Fully-automatic Translation of Open Workflow Net Models into Simple Abstract BPEL Processes

Niels Lohmann[1,2] Jens Kleine[1]

[1] Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany
{nlohmann, jkleine}@informatik.hu-berlin.de

[2] Universität Rostock, Institut für Informatik, 18051 Rostock, Germany

Abstract: On the one hand, Petri net models have a successful history in the modeling, simulation, and verification of workflows and business processes. On the other hand, BPEL is the de facto standard for describing and implementing Web service-based business processes. With abstract BPEL processes, BPEL can also be used as modeling language. However, being a complicated language with many syntactic constraints, abstract BPEL processes impede a straightforward modeling.

In this paper, we introduce a fully-automatic translation of Petri net models into abstract BPEL processes which can be easily refined to executable BPEL processes. This approach combines strengths of Petri nets in modeling and verification with the ability to execute and port BPEL processes. Furthermore, it completes the Tools4BPEL framework to synthesize BPEL processes which are correct by design.

1 Introduction

The *Web Services Business Process Execution Language* (BPEL) [A$^+$07] is emerging as the de facto standard language to describe executable business processes based on Web services. It allows the description of executable Web services in a virtually platform-independent manner. Having the broad support of many companies like Microsoft or IBM, BPEL becomes more and more interesting for both industry and academia. Beside executable Web services, BPEL also allows the description of *abstract processes* — sometimes called *business protocols* — which can be used as a documentation or modeling language for Web services. In abstract BPEL processes, trade secrets (e. g., price calculations), details close to implementation (e. g., variable ranges), or yet unknown details can be hidden or left unspecified, respectively. While this may help the modeler to focus on the modeling aim, abstract processes are still subject to many syntactic constraints which might hamper a straight-forward modeling. However, Petri nets [Rei85] — in particular workflow nets [Aal98] — are an accepted formalism to model business processes. Petri nets combine a simple graphical formalism with a mathematical sound foundation. Due to the absence of syntactical constraints, Petri net models may capture the intuition of the modeler more faithfully. Furthermore, the formal foundation of Petri nets allows for verification of a variety of properties (e. g., soundness [Aal98]). To this end, there exists a broad spectrum of translation approaches from BPEL into Petri net models to apply formal verification to industrial Web services (see [BK06] for a survey).

In this paper, we try to bridge the gap between Petri nets as a formal modeling formalism and BPEL as widespread Web service description and execution language and present an approach to translate a Petri net model into a BPEL process. We thereby focus on the interaction behavior of the modeled service; that is, the Petri net to be translated models the protocol aspects of a Web service that is to be implemented in BPEL. To model the sending and receiving of messages, we extend classical workflow nets with an interface for asynchronous message passing. Such an *open workflow net* (oWFN) is then translated into an abstract BPEL process. This abstract process then implements the business protocol specified by the input oWFN model. Figure 1 depicts the proposed translation approach. The compiler oWFN2BPEL implements the translation.

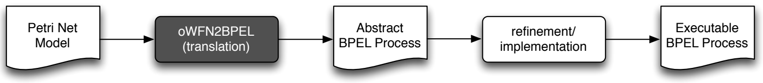


Figure 1: Proposed translation of a Petri net model into a BPEL process.

Beside models generated by translation tools or created by modelers, it is also possible to *synthesize* Petri net models. In particular, there exists an algorithm (cf. [Sch05]) to synthesize a *partner* for an oWFN. This synthesized partner is *compliant* to the original oWFN: It interacts deadlock-freely with the original oWFN. In [LMSW06], we presented with the *Tools4BPEL* framework[1] first steps towards a fully-automatic partner synthesis for a BPEL process and also for complete BPEL-based choreographies [LKLR07]. The BPEL process (or the BPEL choreography) under consideration is translated into an oWFN by the compiler BPEL2oWFN [Loh07]. The resulting model is then analyzed by the verification tool Fiona [LMSW06] which synthesizes a compliant partner oWFN. This paper completes the presented framework (depicted in Fig. 2) by translating this partner oWFN back into a BPEL process with the compiler oWFN2BPEL.

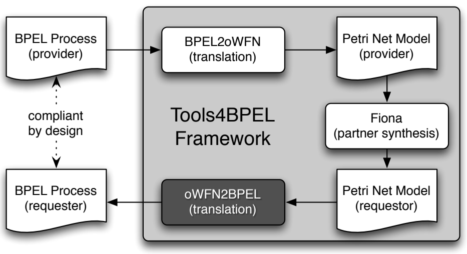


Figure 2: Integration of oWFN2BPEL into the Tools4BPEL framework.

As the generated abstract BPEL processes lacks specific implementation details which cannot be automatically derived from the oWFN model, the process has to be manually refined towards a finally executable BPEL process. To make the refinement process less

[1]See `http://www.informatik.hu-berlin.de/top/tools4bpel`.

error-prone, the generated abstract BPEL code has to be simple; that is, *human-readable*. We claim that BPEL code that avoids control links and uses block-structured elements where possible is readable and easy to understand as it allows a straight-forward decomposition of large processes. Therefore, a brute-force translation has to be avoided as it would produce very lengthy and/or rather counter-intutive code. This makes the translation a non-trivial task as BPEL is a very rich language which frequently offers more than one way to express the same aspect.

The rest of this paper is organized as follows. In Sect. 2, we give a short introduction to abstract BPEL processes and introduce our formal model. The translation from an oWFN to an abstract BPEL process is described in Sect. 3. Section 4 is devoted to validation and related work. Finally, Sect. 5 concludes the paper and gives directions for future work.

2 Background

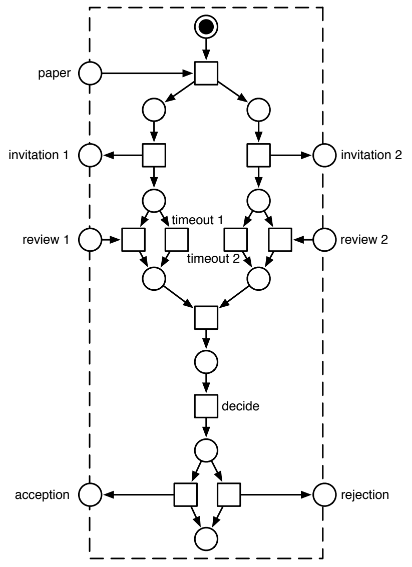
Abstract BPEL Processes The *Web Services Business Process Execution Language* (BPEL) [A$^+$07], is a language for describing the behavior of business processes based on Web services. For the specification of a business process, BPEL provides *activities* and distinguishes between *basic activities* and *structured activities*. The basic activities include ⟨`receive`⟩ to provide web service operations and ⟨`invoke`⟩ to invoke web service operations. A structured activity defines a causal order on the basic activities and can be nested in another structured activity itself. The structured activities include ⟨`sequence`⟩ to process activities sequentially, ⟨`if`⟩ to process activities conditionally, ⟨`while`⟩ to repetitively execute activities, ⟨`pick`⟩ to process events selectively, and ⟨`flow`⟩ to process activities in parallel. Activities embedded to a ⟨`flow`⟩ activity can additionally be ordered by the usage of *control links*.

The specification of abstract BPEL processes further allows to introduce placeholders for details that are either not known yet or are subject of trade secrets and are thus not to be exposed. Such unspecified details are called *opaque*. For example, a condition of an ⟨`if`⟩ activity can be replaced by an opaque condition. In addition, whole activities can be left unspecified. This is done by using the placeholder activity ⟨`opaqueActivity`⟩. To implement an abstract BPEL process, the placeholders have to be replaced by actual conditions, values, and activities, respectively. Furthermore, there are a number of activities to manipulate data, handle faults, or organize compensation. As these activities are very close to the implementation and very "BPEL-specific" and this paper focuses on the interaction behavior of a BPEL process only, we refrain from a presentation. The translation approach presented in this paper will, according to the structure of the input model, decide whether a Petri net node is translated into a interacting activity, an ordering structured activity, or a non-interacting opaque activity.

Open Workflow Nets *Open workflow nets* (oWFNs) [MRS05] are a special class of Petri nets. They generalize classical workflow nets [Aal98] by introducing an interface for asynchronous message passing. Intuitively, an oWFN is a Petri net together with (i) an *interface*, consisting of input and output places, (ii) an initial marking, and (iii) a set of distinguished *final markings*. Final markings represent desired final states of the net and help to distinguish desired final states from unwanted deadlocks. Compared to workflow nets, oWFNs underly no syntactic constraints. In particular, we allow multiple initially marked places as well as multiple final markings. The interplay of two oWFNs N and M is represented by their *composition*, denoted by $N \oplus M$. Thereby, we demand that the nets only share input and output places such that for some input places of N exist corresponding output places of M, and vice versa. The oWFN $N \oplus M$ can then be constructed by merging joint places and merging the initial and final markings.

oWFNs provide a simple but formal foundation to model services and their interaction. They allow — like common Petri nets — for diverse analysis methods of computer-aided verification. The explicit modeling of the interface further allows to analyze the interaction behavior of a service [LMSW06]. An important property of an oWFN is whether it is possible to interact deadlock-freely with it. An oWFN N is called *controllable*, if there exists an oWFN M such that $N \oplus M$ is deadlock free (i. e., N and M are compliant). Like the soundness property for workflow nets, controllability [Sch05] can be regarded as a minimal correctness criterion for interacting services.

First Example: A Paper Reviewing Process As an example for the translation of an oWFN into an abstract BPEL process, consider the oWFN depicted in Fig. 3(a). Upon receiving a paper (cf. the input place paper on the dashed frame), two reviewers are invited. Then, for each reviewer either a review is received or a timeout occurs. Based on the received reviews, a decision is made and communicated to the author.



(a) oWFN modeling a reviewing process

```
<process>
  <sequence>
    <receive operation="paper" />
    <flow>
      <sequence>
        <invoke operation="invitation 1" />
        <pick>
          <onMessage operation="review 1" />
            <empty />
          </onMessage>
          <onAlarm>
            <opqaueActivity name="timeout 1" />
          </onAlarm>
        </pick>
      </sequence>
      <sequence>
        <invoke operation="invitation 2" />
        <pick>
          <onMessage operation="review 2" />
            <empty />
          </onMessage>
          <onAlarm>
            <opaqueActivity name="timeout 2" />
          </onAlarm>
        </pick>
      </sequence>
    </flow>
    <opaqueActivity name="decide" />
    <if>
      <condition opaque="yes" />
      <invoke operation="acception" />
      <else>
        <invoke operation="rejection" />
      </else>
    </if>
  </sequence>
</process>
```

(b) generated abstract BPEL code

Figure 3: An open workflow net (a) and the abstract BPEL code generated by oWFN2BPEL (b).

The oWFN was derived from a workflow net taken from [LA06]. While the original workflow net was annotated with details on the decisions (e. g., expressions or the activity type), our approach solely bases on the structure of the net. Please note that, for instance, "decide" or "timeout 1" are transition names and not labels. These names are only used to name generated activities rather than to choose their type. The output of the compiler oWFN2BPEL implementing the translation rule presented in the next section is depicted in Fig. 3(b).

3 Translation Approach

Our translation approach automatically transforms an oWFN step by step into a single annotated node. From this annotation, BPEL code is generated. As opposed to other translations (cf. [LA06]), we do not annotate the nodes of the net with final BPEL code. During our transformation, the code annotations will reflect a number of BPEL activities that we deem suitable for representing the reduced parts of the net. This enables us to modify, replace, or even remove the code annotations during later reduction steps allowing a more compact and tidy code. While most rules aim at reducing the number of nodes in the intermediate net, some rules will actually *increase* the size of the intermediate net. However, these rules aim at simplifying the structure of the net to enable subsequent transformation rules.

Throughout this paper, we require for an oWFN to be translated that its *inner* (the net without the interface places and their adjacent arcs) is (i) deadlock free, and (ii) it has no dead transitions. These restrictions rule out an incorrect translation result as BPEL processes are (i) deadlock free and (ii) should not contain unreachable code, and are similar to the soundness property [Aal98] of classical workflow nets. For cyclic nets, we further (iii) require that each cycle neither deadlocks nor livelocks. The restrictions can be easily checked using a Petri net model checker such as LoLA [Sch00].

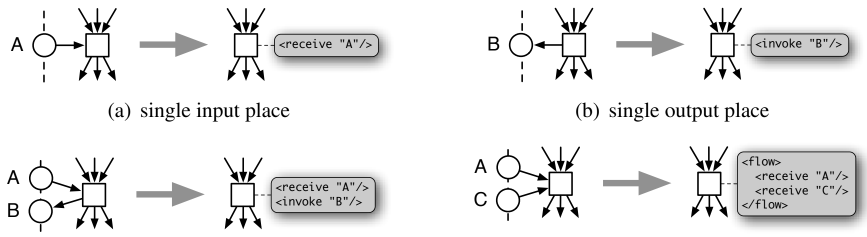
The overall translation consists of 17 rules and 8 additional adjustment rules to overwork and simplify the generated code. Admittedly, we can only present a few aspects of the translation and refer to [Kle07] for the complete set of rules. The first rules we present in the next subsection aim at automatically transforming any safe acyclic oWFN into a BPEL process. These rules range from simple substitutions to more complex replacements. To cover non-save and cyclic nets, additional rules will be introduced later in Sect. 3.2.

3.1 Translation of Simple Models

For the rules presented in this subsection, we require the oWFN to be translated to be acyclic (i. e., the structureof the net contains no cycles) and safe (i. e., there is no reachable marking with more than one token on a place). For example, the oWFN modeling the paper reviewing process (cf. Fig. 3(a)) fulfills these requirements.

Interface places The interface of an oWFN provides us information about inbound and outbound message exchange. The first transformation rule, the INTERFACE rule, transforms the receiving of a message into a ⟨`receive`⟩ activity and the sending of a message into an ⟨`invoke`⟩ activity. Subsequent adjustment rules may later change the basic ⟨`invoke`⟩ activities to structured ⟨`pick`⟩ activities if necessary.

The INTERFACE rule adds code annotations to every transition that has an adjacent interface place. Having all code annotations added, the interface places are removed from the intermediate net. If a transition is connected to multiple interface places, we embed multiple ⟨`receive`⟩ and ⟨`invoke`⟩ activities annotated to the same transition into a ⟨`flow`⟩ activity to simulate the simultaneous exchange of messages. To preserve the behavior of the oWFN, all ⟨`receive`⟩ activities have to be executed *before* the sending ⟨`invoke`⟩ activities. Figure 4 shows the code annotation generated from sending/receiving transitions.[2]

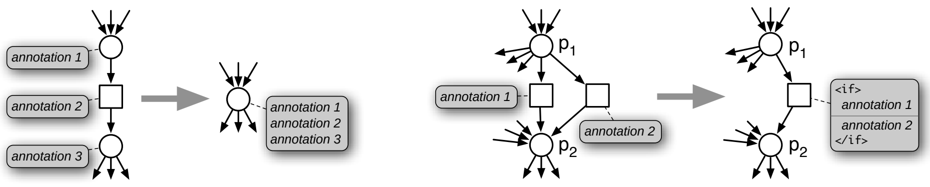


(a) single input place (b) single output place

(c) input and output place (d) multiple input places

Figure 4: Examples for the INTERFACE rule translating message interchange into ⟨`receive`⟩ and ⟨`invoke`⟩ activities.

Sequences Sequences in the intermediate net are collapsed to a single node, but are not explicitly annotated. Instead, we add BPEL ⟨`sequence`⟩ activities only *after* the net is completely translated using an adjustment rule. This allows us to grasp the largest possible sequence of activities and add them only where needed. An example for the SEQUENCE rule is depicted in Fig. 5(a).



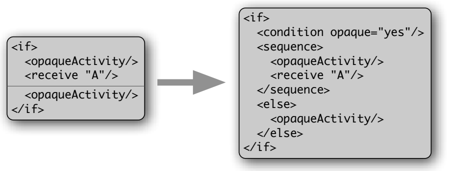
(a) SEQUENCE rule (b) CONDITIONAL rule

Figure 5: Example for the SEQUENCE rule (a) and the CONDITIONAL rule (b).

[2]Note that in Fig. 4(c), the activities in the annotation will be embedded to a ⟨`sequence`⟩ activity later, cf. SEQUENCE rule.

Conditional Behavior The CONDITIONAL rule identifies a place (p_1 in the example shown in Fig. 5(b)) that is the only input place of more than one transition as the start of conditional behavior. All these transitions must have the same single output place (p_2). If these conditions are met, the transitions are replaced by a single transition. This transition is annotated with an ⟨`if`⟩ activity embedding the code annotations of the removed transitions as branches.

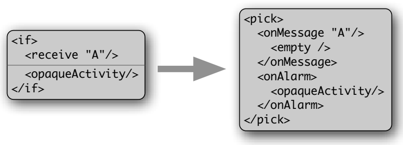
The transformation permits the repeated application with the same start and end place which leads to multiple nested ⟨`if`⟩ activities. A later adjustment rule joins these ⟨`if`⟩ activities: All branches of the nested activities are added as branches to the top activity. While this seems like an unnecessary step, it allows other transformation rules to collapse components between p_1 and p_2 even after the CONDITIONAL rule was first executed.



```
<if>
  <opaqueActivity/>
  <receive "A"/>
  <opaqueActivity/>
</if>
```

```
<if>
  <condition opaque="yes"/>
  <sequence>
    <opaqueActivity/>
    <receive "A"/>
  </sequence>
  <else>
    <opaqueActivity/>
  </else>
</if>
```

(a) adjustment: refinement of ⟨`if`⟩ activity



```
<if>
  <receive "A"/>
  <opaqueActivity/>
</if>
```

```
<pick>
  <onMessage "A"/>
    <empty />
  </onMessage>
  <onAlarm>
    <opaqueActivity/>
  </onAlarm>
</pick>
```

(b) adjustment: changing ⟨`if`⟩ to ⟨`pick`⟩ activity

Figure 6: Example for the adjustment rule to refine (a) or change (b) an ⟨`if`⟩ activity.

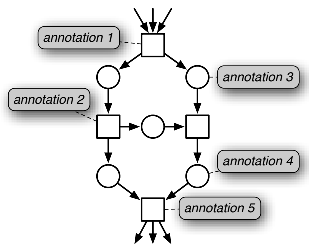
Besides the ⟨`if`⟩ activity to model internal choices, BPEL also provides the ⟨`pick`⟩ activity to model external (i. e., message-triggered) choices. After translating all choices into ⟨`if`⟩ activities, a later adjustment rule (cf. Fig. 6) will change them to ⟨`pick`⟩ activities if necessary; that is, as soon as a message-receiving activity is involved. Note that sending or quiet activities are embedded to ⟨`onAlarm`⟩ branches.[3] If syntactically necessary (e. g., to avoid vacant branches), ⟨`empty`⟩ activities are inserted. Furthermore, the ⟨`if`⟩'s condition is modeled opaquely as it cannot be derived from the structure of the oWFN.

Concurrent Behavior After interface places have been replaced by code annotations and conditional behavior has been treated, the remaining intermediate net (we required acyclic and safe nets) can be straightforwardly mapped to a ⟨`flow`⟩ activity with control links to introduce the required execution order. However, excessive or unnecessary use of control links can result in unreadable and unmaintainable code. To this end, we already introduced the SEQUENCE rule to avoid control links that express simple sequences of activities.

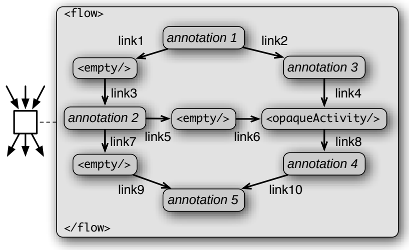
Instead of translating the whole intermediate net into a ⟨`flow`⟩ activity, the CONCURRENT rule tries to identify the smallest suitable subnet in the intermediate net. A subnet is suitable if (i) it is acyclic, because a cyclic control link structure would lead to a deadlocking situation and is prohibited by the BPEL specification [A$^+$07]; (ii) there is a distinct initial node, which is the only node with arcs entering the subnet; (iii) there is a distinct final node whose arcs are only leaving the subnet. The CONCURRENT rule translates the small-

[3]A subsequent manual refinement of the generated abstract process has to fill in necessary timeouts.

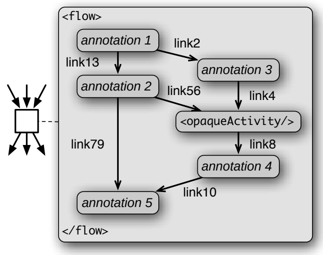
est suitable subnet into a ⟨flow⟩ activity in several steps: Firstly, each node that is not annotated with BPEL code yet is annotated with a BPEL activity to allow the embedding of source and/or target links. To this end, each transition with empty code annotation is annotated with an ⟨opaqueActivity⟩ and each un-annotated place is annotated with an ⟨empty⟩ activity. Then, for each arc $[x, y]$ in the subnet, a control link is generated and annotated as source link of node x and as target link of node y. If the annotation of a transition has more than one incoming control link, a conjunction of these control links is used as join condition. If a place's annotation has more than once incoming control link, a disjunction is used. Finally, the subnet is collapsed to a single node which is annotated with a ⟨flow⟩ activity. The annotations of the former nodes of the subnet are added as branches to this ⟨flow⟩ activity.



(a) suitable subnet



(b) subnet after applying the CONCURRENT rule



(c) subnet after code (i. e., control link) reduction

```
<flow>
  <links>
    <link name="link13" />
    ...
  </links>
  ...
  <opaqueActivity>
    <targets>
      <joinCondition> $link4 and $link56 </joinCondition>
      <target linkName="link4" />
      <target linkName="link56" />
    </targets>
    <sources>
      <source linkName="link8" />
    </sources>
  </opaqueActivity>
  ...
</flow>
```

(d) snippet of the final abstract BPEL code

Figure 7: Example for the CONCURRENT rule to translate concurrent behavior.

Figure 7 depicts the transformation of a small example net. After the application of the CONCURRENT rule, a code reduction rule is applied to remove redundant ⟨empty⟩ activities by aggregating inbound and leaving control links (cf. Fig. 7(c)).

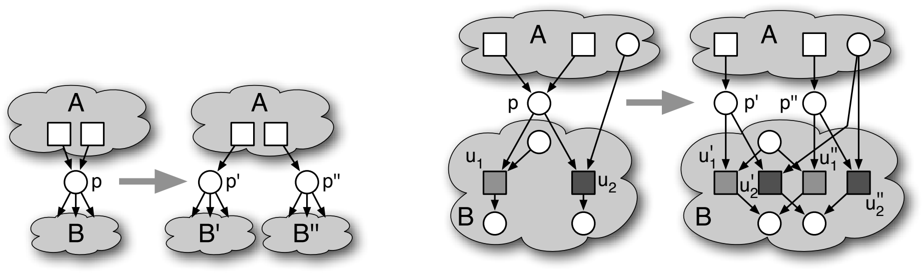
Interim Conclusion With the rules presented so far (INTERFACE, SEQUENCE, CONDITIONAL, and CONCURRENT), we are able to transform any safe acyclic oWFN such as the example depicted in Fig. 3(a). As discussed earlier, the bare translation of those nets into ⟨flow⟩ activities with control links is trivial. With the presented rules, however, we try to avoid as many control links as possible in order to generate compact and readable

BPEL code. As an example, consider the generated abstract BPEL process in Fig. 3(b).[4] The code clearly mirrors the structure of the original oWFN. In fact, if the BPEL process is translated back into an oWFN with the compiler BPEL2oWFN [Loh07], the resulting net is the same.

3.2 Translation of General Models

With the rules presented so far, only safe and acyclic oWFNs can be translated. In this subsection, we further present rules to translate cyclic and non-safe oWFNs as long as they fulfill the restrictions stated in the beginning of this section. The following rules transform bounded cyclic nets to safe acyclic nets, and thus extend the rules of Sect. 3.1.

Multiple Execution of Activities In many complex models, places can get marked multiple times while not being part of a cycle. A BPEL activity, however, can only be executed once unless it is embedded to a repeating structured activity. If it is embedded in such an activity (e. g, ⟨`while`⟩), all the repetitions have to take place *before* subsequent activities are executed. In most cases, this is not the intended behavior in the models that we translate. If we have a fixed number of times a place is marked, we insert copies of this place into the intermediate net. This way, all BPEL activities associated with this place will be executed as often as necessary.



(a) no arcs from subnet A to subnet B (b) general case: arcs from subnet A to subnet B

Figure 8: Examples for the MULTIPLE rule to translate non-safe nets.

The MULTIPLE rule (cf. Fig. 8) can be used if there is a place p with more than one incoming arc. It distinguishes two cases: (a) no further arcs point to subnet A except those originating at place p, and (b) arcs to transitions within the subnet following p are allowed. In the first case, a copy of p and the subnet B is inserted into the intermediate net for every transition in the preset of p (cf. Fig. 8(a)). In the second case, a copy of p and every transition in its postset is inserted. Arcs pointing to these transitions are also copied. Figure 8(b) illustrates this.

[4]The adding of the ⟨`process`⟩ tag and other adjustments were made by applying adjustment rules.

Cycles The CYCLE rule translates cycles in the intermediate net. It is the most complex rule we present in this paper. The rule strives to replace a cycle by a single place annotated with a ⟨`while`⟩ activity. An example for the CYCLE rule is depicted in Fig. 9. In this example, the cycle has one *entry place* (p), and can be exited with two transitions (e_1 and e_2). We firstly add a linearization of the cycle's annotations to the annotation of the entry place (p). Then, we remove the cycle and directly connect all exit transitions to the entry place. Finally, we correct each exit transition's annotation by adding the annotations of all nodes between the entry place and the exit transition. With a second rule (not presented in this paper), we similarly treat cycles with multiple exit transitions *and* multiple entry places.

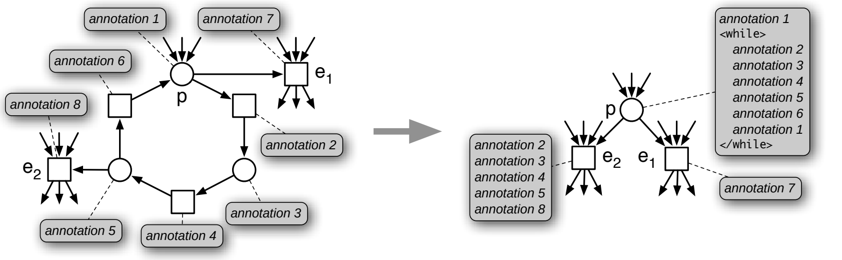
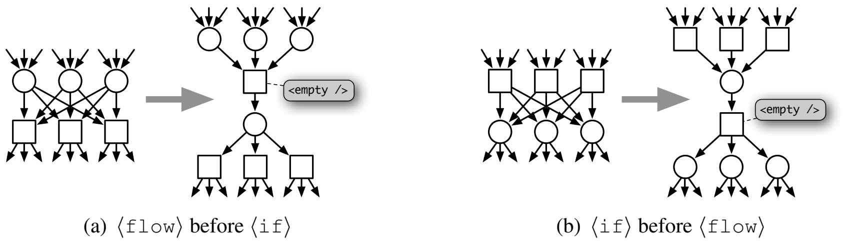


Figure 9: Example of the CYCLE rule translating a cycle with a single entry place (p) and several exit transitions e_1 and e_2 into a ⟨`while`⟩ activity.

Structural Reorganization We use further transformation rules to reorganize (i. e., unfold or restructure) parts of the intermediate net. The reorganized net — though usually larger — has the same behavior, but a simpler structure. This simpler structure might allow the application of a transformation rule which was not applicable prior to the reorganization.



(a) ⟨`flow`⟩ before ⟨`if`⟩ (b) ⟨`if`⟩ before ⟨`flow`⟩

Figure 10: Examples of structural reorganization rules.

Two examples for reorganization rules are depicted in Fig. 10. These rules help to separate interweaved sequential ⟨`flow`⟩ and ⟨`if`⟩ activities by inserting an ⟨`empty`⟩ activity in between. The resulting intermediate net allows the application of the CONCURRENT and CONDITIONAL rules.

3.3 Code Adjustment Rules

After the transformation rules reduced the intermediate net to a single node, adjustment rules are applied to the code annotation of this node. We already mentioned an adjustment rule to refine an ⟨`if`⟩ activity or change it to a ⟨`pick`⟩ activity, and another a rule to simplify the control link structure. Further adjustment rules that are not shown in the paper remove (syntactically) unnecessary ⟨`empty`⟩ activities, add and improve ⟨`sequence`⟩ activities, merge nested ⟨`flow`⟩ activities, and enclose the remaining annotation into a ⟨`process`⟩ activity as root element of the generated BPEL code.

As described earlier, this two-tier approach of firstly generating code annotations which are later possibly changed by adjustment rules, allows for a more flexible translation. As the application of the transformation is local (i. e., the rules are only applied to a subnet), the code annotations only represent a local optimum. Only with a subsequent adjustment, these local optima can be further optimized. For instance, after collecting the largest possible ⟨`sequence`⟩ activities during the translation, these locally best translations can be further improved by combining ⟨`sequence`⟩ activities.

3.4 Algorithm and Implementation in oWFN2BPEL

With the presented rules, we are able to translate any oWFN fulfilling the restrictions described earlier this section into an abstract BPEL process. We thereby proceed as follows:

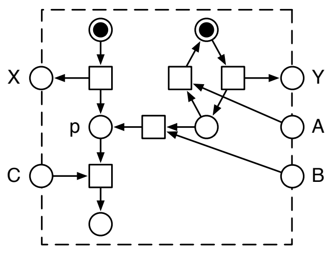
1. The INTERFACE rule is once applied to annotate interacting transitions with ⟨`receive`⟩ and ⟨`invoke`⟩ activities, respectively.
2. Whenever applicable, parts of the intermediate net are translated using rules SEQUENCE, CONDITIONAL, CONCURRENT, MULTIPLE, and CYCLE. In addition, the structural reorganization rules are applied whenever possible.
3. Finally, after the intermediate net has been collapsed to a single node, the adjustment rules are applied to optimize and complete the generated code annotation.

In this paper, we only presented a few examples for each rule. In general, there are many different variants for each rule, each tailored for a specific context. These different "subrules" are usually ordered: for the best translation result, the most specific rule is applied rather than a more general version. The interested reader is referred to [Kle07] for the complete set of rules and the detailed translation algorithm. Each rule either reduces the net to be translated, or simplifies its structure to allow further rules to be applied. Thus, the algorithm will terminate for any net that fulfills the restrictions; that is, reduces it to a single annotated node. If the net, however, violate the restrictions, the translation will stop leaving a partly annotated net for further diagnosis.

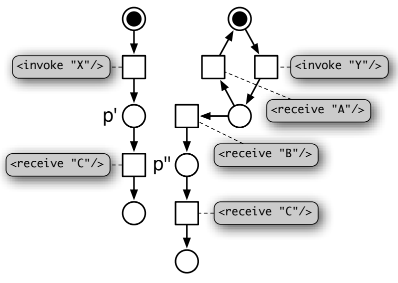
All translation, reorganization, and adjustment rules were implemented in the compiler oWFN2BPEL[5]. It takes an oWFN in Fiona [LMSW06] or PNML file format as input and translates it into an abstract BPEL processes as described in this paper. oWFN2BPEL completes the Tools4BPEL framework (cf. Fig. 2) which now offers a complete tool chain having a BPEL process as input and output. This allows use to apply formal verification techniques in the context of BPEL while "hiding" the formal model from the user of the tools.

3.5 Second Translation Example

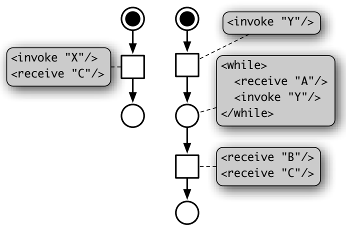
To illustrate the interplay of the presented rules, consider a second example oWFN, depicted in Fig. 11(a). This oWFN has a cycle as well as a non-safe inner: place p can get marked twice. Therefore, the two "threads" are firstly separated, yielding two independent nets (cf. Fig. 11(b)). Then, the cycle of the left net is collapsed into a single node annotated with a ⟨`while`⟩ activity (cf. Fig. 11(c)). Finally, the sequences are further collapsed and embedded into a ⟨`flow`⟩ activity. Figure 11(d) shows the abstract BPEL code after applying the adjustment rules; that is, after adding a ⟨`process`⟩ root element, etc.



(a) oWFN model



(b) after applying the INTERFACE and MULTIPLE rules



(c) after applying the SEQUENCE and CYCLE rules

```
<process>
  <flow>
    <sequence>
      <invoke operation="X" />
      <receive operation="C" />
    </sequence>
    <sequence>
      <invoke operation="Y" />
      <while> <condition opaque="yes" />
        <sequence>
          <receive operation="A" />
          <invoke operation="Y" />
        </sequence>
      </while>
      <receive operation="B" />
      <receive operation="C" />
    </sequence>
  <flow>
</process>
```

(d) generated abstract BPEL code

Figure 11: A second example oWFN (cyclic, non-safe) and its translation to BPEL.

[5]Available at `http://www.informatik.hu-berlin.de/top/tools4bpel/owfn2bpel`.

4 Validation and Related Work

4.1 Validation

In contrast to oWFNs, BPEL is only described informally and has no formal semantics. Therefore, it is impossible to *prove* the correctness of the translation rules presented. Instead, we *validated* the translation approach using several case studies. On the one hand, we used the compiler BPEL2oWFN [Loh07] to translate real-world BPEL processes into oWFNs and re-translated these nets using oWFN2BPEL. In most cases, the generated BPEL process had the same structure as the original BPEL process. In fact, the processes generated by oWFN2BPEL were sometimes "optimized" with respect to the number of used control links. On the other hand, we translated oWFNs using oWFN2BPEL into BPEL processes and these processes back to oWFNs using BPEL2oWFN. Then, we compared the interaction behavior of the original oWFN and the twice translated oWFN by comparing their *operating guidelines* [LMW07] (a characterization of all deadlock-freely interacting partners). In all tested cases, the operating guidelines of the respective pairs of oWFN were equal, showing that the interaction behavior of both nets is equivalent.

4.2 Related Work

There exist two areas related to the translation approach presented in this paper: (i) using BPEL as modeling language, and (ii) modeling another formalisms (e. g., BPMN, UML activity diagrams) and translating these models into BPEL.

Several vendors offer tools to directly "model" or "design" BPEL processes (e. g., Oracle BPEL Process Manager or IBM WebSphere Integration Developer). They introduce a more or less standardized graphical representation of BPEL's activities which can be used to graphically model a BPEL process. These graphical modeling notations are, however, very close to the respective BPEL constructs and underly the same syntactic constraints. Therefore, the user of such BPEL design tools is — knowingly or unknowingly — forced to adapt the model to the BPEL language.[6] We therefore suggest our translation approach to *generate* BPEL code that can be read by such graphical BPEL design tools to *refine* the code towards executable BPEL processes.

To overcome BPEL's shortcomings in the area of modeling, many approaches exists to translate formalisms such as BPMN, UML activity diagrams [ODBH06], or workflow nets [LA06] into BPEL processes. These formalisms are very similar to oWFNs, and therefore we can compare the approach presented in this paper with these works. Table 1 summarizes the comparison with related approaches.

[6]For instance, the choice of whether to use a ⟨`sequence`⟩ or a series of control links to express sequential execution has to be explicitly made by the modeler.

Table 1: Comparison of related approaches to translate business process models to BPEL.

	this approach	Lassen and van der Aalst [LA06]	Ouyang et al. [ODBH06]
A	(low-level) oWFN	(annotated) workflow net	BPMN/UML-AD
B	weak termination	WF-structure, soundness, safeness	none
C	structure	structure and annotation	structure and annotation
D	provisonal, later adjusted	definitive	definitive
E	full-automatic	semi-automatic (user-feedback)	fully-automatic
F	17 rules + adjustment	4 rules + user library (arbitrary)	7 rules + event handlers

A: model – B: constraints – C: activity choice – D: code generation – E: translation – F: transformation rules

Lassen and van der Aaalst The translation approach of [LA06] also uses transformation rules to translate a workflow net into a BPEL process. There are, however, a number of differences:

Firstly, the input model is an *annotated* workflow net. This workflow net does not explicitly model the interaction of the workflow by the help of an interface (as we do). Instead, each transition is annotated with the type of the BPEL activity to be translated to. More importantly, the translation is *semi-automatic*: It only consists of four translation rules, and for each subnet that cannot be translated, the user is prompted to enter suitable BPEL code. Furthermore, cycles are not supported in a systematically: Instead of having general rules to translate arbitrary cycles, only simplest constructs can be translated automatically. For all other cycles, the user has to provide BPEL code. While prompting the user might help to generate very compact and (naturally) intuitive BPEL code, this semi-automatic approach cannot be used in the Tools4BPEL framework (cf. Fig. 2) where the user should be unaware of the formal model and the Petri net models are generated and therefore potentially less understandable.

Finally, the code annotated by the implemented rules (built-in or user-defined) to the nodes is final; that is, once annotated it is not changed by subsequent rules. For example, the tool WorkflowNet2BPEL4WS translates the paper reviewing process (cf. Fig. 3) into a BPEL process with six control links, because it only groups sequences once.

Ouyang et al. The work of [ODBH06] present a translation of *standard process models* (BPMN and UML-AD) into BPEL. The translation is very general and hardly restricts the input model. This rather simple approach, however, differs greatly from our approach:

Several transformation rules are used to identify structures like ⟨`sequence`⟩ or ⟨`flow`⟩ activities. The rest of the model is then treated like a *state machine* and translated into event handlers. These event handlers mimic the pre- and post conditions of each activity by sending messages to themselves that encode the next state. While allowing any construct to be translated, this design decision makes the generated code very lengthy and — due to the mixture of control and message flow — counter-intuitive and hard to maintain. Therefore, this approach is not suitable for the Tools4BPEL framework. Furthermore, an event handler is a very complex construct as it implicitly creates a scope instance on each incoming call and saves a complete scope snapshot for compensation handling. Finally, such processes are not guaranteed to correctly run in any environment as those messages sent from the service to it self might also be received (or sent) by (from) services in the environment. Again, code annotations are final and not changed by subsequent steps.

5 Conclusion

We presented a fully-automatic approach translating an oWFN model into an abstract BPEL process. This approach helps to combine the theoretical foundation and verification techniques of Petri net-based formalisms with the platform independence and executability of BPEL processes. Compared to an existing translation [LA06] from workflow nets into BPEL, we generalized and automated the translation. The translation has been implemented in the compiler oWFN2BPEL which completes the Tools4BPEL framework. This framework allows the fully-automatic partner process synthesis for a given BPEL process. Because the user of the framework does not have to be aware of the underlying formal model, we claim the framework can be easily integrated into industrial BPEL design tools.

In future work, we try to integrate oWFN2BPEL more closely into the Tools4BPEL framework. For example, information about the partner links, operations or variables of the input BPEL process (the provider service) can be stored and later annotated to the synthesized partner BPEL process (the requester service). These information would refine the generated abstract BPEL process and ease the subsequent manual implementation to an executable BPEL process. Furthermore, extensions such as the *WS-BPEL 2.0 Extensions for Sub-Processes*[7] may help to further structure the generated BPEL code. Finally, existing transformations from BPMN and UML-AD to Petri net-based formalisms make the presented approach also applicable to process models other than oWFNs.

Acknowledgments This work is funded by the German Federal Ministry of Education and Research (project Tools4BPEL, project number 01ISE08). We thank the anonymous referees for for detailed comments and very helpful suggestions.

References

[A+07] Alexandre Alves et al. Web Services Business Process Execution Language Version 2.0. OASIS Standard, OASIS, 2007.

[Aal98] Wil M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[BK06] Franck van Breugel and Maria Koshkina. Models and Verification of BPEL. http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf, 2006.

[Kle07] Jens Kleine. Transformation von offenen Workflow-Netzen zu abstrakten WS-BPEL-Prozessen. Diploma thesis (in German), Humboldt-Universität zu Berlin, Berlin, Germany, 2007.

[7] `http://www.ibm.com/developerworks/library/specification/ws-bpelsubproc`

[LA06] Kristian Bisgaard Lassen and Wil M. P. van der Aalst. WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL. In Robert Meersman and Zahir Tari, editors, *OTM Confederated International Conferences, CoopIS, DOA, GADA, and ODBASE 2006, Montpellier, France, October 29 – November 3, 2006. Proceedings, Part I*, volume 4275 of *Lecture Notes in Computer Science*, pages 127–144. Springer-Verlag, 2006.

[LKLR07] Niels Lohmann, Oliver Kopp, Frank Leymann, and Wolfgang Reisig. Analyzing BPEL4Chor: Verification and Participant Synthesis. In Marlon Dumas and Reiko Heckel, editors, *Web Services and Formal Methods, Forth International Workshop, WS-FM 2007, Brisbane, Australia, September 28-29, 2007, Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[LMSW06] Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing Interacting BPEL Processes. In Schahram Dustdar, José Luiz Fiadeiro, and Amit Sheth, editors, *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5–7, 2006, Proceedings*, volume 4102 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2006.

[LMW07] Niels Lohmann, Peter Massuthe, and Karsten Wolf. Operating Guidelines for Finite-State Services. In Jetty Kleijn and Alex Yakovlev, editors, *28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25–29, 2007, Proceedings*, volume 4546 of *Lecture Notes in Computer Science*, pages 321–341. Springer-Verlag, 2007.

[Loh07] Niels Lohmann. A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In Marlon Dumas and Reiko Heckel, editors, *Web Services and Formal Methods, Forth International Workshop, WS-FM 2007, Brisbane, Australia, September 28-29, 2007, Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[MRS05] Peter Massuthe, Wolfgang Reisig, and Karsten Schmidt. An Operating Guideline Approach to the SOA. *Annals of Mathematics, Computing & Teleinformatics*, 1(3):35–43, 2005.

[ODBH06] Chun Ouyang, Marlon Dumas, Stephan Breutel, and Arthur H. M. ter Hofstede. Translating Standard Process Models to BPEL. In Eric Dubois and Klaus Pohl, editors, *Advanced Information Systems Engineering, 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg, June 5–9, 2006, Proceedings*, volume 4001 of *Lecture Notes in Computer Science*, pages 417–432. Springer-Verlag, 2006.

[Rei85] Wolfgang Reisig. *Petri Nets*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[Sch00] Karsten Schmidt. LoLA: A Low Level Analyser. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN 2000, Aarhus, Denmark, June 2000. Proceedings*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474. Springer-Verlag, June 2000.

[Sch05] Karsten Schmidt. Controllability of Open Workflow Nets. In Jörg Desel and Ulrich Frank, editors, *Enterprise Modelling and Information Systems Architectures, Proceedings of the Workshop in Klagenfurt, October 24–25, 2005*, volume 75 of *Lecture Notes in Informatics (LNI)*, pages 236–249. GI, 2005.