Analyzing BPEL processes using Petri nets

H.M.W. Verbeek, W.M.P. van der Aalst

Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
{h.m.w.verbeek,w.m.p.v.d.aalst}@tm.tue.nl

Abstract. Some years ago, BEA, IBM, Microsoft, SAP AG, and Siebel Systems teamed up and proposed the Business Process Execution Language for Web Services (BPEL or BPEL4WS) for application integration within and across organizational boundaries. By now, BPEL has become the de facto standard in this Web services composition arena. However, little effort has been dedicated so far concerning the verification of the modeled business processes. For example, there is no support to detect possible deadlocks, or to detect parts of the process that are not viable. For so-called WF-nets (workflow nets), techniques and tools exist which make it possible to detect such anomalies. Therefore, we could detect these anomalies in a BPEL process model provided that we can successfully map this model onto a WF-net. This papers describes a first attempt to map a BPEL process model onto a WF-net. Although not all BPEL constructs have been mapped yet, the results seem promising, as we are able to map typical examples from the BPEL 1.1 specification onto WF-nets.

1 Introduction

Web Services are rapidly emerging as the principal paradigm for architecting and implementing business collaborations within and across organizational boundaries. According to this paradigm, the functionalities provided by business applications are encapsulated within web services: software components described at a semantical level, which can be invoked by application programs or by other services through a stack of Internet standards including HTTP, XML, SOAP, WSDL, and UDDI [7]. Once deployed, web services provided by various organizations can be inter-connected in order to implement business collaborations, leading to composite web services.

Business collaborations require long-running interactions driven by explicit process models [4]. Accordingly, it is a natural choice to capture the logic of a composite web service using business process modeling languages tailored for web services. Many such languages have recently emerged, including WSCI [21], BPML [6], BPEL(4WS) [8], BPSS [18], and XPDL [22], with little effort being dedicated to the verification of these explicit process models. Questions such as:

- Can an instance of a Web service can always be completed successfully?
- Is completion always clean, that is, is nothing left behind upon completion?

– Are all parts of the Web service viable, that is, are there no dead parts?

are left unanswered.

Existing powerful Petri-net-based techniques could be used to verify the process models by answering the aforementioned questions, provided that it is possible to successfully map the BPEL process model onto a WF-net (a class of Petri nets). In this paper, we make a first attempt to map BPEL process models onto WF-nets. The reason for selecting BPEL is that is the most widely known and used, and has become the de facto standard for this kind of languages.

The remainder of this paper is organized as follows. Section 2 introduces, in a nutshell, both BPEL and WF-nets. Section 3 introduces the current mapping. Section 4 discusses the results of this mapping on some example BPEL process models found in [5]. Next, Section 5 shows that for an small erroneous example the behavior of the Oracle BPEL Process Manager [16] is consistent with the behavior of the mapped WF-net. Section 6 discusses related work. Finally, we conclude the paper.

2 Preliminaries

2.1 BPEL

BPEL, also known as BEPL4WS, builds on IBMs WSFL (Web Services Flow Language) and Microsofts XLANG (Web Services for Business Process Design). Accordingly, it combines the features of a block structured process language (XLANG) with those of a graph-based process language (WSFL). BPEL is intended for modeling two types of processes: executable and abstract processes. An abstract process is a business protocol specifying the message exchange behavior between different parties without revealing the internal behavior of any of them. An executable process specifies the execution order between a number of constituent activities, the partners involved, the messages exchanged between these partners, and the fault and exception handling mechanisms.

A BPEL process specification is a kind of flow-chart. Each element in the process is called an activity. An activity is either primitive or structured. The primitive activity types are:

invoke to invoke an operation of a web service described in WSDL;
receive to wait for a message from an external source;
reply to reply to an external source;
wait to remain idle for some time;
assign to copy data from one data container to another;
throw to indicate errors in the execution;
terminate to terminate the entire service instance; and
empty to do nothing.

To enable the representation of complex structures, the following structured activities are provided:

sequence for defining an execution order;
switch for conditional routing;
while for looping;
pick for race conditions based on timing or external triggers;
flow for parallel routing; and
scope for grouping activities to be treated by the same fault-handler.

Structured activities can be nested. Given a set of activities contained within the same flow, the execution order can further be controlled through (control) links, which allow the definition of dependencies between two activities: the target activity may only start when the source activity has ended. Activities can be connected through links to form directed acyclic graphs.

2.2 WF-nets

We will attempt to map a BPEL process model onto a class of Petri nets known as WF-nets (WorkFlow nets) [1–3], because for this class of nets a soundness property has been defined [1, 14] and a verification tool (Woflan [19, 20]) exists. Basically, a WF-net is a classical Petri net with three additional requirements:

1. There exists a single source place, usually denoted i.
2. There exists a single sink place, usually denoted o.
3. For every node (place or transition) there exists a path from i to o which covers the node.

A token in place i denotes a case (an instance of a Web service, in BPEL terms) that is ready to be started, whereas a token in place o denotes a case that has been completed. The third requirement enforces that every node contributes to the completion of instances.

A WF-net is called sound if and only if it satisfies the following three requirements:

1. For every possible instance, completion is possible, that is, a token can be put into place o).
2. For every completed instance, completion is clean, that is, if o contains a token then o is the only place containing a token.
3. Every node is viable, that is, every node can be activated.

Using existing Petri-net-based techniques, a tool like Woflan can decide soundness of any WF-net, and report any anomalies back to the modeler. Using this report, the modeler can then correct the model.

3 Mapping

In this section, we will map the structured activities onto Petri nets, except for the scope activity. Furthermore, our mapping will provide support for control links. We have used [5] as basis for our mappings.

Note that an activity is not necessarily mapped onto a WF-net, as most of the activities are mapped onto a Petri net that contains multiple source places and multiple sink places. However, the top-most activity, including all its contained activities, will be mapped onto a WF-net, and this WF-net can be verified for soundness.

3.1 Activities

Any BPEL activity can have a name, a join condition, a join-fault setting, a number of sources, and a number of targets. The join condition is a Boolean expression over the activity's targets. The activity can only be started if the join condition has evaluated to "true". The join-fault setting describes what should happen in case the join condition evaluates to "false". If the join condition is set to "yes", then the activity is simply skipped; if set to "no", then a join fault will be thrown which needs to be caught by some element up in the hierarchy. As long as the exception has not been caught, activities will be aborted. For his paper, we assume this setting to be "yes". The sources are the outgoing links that will be traversed if this activity has either been completed or skipped. If skipped, the corresponding link will have a negative status. If completed, then an optional transition condition determines its status. The targets are the incoming links that need to be traversed before this activity can start. A traversed target can have either a positive or a negative status.

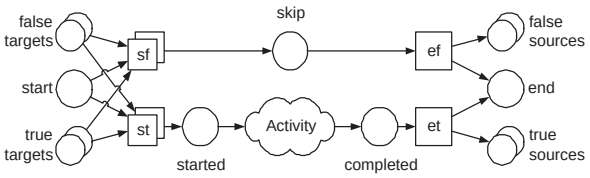

Fig. 1. Mapping for an activity.

Figure 1 shows how we map this onto a Petri net fragment. The place *start* contains a token if the hierarchical structure enables this activity. For every target, a *true* place and a *false* place are present. If the corresponding link has been traversed, either the *true* place contains a token (indicating that the link has a positive status) or the *false* place contains a token (indicating that the link has a negative status). The join condition is mapped onto a set of *st* (start true) transitions: One *st* transition for every valid combination of targets. For every invalid combination of targets, a *sf* (start false) transition is present. If the set of positive targets is valid, that is, if the join condition evaluates to "true", then the activity is started. After the activity has been completed, the corresponding *et* (end true) transition forwards control to both the next activity in the hierarchical structure (through place *end*) and to the sources. If the set of positive targets is

invalid, the corresponding *sf* transition forwards control to the *skip* place, after which the *ef* (end false) transition forwards control to both the next activity in the hierarchical structure (through place *end*) and to the sources. Please note that an activity forwards a positive control to its sources if it was completed, and that it forwards a negative control (for dead path elimination) if it was skipped.

Fig. 2. Mapping for a link-less activity.

Figure 2 shows how we map an activity without sources or targets. Because any source or target must refer to some link defined in some flow activity higher up in the hierarchy, the top activity in the hierarchy cannot have sources or targets. Thus, this top activity will have exactly one source place (*start*) and exactly one sink place (*end*).

3.2 Links

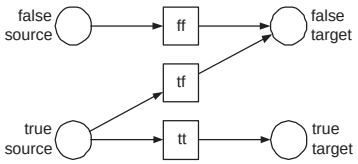

Fig. 3. Mapping for a link.

Figure 3 shows how we map a link onto a Petri net fragment. If the source activity was skipped, then place *false source* contains a token, and we simply have to pass this negative control on (dead path elimination) to the target activity through the place *false target*. If the source activity was completed, then place *true source* contains a token, and the transition condition determines whether a positive control or a negative control needs to be forwarded. If the transition condition evaluates to "true", then a positive control is forwarded through place *true target* (that is, transition *tt* fires), otherwise a negative control is forwarded (that is, transition *tf* fires). Note that transition *tf* (true false) is absent if no transition condition is given for this link.

In the remainder of this section, *we will abstract from links as much as possible*. Instead we will focus on the core of the activity, that is, the part between the

places “started” and “completed”. As a result, we will show places like “start” and “true sources” only if they are relevant for the activity at hand.

3.3 Basic Activities

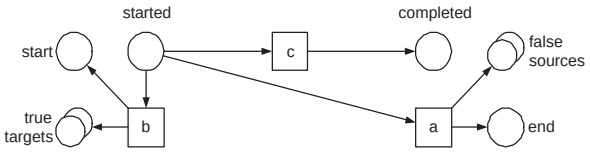

Fig. 4. Mapping for a basic activity. Note that standard connections shown in Figure 1 have been omitted.

A basic activity (like, for example, invoke, receive, and reply) does not contain any other activity. A basic activity can be completed, aborted (by not catching a thrown exception), or rolled back (using a compensation handler). Figure 4 shows how we map this onto a Petri net fragment. Transition c models the completion of the activity, transition a its abortion, and transition b its rolling back. Note that a roll back will set the status of all its incoming links to positive. Because the activity had been started, its join condition must have been evaluated to “true”, and we assume that a join condition will evaluate to “true” if all incoming links have a positive status. Therefore, it seems safe to roll back to a state where all incoming links have a positive status.

3.4 Sequence Activities

Fig. 5. Mapping for a sequence activity.

A sequence activity contains ordered set of activities, and takes care of executing these contained activities in the given order. Figure 5 shows how this can be (trivially) mapped onto a Petri net fragment.

Although it is not shown in Figure 5 (and in many figures to come), it is possible that links exists between enclosed activities. For example, there could be links from *Activity B* to *Activity C*, from *Activity A* to *Activity C*, and there could be even links to and from activities outside the sequence. Figures 1 and

3 show how such links are mapped onto Petri net fragments, which extend the Petri net fragment resulting from Figure 5.

3.5 Switch Activities

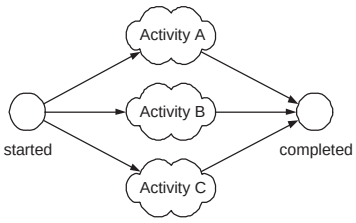

Fig. 6. Mapping for a switch activity.

A switch activity contains an ordered set of activities with associated conditions, and executes the first contained activity for which the associated condition evaluates to "true". The last activity in the ordered set of activities can be an otherwise activity, for which the associated condition always evaluates to "true". If no otherwise activity is present, an empty otherwise activity is considered present. Thus, one of the associated conditions will evaluate to "true". Figure 6 shows how we can map this onto a Petri net fragment. Note that we abstract from the actual conditions and the order in which they are evaluated. Also note that in case all conditions can evaluate to "false", we need to add an empty activity, that is, a transition that forwards control from the *started* place to the *completed* place.

Note that each of the activities in a switch may have links. As a result, parts of the model may not be executed while they contain links that may block concurrent activities. Although [5] is not completely clear about the semantics such constructs, the text "If, during the performance of structured activity S, the semantics of S dictate that activity X nested within S will not be performed as part of the behavior of S, then the status of all outgoing links from X is set to negative." (page 65, [5]) suggest that links need to be taken anyway. A similar problem occurs on the receiving side. In this paper, we will abstract from these problems and assume that both the source and target of a link are traversed or both are skipped. This problem is not limited to switch activity but also applies to other section mechanism (e.g., the pick activity).

3.6 While Activities

A while activity contains one activity and executes this activity as long as an associated condition holds (evaluates to "true"). Figure 7 shows how we can map this onto a Petri net fragment. If the associated condition evaluates to "true",

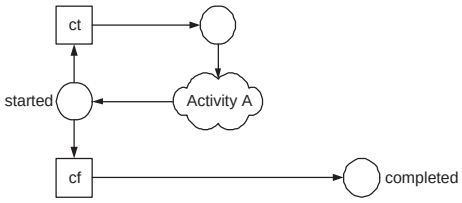

Fig. 7. Mapping for a while activity.

then transition *ct* (condition true) will forward control to the contained activity *A*. When activity *A* completes, it will put the control back. If the associated condition evaluates to "false", transition *cf* (condition false) will forward the control to the *completed* place, indicating that this while activity has now completed.

3.7 Pick Activities

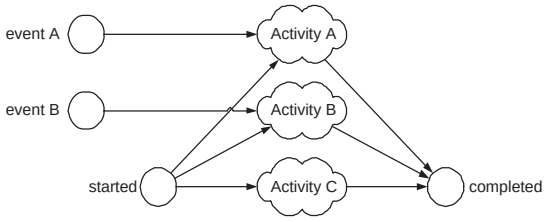

Fig. 8. Mapping for a pick activity.

A pick activity contains a (non-zero) number of *onMessage* elements and a (possibly zero) number of *onAlarm* elements. Each *onMessage* element contains a reference to some event, a possible set of correlations, and an activity. As soon as the event has occurred, the pick activity can start the corresponding activity. Each *onAlarm* element contains either a absolute time or a relative time, and an activity. As soon as the absolute time has been reached or the relative time has been passed, the pick activity can start the corresponding activity. A pick activity starts only one activity. After it has started an activity, a pick activity waits until that activity has completed. A pick activity completes as soon as its started activity completes. Figure 8 shows how we can map this onto a Petri net fragment.

3.8 Flow Activities

A flow activity contains a number of links and a (non-zero) number of activities. In the hierarchical activity structure, all contained activities are enabled in

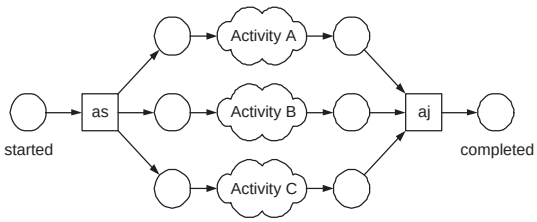

Fig. 9. Mapping for a flow activity.

parallel. However, due to the link structure, some activities might not be able to start immediately. A flow activity completes as soon as all its contained activities have completed. Figure 9 shows how we can map the hierarchical structure onto a Petri net fragment, for the link structure we refer to Figure 1 and Figure 3. In Figure 9, transition *as* (AND-split) enables all activities, and transition *aj* (AND-join) completes the flow activity.

3.9 A Reduction

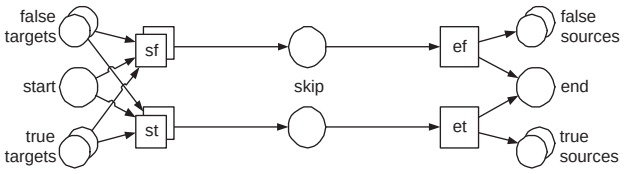

Fig. 10. Mapping for an activity that does not contain a source, target, or pick.

It is straightforward to check, that most of the activities can only lead to problems if they contain sources or targets. The only exception is the pick activity, which may block because it lacks any of the necessary events. Thus, if any activity does not contain any source, target, or pick, then that activity can cause no problems. Hence, we can use the more simple mapping as shown in Figure 10.

4 Examples

The following examples show how the structure of a BPEL process model is mapped onto a WF-net. The examples are all taken (descriptions have been copied in) from the section on structured activities of the BPEL 1.1 specification [5] and, therefore, serve as a nice illustration of the mappings presented in this paper.

4.1 Flow Graph Example

Descriptions In the following example, the activities with the names *getBuyerInformation*, *getSellerInformation*, *settleTrade*, *confirmBuyer*, and *confirmSeller* are nodes of a graph defined through the flow activity. The following links are defined:

- The link named *buyToSettle* starts at *getBuyerInformation* (specified through the corresponding source element nested in *getBuyerInformation*) and ends at *settleTrade* (specified through the corresponding target element nested in *settleTrade*).
- The link named *sellToSettle* starts at *getSellerInformation* and ends at *settleTrade*.
- The link named *toBuyConfirm* starts at *settleTrade* and ends at *confirmBuyer*.
- The link named *toSellConfirm* starts at *settleTrade* and ends at *confirmSeller*.

Based on the graph structure defined by the flow, the activities *getBuyerInformation* and *getSellerInformation* can run concurrently. The *settleTrade* activity is not performed before both of these activities are completed. After *settleTrade* completes the two activities, *confirmBuyer* and *confirmSeller* are performed concurrently again.

```
<flow suppressJoinFailure="yes">
  <links>
    <link name="buyToSettle"/>
    <link name="sellToSettle"/>
    <link name="toBuyConfirm"/>
    <link name="toSellConfirm"/>
  </links>
  <receive name="getBuyerInformation">
    <source linkName="buyToSettle"/>
   </receive>
   <receive name="getSellerInformation">
    <source linkName="sellToSettle"/>
   </receive>
   <invoke name="settleTrade"
           joinCondition="bpws:getLinkStatus('buyToSettle') and
                          bpws:getLinkStatus('sellToSettle')">
    <target linkName="getBuyerInformation"/>
    <target linkName="getSellerInformation"/>
    <source linkName="toBuyConfirm"/>
     <source linkName="toSellConfirm"/>
   </invoke>
  <reply name="confirmBuyer">
    <target linkName="toBuyConfirm"/>
  </reply>
  <reply name="confirmSeller">
    <target linkName="toSellConfirm"/>
  </reply>
</flow>
```

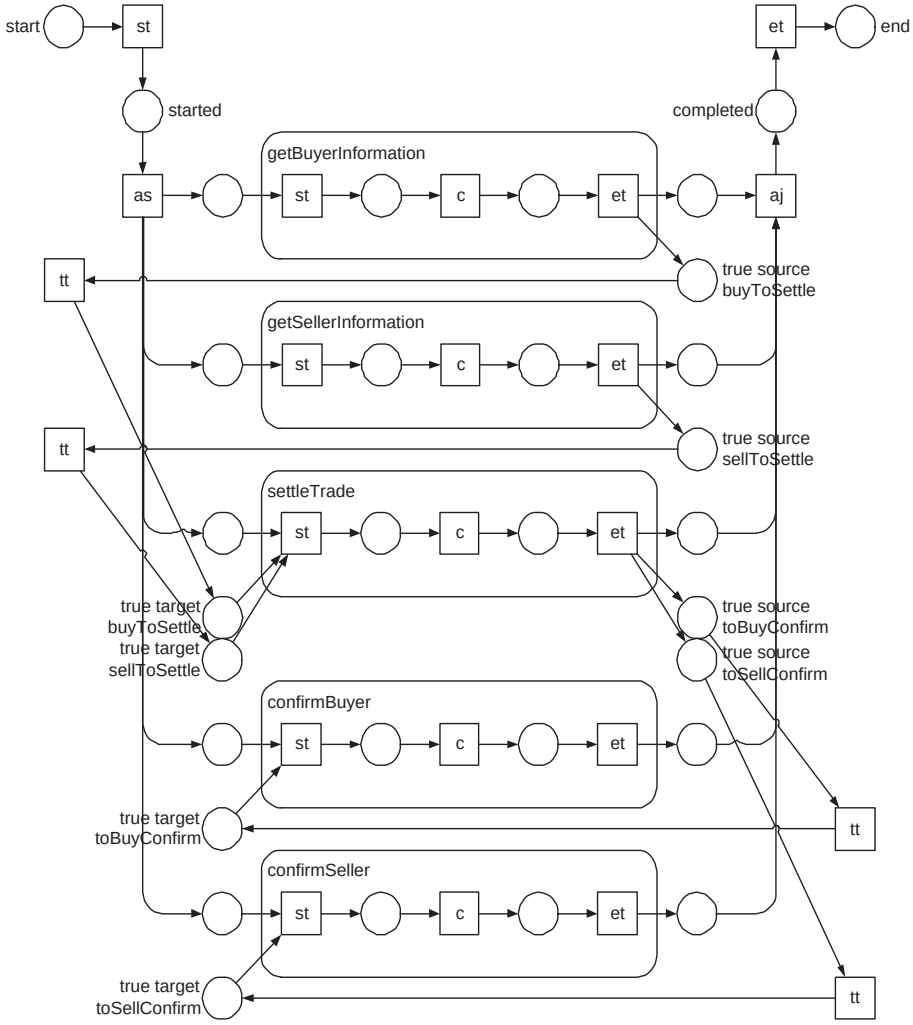

Fig. 11. Intended mapping for the first example.

Mapping Note that the aforementioned example contains references to links named *getBuyerInformation* and *getSellerInformation*, which are activities instead of links. We assume that these references should be replaced by references to *buyToSettle* and *sellToSettle*. Figure 11 shows the intended mapping. This mapping behaves as described in the BPEL 1.1 specification [5].

4.2 Links and Structured Activities

Description Links can cross the boundaries of structured activities. When this happens, care must be taken to ensure the intended behavior of the business process. The following example illustrates the behavior when links target activities within structured constructs.

The following flow is intended to perform the sequence of activities A, B, and C. Activity B has a synchronization dependency on the two activities X and Y outside of the sequence, that is, B is a target of links from X and Y. The join condition at B is missing, and therefore implicitly assumed to be the default, which is the disjunction of the status of the links targeted to B. The condition is therefore true if at least one of the incoming links has a positive status. In this case that condition reduces to the Boolean condition *P(X,B) OR P(Y,B)* based on the transition conditions on the links.

In the flow, the sequence S and the two receive activities X and Y are all concurrently enabled to start when the flow starts. Within S, after activity A is completed, B cannot start until the status of its incoming links from X and Y is determined and the implicit join condition is evaluated. When activities X and Y complete, the join condition for B is evaluated.

Suppose that the expression *P(X,B) OR P(Y,B)* evaluates to false. In this case, the standard fault *bpws:joinFailure* will be thrown, because the environmental attribute *suppressJoinFailure* is set to "no". Thus the behavior of the flow is interrupted and neither B nor C will be performed.

If, on the other hand, the environmental attribute *suppressJoinFailure* is set to "yes", then B will be skipped but C will be performed because the *bpws:joinFailure* will be suppressed by the implicit scope associated with B.

```
<flow suppressJoinFailure="no">
  <links>
    <link name="XtoB"/>
    <link name="YtoB"/>
  </links>

  <sequence name="S">
    <receive name="A" ...>
      ...
    </receive>
    <receive name="B" ...>
      <target linkName="XtoB"/>
      <target linkName="YtoB"/>
      ...
    </receive>
```

```
    <receive name="C" ...>
      ...
    </receive>
  </sequence>

  <receive name="X" ...>
    <source linkName="XtoB" transitionCondition="P(X,B)"/>
    ...
  </receive>

  <receive name="Y" ...>
    <source linkName="YtoB" transitionCondition="P(Y,B)"/>
    ...
  </receive>
</flow>
```

Finally, assume that the preceding flow is slightly rewritten by linking A, B, and C through links (with transition conditions with constant truth-value of "true") instead of putting them into a sequence. Now, B and thus C will always be performed. Because the join condition is a disjunction and the transition condition of link *AtoB* is the constant "true", the join condition will always evaluate to "true", independent from the values of *P(X,B)* and *P(Y,B)*.

```
<flow suppressJoinFailure="no">
  <links>
    <link name="AtoB"/>
    <link name="BtoC"/>
    <link name="XtoB"/>
    <link name="YtoB"/>
  </links>
  <receive name="A">
    <source linkName="AtoB"/>
   </receive>
   <receive name="B">
    <target linkName="AtoB"/>
    <target linkName="XtoB"/>
    <target linkName="YtoB"/>
    <source linkName="BtoC"/>
   </receive>
  <receive name="C">
    <target linkName="BtoC"/>
  </receive>
  <receive name="X">
    <source linkName="XtoB" transitionCondition="P(X,B)"/>
   </receive>
  <receive name="Y">
    <source linkName="YtoB" transitionCondition="P(Y,B)"/>
   </receive>
</flow>
```

Mappings Except for the *suppressJoinFailure="no"*, we can map the second example. Figure 12 shows the intended result. Note that his example is not

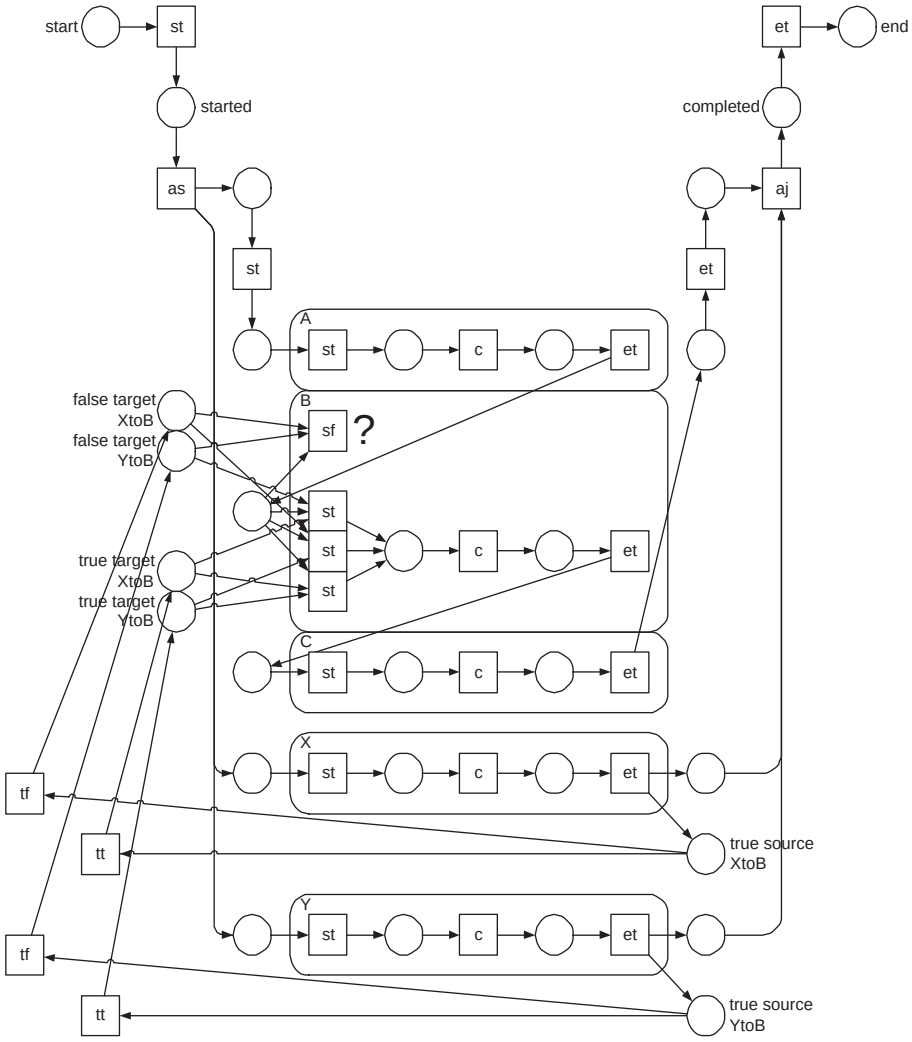

Fig. 12. Intended mapping for the second example.

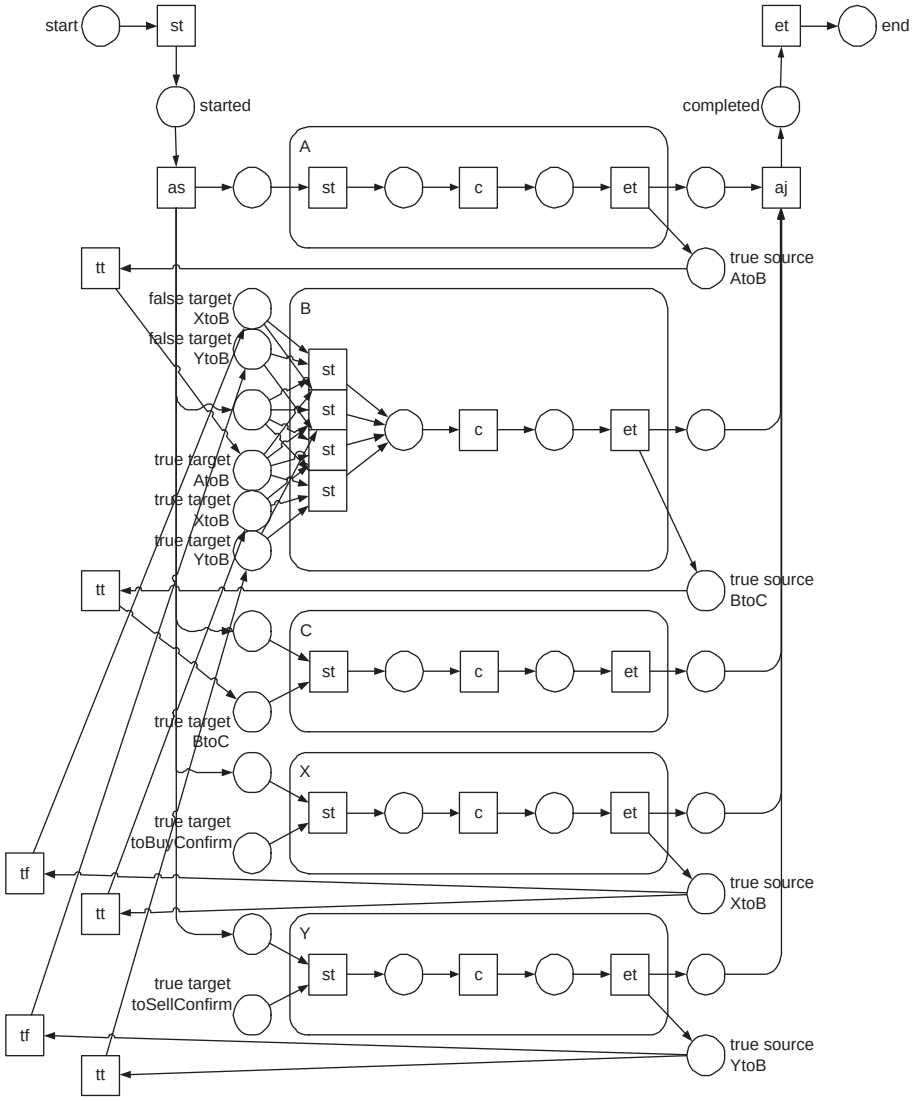

Fig. 13. Intended mapping for the third example.

(yet) mapped onto a WF-net, as the transition *sf* in activity *B* clearly is a sink transition, which is not covered by any path form *start* to *end*. This is due to the fact that the mapping does not (yet) include scopes and fault handlers. In the near future, we hope to add these constructs to the mapping.

Figure 13 shows the intended mapping of the third example. Because in this example the join condition cannot fail, we can map it successfully onto a WF-net.

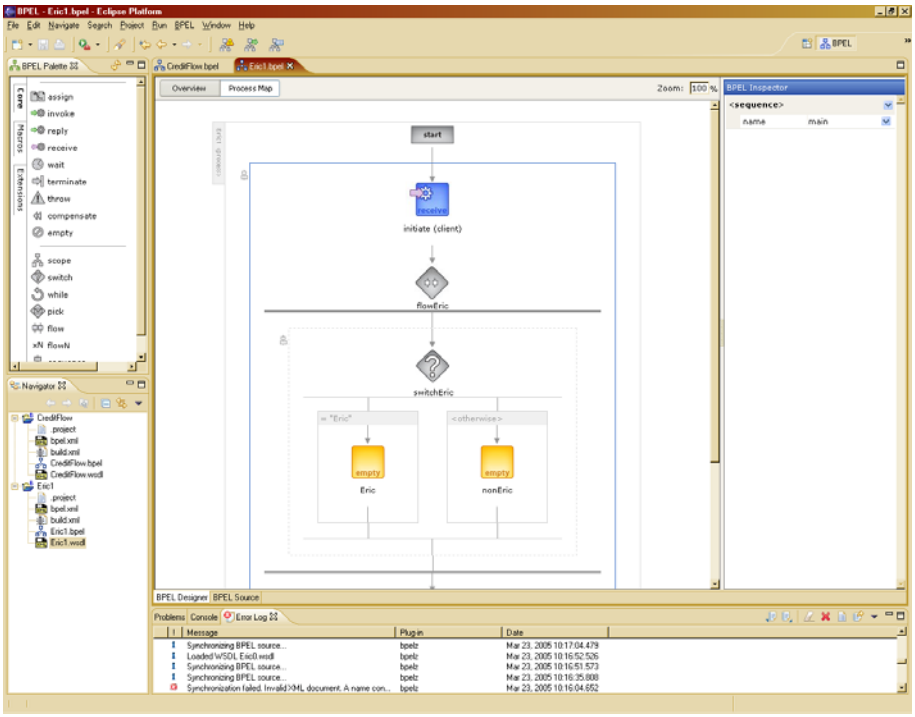

Fig. 14. Oracle BPEL Designer snapshot.

5 Oracle BPEL Process Manager

Using Oracle BPEL Process Manager (and Designer) [16], we have deployed the following, erroneous, BPEL process model.

```
<sequence name="main">
  <receive name="receiveInput" partnerLink="client"
           portType="tns:Eric1" operation="initiate"
           variable="input" createInstance="yes"/>
  <flow name="flowEric">
```

```
    <links>
      <link name="linkEric"/>
    </links>
    <sequence name="flow-sequence-1">
      <switch name="switchEric">
        <case condition="bpws:getVariableData(
              &quot;input&quot;,&quot;payload&quot;,
              &quot;/tns:Eric1Request/tns:input&quot;)=
              &quot;Eric&quot;">
          <empty name="Eric">
            <source linkName="linkEric"/>
          </empty>
        </case>
        <otherwise>
          <empty name="nonEric">
            <target linkName="linkEric"/>
          </empty>
        </otherwise>
      </switch>
    </sequence>
  </flow>
  <invoke name="callbackClient" partnerLink="client"
          portType="tns:Eric1Callback"
          operation="onResult" inputVariable="output"/>
</sequence>
```

Figure 14 shows a snapshot of this model using Oracle BPEL Designer. This process model is clearly erroneous:

- if the condition evaluates to true, the status of the link *linkEric* is set to positive, but this determined status will never be used;
- if the condition evaluates to false, the process model will deadlock because the status of the link *linkEric* can not be determined;
- the (empty) activity labeled "nonEric" is not viable (it cannot be executed).

However, no warnings or errors were issued during the deployment of this process model. This shows that it is not hard to deploy an erroneous BPEL process model, which is clearly not without dangers.

Figures 15 and 16 show the audit trails of two instances, one with "Eric" as input and one with "Wil" as input. The first instance is able to complete. However, no warning is issued for the fact that the status of link *linkEric* is never used. The second instance is not able to complete (it deadlocks) because the status of the link cannot be determined.

We have (manually) mapped the erroneous process onto a WF-net, and have verified that WF-net using Woflan [19, 20]. Figure 17 shows the diagnostic results provided by Woflan. First of all, Woflan reports (the red X in the bottom right corner) that the WF-net is not sound. Furthermore, the conditions (places) that correspond to the link "linkEric" are both improper (unbounded). Obviously, there is something wrong with this link. Even worse, there is no scenario which avoids improper places (as the number of improper scenarios is 0). Also note

Fig. 15. Audit trail "Eric".

Fig. 16. Audit trail "Wil".

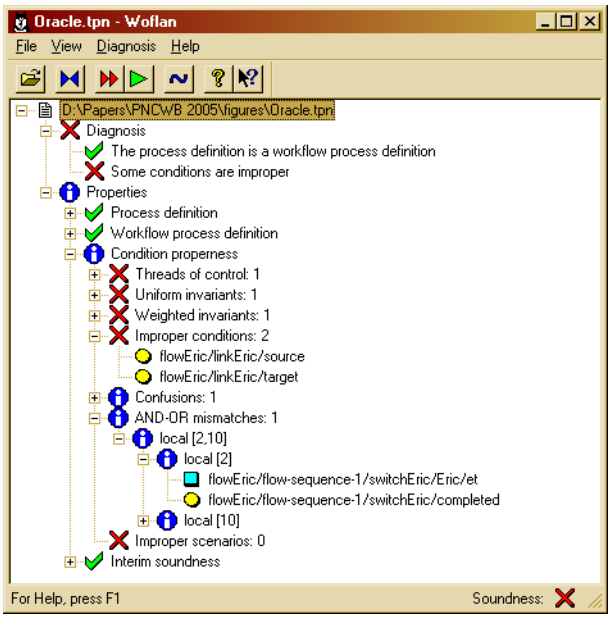

Fig. 17. Woflan report on mapped process model.

that Woflan explicitly mentions the task (transition) "flowEric/flow-sequence-1/switchEric/Eric/et" as a possible cause of the improper places (as it is part of an AND-OR mismatch (TP-handle). Clearly, our approach works and provides the modeler with valuable information towards correcting possible errors.

6 Related Work

Several attempts have been made to capture the behavior of BPEL in some formal way. Some advocate the use of finite state machines [12, 13], others process algebras [11], and yet others abstract state machines [9, 10]. Our approach is specifically tailored towards mapping a BPEL process model onto a WF-net, which enables us to check its soundness. As the aforementioned formalisms do not support the soundness property, we cannot use these mappings. To our knowledge, some researchers also advocate the use of Petri nets (cf. [15, 17]), but the corresponding papers have not been published yet.

7 Conclusion

It is not hard to model an erroneous BPEL process model using existing tools. Even worse, it is also straightforward to deploy such models. Current BPEL tools lack the possibility to verify certain basic properties any BPEL process model should have. To overcome this weakness, we propose to map BPEL process

models onto models for which strong verification techniques and tools exist: Petri nets (WF-nets, to be precise).

We have shown that large parts of BPEL process models can be mapped onto WF-net without any problems. Nevertheless, the current mapping is still not complete: We still need to add support for scopes, fault handlers, and (possibly) correlations. Nevertheless, the results sofar seem encouraging and consistent with the behavior of existing BPEL engines.

In the near future, we plan to extend the current mapping to a fully-fledged mapping. Such a fully-fledged mapping can then be used to verify any BPEL process model before it is deployed. An ongoing point of attention is the fact that we have to be able to map any errors in the resulting WF-net back to the BPEL process model, but we do not expect any difficulties in this area. As a result, the fully-fledged mapping will be able to guide the modeler of an erroneous BPEL process model towards correcting the error(s).

References

1. W.M.P. van der Aalst. Verification of workflow nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426, Toulouse, France, June 1997. Springer, Berlin, Germany.
2. W.M.P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
3. W.M.P. van der Aalst. Workflow verification: Finding control-flow errors using Petri-net-based techniques. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer, Berlin, Germany, 2000.
4. W.M.P. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18:72–76, January/February 2003.
5. BEA, IBM, Microsoft, SAP AG, and Siebel Systems. Business process execution language for web services (version 1.1). ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf (last visited in March 2005), 2003.
6. BPML.org. Business process modeling language. www.bpmi.org (last visited in April 2005), 2002.
7. F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, March 2002.
8. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services, version 1.0. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, 2002.
9. D. Fahland and W. Reisig. ASM-based semantics for BPEL: The negative control flow. In D. Beauquier, E. Brger, and A. Slissenko, editors, *Proc. 12th International Workshop on Abstract State Machines*, pages 131–151, Paris, France, March 2005.
10. R. Farahbod, U. Glässer, and M. Vajihollahi. Specification and validation of the business process execution language for web services. In W. Zimmermann and

B. Thalheim, editors, *Abstract State Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 79–94, Lutherstadt Wittenberg, Germany, May 2004. Springer, Berlin, Germany.
11. A. Ferrara. Web services: a process algebra approach. In *International Conference On Service Oriented Computing: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM Press.
12. J.A. Fisteus, L.S. Fernández, and C.D. Kloos. Formal verification of BPEL4WS business collaborations. In K. Bauknecht, M. Bichler, and B. Proll, editors, *Proceedings of the 5th International Conference on Electronic Commerce and Web Technologies (EC-Web '04)*, volume 3182 of *Lecture Notes in Computer Science*, pages 79–94, Zaragoza, Spain, August 2004. Springer, Berlin, Germany.
13. X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *International World Wide Web Conference: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM Press.
14. K.M. van Hee, N. Sidorova, and M. Voorhoeve. Soundness and separability of workflow nets in the stepwise refinement approach. In W.M.P. van der Aalst and E. Best, editors, *24th International Conference on Application and Theory of Petri Nets (ICATPN 2003)*, volume 2679 of *Lecture Notes in Computer Science*, pages 337–356, Eindhoven, The Netherlands, June 2003. Springer, Berlin, Germany.
15. A. Martens. Analyzing Web Service Based Business Processes. In M. Cerioli, editor, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *Lecture Notes in Computer Science*, pages 19–33. Springer, Berlin, Germany, 2005.
16. Oracle. Oracle BPEL process manager. http://www.oracle.com/technology/bpel (last visited in March 2005), 2005.
17. C. Stahl. Transformation von BPEL4WS in Petrinetze (In German). Master's thesis, Humboldt University, Berlin, Germany, 2004.
18. UN/CEFACT and OASIS. ebXML business process specification schema (version 1.01). www.ebxml.org/specs/ebBPSS.pdf (last visited in November 2002), 2001.
19. H.M.W. Verbeek and W.M.P. van der Aalst. Woflan 2.0: A Petri-net-based workflow diagnosis tool. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 475–484. Springer, Berlin, Germany, 2000.
20. H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnozing workflow processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.
21. W3C. Web service choreography interface (WSCI) 1.0. www.w3.org/TR/wsci (last visited in November 2002), 2002.
22. WfMC. Workflow process definition interface - XML process definition language. www.wfmc.org/standards/docs/TC-1025_10_beta_xpdl_102502.pdf (last visited in April 2005), 2002.