On the Transformation of Control Flow between Block-Oriented and Graph-Oriented Process Modeling Languages

Jan Mendling*
Institute of Information Systems and New Media
Vienna University of Economics and Business Administration
Augasse 2-6, A-1090 Wien, Austria
E-mail: jan.mendling@wu-wien.ac.at
*Corresponding author

Kristian Bisgaard Lassen
Department of Computer Science,
University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
E-mail: k.b.lassen@daimi.au.dk

Uwe Zdun
Distributed Systems Group, Information Systems Institute
Vienna University of Technology
Argentinierstrasse 8/184-1, A-1040 Wien, Austria
E-mail: zdun@acm.org

Abstract: Much recent research work discusses the transformation between different process modeling languages. This work, however, is mainly focussed on specific process modeling languages, and thus the general reusability of the applied transformation concepts is rather limited. In this article, we aim to abstract from concrete transformations by distinguishing two major paradigms for representing control flow in process modeling languages: block-oriented languages (such as BPEL and BPML) and graph-oriented languages (such as BPMN, EPCs, and YAWL). The contribution of this article are generic strategies for transforming from block-oriented process languages to graph-oriented languages, and vice versa.

Keywords: business process management; model transformation; business process modeling languages; workflow; BPEL

Reference to this article should be made as follows: Mendling, J., Lassen, K.B., Zdun, U. (2006) 'On the Transformation of Control Flow between Block-Oriented and Graph-Oriented Process Modeling Languages', Int. J. Business Process Integration and Management, Vol. X, No. X, pp.XX–XX.

Biographical notes: Jan Mendling is a PhD student at the Institute of Information Systems and New Media at the Vienna University of Economics and Business Administration. His research interests include business process management, enterprise modeling, and workflow standardization. He is co-author of the EPC Markup Language (EPML) and co-organizer of the XML4BPM workshop series.
Kristian Bisgaard Lassen is a PhD student in the Colored Petri nets group at the Computer Science Institute at the University of Aarhus, Denmark. The subject of his PhD is model driven software development. His research interests include workflow modeling languages (Coloured Petri nets, BPEL, YAWL) and model state visualization.
Uwe Zdun is currently working as an assistant professor in the Distributed Systems Group at the Vienna University of Technology, Vienna, Austria. Uwe received his doctoral degree from the University of Essen in 2002. His research interests include software patterns, software architecture, SOA, distributed systems, object-orientation, and Web engineering. Uwe has published in numerous conferences and journals, and is co-author of the book "Remoting Patterns" published by J. Wiley & Sons. He has co-organized a number of workshops at conferences such as EuroPLoP, CHI, ECOOP, and OOPSLA, and serves as a program chair for EuroPLoP 2006.

1 Introduction

Business process modeling (BPM) languages play an important role not only for the specification of workflows but also for the documentation of business requirements. Even after more than ten years of standardization efforts (Hollingsworth, 2004), the primary BPM languages are still heterogeneous in syntax and semantics. This problem mainly relates to two issues: Firstly, various BPM language concepts that need to be specified in terms of control flow (van der Aalst et al., 2003) and data flow (Russell et al., 2005) have been identified, and most BPM languages introduce a different subset of these (see (Mendling et al., 2004) for a comparison of BPM concepts). Secondly, the paradigm for representing control flow used in the BPM languages is another source of heterogeneity. This issue has not been discussed in full depth so far, but it is of special importance when transformations between BPM languages need to be implemented. In essence, two control flow paradigms can be distinguished, graph- and block-oriented:

- *Graph-oriented* BPM languages specify control flow via arcs that represent the temporal and logical dependencies between nodes. A graph-oriented language may include different types of nodes. These node types may be different from language to language. Workflow nets (van der Aalst, 1997) distinguish places and transitions similar to Petri nets. EPCs (Keller et al., 1992) include function, event, and connector node types. YAWL (van der Aalst and ter Hofstede, 2005) uses nodes that represent tasks and conditions. Similar to XPDL (Workflow Management Coalition, 2002), these tasks may specify join and split rules.
- *Block-oriented* BPM languages define control flow by nesting control primitives used to represent concurrency, alternatives, and loops. XLANG (Thatte, 2001) is an example of a pure block-oriented language. BPML (Arkin, 2002) and BPEL (Andrews et al., 2003) are also block-oriented languages but they also include some graph-oriented concepts (i.e. links). In BPEL, the control primitives are called structured activities. Due to the widespread adoption of BPEL as a standard, we will stick to BPEL as an example of a block-oriented language. Please note that the concepts presented later are also applicable for other block-oriented languages, but as our definitions of block-oriented control flow are rather BPEL-specific, some effort is needed to customize our concepts to other block-oriented languages.

Transformations between block-oriented languages and graph-oriented languages are useful or needed in a number of scenarios. Many commercial tools support the import and export in other formats and languages, meaning that transformations in both directions are implemented by import and export filters. For instance, many graph-oriented tools are recently enhanced to export BPEL in order to support the standard for interoperability and commercial reasons. Transforming BPEL to Petri nets is for instance done for the purpose of verification (Hinz et al., 2005). BPEL does not have formal semantics and can therefore not be verified. By defining a transformation semantics for BPEL in terms of a mapping to Petri nets, it is possible to investigate behavioral properties, such as dead-locks and live-locks. BPEL process definitions are also transformed to EPCs with the goal to communicate the process behavior e.g. to business analysts in a more visual representation (Mendling and Ziemann, 2005). In the direction from EPCs to BPEL, model-driven development approaches start from a visual graph-oriented BPM language such as UML activity diagrams to generate executable BPEL models (Gardner, 2003). These are only some example scenarios, where BPM transformations are needed. The contribution of this article is to abstract from particular graph-oriented or block-oriented control flow representations, to enable a generic discussion of transformation strategies between both. The presented transformation strategies are independent from a certain application scenario and can be used in any setting where transformations between graph-oriented and block-oriented languages are needed.

The rest of the paper article is structured as follows. Section defines the abstractions that are used throughout this article. In particular, we define an abstraction of graph-oriented BPM languages called *Process Graph* that shares most of its concepts with EPCs and YAWL. Block-oriented languages are abstracted by a language called *BPEL Control Flow*. This language is – as mentioned before – an abstraction of BPEL concepts, but can be mapped to the concepts of other block-oriented languages such as BPML. In Section 2.4 we discuss strategies for transforming BPEL Control Flow to Process Graph, and in Section 3.3 the opposite direction. The strategies are specified using pseudo-code algorithms and their prerequisites, advantages, and shortcomings are discussed. Section 4.5 discusses related work, before Section 4.5 concludes the article.

2 Process Graphs and BPEL Control Flow

2.1 Introductory Example

To discuss transformations between graph-oriented and block-oriented BPM languages in a general way, we have to abstract from specific languages. Before that, we illustrate some features of process graphs and BPEL control flow.

The left part of Figure 1 shows a *process graph* in BPMN notation (BPMI and OMG, 2006). As we are interested in syntax transformations, we give the semantics of process graphs only in an informal manner. A process graph has at least one start event and can have multiple end events. Multiple start events represent mutually exclusive, alterna-

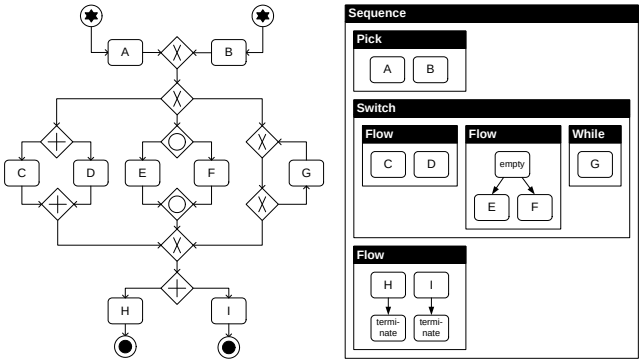

Figure 1: Process graph and BPEL control flow

tive start conditions. End events have explicit termination semantics. This means that when an end event is reached, the complete process is terminated. Connectors represent split and join rules of type OR, XOR, or AND, as they are specified for YAWL (van der Aalst and ter Hofstede, 2005) or EPCs (Keller et al., 1992). All of these elements are connected via arcs which may have an optional guard. Guards are logical expressions that can evaluate to true or false. If a guard of an arc from a split node with type OR or XOR yields false, the target branch of the arc is not executed. If true, execution continues with the target function. After an XOR split, the logical expressions of guards on subsequent arcs must be mutually exclusive.

The right part of Figure 1 gives a *BPEL control flow* with similar control flow semantics as the process graph. In the example, so-called structured activities are used whenever possible. There are structured activities to define alternative start conditions (pick), parallel execution (flow), sequential execution (sequence), conditional repetition (while), and alternative branches (switch). Structured activities can be nested for the definition of complex control flow behavior. Basic activities represent atomic elements of work. There are special basic activities to represent that nothing is done (empty) or that the BPEL control flow is terminated (terminate). Within a flow activity, complex synchronization conditions can be specified via so-called links. Each link can have a transition condition, and each activity that is a target of links can include a join condition of the type OR, XOR, or AND. For BPEL control flow, we adopt the semantics defined in the BPEL specification (Andrews et al., 2003).

2.2 Definition of Process Graphs

To provide a precise description of the transformation strategies, we formalize the syntax of process graphs and those aspects of BPEL that are relevant for a transformation of control flow. We define process graphs to be close to EPCs and YAWL using an EPC-like notation. The respective syntax elements provide the core of graph-based business process modelling languages. Furthermore, AND and XOR connectors can easily be mapped to Petri nets, XPDL, or UML activity diagrams.

Notation 1 (Predecessor and Successor Nodes). Let N be a set of *nodes* and $A \subseteq N \times N$ a binary relation over N defining the arcs. For each *node* $n \in N$, we define the set of *predecessor nodes* $\bullet n = \{x \in N | (x,n) \in A\}$, and the set of *successor nodes* $n\bullet = \{x \in N | (n,x) \in A\}$.

Definition 1 (Process Graph PG). A process graph $PG = (S, E, F, C, l, A, g)$ consists of four pairwise disjoint sets S, E, F, C, a mapping $l : C \rightarrow \{AND, OR, XOR\}$, a binary relation $A \subseteq (S \cup F \cup C) \times (E \cup F \cup C)$, and a mapping $guard : A \rightarrow expr$ such that:

- S denotes the set of start events. $|S| \geq 1$ and $\forall s \in S : |s\bullet| = 1 \wedge |\bullet s| = 0$.
- E denotes the set of end events. $|E| \geq 1$ and $\forall e \in E : |\bullet e| = 1 \wedge |e\bullet| = 0$.
- F denotes the set of functions. $\forall f \in F : |\bullet f| = 1 \wedge |f\bullet| = 1$.
- C denotes the set of connectors. $\forall c \in C : |\bullet c| = 1 \wedge |c\bullet| > 1 \vee |\bullet c| > 1 \wedge |c\bullet| = 1$
- The mapping l specifies the type of a connector $c \in C$ as AND, OR, or XOR.
- A defines the flow as a simple and directed graph. An element of A is called *arc*. Being a simple graph implies that $\forall n \in (E \cup F \cup C) : (n,n) \notin A$ (no reflexive arcs) and that $\forall x, y \in (E \cup F \cup C)\} : |\{(x,y) | (x,y) \in A\}| = 1$ (no multiple arcs).
- The mapping *guard* specifies a guard for an arc $a \in A$. *expr* is a non-terminal symbol to represent a logical expression that defines the guard condition. If and only if this expression yields true, control is propagated to the node following after the guard. Guards of arcs after XOR connector nodes have to be mutually exclusive. Guards are defined on A, however it is only arcs (c,n), where $c \in C$, $l(c) \neq AND$ and $n \in E \cup F \cup C$, that can be expressed as any logical expression. All other guard always yields true; e.g. a guard from an AND-split can never yield false and each function in a sequence is always executed.

Definition 2 (Transitive Closure). Let $PG = (S, E, F, C, l, A, g)$ be defined as in Definition 1. Then $A*$ is the transitive closure of A. That is, if $(n_1, n_2) \in A*$ there is a path from n_1 to n_2 in the process via some arcs of A.

2.3 Definition of BPEL Control Flow

Definition 3 (BPEL Control Flow). A BPEL Control Flow BCF is a tuple $BCF = (Seq, Flow, Switch, While, Pick, Scope, Basic, Empty, Terminate, Link, de, jc, tc)$. BCF consists of pairwise disjoint sets $Seq, Flow, Switch, While, Pick, Scope, Basic, Empty, Terminate$. The set $Str = Seq \cup Flow \cup Switch \cup While \cup Pick \cup Scope$ is called structured activities, the set $Bas = Basic \cup Empty \cup Terminate$ is called basic activities, and the set $Act = Str \cup Bas$ activities. Furthermore, BCF consists of a binary relation $Link \subseteq Act \times Act$, a mapping $de : S \rightarrow \mathbb{P}(A) \setminus \emptyset$, a mapping $jc : A \rightarrow expr$, and a mapping $tc : Link \rightarrow expr$, such that

- *Seq* defines the set of BPEL sequence activities.
- *Flow* defines the set of BPEL flow activities.
- *Switch* defines the set of BPEL switch activities.
- *While* defines the set of BPEL while activities.
- *Pick* defines the set of BPEL pick activities.
- *Scope* defines the set of BPEL scopes.
- *Basic* defines the set of BPEL basic activities without terminate and empty activities. As we are only interested in control flow, the distinction of various basic activities can be neglected here.
- *Empty* defines the set of BPEL empty activities.
- *Terminate* defines the set of BPEL terminate activities.
- *Link* defines a directed graph of BPEL links. These need not to be coherent, but acyclic, and not be connected across the borders of a while activity.
- The mapping *de* denotes a decomposition relation from structured activities to set of nested activities modelled as the power set $\mathbb{P}(A)$. *de* is a tree, i.e. there is no recursive decomposition.
- The mapping *jc* defines the join condition of activities.
- The mapping *tc* defines transition conditions of links.

Definition 4 (Join condition). The join condition, jc, on activities is defined as a $jc : A \rightarrow expr$ using operations such as $\wedge$, $\vee$ and $\veebar$. For an activity x, where $\bullet x = \{y_1, \ldots, y_n\}$ including its predecessor in a structured activity, we use the shorthand AND, OR and XOR for the boolean expressions

$$jc(x) = tc(y_1, x) \wedge \cdots \wedge tc(y_n, x) \text{ (AND)}$$
$$jc(x) = tc(y_1, x) \vee \cdots \vee tc(y_n, x) \text{ (OR)}$$
$$jc(x) = tc(y_1, x) \veebar \cdots \veebar tc(y_n, x) \text{ (XOR)}$$

Definition 5 (Subtree Fragment). Let $Struct \subseteq Act \times Act$ be relation with $(a_1, a_2) \in Struct$ if and only if $a_2 \in de(a_1)$. $Struct*$ is the transitive closure of *Struct*. This implies that a_2 is nested in the subtree fragment of a_1.

Notice that Definition 2.3 do not describe event-, fault-, and compensation handlers. This is because our strategies do not take these into consideration. Also, we do not allow links to cross scope boundaries.

For the purpose of discussing control flow transformations, other BPEL elements than those included in the definition can be neglected. For details on BPEL semantics refer to (Andrews et al., 2003). Note that e.g. BPML has similar syntax elements with comparable semantics (Arkin, 2002). Accordingly, the strategies discussed in the following section can also be applied to define transformations between BPML and process graphs.

2.4 Structural Properties of Process Graphs and BPEL Control Flow

Various transformation choices are bound to certain structural properties of the input model. A process graph can be structured or unstructured and acyclic or cyclic. We define a process graph to be structured by the help of reduction rules. They provide not only a formalization of structuredness but also a means to define a transformation strategy from process graphs to BPEL control flow. Details on this will be explained in Section 3.3.

Definition 6 (Structured Process Graph). A process graph PG is structured if and only if it can be reduced to a single node by the following reduction rules, otherwise it is unstructured. All the reduction rules describe a certain component that is part of the process graph and then how to replace it by a single function.

1. *Sequence reduction:* A sequence is a set of nodes $f_1, \ldots, f_n \in F$ such that $(f_1, f_2), \ldots, (f_{n-1}, f_n) \in A$. *PG* is reduced by removing the sequence and adding a function f_C representing the sequence: $F := (F \cup \{f_C\}) \setminus \{f_1, ..., f_n\}$, $A := (A \cup \{(x, f_C) \mid x \in \bullet f_1\} \cup \{(f_C, x) \mid x \in f_N\bullet\}) \setminus \{(f_1, f_2), ..., (f_{n-1}, f_n)\}$.

2. *Connector pair reduction:* A connecter pair is composed of two connectors $c_1, c_2 \in C$ and functions $c_1\bullet \subseteq F$ such that $|\bullet c_1| = 1$ (split), $|c_2\bullet| = 1$ (join), $l(c_1) = l(c_2)$ and $c_1\bullet = \bullet c_2$. Depending on the image of c_1 and c_2 under l we denote the blocks in the following ways. We call the connector pair an AND-block if $l(c_1) = and$, OR-block if $l(c_1) = or$ and XOR-block if $l(c_1) = xor$. PG is reduced by adding a function f_C representing the connector pair: $F := (F \cup \{f_C\}) \setminus c_1\bullet$, $A := (A \cup \{(x, f_C) \mid x \in \bullet c_1\} \cup \{(f_C, x) | x \in c_2\bullet\}) \setminus (\{(x, c_1) | x \in \bullet c_1\} \cup \{(c_1, x) | x \in c_1\bullet\} \cup \{(x, c_2) | x \in \bullet c_2\} \cup \{(c_2, x) | x \in c_2\bullet\})$ and $C := C \setminus \{c_1, c_2\}$.

3. *XOR-loop reduction:* An XOR-loop is composed of two connectors $c_1, c_2 \in C$ and functions $c_1\bullet \cap \bullet c_2, \bullet c_1 \cap c_2\bullet \subseteq F$ such that $|c_1\bullet| = 1$ (join), $|\bullet c_2| = 1$ (split) and either (1) $c_1\bullet = \{c_2\}$ and $\bullet c_1 \cap c_2\bullet \neq \emptyset$, (2) $c_1\bullet = \bullet c_2$ and $c_2 \in \bullet c_1$, (3) $c_1\bullet = \bullet c_2$ and $\bullet c_1 \cap c_2\bullet \neq \emptyset$ or (4) $c_1\bullet = \{c_2\}$ and $c_2 \in \bullet c_1$. We will refer to an XOR-loop of type (1) as a while-do loop, (2) as a repeat-until loop, (3) as a mixed loop and (4) as the empty loop. PG is reduced by removing the XOR-loop and adding a function f_C representing the XOR-loop: $F := (F \cup \{f_C\}) \setminus ((c_1 \bullet \cap \bullet c_2) \cup (\bullet c_1 \cap c_2\bullet))$, $A := (A \cup \{(x, f_C) | x \in \bullet c_1 \setminus c_2\bullet\} \cup \{(f_C, x) | x \in c_2\bullet \setminus \bullet c_1\}) \setminus (\{(x, c_1) | x \in \bullet c_1\} \cup \{(c_1, x) | x \in c_1\bullet\} \cup \{(x, c_2) | x \in \bullet c_2\} \cup \{(c_2, x) | x \in c_2\bullet\})$ and $C := C \setminus \{c_1, c_2\}$.

4. *Start-block:* A start-block is composed of an XOR connector c and the set of start events S such that $S = \bullet c$. PG is reduced by replacing the start-block by a function f_C: $S := \emptyset$, $F := F \cup \{f_C\}$, $A := (A \cup \{(f_C, x) | x \in c\bullet\}) \setminus (\{(x, c) | x \in \bullet c\} \cup \{(c, x) | x \in c\bullet\})$ and $C := C \setminus \{c\}$.

5. *End-block:* An end-block is composed of an XOR connector c and the set of end events E such that $E = c\bullet$. PG is reduced by replacing the end-block by a function f_C: $E = \emptyset$, $F := F \cup \{f_C\}$, $A := (A \cup \{(x, f_C) | x \in \bullet c\}) \setminus (\{(x, c) | x \in \bullet c\} \cup \{(c, x) | x \in c\bullet\})$ and $C := C \setminus \{c\}$.

Definition 7 (Cyclic versus Acyclic Process Graph). Let $F \cup C$ be the set of functions and connectors of a process graph PG. If $\exists n \in F \cup C : (n, n) \in A*$, then PG is cyclic, otherwise it is acyclic. As a process graph is a simple graph, it holds that $(n, n) \notin A$ (no reflexive arcs). But if $(n, n) \in A*$, there must be a path from n to n via some further nodes $n_1, ..., n_m \in (E \cup F \cup C)$.

Definition 8 (Structured BPEL Control Flow). A BPEL Control Flow BCF is structured if and only if its set $Link = \emptyset$. Otherwise BCF is unstructured.

Furthermore, we define the point wise application of mapping functions which we need in algorithms for the transformation strategies.

Definition 9 (Point Wise Application of Functions). If a function is defined as $f : A \rightarrow B$ then we extend the behavior to sets so that $f(X) = \cup_{x \in X} f(x), X \subseteq A$.

3 From BPEL Control Flow to Process Graph

3.1 Strategy 1: Flattening

Before we present the transformation algorithms, we need to define the mapping function M that transforms a BPEL basic activity to a process graph function.

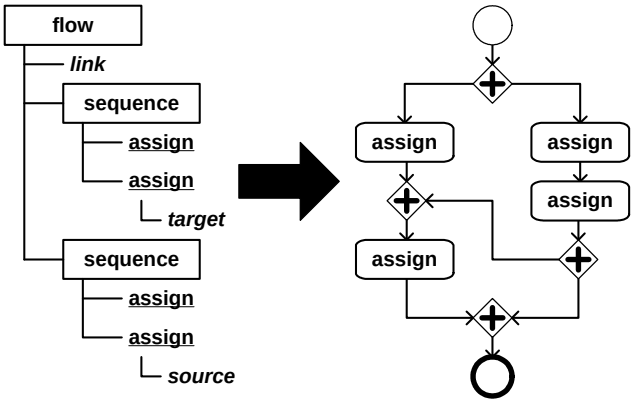

Figure 2: Flattening strategy

Definition 10 (Mapping Function M). Let F be a set of functions of a process graph PG and $Basic$ a set of basic activities of a BCF. The mapping $M : Basic \rightarrow F$ defines a transformation of a BPEL basic activity to a process graph function.

The *general idea* of the Flattening strategy is to map BCF structured activities to respective process graph fragments (see Figure 2). The nested BCF control flow then becomes a flat process graph without hierarchy. For this strategy, there are *no prerequisites*, both structured and unstructured BPEL control flow can be transformed according to this strategy. The *advantage* of Flattening is that the behavior of the whole BPEL control flow is mapped to one process graph. Yet, as a *drawback* the descriptive semantics of structured activities get lost. Such a transformation strategy is useful in a *scenario* where a BPEL control flow has to be communicated to business analysts, because business analysts are used to graph-based visualization of processes.

Algorithm 1 Pseudo code for Flattening strategy

```
procedure: Flattening(BCF)
1: Struct ← Seq ∪ Flow ∪ Switch ∪ While ∪ Pick ∪ Scope
2: S ← {s}; E ← {e}; F ← ∅; C ← ∅; A ← ∅
3: root ← a, where a ∈ Struct ∧ ∄s ∈ Struct : de(s) = a
4: BCFtransform(root, s, e, PG)
5: for all (l1, l2) ∈ Link do
6:     A ← A ∪ {(c1, c2)}
7:     guard(c1, c2) = tc(l1, l2)
8: end for
9: return PG
```

The *algorithm* for the Flattening strategy takes a BCF as input and returns a PG. It recursively traverses the nested structure of BPEL control flow in a top-down manner. This is achieved by identifying the root activity and invoking the *BCFtransform(activity, predecessor, successor, partialResult)* procedure (see Algorithm 1, line 4) which is reinvoked recursively on nested elements. The respective code is given in Algorithm 2. The first parameter *activity* represents the activity to be processed followed by the predecessor and successor node of the output process graph between which the nested structure is hooked in; i.e. *predecessor* and *successor*. For the root activity these are the start and end events s and e. The parameter *partialResult* is used to forward the partial result of the transformation to the procedure. In lines 5–8 links are mapped to arcs and respective join and split connectors around the activity are added.

The $BCFtransform$ procedure (see Algorithm 2) starts with checking whether the current activity serves as target or source for links. If so, respective connectors are added at the beginning and the end of the current activity block. There are four sub-procedures to handle the five structured activities Seq, $Flow$, $Switch$, $While$, and $Pick$. Here, it is assumed that $Pick$ is only used to model alternative start events.[1] The transformation of *Scopes* simply calls the procedure for its nested activity.[2] $Terminate$ is mapped to an end event. Moreover, $Basic$ activities are mapped to functions using M and hooked in the process graph. $Empty$ activities map to an arc between predecessor and successor nodes.

[1] In BPEL, *Pick* can be used at any place where the process waits for concurrent events. As we do not distinguish message-based and other basic activities, decisions are captured by a *Switch* in BCF.

[2] Please note that *Scopes* play an important role in BPEL as a local context for variables, handlers, and also $Terminate$ activities. In the algorithm we abstract from the fact that $Terminate$ only terminates the current *Scope* but not the whole process. Furthermore, we abstract from the fact that a BPEL terminate leads to improper termination.

Algorithm 2 Pseudo code for BCFtransform

procedure: BCFtransform$(activity, pred, succ, PG)$
1: **if** $\exists(l_1, activity) \in Links$ **then**
2: $C \leftarrow C \cup \{c_1\}$; $l(c_1) = jc(activity)$
3: $A \leftarrow A \cup \{(pred, c_1)\}$; $pred \leftarrow c_1$
4: **end if**
5: **if** $\exists(activity, l_2) \in Links$ **then**
6: $C \leftarrow C \cup \{c_2\}$, $l(c_2) = OR$
7: $A \leftarrow A \cup \{(c_2, succ)\}$; $succ \leftarrow c_2$
8: **end if**
9: **if** $activity \in Seq$ **then**
10: $PG \leftarrow$ **transSeq**$(activity, pred, succ, PG)$
11: **else if** $activity \in Flow$ **then**
12: $PG \leftarrow$ **transBlock**$(activity, pred, succ, AND, PG)$
13: **else if** $activity \in Switch$ **then**
14: $PG \leftarrow$ **transBlock**$(activity, pred, succ, XOR, PG)$
15: **else if** $activity \in While$ **then**
16: $PG \leftarrow$ **transWhile**$(activity, pred, succ, PG)$
17: **else if** $activity \in Pick$ **then**
18: $PG \leftarrow$ **transPick**$(activity, pred, succ, PG)$
19: **else if** $activity \in Scope$ **then**
20: $PG \leftarrow$ **BCFtransform**$(de(activity), pred, succ)$
21: **else if** $activity \in Basic$ **then**
22: $F \leftarrow F \cup \{M(activity)\}$
23: $A \leftarrow A \cup \{(pred, activity), (activity, succ)\}$
24: **else if** $activity \in Empty$ **then**
25: $A \leftarrow A \cup \{(pred, succ)\}$
26: **else if** $activity \in Terminate$ **then**
27: $E \leftarrow E \cup \{e\}$
28: $A \leftarrow A \cup \{(pred, e)\}$
29: **end if**
30: **return** PG

The procedures *transSeq*, *transBlock*, *transPick*, and *transWhile* used in the $BCFtransform$ procedure generate the process graph elements that correspond to the respective BCF structured activities. The *transSeq* procedure connects all nested activities of a sequence with process graph arcs. This transformation requires an order defined on the nested activities. For each sub-activities the $BCFtransform$ procedure is invoked again. This is similar to *transBlock*. Here, a split and a join connector are generated. Depending on the label given as a fourth parameter the procedure can transform both *Switch* or *Flow*. The *transPick* replaces the start event of the process graph with one start event for each nested sub-activity. Finally, the *transWhile* procedure generates a loop between an XOR join and XOR split.

3.2 Strategy 2: Hierarchy-Preservation

Many graph-based BPM languages allow to define hierarchies of processes. EPCs for example include hierarchical functions and process interfaces to model sub-processes. In YAWL tasks can be decomposed to sub-workflows. Process graphs can be extended to process graph schemas in a similar way to allow for decomposition.

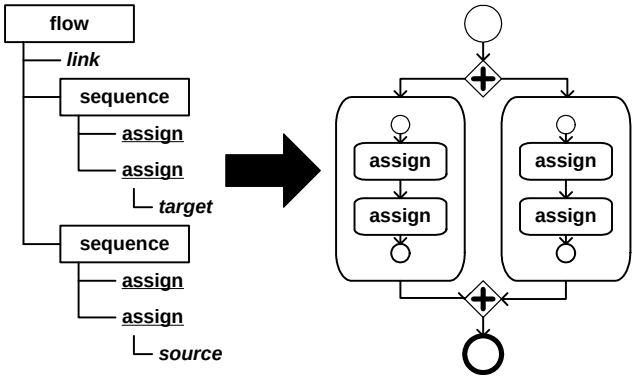

Figure 3: Hierarchy-Preservation strategy

Definition 11 (Process Graph Schema PGS). A process graph schema $PGS = \{PG, s\}$ consists of a set of process graphs PG and a mapping $s : F \rightarrow \{\emptyset, pg\}$ with $pg \in PG$. The mapping s is called subprocess relation. It points from a function to a refining subprocess or, if the function is not decomposed, to the empty set. The relation s is a tree, i.e. there is no recursive definition of sub-processes.

The *general idea* of the Hierarchy-Preservation strategy is to map each BCF structured activity to a process graph of a process graph schema (see Figure 3). The nesting of structured activities is preserved as functions with subprocess relations. The algorithm can be defined in a top-down way similar to the Flattening strategy. Changes have to be defined for the transformation of structured activities as each is mapped to a new process graph. A *prerequisite* of this strategy is that the BCF is structured: links across the border of structured activities cannot be expressed by the subprocess relation. The *advantage* of the Hierarchy-Preservation strategy is that the descriptive semantics of structured activities can be preserved. Furthermore, such a transformation can correctly map the BPEL semantics of *Terminate* activities that are nested in *Scopes*. As a *drawback*, the model hierarchy has to be navigated in order to understand the whole process. This strategy might be useful in a *scenario* where process graphs have to be mapped back to BPEL structured activities, e.g. if a business analyst makes chances in an EPC representation a BPEL control flow and the result is forwarded to a BPEL engineer.

3.3 Strategy 3: Hierarchy-Maximization

One disadvantage of Strategy 2 is that it is bound to structured BPEL control flow. The Hierarchy-Maximization Strategy aims at preserving as much hierarchy as possible with also being applicable to any BPEL control flow – anyway if structured or unstructured. The *general idea* of the strategy is to map those BCF structured activities s to subprocess hierarchies if there are no links nested that cross the border of s (see Figure 4). Accordingly, this strategy is not subject to any structural *prerequisites*. The *advantage*

is that as much structure as possible is preserved. Yet, as a *drawback* the logic of both Strategy 1 and Strategy 2 need to be implemented.

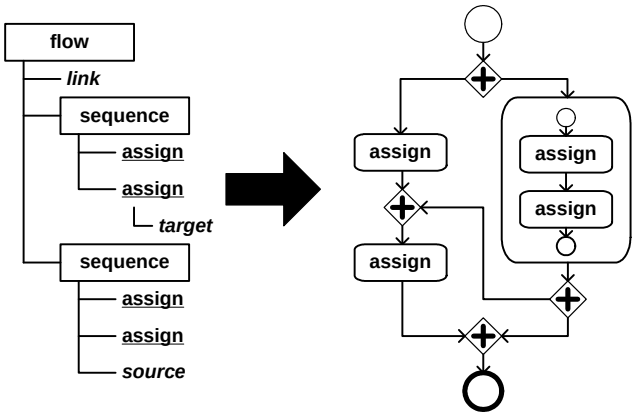

Figure 4: Hierarchy-Maximization strategy

4 From Process Graph to BPEL Control Flow

4.1 Strategy 1: Element-Preservation

In this section we will describe the first strategy for going from process graphs to BCF. Before we go into the details of the Element-Preservation Strategy, we will provide two definitions in which we introduce the notion of an annotated process graph to ease the specification of the strategy. These two definitions, Definition 12 (Annotated Process Graph) and Definition 13 (Annotated Process Graph Node Map), are also relevant for the further strategies.

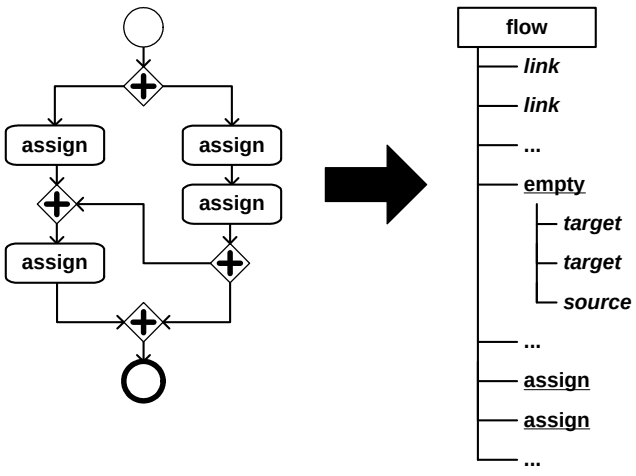

Figure 5: Element-Preservation strategy

Definition 12 (Annotated Process Graph). Let $APG = (S, E, F, C, l, A, B)$ define an annotated graph, where S, E, F, C and l are defined as Definition 1. We define A and B as

- A is a flow relation on the nodes in PG, $A = (S \cup F \cup C \cup B) \times (E \cup F \cup C \cup B)$.
- B is a node in PG that holds a BCF annotation.

One could think of the set B in the annotated process graph, Definition 12, as the set of already translated parts of the process graph. Definition 13 shows how to translate the nodes in an annotated process graph. The *general idea* of this strategy is to map all process graph elements to a *Flow* and map arcs to *Links* (see Figure 5). In particular, start events are mapped to *Basic*,[3] functions are mapped to elements of *Basic*, and connectors are mapped to elements of *Empty*, and end events are translated to elements of *Terminate*. M defines the identity on BPEL constructs.

Definition 13 (Annotated Process Graph Node Map). Let M define a mapping: $E \cup S \cup F \cup C \cup B \rightarrow Basic \cup Empty \cup Terminate \cup B$ and M is defined as

$$M(x) = \begin{cases} Empty(x), & \text{if } x \in C; \\ Basic(x), & \text{if } x \in F \cup S; \\ Terminate(x), & \text{if } x \in E; \\ x, & \text{if } x \in B. \end{cases}$$

an injective translation from the nodes in the graph to activities in BPEL control flow.

It is a *prerequisite* of this strategy that the process graph needs to be acyclic, i.e. $(x, x) \notin A*$. This is because it is not possible to create an activity that logically precedes itself (Andrews et al., 2003). That is, if X precedes Y then Y cannot precede X. The *advantage* of the Element-Preservation strategy is that it is easy to implement and the resulting BPEL control flow will be very similar to the original process graph since there is a one-to-one correspondence between the nodes. As a *drawback*, the resulting BPEL control flow includes more elements than actually needed: connectors are explicitly translated to empty activities in BPEL instead of join condition on nodes. This means that the BPEL control flow might have a lot of nodes which simply act as synchronization points. Furthermore, the resulting BPEL control flow might be difficult to understand compared to strategies in which structured activities, such as the Switch, are chosen to represent some part of the translated graph. If the BPEL control flow is used in a *scenario* where readability is important, then it should be applied only for small process graphs since all elements of the process graph are mapped to *BCF*.

The *algorithm* for the Element-Preservation strategy takes a process graph as input and generates a respective *BCF* as output. Algorithm 3 applies the map M as defined in Definition 13 in lines 1–3. Then, a flow element is added that nests all other activities (lines 4–5). For each arc in the process graph between two nodes a link is added in the BCF between the corresponding two BCF nodes

[3] As a consequence, all alternative start branches are activated when the process is started. Specific transition conditions could be defined to have only one branch being activated. In the algorithm we abstract from this issue.

Algorithm 3 Pseudo Code for Element-Preservation strategy

procedure: Element-Preservation(PG)
1: $Empty \leftarrow M(C)$
2: $Basic \leftarrow M(F \cup S)$
3: $Terminate \leftarrow M(E)$
4: $Flow \leftarrow flow$
5: $de(flow) \leftarrow Empty \cup Basic \cup Terminate$
6: $Link \leftarrow \emptyset$
7: **for all** $(x, y) \in A$ **do**
8: $\quad Link \leftarrow Link \cup (M(x), M(y))$
9: **end for**
10: $jc(x) = \begin{cases} AND, \mid \bullet M^{-1}(x)\mid > 1 \wedge l(M^{-1}(x)) = and; \\ XOR, \mid \bullet M^{-1}(x)\mid > 1 \wedge l(M^{-1}(x)) = xor; \\ OR, otherwise. \end{cases}$
11: $tc(x, y) = guard(M^{-1}(x), M^{-1}(y))$
12: **return**(BCF)

(lines 6–9). The join condition of activities is determined from their corresponding node in the process graph. If it is a connector it will get a similar join condition, i.e. AND for and, OR for or and XOR for xor. Other nodes will get an OR join condition (line 10). If two nodes are connected by a guarded arc then this guard will also be present in the BPEL control flow (line 11).

4.2 Strategy 2: Element-Minimization

This strategy simplifies the generated BCF of Strategy 1. The *general idea* is to remove the empty activities that have been generated from connectors and instead represent splitting behavior by transition conditions of links and joining behavior by join conditions of subsequent activities (see Figure 6).

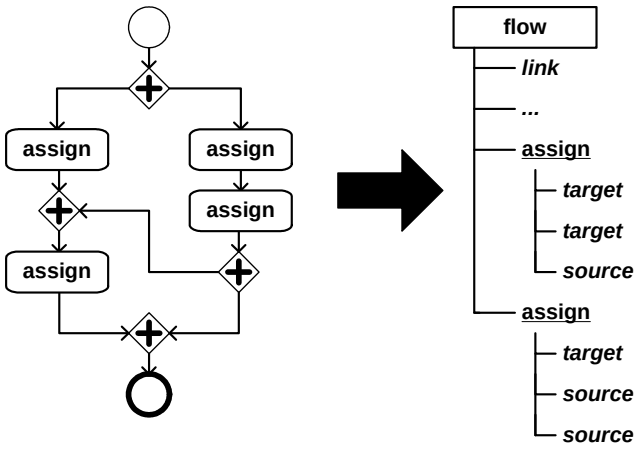

Figure 6: Element-Minimization strategy

As a *prerequisite* the process graph needs to be acyclic, i.e. $(x, x) \notin A*$, in order to make dead path elimination of BPEL work. The *advantage* of the resulting BCF specification is, at least to a greater extent than Strategy 1, that it is in the spirit of BPEL Flow, since it removes empty activities generated from connectors. As a *drawback*, it is less intuitive to identify correspondences between the process graph and the generated BCF specification. This strategy should be used in *scenarios* where the resulting BPEL control flow needs to have as few nodes as possible. This might be the case when performance of the BPEL process matters. In contrast to Strategy 1, the amount of nodes is decreased since all empty activities translated from connector nodes are skipped.

Algorithm 4 Pseudo code for Element-Minimization strategy

procedure: Element-Minimization(PG)
1: $BCF \leftarrow$ **Element-Preservation**(PG)
2: **while** $\exists x \in Empty : M^{-1}(\bullet x) \cap C = \emptyset$ **do**
3: $\quad Link \leftarrow Link \cup \{(y_1, y_2) \mid y_1 \in \bullet x \wedge y_2 \in x\bullet\}$
4: $\quad$ **for all** $y \in x\bullet$ **do**
5: $\quad\quad jc \leftarrow \left(jc'(y') = \begin{cases} jc(y'), & y' \neq y; \\ jc(y') \wedge jc(x), & otherwise. \end{cases}\right)$
6: $\quad$ **end for**
7: $\quad Link \leftarrow Link \setminus (\{(x, y) \mid y \in x\bullet\} \cup \{(x, y) \mid y \in \bullet x\})$
8: $\quad Empty \leftarrow Empty \setminus \{x\}$
9: **end while**
10: **return**(BCF)

The *algorithm* for Element-Minimization translates a PG into a BCF using Algorithm 3 (line 1). Then, there is a loop iterating over all empty activities that have been generated from connectors (line 2) and do not have other translated connector nodes as input links. Finally all translated connector nodes will be removed. For each empty activity x, the nodes having a link to it, are connected to nodes having a link from it. Then, the join conditions of the activities subsequent to x need to be updated. The join condition of an activity is the old join condition it had, before removing x, in conjunction with the join condition of x (lines 4–6). Lines 7–9 defines the actual removal of x. This involves removing all link relations that x occurs in and removing x from the set of Empty activities.

4.3 Strategy 3: Structure-Identification

The *general idea* of this transformation strategy is to identify structured activities in the process graph and apply mappings that are similar to the reduction rules given in Definition 6 on them (see Figure 7). As a *prerequisite* the process graph needs to be structured according to Definition 6. The *advantage* of this strategy is that all control flow is translated to structured activities. For understanding the resulting code this is the best strategy since it reveals the structured components of the process graph. As a *drawback* the relation to the original process graph might not be intuitive to identify. This transformation strategy is appropriate in a *scenario* when the BCF needs to be edited by a BPEL modeling tool or, generally, when understanding the control flow of the process graph is important.

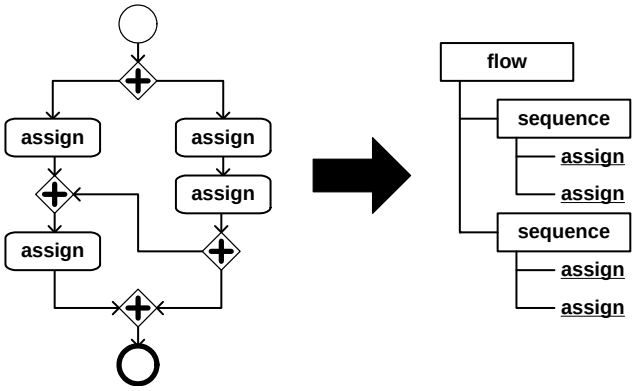

Figure 7: Structure-Identification strategy

This *algorithm* for Structure-Identification uses the reduction rules of Definition 6, but instead of substituting a pattern with a function it is replaced by an annotated node containing the BPEL translation of the process graph fragment. This means, in reducing the process graph we generate an annotated process graph that finally includes only one single annotated node. A single function is mapped to *Basic* in the resulting BCF, whereas annotated nodes are mapped to the set which their annotation is a member of; e.g. *Switch* if a Switch annotation. Each of the rules identifies structure that has an equivalent representation in BPEL control flow as follows:

- A sequence of elements is translated to a BCF sequence with activities in the same order as nodes of the process graph sequence.
- An AND-block is translated to a flow in the BCF. The nodes of the AND-block are translated to nested activities of the flow.
- An OR-block is translated to a flow in the BCF. The nodes of the OR-block are translated to nested activities of the flow with an additional empty activity. This points to each alternative branch and transition conditions are used to activate only a subset of branches. Notice that this translation makes BCF unstructured.
- An XOR-block is translated to a switch in the BCF. Each branch of the XOR-block is mapped to a nested activity of the switch including the respective guard.
- A mixed loop has no direct representation in the BCF. As the rules in Definition 6 state, the graph has the structure $c_1\bullet = \{a_1\}$, $\bullet c_1 \cap c_2\bullet = \{a_2, \ldots, a_n\}$. The condition to leave the loop is `cond`, i.e. the boolean expression $(\underline{\vee}_{x \in A} guard(x)) \wedge \neg(\underline{\vee}_{x \in B} guard(x)), A = \{(c_2, x) | x \in \bullet c_1 \cap c_2\bullet\}$ and $B = \{(c_2, x) | x \notin \bullet c_1 \cap c_2\bullet\}$. However, since exactly one of the arcs from an XOR connector node is true at a time the boolean expression can be reduced to either the left and the right part in the conjunction. Guards in PG are mapped to transition conditions in the BFC. The mixed loop can be mapped to the following BPEL control flow:

```
1: assign(continueLoop,true);
2: while(continueLoop) {
3:   M(a1);
4:   switch {
5:     case cond: assign(continueLoop,false);
6:     case tc(c2,a2)): M(a2);
7:     ...
8:     case tc(c2,an)): M(an);
9:   }
10:}
```

- A while-do loop is translated into a while activity with a switch inside it. It is mapped as the mixed loop with the difference that lines 1, 3, and 5 are omitted and the condition, `cond`, for looping replaces the `continueLoop` in line 2.
- A repeat-until loop has no direct representation in the BCF. It is mapped in a similar way as the mixed loop – lines 6–8 in the pseudo code are omitted.
- An empty loop is translated to an empty activity.
- A start-block is mapped to a Pick containing empty activities for each branch.
- An end-block is translated to a respective AND-, OR-, or XOR-block with each branch followed by a terminate activity.

Algorithm 5 Pseudo code for Structure-Identification strategy

procedure: Structure-Identification(PG)
1: $APG \leftarrow (S, E, F, C, l, A, \emptyset)$
2: **while** $|F \cup C \cup B| > 1$ **do**
3: $APG' \leftarrow match(APG)$ {Using rules in Definition 6}
4: $b \leftarrow translate(APG')$ {Using the described translations above}
5: Reduce APG substituting APG' with b {Using rules in Definition 6}
6: **end while**
7: **return**(BCF)

Algorithm 5 describes the Structure-Identification transformation strategy. Line 1 initializes the annotated process graph. After that, a loop is iterated until the annotated process graph is reduced down to one activity. The reduction rules of Definition 6 are used to substitute components of the process graph by corresponding BCF structured activities in the same way as the function f_C substituted components in Definition 6.

4.4 Strategy 4: Structure-Maximization

The *general idea* of this strategy is to apply the reduction rules of the Structure-Identification strategy as often as possible to identify a maximum of structure (see Figure 8). The remaining annotated process graph is then translated following the Element-Preservation or Element-Minimization strategy. The *advantage* of this strategy is that it can be applied for arbitrary unstructured process graphs as long as its loops can be reduced via the reduction rules of Definition 6. This strategy is also not able to translate arbitrary cycles, i.e. cycles with multiple entrance and/or multiple exit points. A *drawback* of this

strategy is that both the Structure-Identification and the Element-Preservation strategies need to be implemented. The strategy could be used in *scenarios* where models have to be edited by a BPEL modeling tool that uses structured activities as the primal modeling paradigm.

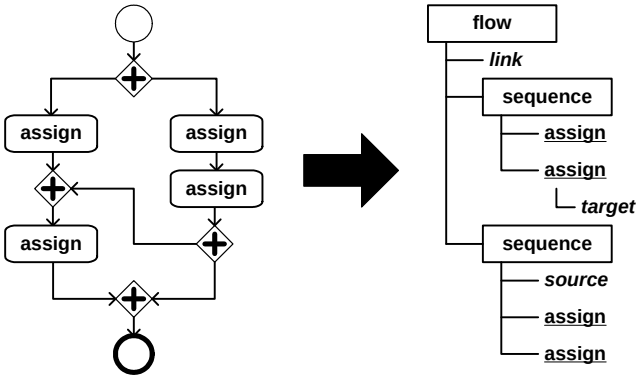

Figure 8: Structure-Maximization strategy

4.5 Strategy 5: Event-Condition-Action-Rules

Similar to the Structure-Maximization strategy, the *general idea* of this strategy to apply the reduction rules of the Structure-Identification strategy as often as possible to identify a maximum of structure (see Figure 9). The remaining annotated process graph is then translated according to the Event-Condition-Action (ECA) rules approach presented in (Ouyang et al., 2006a,b). This approach derives a set of BPEL event handlers that a process calls on itself to capture unstructured control flow. The structured part of the process graph is then encapsulated within BPEL event handlers. Unstructured control flow maps to messages sent from some place in the process to itself where it is fetched by an event handler. In the following we assume the process graph to be reduced according to the Structure-Identification strategy and that the remaining unstructured part is input to Algorithm 6. The general idea of the strategy is similar to an algorithm presented in Ouyang et al. (2006b). First, we introduce three extensions to BPEL control flow:

- *Handler* defines the set of event handlers associated with a BPEL control flow. A precondition set *prc* is associated with a handler and a set of activities *Act* in the expression $handler(prc, Act)$.
- *Invoke* defines the set of invoke activities that a *BCF* recursively calls upon itself. An invoke takes a symbol s as a parameter to represent the message to be sent in the expression $invoke(s)$.
- *Receive* defines the set of receive activities that wait for messages being sent to the *BCF*. A receive takes a symbol s as a parameter to represent the message to be received in the expression $receive(s)$.

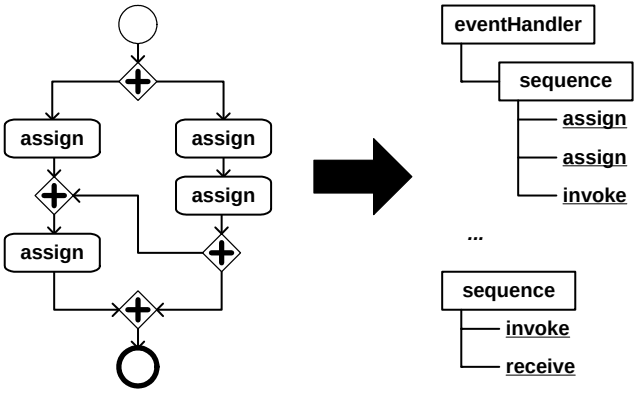

Figure 9: Event-Condition-Action Rules strategy

Algorithm 6 essentially captures the logic defined in Ouyang et al. (2006a). Lines 1–5 generate a sequence to include the first activity of the process graph. When this first activity completes, the $invoke(firstCompleted)$ activity signals this completion to all event handlers that have the first activity in their precondition set. The loop in lines 6–10 adds for each function not being the initial function an individual event handler that is executed when its precondition set evaluates to true.

Algorithm 6 Pseudo code for Event-Condition-Action-Rules strategy

procedure: Event-Condition-Action-Rules(PG)
1: $Receive \leftarrow receive(processStarted)$
2: $Basic \leftarrow M(\textbf{getFirstFunction}(PG))$
3: $Invoke \leftarrow invoke(firstCompleted)$
4: $Sequence \leftarrow sequence$
5: $de(sequence) \leftarrow Receive \cup Basic \cup Invoke$
6: $Handler \leftarrow \emptyset$
7: **for all** $f \in F \setminus \{\textbf{getFirstFunction}(PG)\}$ **do**
8: $prc \leftarrow \textbf{getPreconditionSet}(f)$
9: $Handler \leftarrow Handler \cup handler(prc, M(f))$
10: **end for**

The *advantage* of this strategy is that it is applicable for any process graph, even for those that have unstructured loops. But there are also some *drawbacks*. First, the generated BPEL control flow is more difficult to comprehend than the code generated by the Structure-Maximization strategy, as control flow crosses the border of event handlers. Second, it is so far not clear how the generated BPEL processes behave if the input models have semantic problems. In (Kindler, 2006), it is proven that there are process models with OR-joins waiting for one another that do not have a formal semantics. Basically, the approach presented in (Ouyang et al., 2006b), and also the other strategies, cannot resolve such problems.

5 Case Study on Strategy-based Transformation

We validated the applicability to the strategies for the

implementation of a BPEL export filter of a commercial workflow designer. This workflow designer uses UML Activity Diagrams with product-specific extensions for modeling. In essence, we followed the element-preservation strategy and deviated in order to capture specifics of start and end events, split elements, and a two-level modeling concept. Models built by the workflow designer have exactly one start node and one or more end nodes with implicit termination semantics. As the entry and exit of a BPEL flow exactly captures these semantics, we decided not to transform them to BPEL. The workflow designer offers two split elements that have semantics comparable to an XOR split: switch nodes (two alternatives) and decision nodes (multiple alternatives). We decided to map both of these elements to a BPEL switch that includes empty elements for each alternative that serves as a source for a link to the subsequent activity. This design has been chosen instead of a mapping to empty activities in order to easier distinguish different types of splits when the exported BPEL is re-imported. Furthermore, the workflow designer offers a two-level modeling approach: step nodes similar to process graph functions have to be specified by a sequence of one or multiple step actions. Step nodes are part of the UML model, step actions have no visual representation. Therefore, we map step nodes to BPEL sequences that nest further BPEL activities corresponding to the semantics of the step actions.

The mapping of many proprietary concepts of the workflow designer turned out to be a challenge for our export filter design. These proprietary concepts include sub-workflow elements, step actions, and properties of the individual visual elements:

- Regarding the sub-workflow concept, we decided to map each sub-workflow to a BPEL scope containing a nested subprocess. For a more appropriate mapping, the upcoming BPEL-SPE extension will be very helpful (Kloppmann et al., 2005), especially for variable passing as well as fault and compensation handling.

- Step actions are defined in an abstract class, which is customized by a number of different possible step actions, such as defining a (local) variable, inline Java code, or mail sending. To map these steps, we first defined a generic mapping operation to BPEL in the abstract step action class which is used when no special class overrides the operation. In this case, a BPEL invoke is written to the output, containing the name of the step as a partner link. We also defined mappings for a number of concrete step actions, e.g. transforming the Java code step action is transformed to a BPELJ snippet (Blow et al., 2004).

- All visual elements of the workflow designer can have additional properties. Some of those, such as time-out conditions and escalations, might even have an influence on the control flow. We defined a special XML namespace for these properties and included them as attributes in the respective BPEL activity. Finally, we had to map step actions contained in the step nodes to BPEL basic activities.

In conclusion, the transformation strategies have helped us to find a systematic, initial approach and process for the transformation of the workflow designer's notation to BPEL. They are also useful for explaining the overall design decisions. The case study also shows that the transformation strategies can be mixed. The strategies define ideal, prototypical mappings, but in a complex product like the workflow designer in our case study it is necessary to identify the most suitable transformation strategy for the different parts of the mapping.

6 Related Work

There have been several works on transformations between BPEL control flow and graph-based process modeling languages. We highlight only some of them and refer to (Mendling et al., 2005) for a comprehensive overview.

The transformation of graph-based languages to BPEL control flow can be related to work dedicated to *model-driven development* of executable BPEL process definitions. A conceptual mapping from *EPCs* to BPEL is presented in (Ziemann and Mendling, 2005). The authors choose a transformation based on the Element-Preservation strategy for the reason that it is easy to implement. In (van der Aalst et al., 2005; van der Aalst and Lassen, 2005) a *Workflow-net*-based modeling approach for BPEL including a respective transformation is presented. Similar to the Structure-Identification strategy, Workflow nets are reduced by matching components that are equivalent to BPEL structured activities such as switch and pick. The Structure-Identification strategy has been chosen in order to generate readable BPEL template code and not executable BPEL processes. In transformation based on the Flattering strategy from XML nets to BPEL is reported in (Koschmider and von Mevius, 2005).

Further work takes the modeling standards UML and BPMN as a starting point. In (Gardner, 2003) a BPM-specific profile of *UML* is used to generate BPEL code. From the paper the transformation strategy is not clear, but the figures suggest that the author uses an Element-Preservation strategy and maps sequences to BPEL sequences. The *BPMN* specification (BPMI and OMG, 2006) comes along with a proposal for a mapping to BPEL. The subsection 11.17 of BPMN specification presents a mapping that is close to the Structure-Identification strategy. Yet, the mapping is given rather in prose, a precise algorithm and a definition of required structural properties is missing. (Ouyang et al., 2006a) show a translation from so-called *Standard Process Models* (SPMs) (Kiepuszewski et al., 2003) to BPEL. SPM basically reflects the commonalities of UML Activity Diagrams and BPMN. The authors generate Event-Condition-Action (ECA) rules for each activity in the SPM that describes what event must occur under what condition for an activity to become active.

Each ECA is translated into BPEL as an event handler resulting in the entire BPEL process being a sequence of event handlers that invoke each other. An extension of this work defines the Event-Condition-Action-Rules strategy (Ouyang et al., 2006b) mapping structure to BPEL structured activities.

There is also some work reported on transformations from BPEL to graph-based process modeling languages. The verification of BPEL with Petri net analysis techniques is an important stream of research in this context. Transformations essentially utilizing the Flattening strategy are reported in (Hinz et al., 2005; Ouyang et al., 2005). The latter also provides an overview of BPEL verification approaches in general. In (Mendling and Ziemann, 2005), a transformation to EPCs is presented. The motivation of this work is that executable BPEL processes have to be discussed with business people who are often familiar with the EPC notation. As BPEL does not offer a standardized notation, such a transformation can be used to visualize BPEL processes.

Table 1 summarizes academic work on transformations between BPEL and graph-based process modeling languages as well as the applied transformation strategies. Such transformations are also important to commercial business process management systems. Several of these systems offer a modeling component with a graph-based process modeling notation. As BPEL is becoming accepted as a standard for executable process definitions, these tools face the challenge of importing and exporting BPEL. Some experiences with applying transformation strategies to the development of import and export filters are reported in (Mendling et al., 2006).

Table 1: Transformation strategies and related work

Transformation	Strategy
EPC to BPEL (Ziemann and Mendling, 2005)	Element-Preservation
Workflow nets to BPEL (van der Aalst and Lassen, 2005)	Structure-Identification
UML to BPEL (Gardner, 2003)	Element-Preservation
BPMN to BPEL (BPMI and OMG, 2006)	Structure-Identification
SPM to BPEL (Ouyang et al., 2006a)	Event-Condition-Action-Rules
BPMN to BPEL (Ouyang et al., 2006b)	Event-Condition-Action-Rules
XML nets to BPEL (Koschmider and von Mevius, 2005)	Element-Preservation
BPEL to Petri nets (Hinz et al., 2005)	Flattening
BPEL to Petri nets (Ouyang et al., 2005)	Flattening
BPEL to EPCs (Mendling and Ziemann, 2005)	Flattening

7 Conclusion and Future Work

This article has addressed the problem of transformations between graph-oriented and block-oriented BPM languages. In order to discuss such transformations in a general way, we defined process graphs as an abstraction of graph-oriented BPM languages and BPEL control flow as an abstraction of BPEL that shares most of its concepts with block-oriented languages like BPML. Our major contribution is the identification of different transformation strategies between the two BPM modelling paradigms and their specification as pseudo code algorithms. In particular, we identified the Flattening, Hierarchy-Preservation, and the Hierarchy-Maximization strategies for transformations from BPEL control flow to process graphs. In the other direction we identified the Element-Preservation, Element-Minimization, Structure-Identification, Structure-Maximization, and Event-Condition-Action-Rule strategies. As such, the strategies provide a useful generalization of many current X-to-BPEL and BPEL-to-Y papers, not only for identifying design alternatives but also for discussing design decisions. We checked the applicability of these strategies in two case studies which are reported in (Mendling et al., 2005), one of them is reported in this article.

In future research, we aim to conduct further case studies in order to identify how aspects that are not captured by process graphs and BPEL control flow can be addressed in a systematic way. Another issue is the upcoming new version of BPEL which is expected to be issued as a standard in 2006. It will be interesting to discuss in how far that new version simplifies or complicates the mapping to and from graph-oriented BPM languages. Furthermore, there is potential for additional strategies. The transformation of unstructured process graphs to structured BPEL has been recently discussed in (Zhao et al., 2006). But even though unstructured process graphs including only XOR-splits and -joins can be transformed, arbitrary concurrency cannot be structured (Kiepuszewski et al., 2000). Still future transformation strategies might provide better output in terms of readability or minimal number of constructs for specific subclasses of BPEL processes or process graphs.

REFERENCES

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. (2003). Business Process Execution Language for Web Services, Version 1.1. Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems.

Arkin, A. (2002). Business Process Modeling Language (BPML). Spec., BPMI.org.

Blow, M., Goland, Y., Kloppmann, M., Leymann, F., Pfau, G., Roller, D., and Rowley, M. (2004). BPELJ: BPEL for Java. Whitepaper, BEA and IBM.

BPMI and OMG (2006). *Business Process Modeling Notation Specification. Final Adopted Specification.* http://www.bpmn.org/.

Gardner, T. (2003). UML Modelling of Automated Business Processes with a Mapping to BPEL4WS. In *Proceedings of the First European Workshop on Object Orientation and Web Services at ECOOP 2003*.

Hinz, S., Schmidt, K., and Stahl, C. (2005). Transforming BPEL to Petri Nets. In *Proceedings of BPM 2005*, LNCS 3649, pages 220–235.

Hollingsworth, D. (2004). *The Workflow Handbook 2004*, chapter The Workflow Reference Model: 10 Years On, pages 295–312. Workflow Management Coalition.

Keller, G., Nüttgens, M., and Scheer, A. W. (1992). Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)". Heft 89, Inst. für Wirtschaftsinformatik, Saarbrücken, Germany.

Kiepuszewski, B., ter Hofstede, A. H. M., and Bussler, C. (2000). On structured workflow modelling. In Wangler, B. and Bergman, L., editors, *Advanced Information Systems Engineering, 12th International Conference CAiSE 2000*, volume 1789 of *Lecture Notes in Computer Science*, pages 431–445. Springer.

Kiepuszewski, B., ter Hofstede, A. H. M., and van der Aalst, W. M. P. (2003). Fundamentals of control flow in workflows. *Acta Inf.*, 39(3):143–209.

Kindler, E. (2006). On the semantics of epcs: Resolving the vicious circle. *Data Knowl. Eng.*, 56(1):23–40.

Kloppmann, M., König, D., Leymann, F., Pfau, G., Rickayzen, A., von Riegen, C., Schmidt, P., and Trickovic, I. (2005). WS-BPEL Extension for Sub-processes BPEL-SPE. Joint white paper, IBM and SAP.

Koschmider, A. and von Mevius, M. (2005). A petri net based approach for process model driven deduction of bpel code. In Meersman, R., Tari, Z., and Herrero, P., editors, *OTM Workshops*, volume 3762 of *Lecture Notes in Computer Science*, pages 495–505. Springer.

Mendling, J., Lassen, K., and Zdun, U. (2005). Transformation strategies between block-oriented and graph-oriented process modelling languages. Technical Report JM-2005-10-10, WU Vienna.

Mendling, J., Lassen, K., and Zdun, U. (2006). Experiences in enhancing existing bpm tools with bpel import and export. Technical Report JM-2006-03-10, WU Vienna.

Mendling, J., Nüttgens, M., and Neumann, G. (2004). A Comparison of XML Interchange Formats for Business Process Modelling. In Feltz, F., Oberweis, A., and Otjacques, B., editors, *Proceedings of EMISA 2004*, LNI 56, pages 129–140.

Mendling, J. and Ziemann, J. (2005). Transformation of BPEL Processes to EPCs. In M. Nüttgens and F. J. Rump, editor, *Proceedings of the 4th GI Workshop on Business Process Management with Event-Driven Process Chains (EPK 2005)*, pages 41–53.

Ouyang, C., Dumas, M., Breutel, S., and ter Hofstede, A. (2006a). Translating Standard Process Models to BPEL. In E. Dubois, K. P., editor, *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE'06)*, volume 4001 of *Lecture Notes in Computer Science*.

Ouyang, C., van der Aalst, W., Breutel, S., Dumas, M., ter Hofstede, A., and Verbeek, H. (2005). Formal semantics and analysis of control flow in ws-bpel. BPMCenter Report BPM-05-13, BPMcenter.org.

Ouyang, C., van der Aalst, W., Dumas, M., and ter Hofstede, A. (2006b). Translating bpmn to bpel. BPMCenter Report BPM-06-02, BPMcenter.org.

Russell, N., ter Hofstede, A., Edmond, D., and van der Aalst, W. M. (2005). Workflow data patterns: Identification, representation and tool support. In *Proc. of the 24th International Conference on Conceptual Modeling (ER 2005)*, LNCS.

Thatte, S. (2001). XLANG. Specification, Microsoft Corp.

van der Aalst, W. and Lassen, K. (2005). Translating Workflow Nets to BPEL4WS. BETA Working Paper Series, WP 145, Eindhoven University of Technology, Eindhoven.

van der Aalst, W. M., Jørgensen, J. B., and Lassen, K. B. (2005). Let's Go All the Way: From Requirements via Colored Workflow Nets to a BPEL Implementation of a New Bank System. In Meersman, R. and Z.Tari, editors, *Proceedings of CoopIS/DOA/ODBASE 2005, Cyprus*, LNCS 3760, pages 22–39.

van der Aalst, W. M. P. (1997). Verification of Workflow Nets. In Azéma, P. and Balbo, G., editors, *Application and Theory of Petri Nets*, LNCS 1248, pages 407–426.

van der Aalst, W. M. P. and ter Hofstede, A. H. M. (2005). YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275.

van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. (2003). Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51.

Workflow Management Coalition (2002). Workflow Process Definition Interface – XML Process Definition Language. Document Number WFMC-TC-1025, October 25, 2002, Version 1.0, Workflow Management Coalition.

Zhao, W., Hauser, R., Bhattacharya, K., Bryant, B. R., and Cao, F. (2006). Compiling business processes: untangling unstructured loops in irreducible flow graphs. *Int. Journal of Web and Grid Services*, 2(1):68–91.

Ziemann, J. and Mendling, J. (2005). EPC-Based Modelling of BPEL Processes: a Pragmatic Transformation Approach. In *Proceedings of MITIP 2005, Italy*.