Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages

Jan Mendling[1], Kristian Bisgaard Lassen[2], Uwe Zdun[1]

[1] Institute of Information Systems and New Media
Vienna University of Economics and Business Administration
Augasse 2-6, A-1090 Wien, Austria
{jan.mendling|uwe.zdun}@wu-wien.ac.at
[2] Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
k.b.lassen@daimi.au.dk

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

1 Introduction

Business process modelling (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 [Hol04], 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 [vdAtHKB03] and data flow [RtHEvdA05] have been identified, and most BPM languages introduce a different sub-set of these (see [MNN04] 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 [vdA97] distinguish places and transitions similar to Petri nets. EPCs [KNS92, MN05] include function, event, and connector node types. YAWL [vdAtH05] uses graph nodes that represent tasks and conditions. Similar to XPDL [Wor02], 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 [Tha01] is an example of a pure block-oriented language. BPML [Ark02] and BPEL [ACD$^+$03] 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 done for the purpose of verification [HSS05]. 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 [MZ05]. In the other direction, model-driven development approaches start from a visual graph-oriented BPM language such as UML activity diagrams to generate executable BPEL models [Gar03]. These are only some example scenarios, where BPM transformations are needed. The contribution of this paper is to abstract from particular graph-oriented or block-oriented control flow representation, to enable a generic discussion of transformation strategies between both. The presented transformation strategies are independent from a certain application scenario and can therefore be used in any setting where transformations between graph-oriented and block-oriented languages are needed.

The rest of the paper is structured as follows. Section 2 defines the abstractions that are used throughout this paper. 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 3 we discuss strategies for transforming BPEL Control Flow to Process Graph, and in Section 4 the opposite direction. The strategies are specified using pseudo-code algorithms and their prerequisites, advantages, and shortcomings are discussed. Section 5 discusses related work, and finally Section 6 concludes and discusses future work.

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.

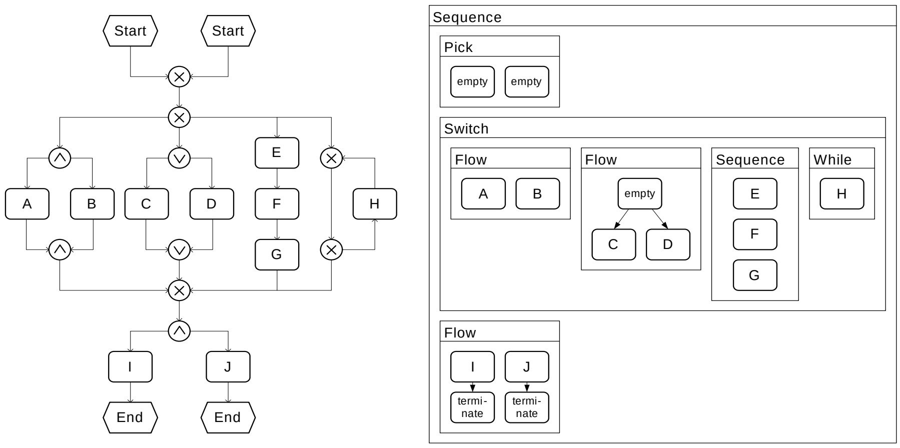

Figure 1: Process graph and BPEL control flow

The left part of Figure 1 shows a *process graph*. 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, alternative 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 [vdAtH05] or EPCs [MN05]. 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 connector 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 of the 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 [ACD+03].

2.2 Definition of Process Graphs

To provide for 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 subsequent to 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 graph 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 the transition condition 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

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

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 [ACD+03]. Note that e.g. BPML has similar syntax elements with comparable semantics [Ark02]. 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 4.

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 reduction rules formally defined in [MLZ05], 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. There are reduction rules for sequences, connector pairs, XOR-loops, and start- and end-blocks. For details refer to [MLZ05].

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. If $\forall n \in F \cup C : (n, n) \notin A*$, then PG 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 BPEL Control Flow to Process Graph Transformation Strategies

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.

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.

Algorithm 1 Pseudo code for Flattening strategy

procedure: Flattening(BCF)

1: $Struct \leftarrow Seq \cup Flow \cup Switch \cup While \cup Pick \cup Scope$
2: $S \leftarrow \{s\}$; $E \leftarrow \{e\}$; $F \leftarrow \emptyset$; $C \leftarrow \emptyset$; $A \leftarrow \emptyset$
3: $root \leftarrow a$, where $a \in Struct \wedge \nexists s \in Struct : de(s) = a$
4: **BCFtransform**$(root, s, e, PG)$
5: **for all** $(l_1, l_2) \in Link$ **do**
6: $\quad A \leftarrow A \cup \{(c_1, c_2)\}$
7: $\quad guard(c_1, c_2) = tc(l_1, l_2)$
8: **end for**
9: **return** PG

The *general idea* of the Flattening strategy is to map BCF structured activities to respective process graph fragments. 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 process 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 process has to be communicated to business analysts.

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 in [MLZ05]. 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 in [MLZ05]) 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

[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.

successor nodes.
The procedures $BCFtransformSeq$, $BCFtransformBlock$, $BCFtransformPick$, and $BCFtransformWhile$ used in the $BCFtransform$ procedure generate the process graph elements that correspond to the respective BCF structured activities. The $BCFtransformSeq$ procedure connects all nested activities of a sequence with process graph arcs. Although not explicitly defined, this transformation requires an order defined on the nested activities. For each sub-activities the $BCFtransform$ procedure is invoked again. This is similar to $BCFtransformBlock$. 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 $BCFtransformPick$ replaces the start event of the process graph with one start event for each nested sub-activity. Finally, the $BCFtransformWhile$ 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.

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. 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 the 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.

3.3 Strategy 3: Hierarchy-Maximization

One disadvantage of Strategy 2 is that it is bound to structured BPEL. 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. Accordingly, this strategy is not subject to any structural *prerequisites*. The advantage is that as much structure as possible is preserved. Yet, the logic of both Strategy 1 and Strategy 2 need to be implemented.

4 Process Graph to BPEL Control Flow Transformation Strategies

4.1 Strategy 1: Element-Preservation

In this section we will describe the first strategy for going from process graphs to BCF. The following Definitions 12 (Annotated Process Graph) and 13 (Annotated Process Graph Node Map) are also relevant of further strategies. Before we go through the strategies we will make some definitions where we introduce the notion of an annotated process graph to ease the notation in 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 an annotation in BCF.

One could think of the set B in the annotated 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 $Links$. In particular, start events are mapped to $Basic$,[3] function 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.

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 [ACD+03]. I.e., if X precedes Y then Y cannot precede X. The *advantage* of the Element-Preservation strategy is that it is simple to implement and the resulting BPEL will be very similar to the original process graph since there is a one-to-one correspondence between

[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.

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 code might have a lot of nodes which simply act as synchronization points. Furthermore, the resulting BPEL might be more difficult to understand than if structured activities, such as the Switch, where chosen to represent some part of the translated graph. If the BPEL code 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.

Algorithm 2 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: $jc(x) = \begin{cases} AND, & |\bullet M^{-1}(x)| > 1 \wedge l(M^{-1}(x)) = and; \\ XOR, & |\bullet M^{-1}(x)| > 1 \wedge l(M^{-1}(x)) = xor; \\ OR, & otherwise. \end{cases}$
11: $tc(x, y) = guard(M^{-1}(x), M^{-1}(y))$
12: **return**(BCF)

The *algorithm* for the Element-Preservation strategy takes a process graph as input and generates a respective BCF as output. The Algorithm 2 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 (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 (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. 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 code 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 3 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* translates a PG into a BCF using Algorithm 2 (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. 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 is to be edited by a BPEL modeling tool or, generally, when understanding the control flow of the process graph is important.
Our *algorithm* 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 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 the 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 rule 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 $(\veebar_{x \in A} guard(x)) \wedge \neg(\veebar_{x \in B} guard(x)), A = \{(c_2, x) | x \in \bullet c_1 \cap c_2\bullet\}$ 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 both 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 pseudo code:

```
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 through 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 4 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

Algorithm 4 Pseudo code for Structure-Identification strategy

procedure: Structure-Identification(PG)

$APG \leftarrow (S, E, F, C, l, A, \emptyset)$
while $|F \cup C \cup B| > 1$ **do**
  $APG' \leftarrow match(APG)$ {Using rules in Definition 6}
  $b \leftarrow translate(APG')$ {Using the described translations above}
  Reduce APG substituting APG' with b {Using rules in Definition 6}
end while
return(BCF)

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. 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. Still 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 strategy and at least the Element-Preservation strategy needs 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.

5 Related Work

A lot of work exists on transformation between BPEL and other process languages. We highlight only a part and refer to [MLZ05] for a more complete discussion of related work. One branch of related work is dedicated to *model-driven development* of executable BPEL process definitions. In [Gar03] a BPM-specific profile of UML is used to generate BPEL code. The aim is rather to prove the feasibility of such an approach than the discussion of different transformation alternatives. It is not clear from the paper which strategy the author choose. The code fragments suggest that an Element-Preservation strategy is taken and sequences are mapped to BPEL sequence. The Element-Preservation strategy can also be found in a mapping from EPCs to BPEL [ZM05]. The BPMN specification [Whi04] comes along with a proposal for a mapping to BPEL. As BPMN is a graph-oriented BPM language similar to process graphs, the strategies of Section 4 can be applied. The subsection 6.17 of BPMN spec presents a mapping that is close to the Structure-Identification strategy proposed in this paper. The authors introduce so-called conceptual tokens to identify structure. Yet, the mapping is given rather in prose, a precise algorithm and a definition of required structural properties is missing. Further work using a Structure-Identification strategy is reported in [vdAJL05] where Workflow nets, and in [KvM05] where XML nets are mapped to BPEL.

A second branch of research is related to *conceptual mappings* in order to better understand BPEL behavior and its relation to other BPM languages. In [HSS05] a transformation from BPEL to Petri Nets is presented in order to give BPEL formal semantics. The authors use a Flattening strategy to generate a Petri Net that covers BPEL behavior including exceptional behavior. The generated Petri Net is used for formal static analysis of the BPEL model.

6 Conclusion and Future Work

In this paper, we 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 identify the Flattening, Hierarchy-Preservation, and the Hierarchy-Maximization strategy for transformations from BPEL control flow to process graphs. In the other direction we identify Element-Preservation, Element-Minimization, Structure-Identification, and Structure-Maximization strategy. 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 [MLZ05].

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 the beginning of 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.

References

[ACD+03] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems, 2003.

[Ark02] A. Arkin. Business Process Modeling Language (BPML). Spec., BPMI.org, 2002.

[Gar03] Tracy Gardner. 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*, 2003.

[Hol04] David Hollingsworth. *The Workflow Handbook 2004*, chapter The Workflow Reference Model: 10 Years On, pages 295–312. Workflow Management Coalition, 2004.

[HSS05] Sebastian Hinz, Karsten Schmidt and Christian Stahl. Transforming BPEL to Petri Nets. In *Proceedings of BPM 2005*, LNCS 3649, pages 220–235, 2005.

[KNS92] G. Keller, M. Nüttgens and A. W. Scheer. Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)". Heft 89, Institut für Wirtschaftsinformatik, Saarbrücken, Germany, 1992.

[KvM05] Agnes Koschmider and Marco von Mevius. A Petri Net Based Approach for Process Model Driven Deduction of BPEL Code. In Robert Meersman, Zahir Tari and Pilar Herrero, editors, *OTM Workshops*, volume 3762 of *Lecture Notes in Computer Science*, pages 495–505. Springer, 2005.

[MLZ05] J. Mendling, K. Lassen and U. Zdun. Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages. Technical Report JM-2005-10-10, WU Vienna, October 2005.

[MN05] Jan Mendling and Markus Nüttgens. EPC Markup Language (EPML) - An XML-Based Interchange Format for Event-Driven Process Chains (EPC). Technical Report JM-2005-03-10, WU Wien, Austria, 2005.

[MNN04] Jan Mendling, Markus Nüttgens and Gustaf Neumann. A Comparison of XML Interchange Formats for Business Process Modelling. In F. Feltz, A. Oberweis and B. Otjacques, editors, *Proceedings of EMISA 2004*, LNI 56, pages 129–140, 2004.

[MZ05] J. Mendling and J. Ziemann. EPK-Visualisierung von BPEL4WS Prozessdefinitionen. In *Proc. of Workshop on Software Reengineering, Germany*, 2005.

[RtHEvdA05] Nick Russell, A.H.M. ter Hofstede, D. Edmond and Wil M.P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In *Proc. of the 24th International Conference on Conceptual Modeling (ER 2005)*, LNCS, 2005.

[Tha01] S. Thatte. XLANG. Specification, Microsoft Corp., 2001.

[vdA97] W. M. P. van der Aalst. Verification of Workflow Nets. In Pierre Azéma and Gianfranco Balbo, editors, *Application and Theory of Petri Nets*, LNCS 1248, pages 407–426, 1997.

[vdAJL05] Wil M.P. van der Aalst, Jens Bæk Jørgensen and Kristian Bisgaard Lassen. Let's Go All the Way: From Requirements via Colored Workflow Nets to a BPEL Implementation of a New Bank System. In R. Meersman and Z.Tari, editors, *Proceedings of CoopIS/DOA/ODBASE 2005, Agia Napa, Cyprus*, LNCS 3760, pages 22–39, 2005.

[vdAtH05] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[vdAtHKB03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski and Alistair P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

[Whi04] S. A. White. Business Process Modeling Notation. Specification, BPMI.org, 2004.

[Wor02] Workflow Management Coalition. Workflow Process Definition Interface – XML Process Definition Language. Document Number WFMC-TC-1025, October 25, 2002, Version 1.0, Workflow Management Coalition, 2002.

[ZM05] J. Ziemann and J. Mendling. EPC-Based Modelling of BPEL Processes: a Pragmatic Transformation Approach. In *Proceedings of MITIP 2005, Italy*, 2005.