SET of a dual of X with respect to another service Y. Basically, when X receives a message m from Y, a dual of X with respect to Y (denoted by $SET(\overline{X_Y})$) acts somewhat "as Y should" and sends a message m to X, and vice-versa. One obtains $SET(\overline{X_Y})$ from $SET(X_Y)$ by replacing asynchronous *invoke*s with *receive*s (and vice-versa), and synchronous *invoke*s with pairs *receive* → *reply* (and vice-versa). $SET(\overline{ME_{CC}})$ and $SET(\overline{CC_{ME}})$ are depicted in Figure 3(a) and (b), respectively.

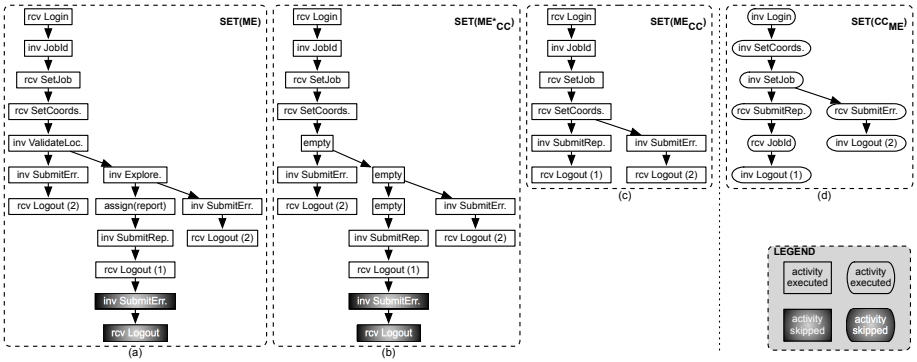

Fig. 2. (a) $SET(ME)$, (b) $SET(ME^*_{CC})$, (c) $SET(ME_{CC})$, and (d) $SET(CC_{ME})$.

Adapter SET. The SET of an adapter A ($SET(A)$) mediating the interaction of two services, C and S, is obtained by suitably merging $SET(\overline{C_S})$ with $SET(\overline{S_C})$. This process consists of two steps, as follows.

During the first step, we *match* the activities of $SET(\overline{C_S})$ with the activities of $SET(\overline{S_C})$ with the following two rules: (1) An asynchronous *invoke Op* of $SET(\overline{C_S})$ matches a *receive Op* of $SET(\overline{S_C})$, and vice-versa, and (2) a synchronous *invoke Op* of $SET(\overline{C_S})$ matches a pair *receive Op* $\rightarrow$ *reply Op* of $SET(\overline{S_C})$, and vice-versa. Then, we express each match as a *data-flow dependency* (or dependency, for short), which emerges at the *receive* and targets the *invoke*, in the case of asynchronous message exchanges, or as a pair of dependencies, one emerging at the *receive* and targeting the *invoke*, and another one emerging at the *invoke* and targeting the *reply*, in the case of synchronous message exchanges. We call an activity that is target of at least one dependency as *constrained*. Otherwise, we say that the activity is *unconstrained* (with respect to the data-flow dependencies between the two SETs). For example, *invoke Login* and *receive JobId* of $SET(\overline{ME_{CC}})$ match *receive Login* and *invoke JobId*, respectively, of $SET(\overline{CC_{ME}})$. (See Figure 3(a) and (b).) Informally, a dependency indicates that the adapter has to wait first a message from one of the two services, and then (possibly at a later moment) it forwards it to the other service. In other words, a dependency from X to Y says that the adapter has to execute X before executing Y. Note that the interpretation in the case of multiple dependencies emerging from different activities X_k and targeting an activity Y, is that for the execution of Y it suffices to execute only one activity X_k. This is the case for *invoke Logout (1)* and *invoke Logout (2)* of $SET(\overline{ME_{CC}})$.

As previously mentioned, each path in $SET(X)$ is an execution trace of X. During the second step, we compute the merge of all possible pairs of traces $(\overline{c}, \overline{s})$, where $\overline{c} =< \overline{c_1}, \overline{c_2}, \ldots, \overline{c_n} >$ is a trace of $SET(\overline{C_S})$, and $\overline{s} =< \overline{s_1}, \overline{s_2}, \ldots, \overline{s_m} >$ is a trace of $SET(\overline{S_C})$. Such a merge can lead either to a success, or to a failure. In the former case, the merge of $\overline{c}$ and $\overline{s}$ gives a (successful) trace a of the adapter A (and consequently a path in $SET(A)$). At each step, the merge process compares nodes $\overline{c_i}$ and $\overline{s_j}$, by starting from the roots of the two traces, and it produces a node a_k. In terms of BPEL activities, one may think of the node a_k as a sequence

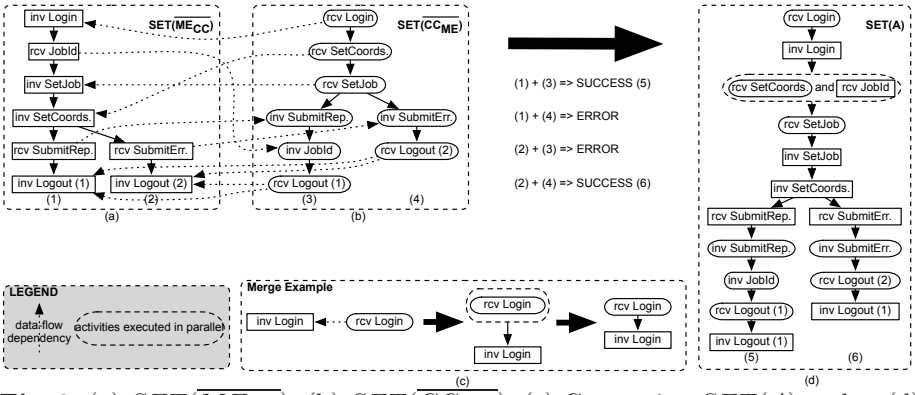

Fig. 3. (a) $SET(\overline{ME_{CC}})$, (b) $SET(\overline{CC_{ME}})$, (c) Generating $SET(A)$ nodes, (d) $SET(A)$.

containing a flow. The merge algorithm basically adds activities of the two nodes ($\overline{c_i}$ and $\overline{s_j}$) either inside the flow, or inside the sequence yet following the flow. For simplicity, we informally describe hereafter the algorithm of merging two nodes containing each one activity only, and each being the target of at most one dependency. (The general case of merging nodes with multiple activities and multiple constraints is analogous.) If $\overline{c_i}$ is unconstrained, then add $\overline{c_i}$ to the flow inside a_k (e.g., merging *receive JobId* and *receive SetCoords*). Please note that in the case of an unconstrained *invoke* activity, the merge process (of the two traces) returns with a *failure*. We do so in order to avoid the generation of (arbitrary) messages by the adapter. Otherwise, if $\overline{c_i}$ is constrained by $\overline{s_J}$ such that $J < j$ (i.e., from the point of view of executing the trace $\overline{s}$, activity of $\overline{s_J}$ has already been executed), then add $\overline{c_i}$ to the flow (e.g., merging *invoke SetCoords* and *invoke SubmitRep*). Otherwise, if $\overline{c_i}$ is constrained by $\overline{s_J}$ such that $J = j$ (i.e., activity of $\overline{s_J}$ is ready to be executed), then add $\overline{c_i}$ to the sequence, following the flow (e.g., merging *invoke Login* and *receive Login*). Otherwise, if $\overline{c_i}$ is constrained by $\overline{s_J}$ such that $j < J$ (i.e., activity of $\overline{s_J}$ is not executable yet), then we say that the trace $\overline{c}$ is "stalled" (e.g., assume merging *invoke SetJob* and *receive SetCoords*). Next, the algorithm does the same for $\overline{s_j}$. For example, one may see in Figure 3(c) the result of merging the roots of $\overline{ME_{CC}}$, and $\overline{CC_{ME}}$. (The elimination of the flow is due to the fact that it contains one activity only.) If both traces are stalled, then we have a lock between the two traces, and hence a *failure* in merging the two traces. Otherwise, the algorithm continues by comparing the node $\overline{c_i}$ (if $\overline{c}$ is stalled) or $\overline{c_{i+1}}$ (if $\overline{c}$ is not stalled) with the node $\overline{s_j}$ (if $\overline{s}$ is stalled) or $\overline{s_{j+1}}$ (if $\overline{s}$ is not stalled). If the merge has added to the trace a all nodes of one of the two traces ($\overline{c}/\overline{s}$), it simply appends at the end of a the remaining sequence of nodes of the other trace ($\overline{s}/\overline{c}$). If all nodes of both $\overline{c}$ and $\overline{s}$ have been added to a, then we have a *success*, and a represents a (successful) trace of the adapter A.

Next, we derive $SET(A)$ by merging all successful traces a of A. If no such successful traces exist, then the algorithm generating the adapter fails, as the mismatches between the two interacting processes cannot be solved. For example,

if the root of $SET(\overline{C_S})$ consists of an *invoke Op1* and if the root of $SET(\overline{S_C})$ consists of another *invoke Op2*, then we have a deadlock as each service is waiting to receive a message from the other. Consider a set $\{a^1, a^2, \ldots, a^p\}$ of successful adapter traces. The merge algorithm, in this case, starts by considering $SET(A)$ to be a^1. Then, for all nodes a_i^k of the other traces a^k, it checks whether a_i^k is contained in $SET(A)$ at depth i. If so, it marks the respective position in the tree, and it choses the next node in the sequence (i.e., a_{i+1}^k). Otherwise, it adds the rest of the trace a^k, including the node a_i^k, as a branch splitting from the last marked node in $SET(A)$. For our example, we get only two successful traces of the adapter. The first one, denoted by (5) in Figure 3(d) is obtained by merging the traces denoted by (1) and (3) of $SET(\overline{ME_{CC}})$ and $SET(\overline{CC_{ME}})$, respectively, while the second one, denoted by (6) is obtained by merging traces denoted by (2) and (4). These two adapter traces are then merged into the adapter given in Figure 3(d).

Adapter Workflow. If the adapter has at least one successful trace, then the adaptation process generates next the YAWL workflow of the adapter A from $SET(A)$ as described hereafter. Initially, it generates the *Begin(Process)* and the *Begin(Sequence)*, as well as the *End(Sequence)* and the *End(Process)* patterns [7], which logically mark the initiation of the business process and of its activity, as well as their termination, respectively. The former two, as well as the last two are to be linked in a sequence. (See Figure 4.) Basically, generating the pattern of a basic activity simply consists of instantiating the *Basic Pattern Template* defined in [7] (e.g., setting the name, inputs, and outputs of its *Activity Specific Task*), while generating the pattern of a structured one reduces to instantiating its *Begin* and its *End* patterns, and the pattern of each child activity (, as well as the patterns for handling the exceptional behaviour, if any). For each node n in $SET(A)$, starting with its root, the algorithm generates and adds to the workflow the pattern(s) corresponding to the activity (activities) contained in n. If n consists of one activity only, then the pattern of its (basic) activity is produced and suitably linked in the workflow as output of the pattern corresponding to the parent node of n (or to *Begin(Sequence)* if n is the root). For example, the *receive Login* root of $SET(A)$ leads to a *Receive* pattern linked as output of *Begin(Sequence)*. If n consists of multiple activities, then the pattern given by the node is a *Flow*, which includes the patterns of each activity in the node. Next, if n has one child node only, the adaptation process continues with its child. Otherwise, if n has more than one successor, then we have three possibilities: (1) If all child nodes of n contain each one *receive* only, and if there are no conditions constraining their execution[4] then the resulting pattern is a *Pick* having the respective *receive*s as *onMessage* tasks in *Begin(Pick)*, and for each branch is generated a *Sequence* pattern. The generation process continues then on each subtree having as root a child of n (excluding the child of n already considered as *onMessage* inside the *Pick*). (2) If all child nodes of n are constrained by (disjoint)

[4] We recall that such conditions are due to the guards of *switch* activities and synchronisation links.

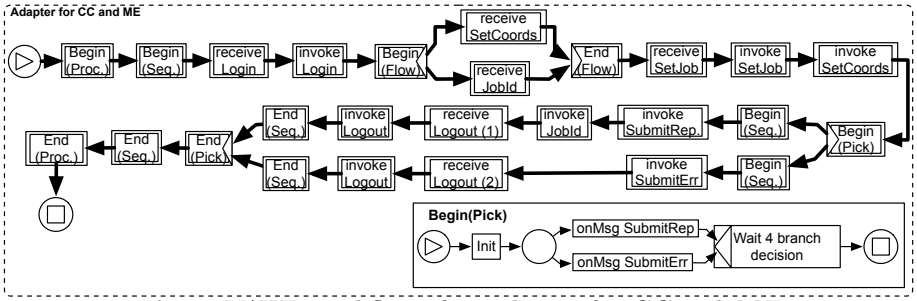

Fig. 4. YAWL workflow of an adapter for CC and ME.

conditions, then a *Switch* pattern is produced with the respective conditions as guards, and for each branch of the *Switch*, a *Sequence* pattern is generated. The algorithm continues next on each branch of the subtree with the root n. (3) In all other cases, the adaptation process aborts, as the adapter cannot be successfully constructed due to a non-deterministic (other than *pick*) behaviour. For example, if n has two unconstrained children, one *invoke Op1* and one *receive Op2*, then the adapter cannot "know" whether it should wait for a message, or whether it should send a message. The YAWL adapter one obtains for our example is presented in Figure 4.

3.3 Lock Analysis

In [4] we show how reachability graphs (or modified reachability trees) can be employed to check the lock-freedom of aggregations of YAWL workflows (e.g., a non-final node of the reachability graph/tree without outgoing links corresponds to a deadlock). Hence, through this methodology one may check whether the aggregation [4,6] of the workflows of C, A, and S locks. If all traces of the aggregate are lock-free, then A is a *full adapter* for C and S. Otherwise, if some (yet not all) of the traces of the aggregate are lock-free, then A is a *partial adapter*, as there are interaction scenarios that cannot be resolved. Finally, if the aggregate does not have lock-free traces, then we consider that the adaptation has failed. (Although space limitations do not allow us to demonstrate it, note that the adapter for CC and ME given in Figure 4 is a full adapter.)

3.4 Adapter Deployment

If the Lock Analysis phase has validated A as a full/partial adapter, then the Adapter Deployment phase generates the BPEL process of the adapter A from its YAWL workflow. The deployment process works by parsing the YAWL workflow with respect to the patterns defined in our BPEL2YAWL translator [7]. For example, the *Pick* pattern in Figure 4 leads to the generation of a BPEL *pick* with two branches guarded by *onMessage SubmitRep*, and *onMessage SubmitErr*, where each branch activity is a *sequence*. Although not explicitly represented in the figures, the YAWL patterns translating BPEL activities contain all the necessary information for the inverse, YAWL2BPEL translator (e.g., *partnerLink*, *portType*, *operation*, and *variable* attributes in the case of a *receive*, and so on).

The YAWL workflow of the adapter in Figure 4, leads to the following BPEL (adapter) process:

```
<process name="Adapter_for_CC_and_ME"><sequence>
  <receive op="Login" from CommandCentre var="loginInfo"/>
  <invoke op="Login" of MarsExplorer var="loginInfo"/>
  <flow>
    <receive op="SetCoords" from CommandCentre var="coords"/>
    <receive op="JobID" from MarsExplorer var="id"/></flow>
  <receive op="SetJob" from CommandCentre var="jobDetails"/>
  <invoke op="SetJob" of MarsExplorer var="jobDetails"/>
  <invoke op="SetCoords" of MarsExplorer var="coords"/>
  <pick>
    <onMsg op="SubmitRep" from MarsExplorer var="report"><sequence>
      <invoke op="SubmitRep" of CommandCentre var="report">
      <invoke op="JobID" of CommandCentre var="id"/>
      <receive op="Logout" from CommandCentre/>
      <invoke op="Logout" of MarsExplorer/></sequence></onMsg>
    <onMsg op="SubmitErr" from MarsExplorer var="error"><sequence>
      <invoke op="SubmitErr" of CommandCentre var="error">
      <receive op="Logout" from CommandCentre/>
      <invoke op="Logout" of MarsExplorer/>
    </sequence></onMsg></pick></sequence></process>
```

4 Concluding Remarks

In this paper we have outlined a methodology for the automated generation of (service) adapters capable of solving behavioural mismatches between BPEL processes. Its main features are: (1) It automatically synthesises a full/partial BPEL adapter (if possible) from two input BPEL processes, (2) it generates the YAWL workflow of the adapter, which can be used to check properties (e.g., lock-freedom, reachability, liveness, and so on) of the interaction with the adapted services, as well as (3) it can be straightforward integrated with the ontology-enriched service customisation [5] and service aggregation [4,6] approaches, as all use service contracts with YAWL as intermediate language to represent the service behaviour.

Web *service adaptation* is in its early stages and current approaches feature only partial solutions to the issues of adaptation. Iyer et al. [9] employ XML scripts and XSL to (manually) achieve the signature-level interoperability of SOAP services. Syu [12] describes an OWL-S based approach to deal with only three cases of adaptation of input parameters: permutation, modification, and combination. Hau et al. [8] provide a framework for semantic matchmaking and service adaptation, which deals with signature mismatches, yet not with behavioural ones. Ponnekanti and Fox [11] propose a framework for coping with structural, value, encoding, and semantic incompatibilities among services. Yet, their approach – as [8,9,12] – relies on black-box (viz., behaviour-less) views of services. A methodology for generating service adapters to solve behavioural mismatches was presented by Brogi et al. in [3], yet it assumes the availability of an adapter specification to be manually generated. Benatallah et al. [1] describe an approach for the generation of replaceability adapters based on mismatch patterns. However, their approach cannot capture complex behavioural mismatches (through pattern compositions), and the generation of the adapter code relies on the designer (e.g., the provision of the template parameters).

It is worth noting that our adaptation methodology can be successfully employed to generate *replaceability adapters*, viz., adapters that wrap Web services so that they become compliant with other services (e.g., wrapping new service versions for backwards compatibility). Given two services, S and S^*, wrapping S^* so as to behave like S with respect to clients C can be achieved by computing $SET(A)$ as the merge of $SET(S_C)$ and $SET(\overline{S^*_C})$. Furthermore, *behavioural service customisation*, viz., the generation of adapters that wrap services S^* into exposing to clients C a partial behaviour S, can be achieved again by computing $SET(A)$ as the merge of $SET(S_C)$ and $SET(\overline{S^*_C})$.

Two main lines of future work are the development of adapters capable of solving behavioural mismatches among several interacting BPEL processes, as well as enhancing the adaptation methodology to cope with ontology mismatches along the lines of [4,5].

References

1. B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani. Developing Adapters for Web Services Integration. In *Proc. of CAiSE, LNCS vol. 3520*, pages 415–429, 2005.
2. BPEL4WS Coalition. Business Process Execution Language for Web Services v1.1.
3. A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing Web Service Choreographies. In *Proc. of WS-FM'04, ENTCS 105*, pages 73–94, 2004.
4. A. Brogi and R. Popescu. Contract-based Service Aggregation. Technical Report, University of Pisa, Sep. 2006. `(http://www.di.unipi.it/~popescu/CoSA.pdf)`.
5. A. Brogi and R. Popescu. Service Adaptation through Trace Inspection. In *Proc. of SOBPI'05*, pages 44–58, 2005.
6. A. Brogi and R. Popescu. Towards Semi-automated Workflow-Based Aggregation of Web Services. In *Proc. of ICSOC'05, LNCS vol. 3826*, pages 214–227, 2005.
7. A. Brogi and R. Popescu. From BPEL Processes to YAWL Workflows. In *Proc. of WS-FM'06, LNCS vol. 4184*, pages 107–122, 2006.
8. J. Hau, W. Lee, and S. Newhouse. The ICENI Semantic Service Adaptation Framework. In UK e-Science All Hands Meeting, 2003. `(http://www.nesc.ac.uk/events/ahm2003/AHMCD/pdf/017.pdf)`.
9. A. Iyer, G. Smith, P. Roe, and J. Pobar. An Example of Web Service Adaptation to Support B2B Integration. `(http://ausweb.scu.edu.au/aw02/papers/refereed/smith2/paper.html)`.
10. M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. *Communications of the ACM*, 46(10):24–28, 2003.
11. S. R. Ponnekanti and A. Fox. Interoperability among independently evolving web services. In *Proc. of the 5th ACM Int. Conf. on Middleware*, pages 331–351, 2004.
12. J.-Y. Syu. *An Ontology-Based Approach to Automatic Adaptation of Web Services*. Department of Information Management National Taiwan University, 2004. `(http://www.im.ntu.edu.tw/IM/Theses/r92/R91725051.pdf)`.
13. W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Inf. Syst.*, 30(4):245–275, 2005.
14. F.-Y. Wang, Y. Gao, and M. Zhou. A Modified Reachability Tree Approach to Analysis of Unbounded Petri Nets. *IEEE Transactions on Systems, Man and Cybernetics – Part B*, 34(1):303–308, 2004.
15. WSDL Coalition. Web Service Description Language (WSDL) v1.1.