From Business Process Models to Process-oriented Software Systems: The BPMN to BPEL Way*

Chun Ouyang[1], Marlon Dumas[1], Wil M.P. van der Aalst[2,1], and Arthur H.M. ter Hofstede[1]

[1] Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane QLD 4001, Australia
{c.ouyang,m.dumas,a.terhofstede}@qut.edu.au

[2] Department of Computer Science, Eindhoven University of Technology, GPO Box 513, NL-5600 MB, The Netherlands
{w.m.p.v.d.aalst}@tue.nl

Abstract. Emerging methods for enterprise systems analysis rely on the representation of work practices in the form of business process models. A standard for representing such models is the Business Process Modeling Notation (BPMN). BPMN models are mainly intended for communication and decision-making between domain analysts, but often they are also given as input to software development projects. Meanwhile, development methods for process-oriented systems rely on detailed process definitions that are executed by process engines. These process definitions refine BPMN models by adding data manipulation, application binding and other implementation details. A major standard for process implementation is the Business Process Execution Language for Web Services (BPEL4WS, or BPEL for short). Accordingly, a natural method for end-to-end development of process-oriented systems is to translate BPMN models to BPEL definitions for subsequent refinement. However, instrumenting this method is challenging because BPEL imposes far more syntactic restrictions than BPMN so as to ensure correctness. Existing techniques for translating BPMN to BPEL only work for limited classes of BPMN models. This paper proposes techniques that overcome these limitations. Beyond its direct relevance in the context of BPMN and BPEL, the techniques presented in this paper address issues that arise generally when translating from graphical/unstructured to textual/structured (i.e. more programming-like) languages.

1 Introduction

Business Process Management is an established discipline for building, maintaining and evolving large enterprise systems on the basis of business process models [9]. A business process model is a flow-oriented representation of a set of work practices aimed at achieving a goal, such as processing a customer request or complaint, satisfying a regulatory requirement, etc.

* This work is supported by the Australian Research Council under the Discovery Grant "Expressiveness Comparison and Interchange Facilitation between Business Process Execution Languages" (DP0451092).

The *Business Process Modeling Notation* (BPMN) [19] has attained a certain level of adoption among domain analysts as a language for defining business process models [23]. Despite being a recent proposal, BPMN is already supported by more than 30 tools (see `www.bpmn.org`). The main purpose of business process models generally, and BPMN models in particular, is to facilitate communication between domain analysts as well as strategic decision-making based on techniques such as cost analysis, scenario analysis and simulation [23, 25]. However, oftentimes, BPMN models are also used as a basis for specifying software system requirements, and in such cases, they are handed over to software development teams. In this setting, the motivating question of this paper is: What can developers do with the process models they are handed over?

Meanwhile, the *Business Process Execution Language for Web Services* (BPEL) [8] is emerging as a de facto standard for implementing business processes on top of Web services technology. Numerous platforms (such as Oracle BPEL Process Manager, IBM WebSphere Application Server Enterprise, IBM WebSphere Studio Application Developer Integration Edition, and Microsoft BizTalk Server 2004) support the execution of BPEL processes. Some of these platforms also provide graphical editing tools for defining BPEL processes. However, these tools directly follow the syntax of BPEL, as opposed to a syntax closer to BPMN. The fundamental reason for this phenomenon is that BPEL imposes far more syntactic restrictions than BPMN. A BPMN model consists of nodes that can be connected through flow relations in arbitrary ways. The lack of restrictions makes it possible to obtain models with undesirable execution semantics, such as livelocks and deadlocks. Generally, this is not seen as a problem in the context of BPMN: Since domain analysts are not concerned with the direct executability of their models, one can argue that these users should not be imposed constraints driven by such concerns. On the other hand, BPEL imposes strict syntactic restrictions in order to avoid process definitions with undesirable execution semantics, as executability is the main concern of BPEL. In particular, the majority of constructs in BPEL are block-structured, e.g. block-structured loops. There is also a graph-oriented flow construct, but it must be applied with syntactical limitations such that the arbitrary control flow relations are allowed only when they do not cause cycles and they remain contained within block-structured loops.

The premise of this paper is that BPEL process definitions are refinements of executable BPMN models. Indeed, BPEL process definitions add data manipulation, Web service bindings and other implementation details not specified in the BPMN models. Thus, a natural method to approach a process-oriented systems development project taking BPMN models as input, is to translate these models into BPEL process definitions for subsequent refinement. However, the instrumentation of this method is hindered by the fundamental mismatch between BPMN and BPEL [24]. Previous attempts to define mappings between BPMN and BPEL [16, 19] impose restrictions on the structure of the source models. For example, they are restricted to BPMN models such that every loop has one single entry point and one single exit point and such that each point where the flow of control branches has a corresponding point where the resulting branches merge back, etc.

The ensuing problem is to some extent similar to that of translating unstructured flowcharts into structured ones (or GOTO programs into WHILE programs [20]). However, the main difference is that process modelling languages include constructs for capturing parallel execution, as well as constructs for capturing choices driven by the environment (also called event-driven choices), as opposed to choices driven by data such as those found in flowcharts. It turns out that due to parallelism, the class of structured process models is strictly contained in the class of unstructured process models (assuming the setting described in [12]). This raises the following question:

> Can every BPMN model be translated into a BPEL model?

This paper shows that, for a core subset of BPMN which includes parallelism and event-driven choice, the answer is yes. However, the resulting translation heavily uses a construct in BPEL known as "event handler" which serves to encode event-action rules. Ultimately, the process model is decomposed into a large number of event-action rules that trigger one another to capture the process flow. Arguably, the resulting BPEL code is not readable and thus unsuitable for refinement by developers. It would be preferable that the generated BPEL code used the constructs of BPEL specifically designed for capturing control flow dependencies as opposed to a construct intended for event handling. But since BPEL's control flow constructs are syntactically restricted, this turns out to be not always possible. Therefore, the paper also addresses the following question:

> Are there classes of BPMN models that can be translated to BPEL models using the syntactically constrained control flow constructs of BPEL?

This paper identifies two such classes of BPMN models. The first one corresponds to the class of structured process models as defined in [12]. Such models can be translated into the five structured control flow constructs of BPEL. The second class corresponds to the class of synchronising process models as defined in [11], which can be translated into BPEL using a construct called "control link". A BPMN model, or a fragment thereof, falls under this class if it satisfies certain semantic conditions: no deadlock and no possibility of having multiple parallel instances of the same action. To test these conditions, a formal semantics of BPMN is needed. Accordingly, the paper also defines an abstract syntax and a Petri net-based semantics [17] for a core subset of BPMN. The paper focuses on a subset of BPMN because BPMN contains a large number of constructs, making an exhaustive translation daunting, and because some constructs in BPMN have an incompletely specified meaning.

The paper also shows how the proposed translation techniques can be combined, such that a technique yielding less readable code is only applied when the other techniques can not, and only for model fragments of minimal size. The combined translation technique has been implemented as an open-source tool, namely BPMN2BPEL.[3]

[3] `http://www.bpm.fit.qut.edu.au/projects/babel/tools`

Beyond its direct relevance in the context of BPMN and BPEL, this paper address difficult problems that arise generally when translating between flow-based languages with parallelism. In particular, the main results are still largely applicable if we replace BPMN and BPEL by one of their predecessors, such as UML Activity Diagrams [18] or XLANG [26] respectively, or if we needed to translate graph-oriented models specified using YAWL [4] into readable BPEL.

The rest of the paper is structured as follows: Section 2 gives an overview of BPMN and BPEL. Section 3 defines an abstract syntax and formal semantics for BPMN. Section 4 presents an algorithm for translating BPMN into BPEL. The translation algorithm is then illustrated through case studies in Section 5. Finally, Section 6 compares the proposal with related work while Section 7 concludes and outlines future work.

2 Background: BPMN and BPEL

2.1 Business Process Execution Language for Web Services (BPEL)

BPEL [8] is essentially an extension of imperative programming languages with constructs specific to Web service implementations. A BPEL process definition relates a number of *activities*. An activity is either a basic or a structured activity. *Basic activities* correspond to atomic actions such as: *invoke*, invoking an operation on a Web service; *receive*, waiting for a message from a partner; *exit*, terminating the entire service instance; *empty*, doing nothing; and etc. To enable the presentation of complex structures the following *structured activities* are defined: *sequence*, for defining an execution order; *flow*, for parallel routing; *switch*, for conditional routing; *pick*, for race conditions based on timing or external triggers; *while*, for structured looping; and *scope*, for grouping activities into blocks to which event, fault and compensation handlers may be attached.

An *event handler* is an *event-action rule* associated with a scope. It is enabled when the scope is under execution and may execute concurrently with the scope's main activity. When an occurrence of the event (a message receipt or a timeout) associated with an enabled event handler is registered, the body of the handler is executed. The completion of the scope as a whole is delayed until all active event handlers have completed. *Fault* and *compensation handlers* are designed for exception handling and are not used further in this report.

In addition, BPEL provides a non-structured construct known as *control links* which, together with the associated notions of *join condition* and *transition condition*, allow the definition of directed graphs. The graphs can be nested but must be acyclic. A control link between activities A and B indicates that B cannot start before A has either completed or has been skipped. Moreover, B can only be executed if its associated join condition evaluates to true, otherwise B is skipped. This join condition is expressed in terms of the tokens carried by control links leading to B. These tokens may take either a *positive* (true) or a *negative* (false) value. An activity X propagates a token with a positive value along an outgoing link L if and only if X was executed (as opposed to being skipped) and the transition condition associated to L evaluates to true.

Transition conditions are boolean expressions over the process variables (just like the conditions in a *switch* activity). The process by which positive and negative tokens are propagated along control links, causing activities to be executed or skipped, is called *dead path elimination*.

There are over 20 execution engines supporting BPEL (see `http://en.wikipedia.org/wiki/BPEL` for a list). Many of them come with an associated graphical editing tool. However, the notation supported by these tools directly reflects the underlying code, thus forcing users to reason in terms of BPEL constructs (e.g., block-structured activities and syntactically restricted links). Current practice suggests that the level of abstraction of BPEL is unsuitable for business process analysts and designers. Instead, such users rely on languages perceived as "higher-level" such as BPMN and various flavours of UML diagrams, thus justifying the need for mapping languages such as BPMN into BPEL.

2.2 Business Process Modelling Notation (BPMN)

BPMN [19] essentially provides a graphical notation for business process modelling, with an emphasis on control-flow. It defines a *Business Process Diagram* (BPD), which is a kind of flowchart incorporating constructs tailored to business process modelling, such as AND-split, AND-join, XOR-split, XOR-join, and deferred (event-based) choice.

A BPD is made up of BPMN elements. We consider a core subset of BPMN elements that can be used to build BPDs covering the fundamental control flows in BPMN. These elements are shown in Figure 1. There are *objects* and *sequence flows*. A sequence flow links two objects in a BPD and shows the control flow relation (i.e. execution order). An object can be an *event*, a *task* or a *gateway*. An event may signal the start of a process (*start event*), the end of a process (*end event*), a message that arrives, or a specific time-date being reached during a process (*intermediate message/timer event*). A task is an atomic activity and stands for work to be performed within a process. There are seven task types: *service*, *receive*, *send*, *user*, *script*, *manual*, and *reference*. For example, a receive task is used when the process waits for a message to arrive from an external partner. Also, a task may be none of the above types, which we refer to as

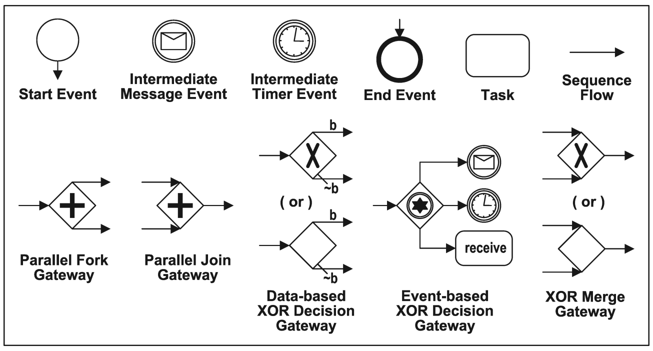


Figure 1. A core subset of BPMN elements.

a *blank* task. A gateway is a routing construct used to control the divergence and convergence of sequence flow. There are: *parallel fork gateways* for creating concurrent sequence flows, *parallel join gateways* for synchronizing concurrent sequence flows, *data/event-based XOR decision gateways* for selecting one out of a set of mutually exclusive alternative sequence flows where the choice is based on either the process data (data-based) or external event (event-based), and *XOR merge gateways* for joining a set of mutually exclusive alternative sequence flows into one sequence flow. In particular, an event-based XOR decision gateway must be followed by either receive tasks or intermediate events to capture race conditions based on timing or external triggers (e.g. the receipt of a message from an external partner).

It is worth noting that for some other BPMN objects such as *error intermediate events* and *OR gateways*, we decide not to include them into the above core subset of BPMN elements, because they do not have clearly specified semantics. For example, an error intermediate event is used either to "throw" an exception during the normal flow or to "catch" an exception thus creating the exception flow. However, it is not clear whether such "throw-catch" behaviour is defined to capture a strictly hierarchical faulting mechanism (e.g., try-catch blocks in most programming languages as well as fault handling in BPEL), or a parallel-thread interruption signaling mechanism (e.g., using an "interrupt" event to communicate exception on one thread to interrupt an activity on another thread). As another example, an *OR merge gateway* (also known as *OR-join*) is defined as to synchronise "all sequence flows that were actually produced by an upstream" (Section 9.5.3 on page 80 of [19]). Such definition is however incomplete, e.g., an OR-join in the context of loops has no clear semantics and leads to a paradox (cf. the "vicious circle" described in [3]).

Finally, a BPD, which is made up of the core subset of BPMN elements shown in Figure 1, is hereafter referred to as a *core BPD*. In the next section, we shall define the syntax and semantics of core BPDs.

3 Abstract Syntax and Semantics of BPMN

In BPMN, business process models are captured as BPDs. In this section, we first define the syntax of core BPDs. We then discuss the transformation of a BPD from a graph structure to a block structure. We use the term "components" to refer to subsets of a BPD, which can be mapped onto suitable "BPEL blocks" (these will be used in Section 4). Finally, this section specifies the semantics of components in terms of Petri nets, and analyses some of their properties.

3.1 Abstract Syntax of BPDs

Definition 1 (Core BPD). *A core BPD is a tuple* $\mathcal{BPD} = (\mathcal{O}, \mathcal{T}, \mathcal{E}, \mathcal{G}, \mathcal{T}^R, \mathcal{E}^S, \mathcal{E}^I, \mathcal{E}^E, \mathcal{E}^I_M, \mathcal{E}^I_T, \mathcal{G}^F, \mathcal{G}^J, \mathcal{G}^D, \mathcal{G}^M, \mathcal{G}^V, \mathcal{F}, \mathsf{Cond})$ *where:*

– $\mathcal{O}$ *is a set of objects which can be partitioned into disjoint sets of tasks* $\mathcal{T}$*, events* $\mathcal{E}$*, and gateways* $\mathcal{G}$*,*

- $\mathcal{T}^R \subseteq \mathcal{T}$ *is a set of receive tasks,*
- $\mathcal{E}$ *can be partitioned into disjoint sets of start events* $\mathcal{E}^S$*, intermediate events* $\mathcal{E}^I$*, and end events* $\mathcal{E}^E$*,*
- $\mathcal{E}^I$ *can be partitioned into disjoint sets of intermediate message events* $\mathcal{E}^I_M$ *and timer events* $\mathcal{E}^I_T$*,*
- $\mathcal{G}$ *can be partitioned into disjoint sets of parallel fork gateways* $\mathcal{G}^F$*, parallel join gateways* $\mathcal{G}^J$*, data-based XOR decision gateways* $\mathcal{G}^D$*, event-based XOR decision gateways* $\mathcal{G}^V$*, and XOR merge gateways* $\mathcal{G}^M$*,*
- $\mathcal{F} \subseteq \mathcal{O} \times \mathcal{O}$ *is the control flow relation,*
- Cond*:* $\mathcal{F} \nrightarrow \mathcal{B}$ *is a function mapping sequence flows emanating from data-based XOR decision gateways to conditions,*[4] *i.e.* $dom(\mathsf{Cond}) = \mathcal{F} \cap (\mathcal{G}^D \times \mathcal{O})$*.*

The relation $\mathcal{F}$ defines a directed graph with nodes (objects) $\mathcal{O}$ and arcs (sequence flows) $\mathcal{F}$. For any node $x \in \mathcal{O}$, input nodes of x are given by $\mathsf{in}(x) = \{y \in \mathcal{O} \mid y\mathcal{F}x\}$ and output nodes of x are given by $\mathsf{out}(x) = \{y \in \mathcal{O} \mid x\mathcal{F}y\}$.

Definition 1 allows for graphs which are unconnected, not having start or end events, containing objects without any input and output, etc. Therefore we need to restrict the definition to *well-formed core BPDs*.

Definition 2 (Well-formed core BPD). *A core BPD given in Definition 1 is well formed if relation* $\mathcal{F}$ *satisfies the following requirements:*

- $\forall\ s \in \mathcal{E}^S$, $\mathsf{in}(s) = \varnothing \wedge \mid \mathsf{out}(s) \mid = 1$, *i.e. start events have an indegree of zero and an outdegree of one,*
- $\forall\ e \in \mathcal{E}^E$, $\mathsf{out}(e) = \varnothing \wedge \mid \mathsf{in}(e) \mid = 1$, *i.e., end events have an outdegree of zero and an indegree of one,*
- $\forall\ x \in \mathcal{T} \cup \mathcal{E}^I$, $\mid \mathsf{in}(x) \mid = 1$ *and* $\mid \mathsf{out}(x) \mid = 1$, *i.e. tasks and intermediate events have an indegree of one and an outdegree of one,*
- $\forall\ g \in \mathcal{G}^F \cup \mathcal{G}^D \cup \mathcal{G}^V$*:* $\mid \mathsf{in}(g) \mid = 1 \wedge \mid \mathsf{out}(g) \mid > 1$, *i.e. fork or decision gateways have an indegree of one and an outdegree of more than one,*
- $\forall\ g \in \mathcal{G}^J \cup \mathcal{G}^M$, $\mid \mathsf{out}(g) \mid = 1 \wedge \mid \mathsf{in}(g) \mid > 1$, *i.e. join or merge gateways have an outdegree of one and an indegree of more than one,*
- $\forall\ g \in \mathcal{G}^V$, $\mathsf{out}(g) \subseteq \mathcal{E}^I \cup \mathcal{T}^R$, *i.e. event-based XOR decision gateways must be followed by intermediate events or receive tasks,*
- $\forall g \in \mathcal{G}^D$, $\exists\, x \in \mathsf{out}(g)$, (g, x) *is a default flow among all the outgoing flows from* g*, i.e. if none of the conditions on the outgoing flows from* g *evaluate to true, then the default flow* (g, x) *will be chosen,*
- $\forall\ x \in \mathcal{O}$, $\exists\ (s, e) \in \mathcal{E}^S \times \mathcal{E}^E$, $s\mathcal{F}^*x \wedge x\mathcal{F}^*e$,[5] *i.e. every object is on a path from a start event to an end event.*

[4] $\mathcal{B}$ is the set of all possible conditions. A condition is a boolean function operating over a set of propositional variables. Note that we abstract from these variables in the control flow definition. We simply assume that a condition evaluates to true or false, which determines whether or not the associated sequence flow is taken during the process execution.

[5] $\mathcal{F}^*$ is a reflexive transitive closure of $\mathcal{F}$, i.e. $x\mathcal{F}^*y$ if there is a path from x to y and by definition $x\mathcal{F}^*x$.

In the remainder we only consider well-formed core BPDs, and will use a simplified notation $\mathcal{BPD} = (\mathcal{O}, \mathcal{F}, \mathsf{Cond})$ for their representation. Moreover, we assume that both $\mathcal{E}^S$ and $\mathcal{E}^E$ are singletons, i.e. $\mathcal{E}^S = \{s\}$ and $\mathcal{E}^E = \{e\}$.[6]

3.2 Decomposing a BPD into Components

We would like to achieve two goals when mapping BPMN onto BPEL. One is to define an algorithm which allows us to translate each well-formed core BPD into a valid BPEL process, the other is to generate readable and compact BPEL code. To map a BPD onto (readable) BPEL code, we need to transform a graph structure into a block structure. For this purpose, we decompose a BPD into *components*. A component is a subset of the BPD that has one entry point and one exit point. We then try to map components onto suitable "BPEL blocks". For example, a component holding a purely sequential structure should be mapped onto a BPEL *sequence* construct while a component holding a parallel structure should be mapped onto a *flow* construct. Below, we formalise the notion of components in a BPD. To facilitate the definitions, we specify an auxiliary function elt over a domain of singletons, i.e., if $X=\{x\}$, then $\mathsf{elt}(X)=x$.

Definition 3 (Component). *Let $\mathcal{BPD} = (\mathcal{O}, \mathcal{F}, \mathsf{Cond})$ be a well-formed core BPD. $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$ is a component of $\mathcal{BPD}$ if and only if:*

- *$\mathcal{O}_c \subseteq \mathcal{O} \backslash (\mathcal{E}^S \cup \mathcal{E}^E)$, i.e., a component does not have any start or end event,*
- *$|\, (\bigcup_{x \in \mathcal{O}_c} \mathsf{in}(x)) \backslash \mathcal{O}_c \,| = 1$, i.e., there is a single entry point outside the component,[7] which can be denoted as $\mathsf{entry}(\mathcal{C}) = \mathsf{elt}((\bigcup_{x \in \mathcal{O}_c} \mathsf{in}(x)) \backslash \mathcal{O}_c)$,*
- *$|\, (\bigcup_{x \in \mathcal{O}_c} \mathsf{out}(x)) \backslash \mathcal{O}_c \,| = 1$, i.e., there is a single exit point outside the component, which can be denoted as $\mathsf{exit}(\mathcal{C}) = \mathsf{elt}((\bigcup_{x \in \mathcal{O}_c} \mathsf{out}(x)) \backslash \mathcal{O}_c)$,*
- *there exists a unique source object $i_c \in \mathcal{O}_c$ and a unique sink object $o_c \in \mathcal{O}_c$ and $i_c \neq o_c$, such that $\mathsf{entry}(\mathcal{C}) \in \mathsf{in}(i_c)$ and $\mathsf{exit}(\mathcal{C}) \in \mathsf{out}(o_c)$,*
- *$\mathcal{F}_c = \mathcal{F} \cap (\mathcal{O}_c \times \mathcal{O}_c)$,*
- *$\mathsf{Cond}_c = \mathsf{Cond}[\mathcal{F}_c]$, i.e. the Cond function where the domain is restricted to $\mathcal{F}_c$.*

Note that all event objects in a component are intermediate events. Also, a component contains at least two objects: the source object and the sink object. A BPD without any component, which is referred to as a *trivial BPD*, has only a single task or intermediate event between the start event and the end event. Translating a trivial BPD into BPEL is straightforward and will be covered by the final translation algorithm (see Section 4.4).

The decomposition of a BPD helps to define an iterative approach which allows us to incrementally transform a "componentized" BPD into a block-structured BPEL process. Below, we define function Fold that replaces a component by a single (blank) task object in a BPD. This function can be used to

[6] A BPD with multiple start events can be transformed into a BPD with a unique start event by using an event-based XOR decision gateway. A BPD with multiple end events can be transformed into a BPD with a unique end event by using an OR-join gateway which is however not covered in this paper.

[7] Note that $\mathsf{in}(x)$ is not defined with respect to the component but refers to the whole BPD. Similarly, this is also applied to $\mathsf{out}(x)$ in the definition.

perform iterative reduction of a componentized BPD until no component is left in the BPD. The function will play a crucial role in the final translation algorithm where we incrementally replace BPD components by BPEL constructs.

Definition 4 (Fold). *Let $\mathcal{BPD} = (\mathcal{O}, \mathcal{F}, \mathsf{Cond})$ be a well-formed core BPD and $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$ be a component of $\mathcal{BPD}$. Function Fold replaces $\mathcal{C}$ in $\mathcal{BPD}$ by a task object $t_c \notin \mathcal{O}$, i.e. $\mathsf{Fold}(\mathcal{BPD}, \mathcal{C}, t_c) = (\mathcal{O}', \mathcal{F}', \mathsf{Cond}')$ with:*

- $\mathcal{O}' = (\mathcal{O} \backslash \mathcal{O}_c) \cup \{t_c\}$,
- *$\mathcal{T}_c$ is the set of tasks in $\mathcal{C}$, i.e. $\mathcal{T}_c = \mathcal{O}_c \cap \mathcal{T}$,*
- *$\mathcal{T}' = (\mathcal{T} \backslash \mathcal{T}_c) \cup \{t_c\}$ is the set of tasks in $\mathsf{Fold}(\mathcal{BPD}, \mathcal{C}, t_c)$,*
- *$\mathcal{T}^{R'} = (\mathcal{T}^R \backslash \mathcal{T}_c)$ is the set of receive tasks in $\mathsf{Fold}(\mathcal{BPD}, \mathcal{C}, t_c)$,*
- $\mathcal{F}' = (\mathcal{F} \cap (\mathcal{O}\backslash\mathcal{O}_c \times \mathcal{O}\backslash\mathcal{O}_c)) \cup \{(\mathsf{entry}(\mathcal{C}), t_c), (t_c, \mathsf{exit}(\mathcal{C}))\}$,
- $\mathsf{Cond}' = \begin{cases} \mathsf{Cond}[\mathcal{F}'] & \textit{if } \mathsf{entry}(\mathcal{C}) \notin \mathcal{G}^D \\ \mathsf{Cond}[\mathcal{F}'] \cup \{((\mathsf{entry}(\mathcal{C}), t_c), \mathsf{Cond}(\mathsf{entry}(\mathcal{C}), i_c))\} & \textit{otherwise} \end{cases}$

3.3 Petri-net Semantics

We use Petri nets [17] to define formal semantics for the core subset of BPMN. Note that the current BPMN specification [19] describes BPMN in natural language and does not contain a formal semantics of BPMN. In the following, we first introduce the basic Petri net terminology and notations. Readers familiar with Petri nets can skip this introduction. Then, as the second part, we define a mapping from a BPD component to Petri nets.

3.3.1 Petri Nets

The classical Petri net is a directed bipartite graph with two types of nodes called *places* and *transitions*. The nodes are connected via directed *arcs*, and connections between two nodes of the same type are not allowed. Places are graphically represented by circles and transitions by rectangles.

Definition 5 (Petri net). *A Petri net is a triple $PN = (P, T, F)$:*

- *P is a finite set of places,*
- *T is a finite set of transitions ($P \cap T = \varnothing$),*
- *$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation).*

A place p is called an *input place* of a transition t iff there exists a directed arc from p to t. Place p is called an *output place* of transition t iff there exists a directed arc from t to p. We use $\bullet t$ to denote the set of input places for a transition t. The notations $t\bullet$, $\bullet p$ and $p\bullet$ have similar meanings, e.g., $p\bullet$ is the set of transitions sharing p as an input place.

At any time a place contains zero or more *tokens*. The *state*, often referred to as *marking*, is the distribution of tokens over places. We will represent a state as follows: $1p_1 + 2p_2 + 0p_3$ is the state with one token in place p_1, two tokens in p_2, and no tokens in p_3. We can also represent this state as $p_1 + 2p_2$. A Petri net PN and its initial marking M are denoted by (PN, M).

The number of tokens may change during the execution of the net. Transitions are the active components in a Petri net: they change the state of the net according to the following *firing rule*:

(1) A transition t is said to be *enabled* iff each input place p of t contains at least one token.
(2) An enabled transition may *fire*. If transition t fires, then t *consumes* one token from each input place p of t and *produces* one token for each output place p of t.

The firing rule specifies how a Petri net can move from one state to another. If at any time multiple transitions are enabled, a non-deterministic choice is made. A firing sequence $\sigma = t_1 t_2 ... t_n$ is enabled if, starting from the initial marking, it is possible to subsequently fire $t_1, t_2, ..., t_n$. A marking M is reachable from the initial marking if there exists an enabled firing sequence resulting in M. Using these notions we define some standard properties for Petri nets.

Definition 6 (Live). *A Petri net (PN, M) is live iff, for every reachable state M' and every transition t, there is a state M'' reachable from M' which enables t.*

Definition 7 (Bounded, safe). *A Petri net (PN, M) is bounded iff for each place p there is a natural number n such that for every reachable state the number of tokens in p is less than n. The net is safe iff for each place the maximum number of tokens does not exceed 1.*

3.3.2 Semantics of Components

Figure 2 depicts the mapping from core BPMN objects to Petri-net modules. It covers all BPMN objects that may be contained within a component of a well-formed core BPD. A task or an intermediate event is transformed into

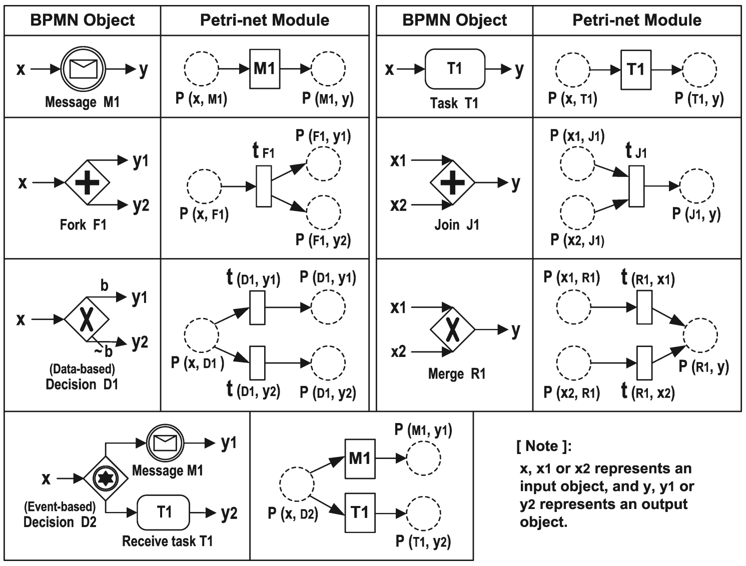


Figure 2. Mapping BPMN objects onto Petri-net modules.

a transition with one input place and one output place. The transition is directly named after the task or event. Gateways are mapped onto small Petri-net modules which describe their routing behaviour explicitly. In particular, for an event-based gateway the race condition between events or receive tasks is captured in such a way that all the corresponding event/task transitions share the same output place of the preceding object of that gateway. Since the transitions appearing in the different mappings of fork, join, data-based decision, and merge gateways are only used to capture the routing behaviour, they can be considered as "silent" transitions [17]. In the case of a fork or join gateway, only one transition is used and thus can be uniquely identified by that gateway. For a data-based decision gateway, multiple transitions are used and each of them is identified by both the gateway and one of the gateway's *output* objects. A merge gateway is also mapped onto a number of transitions, and each of these transitions is identified by both the gateway and one of the gateway's *input* objects. Finally, places are used to link the Petri net modules of two connected BPMN objects and therefore can be identified by both objects. Also, places are drawn in dashed borders to indicate that their usage is not unique to one module. For example, if message event "M1" is followed by task "T1", the output place of transition M1 becomes the input place of transition T1. A formal definition of the mapping from BPD components to Petri nets is given as follows.

Definition 8 (Petri net semantics of components). *Let $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$ be a component of a well-formed core BPD. By using the similar set notations as in the definition of a BPD (see Definition 1), we can write that $\mathcal{O}_c = \mathcal{T}_c \cup \mathcal{E}_c \cup \mathcal{G}_c$ and $\mathcal{G}_c = \mathcal{G}_c^F \cup \mathcal{G}_c^J \cup \mathcal{G}_c^D \cup \mathcal{G}_c^M \cup \mathcal{G}_c^V$. Also, i_c denotes the single source object and o_c denotes the single target object in $\mathcal{C}$. $\mathcal{C}$ can be mapped onto a Petri net $PN_c = (P', T', F')$ where:*

$$
\begin{aligned}
P' = {} & \{p_{(\mathsf{entry}(\mathcal{C}), i_c)}, p_{(o_c, \mathsf{exit}(\mathcal{C}))}\} \cup && \textit{– source/sink place}\\
& \{p_{(x,y)} \mid x\mathcal{F}_c y \wedge x \notin \mathcal{G}_c^V\} && \textit{– other places}
\end{aligned}
$$

$$
\begin{aligned}
T' = {} & \mathcal{T}_c \cup \mathcal{E}_c \cup && \textit{– task/event}\\
& \{t_x \mid x \in \mathcal{G}_c^F \cup \mathcal{G}_c^J\} \cup && \textit{– fork/join}\\
& \{t_{(x,y)} \mid x \in \mathcal{G}_c^D \wedge y \in \mathsf{out}(x)\} \cup && \textit{– data decision}\\
& \{t_{(x,y)} \mid x \in \mathcal{G}_c^M \wedge y \in \mathsf{in}(x)\} \cup && \textit{– merge}
\end{aligned}
$$

$$
\begin{aligned}
F' = {} & \{(p_{(x,y)}, y) \mid y \in \mathcal{T}_c \cup \mathcal{E}_c \wedge x \in \mathsf{in}(y) \wedge x \notin \mathcal{G}_c^V\} \cup && \\
& \{(y, p_{(y,z)})\} \mid y \in \mathcal{T}_c \cup \mathcal{E}_c \wedge z \in \mathsf{out}(y)\} \cup && \textit{– task/event}\\
& \{(p_{(x,y)}, t_y)\} \mid y \in \mathcal{G}_c^F \cup \mathcal{G}_c^J \wedge x \in \mathsf{in}(y)\} \cup && \\
& \{(t_y, p_{(y,z)})\} \mid y \in \mathcal{G}_c^F \cup \mathcal{G}_c^J \wedge z \in \mathsf{out}(y)\} \cup && \textit{– fork/join}\\
& \{(p_{(x,y)}, t_{(y,z)}) \mid y \in \mathcal{G}_c^D \wedge x \in \mathsf{in}(y) \wedge z \in \mathsf{out}(y)\} \cup && \\
& \{(t_{(y,z)}, p_{(y,z)}) \mid y \in \mathcal{G}_c^D \wedge z \in \mathsf{out}(y)\} \cup && \textit{– data decision}\\
& \{(p_{(x,y)}, t_{(y,x)}) \mid y \in \mathcal{G}_c^M \wedge x \in \mathsf{in}(y)\} \cup && \\
& \{(t_{(y,x)}, p_{(y,z)}) \mid y \in \mathcal{G}_c^M \wedge x \in \mathsf{in}(y) \wedge z \in \mathsf{out}(y)\} \cup && \textit{– merge}\\
& \{(p_{(x,y)}, z) \mid y \in \mathcal{G}_c^V \wedge x \in \mathsf{in}(y) \wedge z \in \mathsf{out}(y)\} && \textit{– event decision}
\end{aligned}
$$

In the above definition, generally any sequence flow in a BPD is mapped onto a place except for event-based decision gateways. With an event-based decision gateway (y), the choice is delayed until one of its immediately following events or tasks ($z \in \mathsf{out}(y)$) is triggered. Let x denote the preceding object of y (i.e. $x \in \mathsf{in}(y)$). The place $p_{(x,y)}$, which models the sequence flow from x to y, is directly connected to each transition modelling the event or task z. This way the mapping captures, for an event-based decision gateway, the moment of choice when one of its alternative branches is actually started.

3.4 Soundness and Safeness

The main goal of providing a Petri net mapping for BPMN is that it allows us to discuss the semantics and correctness in a concise and unambiguous manner. The application of the mapping in Definition 8 to a BPD component results in a Petri net satisfying some desirable properties which make automated analysis easier. To capture these properties, it is necessary to introduce the concepts of *WorkFlow nets* (WF-net) [1] and *free-choice nets* [10]. A WF-net is a Petri net which models the control-flow dimension of a workflow, while free-choice nets are an important subclass of Petri nets for which strong theoretical results exist.

Definition 9 (Free-choice WF-net). *A Petri net $PN = (P, T, F)$ is a WF-net (workflow net) if and only if:*

(i) There is one source place $i \in P$ such that $\bullet i = \varnothing$.
(ii) There is one sink place $o \in P$ such that $o \bullet = \varnothing$.
(iii) Every node $x \in P \cup T$ is on a path from i to o.

Also, PN is a free-choice net if and only if for every two transitions $t_1, t_2 \in T$, $\bullet t_1 \cap \bullet t_2 \neq \varnothing$ implies that $\bullet t_1 = \bullet t_2$.

It can be seen that a WF-net has exactly one input place (called *source place*) and one output place (*sink place*). A token in the source place corresponds to a case (i.e. process instance) which needs to be handled, and a token in the sink place corresponds to a case which has been handled. Also, in a WF-net there are no dangling tasks and/or conditions. Tasks are modelled by transitions and conditions by places. Therefore, every transition or place should be located on a path from source place to sink place in a WF-net.

Given a WF-net $PN = (P, T, F)$, we want to decide whether PN is *sound*. *Soundness* is a notion of correctness. A procedure modelled by a WF-net is sound iff it satisfies the following two requirements: (1) *for any case, the procedure will terminate eventually and the moment the procedure terminates there is a token in the sink place and all the other places are empty*;[8] and (2) *there should be no dead tasks, i.e., it should be possible to execute an arbitrary task by following the appropriate route through the WF-net.* In [1] we have shown that soundness corresponds to liveness and boundedness. To link soundness to liveness and boundedness, we define an extended net $\overline{PN} = (\overline{P}, \overline{T}, \overline{F})$. $\overline{PN}$ is the Petri net obtained by adding an additional transition t^* which connects sink place o and source place i. The extended Petri net $\overline{PN} = (\overline{P}, \overline{T}, \overline{F})$ is defined as follows: $\overline{P} = P$, $\overline{T} = T \cup \{t^*\}$, and $\overline{F} = F \cup \{o, t^*), (t^*, i)\}$. We call such an

[8] Sometimes the term "proper termination" is used to describe this requirement.

extended net the *short-circuited* net of PN. Also, we use i to denote the state with only one token in place i and thereby use $(\overline{PN}, i)$ to denote the Petri net $\overline{PN}$ with an initial state i. The short-circuited net allows for the formulation of the following theorem, which shows that standard Petri net-based analysis techniques can be used to verify soundness of WF-nets.

Theorem 1. *A WF-net PN is sound if and only if* $(\overline{PN}, i)$ *is live and bounded.*

Proof. See [1]. □

With Theorem 1 we can also use very efficient analysis techniques. In order to do this, we need to show that any BPD component corresponds to a free-choice net. Free-choice Petri nets have been studied extensively and are characterised by strong theoretical results and efficient analysis techniques. In fact, soundness can be determined in polynomial time for free-choice WF-nets [1]. Moreover, we require a WF-net to be safe, i.e., no marking reachable from (PN, i) marks a place twice. Although safeness is defined with respect to some initial marking, we extend it to WF-nets and simply state that a WF-net is safe or not (given an initial state i). A sound free-choice WF-net is guaranteed to be safe [2].

Theorem 2. *Let* $\mathcal{C}$ *be a component of a well-formed core BPD.* PN_c*, which denotes the Petri net mapping of* $\mathcal{C}$ *as given in Definition 8, is a free-choice WF-net.*

Proof. We first prove that PN_c is a WF-net. There is one source place $p_{(\mathsf{entry}(\mathcal{C}),i_c)}$ and one sink place $p_{(o_c,\mathsf{exit}(\mathcal{C}))}$. Moreover, every node (place or transition) is on a path from $p_{(\mathsf{entry}(\mathcal{C}),i_c)}$ to $p_{(o_c,\mathsf{exit}(\mathcal{C}))}$ since in the corresponding component $\mathcal{C}$ all objects are on a path from the source object to the sink object and all sequence flows (connecting the objects) are preserved by the mapping given in Definition 8.

Next we prove that PN_c is free-choice. Considering places with multiple output arcs, these places all correspond to decision gateways. All the other places have only one output arc (except the sink place $p_{(o_c,\mathsf{exit}(\mathcal{C}))}$ which has none). All outgoing sequence flows of a decision gateway are mapped onto transitions with only one input place. If $PN_c = (P_c, T_c, F_c)$, the above indicates that for all $(p, t) \in F_c$: $|p\bullet| > 1$ implies $|\bullet t| = 1$. Hence, PN_c is free-choice. □

The two theorems in this section demonstrate that results related to safeness and boundedness of free-choice nets can be used to check the soundness of a core BPMN model in polynomial time. This can be formalized as follows. Let $\overline{PN_c}$ be the short-circuited net of PN_c, we use $p_{(\mathsf{entry}(\mathcal{C}),i_c)}$ to denote the state with only one token in the source place $p_{(\mathsf{entry}(\mathcal{C}),i_c)}$ of PN_c and thereby use $(\overline{PN_c}, p_{(\mathsf{entry}(\mathcal{C}),i_c)})$ to denote the Petri net $\overline{PN_c}$ with an initial state $p_{(\mathsf{entry}(\mathcal{C}),i_c)}$. The WF-net PN_c is sound iff $(\overline{PN_c}, p_{(\mathsf{entry}(\mathcal{C}),i_c)})$ is live and bounded (and this can be checked in polynomial time). Moreover, using the results presented in [2], we can also show that if PN_c is sound, it is guaranteed to be safe. This implies that in sound BPMN models an object cannot be activated multiple times for the same instance. We will use this observation in our translation from BPMN to BPEL.

4 Mapping BPMN onto BPEL

This section presents a mapping from BPMN models to BPEL processes. As mentioned before, the basic idea is to map BPD components onto suitable "BPEL blocks" and thereby to incrementally transform a "componentized" BPD into a block-structured BPEL process. We apply *three different approaches* to the mapping of components.[9] A component may be *well-structured* so that it can be directly mapped onto BPEL structured activities. If a component is not well-structured but is acyclic, it may be possible to map the component to control link-based BPEL code. Otherwise, if a component is neither well-structured nor can be translated using control links (e.g. a component that contains unstructured cycles), the mapping of the component will rely on BPEL event handlers via the usage of *event-action rules* (this will always work but the resulting BPEL code will be less readable). We identify the above categories of components and introduce the corresponding translation approaches one by one. Finally, we propose the algorithm for mapping an entire BPD onto a BPEL process.

4.1 Structured Activity-based Translation

As mentioned before, one of our goals for mapping BPMN onto BPEL is to generate readable BPEL code. For this purpose, BPEL structured activities comprising *sequence*, *flow*, *switch*, *pick* and *while*, have the first preference if the corresponding structures appear in the BPD. Components that have a direct and intuitive correspondence to one of these five structured constructs are considered *well-structured*. Below, we classify different types of well-structured components resembling these five structured constructs.

Definition 10 (Well-structured components). *Let $\mathcal{BPD} = (\mathcal{O}, \mathcal{F}, \mathsf{Cond})$ be a well-formed core BPD and $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$ be a component of $\mathcal{BPD}$. i_c is the source object of $\mathcal{C}$ and o_c is the sink object of $\mathcal{C}$. The following components are well-structured:*

(*a*) *$\mathcal{C}$ is a SEQUENCE-component if $\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I$ (i.e. $\forall\, x \in \mathcal{O}_c$, $|\mathsf{in}(x)|$=$|\mathsf{out}(x)|$ $= 1$) and $\mathsf{entry}(\mathcal{C}) \notin \mathcal{G}^V$. $\mathcal{C}$ is a maximal SEQUENCE-component if $\mathcal{C}$ is a SEQUENCE-component and there is no other SEQUENCE-component $\mathcal{C}'$ such that $\mathcal{O}_c \subset \mathcal{O}_{c'}$ where $\mathcal{O}_{c'}$ is the set of objects in $\mathcal{C}'$,*

(*b*) *$\mathcal{C}$ is a FLOW-component if*
- *$i_c \in \mathcal{G}^F \wedge o_c \in \mathcal{G}^J$,*
- *$\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I \cup \{i_c, o_c\}$,*
- *$\forall\, x \in \mathcal{O}_c \backslash \{i_c, o_c\}$, $\mathsf{in}(x) = \{i_c\} \wedge \mathsf{out}(x) = \{o_c\}$.*

(*c*) *$\mathcal{C}$ is a SWITCH-component if*
- *$i_c \in \mathcal{G}^D \wedge o_c \in \mathcal{G}^M$,*
- *$\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I \cup \{i_c, o_c\}$,*
- *$\forall\, x \in \mathcal{O}_c \backslash \{i_c, o_c\}$, $\mathsf{in}(x) = \{i_c\} \wedge \mathsf{out}(x) = \{o_c\}$.*

(*d*) *$\mathcal{C}$ is a PICK-component if*

[9] It should be noted that the first two approaches are inspired by the mapping from Petri nets to BPEL as described in [7, 14].

- $i_c \in \mathcal{G}^V \wedge o_c \in \mathcal{G}^M$,
- $\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I \cup \{i_c, o_c\}$,
- $\forall\ x \in \mathcal{O}_c \backslash (\{i_c, o_c\} \cup \mathsf{out}(i_c))$, $\mathsf{in}(x) \subset \mathsf{out}(i_c) \wedge \mathsf{out}(x) = \{o_c\}$.[10]

(*e*) $\mathcal{C}$ *is a WHILE-component if*
- $i_c \in \mathcal{G}^M \wedge o_c \in \mathcal{G}^D \wedge x \in \mathcal{T} \cup \mathcal{E}^I$,
- $\mathcal{O}_c = \{i_c, o_c, x\}$,
- $\mathcal{F}_c = \{(i_c, o_c), (o_c, x), (x, i_c)\}$.

(*f*) $\mathcal{C}$ *is a REPEAT-component if*
- $i_c \in \mathcal{G}^M \wedge o_c \in \mathcal{G}^D \wedge x \in \mathcal{T} \cup \mathcal{E}^I$,
- $\mathcal{O}_c = \{i_c, o_c, x\}$,
- $\mathcal{F}_c = \{(i_c, x), (x, o_c), (o_c, i_c)\}$.

(*g*) $\mathcal{C}$ *is a REPEAT+WHILE-component if*
- $i_c \in \mathcal{G}^M \wedge o_c \in \mathcal{G}^D \wedge x_1, x_2 \in \mathcal{T} \cup \mathcal{E}^I \wedge x_1 \neq x_2$,
- $\mathcal{O}_c = \{i_c, o_c, x_1, x_2\}$,
- $\mathcal{F}_c = \{(i_c, x_1), (x_1, o_c), (o_c, x_2), (x_2, i_c)\}$.

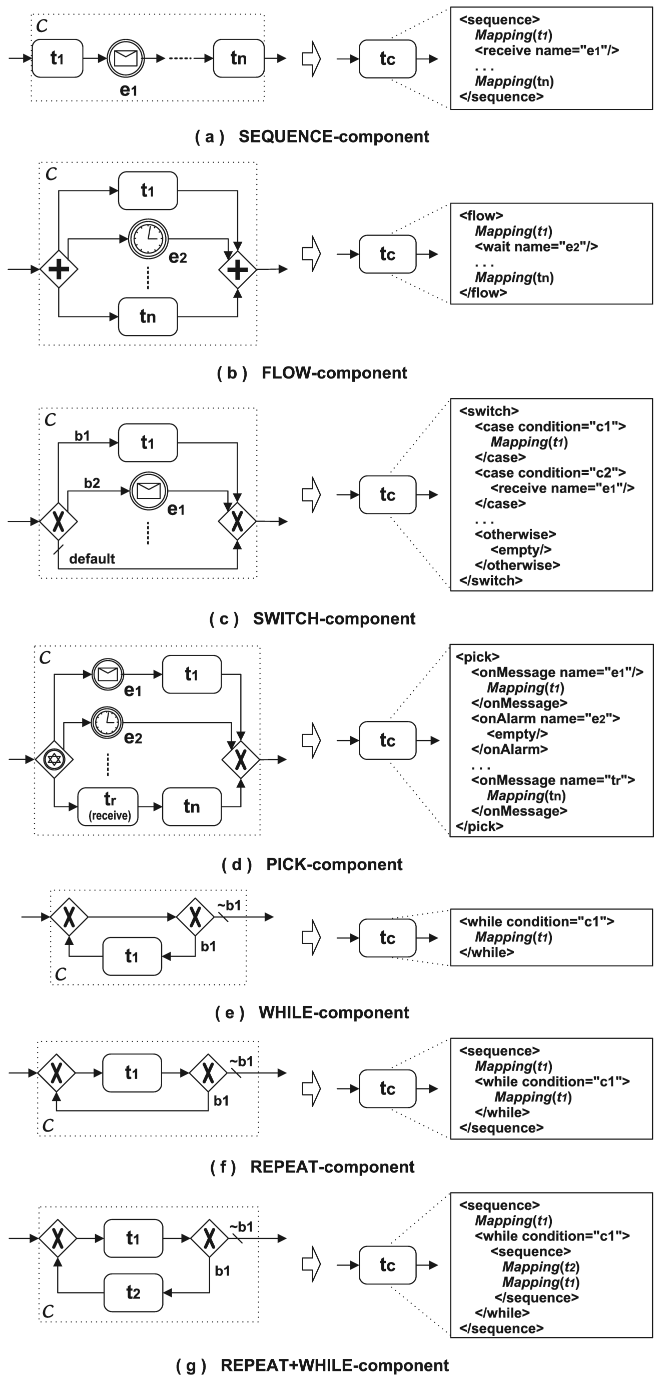
Figure 3 illustrates how to map each of the components mentioned above onto the corresponding BPEL structured activities. Using function Fold in Definition 4, a component $\mathcal{C}$ is replaced by a single task t_c attached with the corresponding BPEL translation of $\mathcal{C}$. Note that the BPEL code for the mapping of each task t_i ($i = 1, ..., n$) is denoted as $\mathsf{Mapping}(t_i)$. Based on the nature of these task objects they are mapped onto the proper types of BPEL activities. For example, a *service* task is mapped onto an invoke activity, a *receive* task (like t_r in Figure 3(d)) is mapped onto a receive activity, and a *user* task may be mapped onto an invoke activity followed by a receive activity[11]. Also, a task t_i may result from the folding of a previous component $\mathcal{C}'$, in which case, $\mathsf{Mapping}(t_i)$ is the code for the mapping of component $\mathcal{C}'$.

In Figure 3(a) to (e), the mappings of the five components, SEQUENCE, FLOW, SWITCH, PICK and WHILE, are straightforward. Note that in a PICK-component (Figure 3(d)), an event-based XOR decision gateway must be followed by receive tasks or intermediate message or timer events. For this reason, a SEQUENCE-component (Figure 3(a)) cannot be preceded by an event-based XOR decision gateway (as defined by $\mathsf{entry}(\mathcal{C}) \notin \mathcal{G}^V$ in Definition 10(a)).

In Figure 3(f) and (g), the two components, REPEAT and REPEAT+WHILE, represent repeat loops. A while loop (see Figure 3(e)) evaluates the loop condition *before* the body of the loop is executed, so that the loop is not executed if the condition is initially false. In a repeat loop, the condition is checked *after* the body of the loop is executed, so that the loop is always executed at least once. In Figure 3(f), a repeat loop of task t_1 is equivalent to a single execution

[10] Note that $\mathsf{out}(i_c) \subseteq \mathcal{T}^R \cup \mathcal{E}^I$ is the set of receive tasks and intermediate events following the event-based XOR decision gateway i_c. Between the merge gateway o_c and each of the objects in $\mathsf{out}(i_c)$ there is at most one task or event object.

[11] Since the goal of this paper is to define an approach for translating BPDs with arbitrary topologies to valid BPEL processes, we do not discuss further how to map simple tasks in BPMN onto BPEL. Interested reader may refer to [19] for some guidelines on mapping BPMN tasks into BPEL activities.



```
<sequence>
  Mapping(t1)
  <receive name="e1"/>
  . . .
  Mapping(tn)
</sequence>
```

```
<flow>
  Mapping(t1)
  <wait name="e2"/>
  . . .
  Mapping(tn)
</flow>
```

```
<switch>
   <case condition="c1">
      Mapping(t1)
   </case>
   <case condition="c2">
      <receive name="e1"/>
   </case>
   . . .
   <otherwise>
      <empty/>
   </otherwise>
</switch>
```

```
<pick>
   <onMessage name="e1"/>
      Mapping(t1)
   </onMessage>
   <onAlarm name="e2">
      <empty/>
   </onAlarm>
   . . .
   <onMessage name="tr">
      Mapping(tn)
   </onMessage>
</pick>
```

```
<while condition="c1">
  Mapping(t1)
</while>
```

```
<sequence>
  Mapping(t1)
  <while condition="c1">
     Mapping(t1)
  </while>
</sequence>
```

```
<sequence>
  Mapping(t1)
  <while condition="c1">
     <sequence>
        Mapping(t2)
        Mapping(t1)
     </sequence>
  </while>
</sequence>
```

Figure 3. Mapping a well-structured component $\mathcal{C}$ onto a BPEL structured activity and folding $\mathcal{C}$ into a single task object t_c attached with the BPEL code for mapping.

of t_1 followed by a while loop of t_1. In Figure 3(g), a repeat loop of task t_1 is combined with a while loop of task t_2, and both loops share one loop condition. Task t_1 is always executed once before the initial evaluation of the condition, which is followed by a while loop of sequential execution of t_2 and t_1.

4.2 Control Link-based Translation

Since BPMN is a graph-oriented language in which nodes can be connected almost arbitrarily, a BPD may contain non-well-structured components, i.e. components that do not match any of the "patterns" given in Definition 10. Recall that BPEL provides a non-structured construct called control link, which allows for the definition of directed acyclic graphs and thus can be used for the translation of a large subset of acyclic BPMN components. We use the term "control link-based flow construct" to refer to a flow activity in which all sub-activities are connected through links to form directed acyclic graphs.

In addition to the above, it is important to emphasize the following two issues related to link semantics. First, the use of control links may hide errors such as deadlocks.[12] This means that the designer makes a modelling error that in other languages would result in a deadlock, however, given the dead-path elimination semantics of BPEL the error is masked. Second, since an activity cannot "start until the status of all its incoming links has been determined and the join condition associated with the activity has been evaluated" (Section 12.6.1 of [8]), each execution of an activity can trigger at most one execution of any subsequent activity to which it is connected through a control link. Taking into account these two issues, one needs to ensure that an acyclic component is sound (e.g. deadlock-free) and safe (i.e. each activity is executed at most once) before attempting to translate it into a graph of BPEL activities connected through control links.

For example, Figure 4 depicts two acyclic components that are not well-structured. The one shown in (a) can be mapped onto a control link-based flow construct without any problem. However, for the one shown in (b), if condition b does not hold at the data-based decision gateway D2, task T4 will never be performed, causing the component to deadlock at the join gateway J4 (which requires that both tasks T3 and T4 are executed). Also, task T3 will be executed twice, a behaviour that cannot be represented using control links which only trigger the target activity at most once. In Petri nets terminology, such a component cannot be qualified as being "sound" and "safe".

4.2.1 Components for Control Link-based Translation

We use the term "synchronising process component" to refer to a component that can be mapped to a control link-based flow construct preserving the same

[12] While control links do not create deadlocks, they can lead to models where one or several actions are "unreachable", which means that the action in question will never be executed. See [22] for examples of such undesirable models.

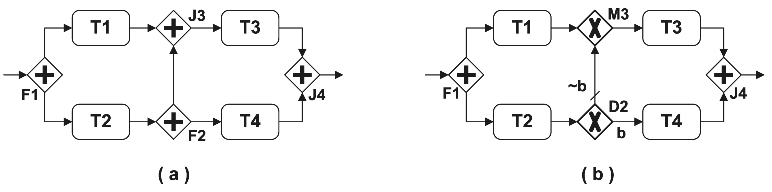


Figure 4. Two non-well-structured acyclic components.

semantics. In synchronising process models, "an activity can receive two types of tokens, a true token or a false token. Receipt of a true token enables the activity, while receipt of a false token leads to the activity being skipped and the token to be propagated" [11]. This way the semantics of control links are well captured and thus the activities can be viewed as being connected via control links.

Definition 11 (Synchronising process component). *Let $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$ be a component of a well-formed core BPD. $\mathcal{C}$ is a synchronising process component if it satisfies all the following three conditions.*
(a) *There are no cycles (i.e., $\forall\, x \in \mathcal{O}_c$, $(x, x) \notin \mathcal{F}^*$);*
(b) *There are no event-based gateways (i.e., if $\mathcal{G}^V$ denotes the set of event-based gateways in the BPD, then $\mathcal{O}_c \cap \mathcal{G}^V = \varnothing$); and*
(c) *PN_c, which denotes the Petri net mapping of $\mathcal{C}$ as given in Definition 8, is a safe and sound free-choice WF-net.*

Note that some well-structured components such as SEQUENCE-component, FLOW-component and SWITCH-component are also synchronising process components. When mapping a BPD onto BPEL we will always try to use the structured activity-based translation described in Section 4.1, until there are no well-structured components left in the BPD. Therefore, the control link-based translation only applies to a subset of synchronising process components that are not well-structured. In addition, we will always try to detect a *minimal* synchronising process component for translation. A synchronising process component $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$ is *minimal* if there is no any other component $\mathcal{C}' = (\mathcal{O}_{c'}, \mathcal{F}_{c'}, \mathsf{Cond}_{c'})$ such that $\mathcal{O}_{c'} \subset \mathcal{O}_c$. It is easy to discover that such a component always starts with a fork or data-based decision gateway and ends with a join or merge gateway, given the fact that there are no well-structured components left (they have been iteratively removed).

4.2.2 Control Link-based Translation Algorithm

The basic idea behind this algorithm is to translate the control-flow relation between all task and event objects within a synchronising process component into a set of control links. Before translation, it is necessary to pre-process the component in the following two steps. First, as aforementioned, a minimal synchronising process component always has a gateway as its source or sink object. Since control links connects only task/event objects, it is necessary to insert an empty task (i.e. a task of doing nothing) before the source gateway object of the component, and to insert an empty task after the sink gateway object of the component. We call the resulting component a *wrapped component*.

Definition 12 (Wrapped component). *Let $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$ be a component of a well-formed core BPD. By inserting an empty task a_h before the source object i_c of $\mathcal{C}$ and an empty task a_t after the sink object o_c of $\mathcal{C}$, we obtain the wrapped component of $\mathcal{C}$ as being the component $\mathcal{C}' = (\mathcal{O}_{c'}, \mathcal{F}_{c'}, \mathsf{Cond}_{c'})$ where:*
- *$\mathcal{O}_{c'} = \mathcal{O}_c \cup \{a_h, a_t\}$,*
- *$\mathcal{F}_{c'} = \mathcal{F}_c \cup \{(a_h, i_c), (o_c, a_t)\}$,*
- *$\mathsf{Cond}_{c'} = \mathsf{Cond}_c$*

Next, the BPMN specification states that the conditional branches of a data-based decision gateway "should be evaluated in a specific order" (Section 9.5.2 on page 72 of [19]). In more detail, "the first one that evaluates as TRUE will determine the Sequence Flow that will be taken. Since the behavior of this Gateway is exclusive, any other conditions that may actually be TRUE will be ignored". Also, the default branch, which is always the last branch considered, will be chosen if none of the other branches evaluate to true (see Definition 2). When using control links to replace a data-based decision gateway, we need to ensure that the above semantics of the gateway are preserved. This can be done by refining the conditions on each of the outgoing flows of a data-based decision gateway. We use $\{f_1, ..., f_n\}$ to denote the set of outgoing flows from a data-based decision gateway and use $\mathsf{Cond}(f_i)$ $(1 \leqslant i \leqslant n)$ to denote the condition on flow f_i. Assume that $\mathsf{Cond}(f_i)$ is evaluated in the order from f_1 to f_n, and f_n is the default branch. The refined condition on flow f_i is given by

$$\mathsf{RefinedCond}(f_i) = \begin{cases} \mathsf{Cond}(f_1) & i = 1 \\ \neg(\mathsf{Cond}(f_1) \wedge ... \wedge \mathsf{Cond}(f_{i-1})) \wedge \mathsf{Cond}(f_i) & 1 < i < n \\ \neg(\mathsf{Cond}(f_1) \wedge ... \wedge \mathsf{Cond}(f_{n-1})) & i = n \end{cases}$$

Note that it is easy to prove that the above pre-processing of a synchronising process component will not change the nature of the component, i.e. the resulting component still satisfies the three requirements given in Definition 11.

We now derive from the structure of a pre-processed synchronising component, the set of control links used to connect all tasks and events in the component. First, we would like to capture the control flow logic between every two task/event objects that are directly or indirectly (via gateways) connected within the component. To this end, we define two functions as shown in Figure 5. One named PreTEC-Sets (line 1), takes an object x and generates the set of sets each containing the preceding tasks, events, and/or conditions[13] for x by relying on the other function named PreTEC-SetsFlow (lines 2-17). This function produces the same type of output as PreTEC-Sets but takes as input a flow rather than an object. It operates based on the type of the source object of the flow. If the flow's source is a task or an event, a set is returned containing a singleton set of that task or event (line 5). Otherwise, if the flow's source is a gateway, the algorithm keeps working backwards through the component, traversing other gateways, until reaching a task or an event. In particular, if a flow originates from a data-based decision gateway, the (refined) condition on the flow is added

[13] These are the conditions specified on the outgoing flows of data-based decision gateways.

to each of the set elements in the resulting set (line 11). This captures the fact that the condition specified on an outgoing flow of a data-based decision gateway is part of each trigger that enables the corresponding object following the gateway. In the case of a flow originating from a merge or a join gateway, the function is recursively called for each of the flows leading to this gateway. For a merge gateway, the union of the resulting sets captures the fact that when *any* of these flows is taken, the gateway may be executed (line 14). Similarly, for a join gateway, the cartesian product of the resulting sets captures the fact that when *all* of these flows are taken, the gateway may be executed (line 16).

```
1:  function PreTEC-Sets($x$: Object) = $\bigcup_{y \in \mathsf{in}(x)}$ PreTEC-SetsFlow($y, x$)
2:  function PreTEC-SetsFlow($y$: Object, $x$: Object)
3:    begin
4:      if $y$ is a task or an event then
5:        PreTEC-SetsFlow := $\{\{y\}\}$
6:      else
7:        if $y$ is a fork gateway then
8:          PreTEC-SetsFlow := PreTEC-Sets($y$)
9:        else
10:         if $y$ is a data-based decision gateway then
11:           PreTEC-SetsFlow := AddCond(RefinedCond($y, x$), PreTEC-Sets($y$))
12:         else
13:           if $y$ is a merge gateway then
14:             PreTEC-SetsFlow := $\bigcup_{z \in \mathsf{in}(y)}$ PreTEC-SetsFlow($z, y$)
15:           else // i.e. $y$ is a join gateway
16:             PreTEC-SetsFlow := $\prod_{z \in \mathsf{in}(y)}$ PreTEC-SetsFlow($z, y$)
17:   end
18: // Note the above function makes use of the following auxiliary function:
19: // AddCond($b, \{p_1, p_2, ..., p_n\}$) = $\{p_1 \cup \{b\}, p_2 \cup \{b\}, ..., p_n \cup \{b\}\}$
```

Figure 5. Two functions for deriving the set of preceding tasks and/or events sets for an object within a pre-processed synchronising process component.

Based on the above, Figure 6 defines an algorithm which takes an input of a pre-processed synchronising process component $\mathcal{C}$, and produces the set of control links with their associated transition conditions (given by TransCond) for connecting all the tasks and events in $\mathcal{C}$ and also the join conditions for each of these tasks and events (given by JoinCond). A transition condition associated with a control link is a boolean expression that functions as a guard on the link, and a join condition associated with a task/event object is specified as a boolean expression over the status of each of the incoming links to the object. In Figure 6, the algorithm begins by collecting the set of tasks and events in component $\mathcal{C}$ (line 5). In the set returned by function PreTEC-Sets for each task/event object x, each element p is a set containing task/event objects and optionally conditions. To this end, we define function TE-SetIn (line 8) which extracts the task/event objects from p. Since each element a in TE-SetIn(p) is a preceding task/event object of x, we define a link $l_{a,x}$ connecting a (source object) and x (target object). The transition condition of $l_{a,x}$ is specified as a conjunction of all the conditions appearing in element p (line 9). The join condition is computed for each task/event object x. On the one hand, since all the preceding task/event objects in $\mathsf{TE\text{-}SetIn}(p)$ capture *one* possible way to reach x, *all* the resulting incoming links to x must carry a true token to enable x and thereby a conjunction of these link status is applied. On the other hand, since each element p in $\mathsf{PreTEC\text{-}Sets}(x)$ represents one possible way to reach x, the join condition for x is then defined as a disjunction of the above results (line 10).

```
1:  input: $\mathcal{C}$: pre-processed synchronising process component
2:  output: TransCond: Set(Link, Bool-Expr);
3:           JoinCond: Set(Object, Bool-Expr)
4:  begin
5:    let $\mathcal{TE}_c$ = the set of tasks and events in $\mathcal{C}$
6:    for each $x \in \mathcal{TE}_c$
7:      for each $p \in \mathsf{PreTEC\text{-}Sets}(x)$
8:        $\mathsf{TE\text{-}SetIn}(p) := \bigcup_{te \in p \cap \mathcal{TE}_c} te$
9:        $\mathsf{TransCond} := \mathsf{TransCond} \cup (\bigcup_{a \in \mathsf{TE\text{-}SetIn}(p)} \{(l_{a,x}, \bigwedge_{c \in p \setminus \mathsf{TE\text{-}SetIn}(p)} c)\})$
10:     $\mathsf{JoinCond} := \mathsf{JoinCond} \cup \{(x, \bigvee_{p \in \mathsf{PreTE\text{-}Sets(x)}} (\bigwedge_{a \in \mathsf{TE\text{-}SetIn(p)}} l_{a,x}))\}$
11: end
```

Figure 6. Algorithm for deriving the set of control links with their associated transition conditions and the join conditions for each of the tasks and events within a pre-processed synchronising process component.

Finally, the set of control links with their associated transition conditions and the join conditions for each of the task/event objects derived from the above, can be specified using the BPEL syntax thus generating the target process definition. For example, assume that both task objects x and y are mapped onto *invoke* activities. The definition of link $l_{x,y}$ can be BPEL-encoded as follows:

```
<flow>
   <links>
      <!-- declaration of all control links -->
      <link name="l_{x,y}" condition="TransCond(l_{x,y})"/>
      ...
   </links>
   <!-- all activities (tasks and events) -->
   <invoke name="x" ...>
      <!-- if there are any incoming links to x -->
      <target .../>
      ...
      <!-- all outgoing links from x -->
      <source linkName="l_{x,y}"/>
      ...
   </invoke>
   <invoke name="y" joinCondition="JoinCond(y)">
      <!-- all incoming links to y -->
      <target linkName="l_{x,y}"/>
      ...
```

```
        <!-- if there are any outgoing links from y -->
        <source .../>
        ...
     </invoke>
     ...
  </flow>
```

A detailed example of applying control link-based algorithm to the translation of a BPMN process model is given in Section 5.2.

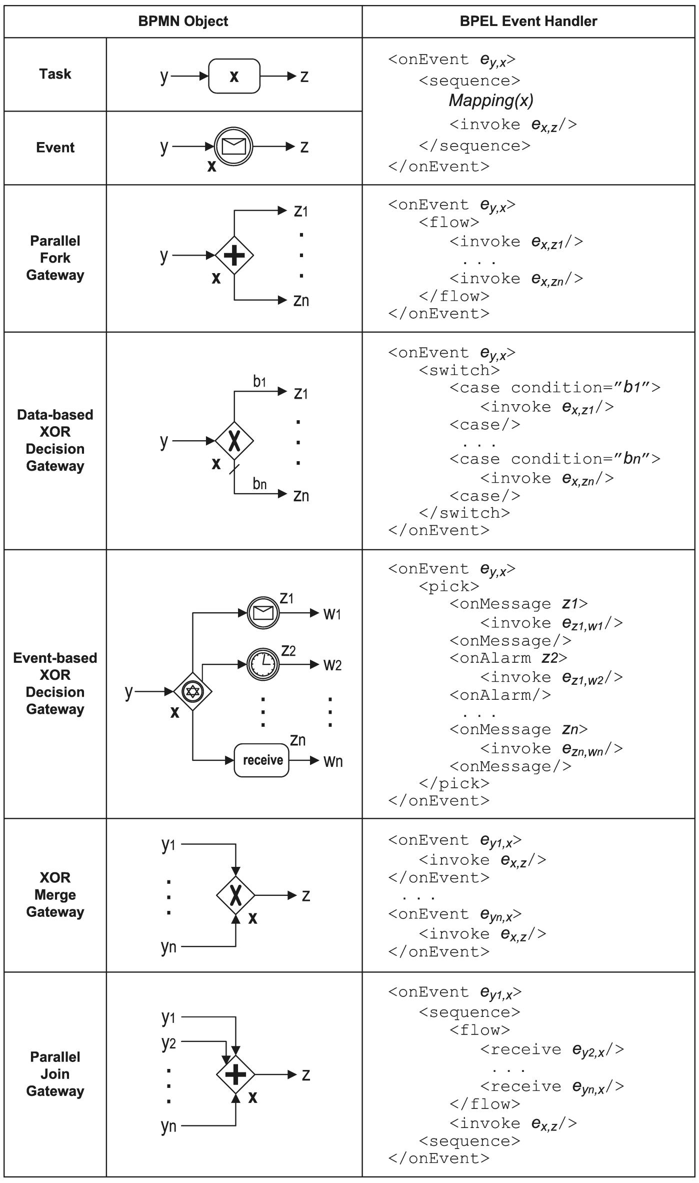
It is important to mention the interplay between the structured activity-based approach (Section 4.1) and the control link-based approach for translating BPDs into BPEL. First the structured activity-based translation is applied iteratively. If there are no longer well-structured components, the control link-based translation is used. Applying the control link-based translation may again enable a structured activity-based translation, etc. Hence it is possible that both types of reductions alternate. Unfortunately, there are BPDs that cannot be translated into BPEL using these two approaches. The next subsection shows a "brute force" approach that can be used as a last resort.

4.3 Event-Action Rule-based Translation

A well-formed core BPD may also contain components that are neither well-structured nor can be translated using control links, e.g. components capturing multi-merge patterns [5] or unstructured loops. We present an approach that can be used to translate such component into a *scope* activity by exploiting the "event handler" construct in BPEL. Since an event handler is an *event-action rule* associated with a scope, we name the approach, in a more general sense, *event-action rule-based translation approach*. It should be mentioned that this approach can be applied to translating any component to BPEL. However, it produces less readable BPEL code and hence we resort only to this approach when there are no components left in the BPD, which are either well-structured or which can be translated using control links.

The basic idea behind the event-action rule-based approach is to map each object (task, event or gateway) onto event handler(s). An incoming flow of the object captures the occurrence of an "event" that triggers the corresponding event handler. The actions taken by the event handler must ensure to invoke "events", which signal the completion of the object being executed, in the correct logic order. Note that the "events" we mention here are events within the context of event-action rules, and thus are different from BPMN event objects.

Figure 7 illustrates how to map each type of BPMN objects onto BPEL event handlers. We use $e_{y,x}$ to denote an "event" captured by the sequence flow connecting from object y to object x. This "event" signals the completion of object y so that the execution of object x may start. In more detail, each task, event, fork gateway, or decision gateway object is mapped onto one event handler, which is triggered upon the occurrence of the "event" from the only incoming flow of the object. For a task or event object x, the resulting event handler first executes that task or event, whose mapping is denoted as $\mathsf{Mapping}(x)$, and



BPMN Object		BPEL Event Handler
Task	y → x → z	`<onEvent e_{y,x}>` `<sequence>` *Mapping(x)* `<invoke e_{x,z}/>` `</sequence>` `</onEvent>`
Event	y → x → z	
Parallel Fork Gateway	y → x → z1 . . . zn	`<onEvent e_{y,x}>` `<flow>` `<invoke e_{x,z1}/>` `. . .` `<invoke e_{x,zn}/>` `</flow>` `</onEvent>`
Data-based XOR Decision Gateway	y → x; b1 → z1 . . . bn → zn	`<onEvent e_{y,x}>` `<switch>` `<case condition="b1">` `<invoke e_{x,z1}/>` `<case/>` `. . .` `<case condition="bn">` `<invoke e_{x,zn}/>` `<case/>` `</switch>` `</onEvent>`
Event-based XOR Decision Gateway	y → x; z1 → w1, z2 → w2, . . . , zn (receive) → wn	`<onEvent e_{y,x}>` `<pick>` `<onMessage z1>` `<invoke e_{z1,w1}/>` `<onMessage/>` `<onAlarm z2>` `<invoke e_{z1,w2}/>` `<onAlarm/>` `. . .` `<onMessage zn>` `<invoke e_{zn,wn}/>` `<onMessage/>` `</pick>` `</onEvent>`
XOR Merge Gateway	y1 . . . yn → x → z	`<onEvent e_{y1,x}>` `<invoke e_{x,z}/>` `</onEvent>` `. . .` `<onEvent e_{yn,x}>` `<invoke e_{x,z}/>` `</onEvent>`
Parallel Join Gateway	y1, y2 . . . yn → x → z	`<onEvent e_{y1,x}>` `<sequence>` `<flow>` `<receive e_{y2,x}/>` `. . .` `<receive e_{yn,x}/>` `</flow>` `<invoke e_{x,z}/>` `<sequence>` `</onEvent>`

Figure 7. Mapping BPMN objects onto BPEL event handlers.

then invokes the "event", which signals the completion of the execution of x and is captured by the only outgoing flow of x. For a fork or decision gateway, the corresponding event handler invokes the "events" to be captured by their outgoing flows in a logic order as defined by the gateway. To this end, the *flow* activity is used for the mapping of a fork gateway, *switch* is used for the mapping of a data-based decision gateway, and *pick* is used for the mapping of an event-based decision gateway with the immediately followed events and/or receive tasks. Next, a merge gateway is mapped onto multiple event handlers in a way that each of them can be triggered upon the occurrence of the "event" from *one* of the multiple incoming flows of the gateway. Finally, for a join gateway, the mapping is less straightforward because BPEL only supports the situation where an event handler is triggered by the occurrence of a *single* event. As shown in Figure 7, a join gateway x can be mapped onto one event handler by separating the "event" ($e_{y_1,x}$) on the first incoming flow from those ($e_{y_2,x}$, ..., $e_{y_n,x}$) on the rest of the incoming flows. Although the resulting event handler can be triggered by the occurrence of $e_{y_1,x}$, the real action, i.e. invoking $e_{x,z}$, will not be performed until all the remaining "events" $e_{y_2,x}$ to $e_{y_n,x}$ have occurred.

Based on the above, we can map a component $\mathcal{C}$ onto a BPEL scope. The scope has a number of event handlers as mappings of all the objects in $\mathcal{C}$. Let $e_{\mathsf{entry}(C),i_c}$ denote the "event" captured by the sequence flow connecting to the source object i_c from the entry point $\mathsf{entry}(\mathcal{C})$ outside the component $\mathcal{C}$. The resulting scope may be encoded as follows:

```
<scope>
   <onEvent $e_{\mathsf{entry}(\mathcal{C}),i_c}$> . . . </onEvent>
   . . .
   <onEvent ...> . . . </onEvent>
   <invoke $e_{\mathsf{entry}(\mathcal{C}),i_c}$/>
</scope>
```

The main activity of the scope is to invoke "event" $e_{\mathsf{entry}(\mathcal{C}),i_c}$. The occurrence of $e_{\mathsf{entry}(\mathcal{C}),i_c}$ triggers the execution of the source object of $\mathcal{C}$, and the entire scope completes after its main activity and all active event handlers have completed.

Finally, it should be mentioned that the above events for triggering event handlers are performed by an invoke activity via a *local* partner link between the final BPEL process (i.e. mapping of the BPD to which component $\mathcal{C}$ belongs) and itself. The interested reader may refer to [21] for definitions of a local partner link and an event being invoked or consumed via a local partner link.

4.4 Overall Translation Algorithm

Based on the mapping of each of the components aforementioned, we define an algorithm for translating a well-formed core BPD into BPEL. Figure 8 shows this algorithm which takes a well-formed core BPD with one start event and one end event, and produces the corresponding BPEL process. For such a BPD Q, we use $\mathcal{O}_Q$ to denote the set of objects in Q, and $[Q]_c$ the set of components in Q. If Q is a trivial BPD, its mapping to BPEL is straightforward (lines 5-9). Otherwise, for a non-trivial BPD, the basic idea is to select a component in the BPD, provide its BPEL translation, and fold the component (lines 10-28). This is repeated until no component is left in the BPD, and the resulting BPEL process definition is then given by the mapping of the task object t_c^i that is created from the last folding (line 29).

More specifically, for a non-trivial BPD, the component mapping always starts from a maximal SEQUENCE-component after each folding (lines 13 to 15). When there are no sequences left in the BPD, other well-structured components

```
1: input: $Q$: a well-formed core BPD with one start event and one end event
2: output: $P$: String
3: begin
4:    let $s :=$ start event of $Q$; $e :=$ end event of $Q$
5:    if $[Q]_c = \varnothing$
6:    then // Mapping of a trivial BPD
7:       $x := \mathcal{O}_Q \backslash \{s, e\}$
8:       BPELcode := basic activities corresponding to $x$
9:       $P :=$ <process> BPELcode </process>
10:   else // Mapping a non-trivial BPD
11:      $i := 0$; Mapping := {}
12:      while $[Q]_c \neq \varnothing$ do
13:         if $\exists$ a maximal SEQUENCE-component $\mathcal{C} \in [Q]_c$
14:         then BPELcode := sequence activity corresponding to $\mathcal{C}$
15:                          according to Figure 3(a) in Section 4.1
16:         else if $\exists$ a well-structured (non-sequence) component $\mathcal{C} \in [Q]_c$
17:              then BPELcode := structured activity corresponding to $\mathcal{C}$
18:                               according to Figure 3(b) to (g) in Section 4.1
19:              else select a minimal non-well-structured component $\mathcal{C} \in [Q]_c$
20:                 if $\mathcal{C}$ is a synchronising process component
21:                 then BPELcode := control link-based flow construct corresponding
22:                                  to $\mathcal{C}$ as specified in Section 4.2.2
23:                 else BPELcode := scope activity with event handlers corresponding
24:                                  to $\mathcal{C}$ as specified in Section 4.3
25:         $i := i + 1$
26:         create a new task object $t_c^i$
27:         Mapping := Mapping $\cup \{(t_c^i,$ BPELcode$)\}$
28:         $Q :=$ Fold$(Q, \mathcal{C}, t_c^i)$
29:      $P :=$ <process> Mapping$(t_c^i)$ </process>
30: end
```

Figure 8. Algorithm for translating a well-formed core BPD into a BPEL process.

are processed (lines 16 to 18). Since all well-structured non-sequence components are disjoint, the order of mapping these components is irrelevant. Next, when no well-structured components are left, the algorithm selects a *minimal* non-well-structured component for translation (line 19). Note that $\mathcal{C}$ is a *minimal* non-well-structured component, if within the same BPD there is no other component $\mathcal{C}'$ such that the set of nodes in $\mathcal{C}'$ is a subset of the set of nodes in $\mathcal{C}$. The algorithm selects a *minimal* non-well-structured component $\mathcal{C}$ and not a maximal one to avoid missing any "potential" well-structured component that may appear after the folding of $\mathcal{C}$. This means that there is always a preference for smaller structured activities rather than large flows. The algorithm then checks if $\mathcal{C}$ is a synchronising process component so that the control link-based translation approach can be applied (lines 20-22). Otherwise, the event-action rule-based translation approach is used as a last resort (lines 23-24). Using the event-action rule-based translation only as a last resort, reflects the desire to produce readable BPEL code. In most cases, event-action rule-based translations can be avoided or play a minor part in the translation. This is illustrated by the examples in the next section and by empirical studies [14].

5 Case Studies

This section provides two examples of business processes modelled using BPMN. We show how these two models can be translated into BPEL using the algorithm presented in the previous section.

5.1 Example 1: Complaint Handling Process

Consider the complaint handling process model shown in Figure 9. It is described as a well-formed core BPD. First the complaint is registered (task *register*), then in parallel a questionnaire is sent to the complainant (task *send questionnaire*) and the complaint is processed (task *process complaint*). If the complainant returns the questionnaire in two weeks (event *returned-questionnaire*), task *process questionnaire* is executed. Otherwise, the result of the questionnaire is discarded (event *time-out*). In parallel the complaint is evaluated (task *evaluate*). Based on the evaluation result, the processing is either done or continues to task *check processing*. If the check result is not OK, the complaint requires re-processing. Finally, task *archive* is executed. Note that labels *DONE*, *CONT*, *OK* and *NOK* on the outgoing flows of each data-based XOR decision gateway, are abstract representations of conditions on these flows.

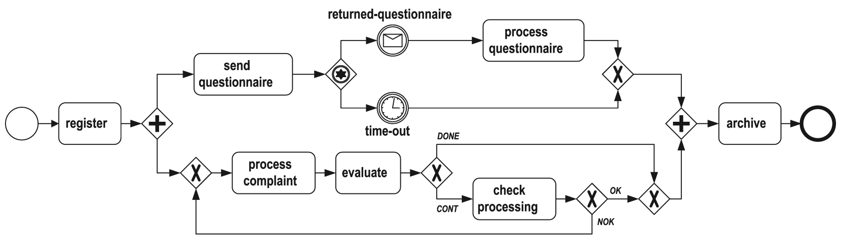


Figure 9. A complaint handling process model.

Following the algorithm in Section 4, we now translate the above BPD to BPEL. Figure 10 sketches the translation procedure which shows how this BPD can be reduced to a trivial BPD. Six components are identified. Each component is named C_i where i specifies in what order the components are processed, and C_i is folded into a task object named t_c^i. Also, we assign an identifier a_i to each task or intermediate event and an identifier g_i to each gateway in the initial BPD. We use these identifiers to refer to the corresponding objects in the following translation. It should be mentioned that since we focus on the control-flow perspective, the resulting BPEL process definition will be presented in simplified BPEL syntax which defines the control flow for the process but omits all details related to data definitions such as partners, messages and variables.

1st Translation. The algorithm first tries to locate SEQUENCE-components. In the initial BPD shown in Figure 9, the component C_1 consisting of tasks a_6 and a_7 is the only SEQUENCE-component that can be identified. Hence, C_1 is folded into a task t_c^1 attached with the BPEL translation sketched as:

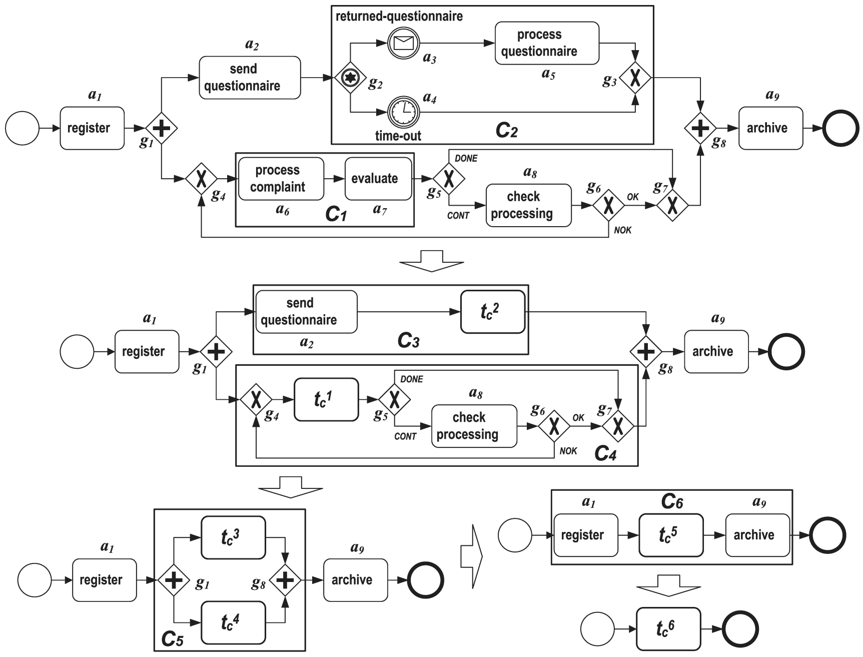


Figure 10. Translating the complaint handling process model in Figure 9 into BPEL.

```
<sequence name="$t_c^1$">
   <invoke name="process complaint".../>
   <invoke name="evaluate".../>
</sequence>
```

2nd Translation. When no SEQUENCE-components can be identified, the algorithm tries to discover any well-structured non-sequence component. As a result, the component C_2 is selected. It is a PICK-component and is folded into a task t_c^2 attached with the BPEL translation sketched as:

```
<pick name="$t_c^2$">
   <onMessage operation="returned-questionnaire"...>
      <invoke name="process questionnaire".../>
   </onMessage>
   <onAlarm for='P14DT'>
      <empty/>
   </onAlarm>
</pick>
```

Assume that the maximal waiting period for the returned questionnaire is two weeks, i.e. 14 days. In BPEL, this is encoded as P14DT.

3rd Translation. Folding C_2 into t_c^2 introduces a new SEQUENCE-component C_3 consisting of tasks a_2 and t_c^2. C_3 is folded into a task t_c^3 attached with the BPEL translation sketched as:

```
<sequence name="$t_c^3$">
   <invoke name="send questionnaire".../>
   <pick name="$t_c^2$"> ... </pick>
</sequence>
```

4th Translation. After the above three components C_1 to C_3 have been folded into the corresponding tasks t_c^1 to t_c^3, there is no well-structured components left in the BPD. The algorithm continues to identify any minimal non-well-structured component. As a result, the component C_4 is selected. Since C_4 contains cycles, it is not a synchronising process component. Below, we map C_4 onto a scope with event handlers.

```
<scope name="$t_c^4$">
   <!-- mapping of $g_4$ -->
   <onEvent $e_{g_1,g_4}$>
      <invoke $e_{g_4,t_c^1}$/>
   </onEvent>
   <onEvent $e_{g_6,g_4}$>
      <invoke $e_{g_4,t_c^1}$/>
   </onEvent>
   <!-- mapping of $t_c^1$ -->
   <onEvent $e_{g_4,t_c^1}$>
      <sequence>
         <sequence name="$t_c^1$"> ... </sequence>
         <invoke $e_{t_c^1,g_5}$/>
      </sequence>
   </onEvent>
   <!-- mapping of $g_5$ -->
   <onEvent $e_{t_c^1,g_5}$>
      <switch>
         <case condition="DONE">
            <invoke $e_{g_5,g_7}$/>
         </case>
         <case condition="CONT">
            <invoke $e_{g_5,a_8}$/>
         </case>
      </switch>
   </onEvent>
   <!-- mapping of $a_8$ -->
   <onEvent $e_{g_5,a_8}$>
      <sequence>
         <invoke name="check processing".../>
         <invoke $e_{a_8,g_6}$/>
      </sequence>
   </onEvent>
   <!-- mapping of $g_6$ -->
   <onEvent $e_{a_8,g_6}$>
```

```
      <switch>
         <case condition="OK">
            <invoke $e_{g_6,g_7}$/>
         </case>
         <case condition="NOK">
            <invoke $e_{g_6,g_4}$/>
         </case>
      </switch>
   </onEvent>
   <!-- mapping of $g_7$ -->
   <onEvent $e_{g_5,g_7}$>
      <invoke $e_{g_7,g_8}$/>
   </onEvent>
   <onEvent $e_{g_6,g_7}$>
      <invoke $e_{g_7,g_8}$/>
   </onEvent>
   <!-- to trigger source object $g_4$ -->
   <invoke $e_{g_1}$/>
</scope>
```

5th Translation. Folding C_3 to t_c^3 and C_4 to t_c^4 introduces a FLOW-component C_5. C_5 is folded into a task t_c^5 attached with the BPEL code sketched as:

```
<flow name="$t_c^5$">
   <sequence name="$t_c^3$"> ... </sequence>
   <scope name="$t_c^4$"> ... </scope>
</flow>
```

6th Translation. After C_5 has been folded into t_c^5, a new SEQUENCE-component C_6 is introduced. This is also the only component left between the start event and the end event in the BPD. Folding C_6 into task t_c^6 leads to the end of the translation, and the final BPEL process is sketched as:

```
<process name="complaint handling">
   <sequence name="$t_c^6$">
      <invoke name="register">
      <flow name="$t_c^5$"> ... </flow>
      <invoke name="archive">
   </sequence>
</process>
```

5.2 Example 2: Order Fulfillment Process

Figure 11 depicts an order fulfillment process at the customer side using BPMN. The process starts by making a choice between two conditional branches, depending on whether the shipper supports the Universal Business Language (UBL) or the Electronic Data Interchange (EDI) standard. The choice between these two standards is exclusive and EDI is always the default one to choose. If UBL is used, the process needs to receive both the despatch advice and the invoice from the shipper before it can continue. Alternatively, if EDI is used, the process needs to receive both EDI 856 for the Advanced Shipment Notice (ASN) and EDI 810 for the Invoice before it can proceed. Next, upon the receipt of either EDI 810 or the invoice (formatted in UBL), a payment request can be sent to the shipper. Once the payment request has been sent out and either EDI 856 or the despatch advice (formatted in UBL) has been received, the customer sends the fulfillment notice and by then the process completes.

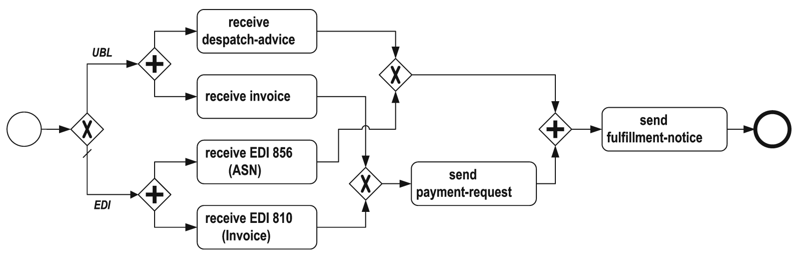


Figure 11. An order fulfillment process model.

Figure 12 sketches how the above BPD can be reduced to a trivial BPD. Two components are identified. Below, we describe the translation step by step.

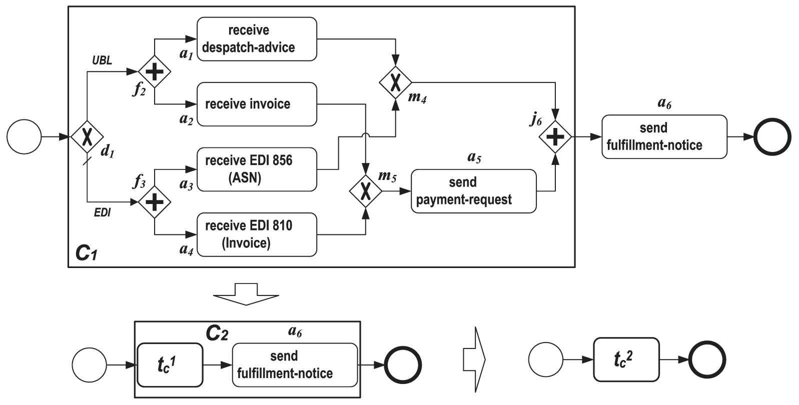


Figure 12. Translating the order fulfillment process model in Figure 11 into BPEL.

1st Translation. Initially, no well-structured components can be detected in the BPD shown in Figure 11. The component C_1 consisting of tasks a_1 to a_5 is the only minimal non-well-structured component identified. It is acyclic and has no event-based gateway. Figure 13 shows the Petri net mapping of C_1, which is proven to be sound and safe. Thus, the component C_1 is a synchronising process component and can be mapped to a control link-based BPEL flow construct.

First, we pre-process the component C_1 as illustrated in Figure 14. Two empty tasks a_h and a_t are inserted respectively before the data-based decision gateway d_1 (source object of C_1) and after the join gateway j_6 (sink object of C_1).

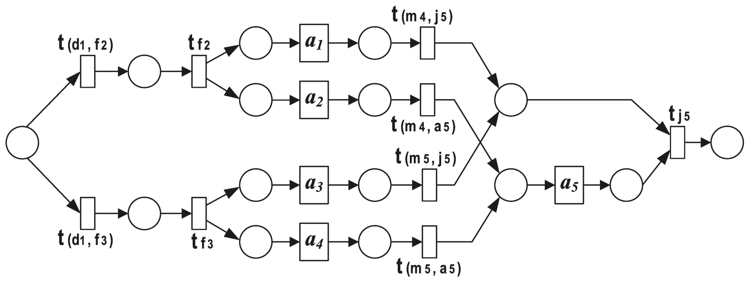


Figure 13. The Petri net mapping of component C_1 shown in Figure 12.

Also, the conditions on the outgoing flows of d_1 are refined. The component C_1, after the above pre-processing, is then renamed C_1'.

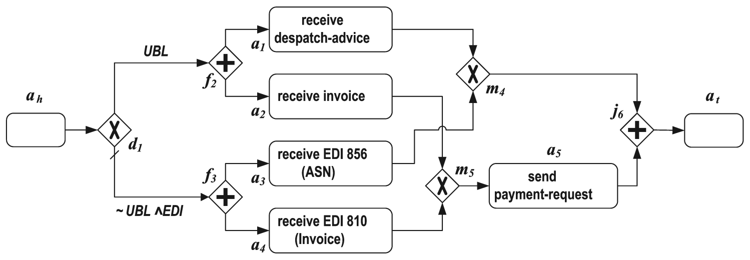


Figure 14. Pre-processing the component C_1 shown in Figure 12.

Second, we generate the set of preceding tasks for each task object in component C_1' (C_1' has no event objects). There are totally seven sets as listed below:

$\mathsf{PreTEC\text{-}Sets}(a_h) = \varnothing$,
$\mathsf{PreTEC\text{-}Sets}(a_1) = \mathsf{PreTEC\text{-}Sets}(a_2) = \{\{a_h,\ \mathit{UBL}\}\}$,
$\mathsf{PreTEC\text{-}Sets}(a_3) = \mathsf{PreTEC\text{-}Sets}(a_4) = \{\{a_h,\ \neg \mathit{UBL} \wedge \mathit{EDI}\}\}$,
$\mathsf{PreTEC\text{-}Sets}(a_5) = \{\{a_2\},\ \{a_4\}\}$, and
$\mathsf{PreTEC\text{-}Sets}(a_t) = \{\{a_1,\ a_5\},\ \{a_3,\ a_5\}\}$

Third, we can then derive the set of control links with their associated transition conditions for connecting all the tasks in component C_1' and the join conditions for each of these tasks. These are:

$$\begin{aligned}\mathsf{TransCond}=\{&(l_{a_h,a_1}, \mathit{UBL}), (l_{a_h,a_2}, \mathit{UBL}), (l_{a_h,a_3}, \neg\mathit{UBL}\wedge\mathit{EDI}), (l_{a_h,a_4}, \neg\mathit{UBL}\wedge\mathit{EDI}),\\ &(l_{a_2,a_5}, \mathtt{TRUE}), (l_{a_4,a_5}, \mathtt{TRUE}), (l_{a_1,a_t}, \mathtt{TRUE}), (l_{a_3,a_t}, \mathtt{TRUE}), (l_{a_5,a_t}, \mathtt{TRUE})\}\end{aligned}$$

$$\begin{aligned}\mathsf{JoinCond}=\{&(a_h, \mathtt{TRUE}), (a_1, l_{a_h,a_1}), (a_2, l_{a_h,a_2}), (a_3, l_{a_h,a_3}), (a_4, l_{a_h,a_4}),\\ &(a_5, l_{a_2,a_5} \vee l_{a_4,a_5}), (a_t, (l_{a_1,a_t} \wedge l_{a_5,a_t}) \vee (l_{a_3,a_t} \wedge l_{a_5,a_t}))\}\end{aligned}$$

Note that task a_h is the source object of component C_1' and has no incoming links. Hence, $\mathsf{JoinCond}(a_h) = \mathtt{TRUE}$ implies that no join condition needs to be specified for a_h in the corresponding BPEL definition.

Finally, based on the above, component C_1 can be folded into a task t_c^1 attached with the BPEL translation sketched as:

```
<flow name="$t_c^1$">
   <links>
      <link name="$l_{a_h,a_1}$" condition="$\mathit{UBL}$"/>
      <link name="$l_{a_h,a_2}$" condition="$\mathit{UBL}$"/>
      <link name="$l_{a_h,a_3}$" condition="$\neg\mathit{UBL}\wedge\mathit{EDI}$"/>
      <link name="$l_{a_h,a_4}$" condition="$\neg\mathit{UBL}\wedge\mathit{EDI}$"/>
      <link name="$l_{a_2,a_5}$" condition="TRUE"/>
      <link name="$l_{a_4,a_5}$" condition="TRUE"/>
      <link name="$l_{a_1,a_t}$" condition="TRUE"/>
      <link name="$l_{a_3,a_t}$" condition="TRUE"/>
      <link name="$l_{a_5,a_t}$" condition="TRUE"/>
   </links>
   <empty name="$a_h$">
      <source linkName="$l_{a_h,a_1}$"/>
      <source linkName="$l_{a_h,a_2}$"/>
      <source linkName="$l_{a_h,a_3}$"/>
      <source linkName="$l_{a_h,a_4}$"/>
   </empty>
   <invoke name="receive despatch-advice"
           joinCondition="bpws:getLinkStatus($l_{a_h,a_1}$)">
      <target linkName="$l_{a_h,a_1}$"/>
      <source linkName="$l_{a_1,a_t}$"/>
   </invoke>
   <invoke name="receive invoice"
           joinCondition="bpws:getLinkStatus($l_{a_h,a_2}$)">
      <target linkName="$l_{a_h,a_2}$"/>
      <source linkName="$l_{a_2,a_5}$"/>
   </invoke>
   <invoke name="receive EDI 856"
           joinCondition="bpws:getLinkStatus($l_{a_h,a_3}$)">
      <target linkName="$l_{a_h,a_3}$"/>
      <source linkName="$l_{a_3,a_t}$"/>
   </invoke>
   <invoke name="receive EDI 810"
           joinCondition="bpws:getLinkStatus($l_{a_h,a_4}$)">
      <target linkName="$l_{a_h,a_4}$"/>
      <source linkName="$l_{a_4,a_5}$"/>
   </invoke>
   <invoke name="send payment-request"
           joinCondition="bpws:getLinkStatus($l_{a_2,a_5}$) or
                          bpws:getLinkStatus($l_{a_4,a_5}$)">
      <target linkName="$l_{a_2,a_5}$"/>
      <target linkName="$l_{a_4,a_5}$"/>
      <source linkName="$l_{a_5,a_t}$"/>
   </invoke>
   <empty name="$a_t$"
          joinCondition="(bpws:getLinkStatus($l_{a_1,a_t}$) and
                          bpws:getLinkStatus($l_{a_5,a_t}$)) or
                         (bpws:getLinkStatus($l_{a_3,a_t}$) and
                          bpws:getLinkStatus($l_{a_5,a_t}$))">
      <target linkName="$l_{a_1,a_t}$"/>
      <target linkName="$l_{a_3,a_t}$"/>
      <target linkName="$l_{a_5,a_t}$"/>
   </empty>
</flow>
```

2nd Translation. After C_1 has been folded into t_c^1, a new SEQUENCE-component C_2 is introduced. This is also the only component left between the start event and the end event in the BPD. Folding C_2 into task t_c^6 leads to the end of the translation, and the final BPEL process is sketched as:

```
<process name="order fulfillment">
   <sequence name="$t_c^2$">
      <flow name="$t_c^1$"> ... </flow>
      <invoke name="send fulfillment-notice">
   </sequence>
</process>
```

6 Related Work

White [19,27] informally sketches a translation from BPMN to BPEL. However, as acknowledged in [19] this translation is fundamentally limited, e.g. it excludes diagrams with arbitrary topologies. Specifically, [19] states that acyclic graphs with arbitrary topologies could be translated to control links, which is however left as future work. Also, a method for translating some types of unstructured cycles is outlined, but no automated and general method is given. In addition, several steps in White's translation require human input to identify patterns in the source model. A detailed review of White's translation from BPMN to BPEL can be found in [21].

Research into structured programming in the 60s and 70s led to techniques for translating unstructured flowcharts into structured ones. However, these techniques are not applicable when AND-splits and AND-joins are introduced. An identification of situations where unstructured process diagrams cannot be translated into equivalent structured ones (under weak bisimulation equivalence) can be found in [12,15], while an approach to overcome some of these limitations for processes without parallelism is sketched in [13]. However, these related work only address a piece of the puzzle of translating from graph-oriented process modelling languages to BPEL.

This paper combines insights from two of our previous studies. In [6], we describe a case study where the requirements of a bank system are captured as Coloured Workflow nets (a subclass of Coloured Petri nets) and the system is then implemented in BPEL. In this study we use a semi-automated mapping from Coloured Petri nets to BPEL that has commonalities with a subset of the translation discussed in this paper. This mapping has been implemented in a tool called Workflownet2BPEL4WS [7, 14] and is also supported by recent versions of ProM[14] Importantly, this tool does not attempt to automatically map every Coloured Petri net. Instead, it maps as many fragments of the net as it can using a pre-defined library of patterns. When the tool can not apply any of the available patterns, the user must intervene to identify at least one remaining fragment that can be translated by some means, possibly leading to a new translation pattern being added into the library. The approach has been empirically tested on 100 real-life Protos[15] process models created in student projects [14]. Meanwhile, in [21] we present a mapping from a graph-oriented language supporting AND-splits, AND-joins, XOR-splits, and XOR-joins, into sets of BPEL event handlers. In this paper, we have extended this previous mapping to cover a broader set of BPMN constructs and to improve the readability of the generated code. Whereas in [21] the generated code relies heavily on BPEL event handlers, in this paper we make greater use of BPEL's block-structured constructs and control links.

In parallel with our work, Mendling et al. [16] have developed four strategies to translate from graph-oriented process modelling languages (such as BPMN) to BPEL. Firstly, the so-called *Structure-Identification Strategy* works like the *structured activity-based translation approach* presented in this paper. Next, the *Element-Preservation Strategy* and the *Element-Minimization Strategy* translate acyclic graph-oriented models into BPEL process definitions with control links. But the authors simply point out that the two strategies can be applied to the transformation of acyclic graphs, while in this paper, we have formally characterised the set of acyclic process models that can be mapped to BPEL using control links. Also, the *Element-Preservation Strategy* generates BPEL processes that contain many unnecessary empty activities. The *Element-Minimization Strategy* then tries to mitigate this problem but it does not completely eliminate all these empty activities. Meanwhile, our proposal for translating synchronising BPMN process models into BPEL control links does not generate such unnecessary empty activities. Finally, the *Structure-Maximization Strategy* tries to derive a BPEL process with as many structured activities as possible and for the remaining unstructured fragments, it tries to apply the strategies that rely on control links. None of these four strategies permit the translation of processes with arbitrary cycles as we do in our proposal.

7 Conclusion

This paper presented an integrated set of techniques to translate models captured using a core subset of BPMN into BPEL. The proposed techniques are capable of

[14] The ProM framework offers a wide range of tools related to process mining (`http://www.processmining.org`).

[15] Protos is a process modelling tool developed by Pallas Athena (`http://uk.pallas-athena.com/`).

generating readable BPEL code by discovering “patterns” in the BPMN models that can be mapped onto BPEL block-structured constructs or acyclic graphs of control links. One of the techniques can deal with unstructured BPMN models by translating the control dependencies in the BPMN model into a collection of BPEL event handlers that trigger one another to emulate these dependencies. This latter technique enables any core BPMN process model to be translated into BPEL, but at the price of reduced readability. The integration of the proposed techniques is therefore defined in a way that maximises the use of structured BEL constructs and minimises the use of event handlers. The integrated technique has been implemented as an open-source tool, namely BPMN2BPEL, available at `http://www.bpm.fit.qut.edu.au/projects/babel/tools`. Testing has been performed against the case studies presented in this paper as well as examples extracted from the BPMN standard specification. The correctness of the generated BPEL process definitions has been validated by loading them into the Oracle BPEL Process Manager (version 10.1.2)[16].

A possible avenue for future work is to extend the proposed techniques to cover a larger subset of BPMN models, e.g. models involving exception handling and other advanced constructs such as OR-joins. Unfortunately, many advanced constructs of BPMN are under-specified and are still being refined by the relevant standardisation body. A preliminary step to extend the translation is therefore to unambiguously define these constructs, for example by extending the Petri net semantics of core BPMN models defined in this paper (e.g. using YAWL as an intermediate step).

The work reported in this paper is motivated by the fact that business process models, while primarily intended for process documentation, communication and improvement, are often also used as input for developing process-oriented software systems. Thus a translation between BPMN models and languages used by developers, e.g. BPEL, is a first step in instrumenting end-to-end methods for this class of systems. But as the BPEL process definition is modified during implementation, inconsistencies may arise between the original business process models and the implemented process definitions. To tackle this issue, it would be desirable to have reversible transformations, so that the modified BPEL models can be viewed in BPMN and any deviations with respect to the original BPMN model can be easily identified. We conjecture that for the class of structured and synchronising process models, such reversible transformations are possible. However, characterising larger classes of BPMN models for which reversible transformations can be defined is a challenging problem. In addition, defining the notion of “reversibility” in the context of BPMN-to-BPEL translations may prove to be a challenge on its own.

Acknowledgments. The authors are extremely grateful to Stephan Breutel for his efforts in implementing the BPMN2BPEL tool. Moreover, we thank Kristian Bisgaard Lassen for developing and implementing some of the ideas in a pure Petri net setting and implementing this in the context of CPN Tools and ProM.

[16] `http://www.oracle.com/technology/products/ias/bpel/`

References

1. W.M.P. van der Aalst. Verification of workflow nets. In P. Azéma and G. Balbo, editors, *Proceedings of 18th International Conference on Application and Theory of Petri Nets*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426, Toulouse, France, June 1997. Springer-Verlag.
2. W.M.P. van der Aalst. Workflow verification: Finding control-flow errors using Petri-net-based techniques. In *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer-Verlag, 2000.
3. W.M.P. van der Aalst, J. Desel, and E. Kindler. On the Semantics of EPCs: A Vicious Circle. In M. Nüttgens and F.J. Rump, editors, *Proceedings of the EPK 2002: Business Process Management using EPCs*, pages 71–80, Trier, Germany, November 2002. Gesellschaft für Informatik, Bonn.
4. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2004.
5. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.
6. W.M.P. van der Aalst, J.B. Jørgensen, and K.B. Lassen. Let's go all the way: From requirements via colored workflow nets to a BPEL implementation of a new bank system. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, volume 3760 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 2005.
7. W.M.P. van der Aalst and K.B. Lassen. Translating workflow nets to BPEL4WS. BETA Working Paper Series, WP 145, Eindhoven University of Technology, Eindhoven, 2005.
8. A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu, editors. *Web Services Business Process Execution Language Version 2.0.* Committee Draft. WS-BPEL TC OASIS, December 2005. Available via `http://www.oasis-open.org/committees/download.php/16024/`.
9. J. Becker, M. Kugeler, and M. Rosemann, editors. *Process Management. A Guide for the Design of Business Processes.* Springer-Verlag, 2003.
10. J. Desel and J. Esparza, editors. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, UK, 1995.
11. B. Kiepuszewski, A.H.M. ter Hofstede, and W.M.P. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, 2003.
12. B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On structured workflow modelling. In *Proceedings of 12th International Conference on Advanced Information Systems Engineering (CAiSE 2000)*, volume 1789 of *Lecture Notes in Computer Science*, pages 431–445, London, UK, 2000. Springer-Verlag.
13. J. Koehler and R. Hauser. Untangling unstructured cyclic flows - A solution based on continuations. In R. Meersman, Z. Tari, W.M.P. van der Aalst, C. Bussler, and A. Gal et al., editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004*, volume 3290 of *Lecture Notes in Computer Science*, pages 121–138, 2004.
14. K.B. Lassen and W.M.P. van der Aalst. WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL. Accepted for

14th International Conference on Copperative Information Systems (CoopIS 2006), also published as BETA Working Paper Series, WP 167, Eindhoven University of Technology, Eindhoven, 2006.

15. R. Liu and A. Kumar. An analysis and taxonomy of unstructured workflows. In *Proceedings of the International Conference on Business Process Management (BPM2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 268–284, Nancy, France, 2005. Springer-Verlag.
16. J. Mendling, K.B. Lassen, and U. Zdun. Transformation strategies between block-oriented and graph-oriented process modelling languages. In F. Lehner, H. Nösekabel, and P. Kleinschmidt, editors, *Multikonferenz Wirtschaftsinformatik 2006. Band 2*, pages 297–312. GITO-Verlag, Berlin, Germany, 2006.
17. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
18. OMG. *Unified Modeling Language: Superstructure.* UML Superstructure Specification v2.0, formal/05-07-04. OMG, August 2005. Available via `http://www.omg.org/cgi-bin/doc?formal/05-07-04`.
19. OMG. *Business Process Modeling Notation (BPMN) Version 1.0.* OMG Final Adopted Specification. OMG, February 2006. Available via `http://www.bpmn.org/`.
20. G. Oulsnam. Unravelling unstructured programs. *Computer Journal*, 25(3):379–387, 1982.
21. C. Ouyang, M. Dumas, S. Breutel, and A.H.M. ter Hofstede. Translating Standard Process Models to BPEL. In *Proceedings of 18th International Conference on Advanced Information Systems Engineering (CAiSE 2006)*, volume 4001 of *Lecture Notes in Computer Science*, pages 417–432, Luxembourg, 2006. Springer-Verlag. An extended version as a technical report is available via `http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-27.pdf`.
22. C. Ouyang, H.M.W. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, and A.H.M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. Technical Report BPM-05-15, BPMcenter.org, September 2005. Available via `http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-15.pdf`.
23. J. Recker, M. Indulska, M. Rosemann, and P. Green. Do process modelling techniques get better? A comparative ontological analysis of BPMN. In D. Bunker B. Campbell, J. Underwood, editor, *Proceedings of the 16th Australasian Conference on Information Systems.* Australasian Chapter of the Association for Information Systems, Sydney, Australia, 2005.
24. J. Recker and J. Mendling. On the translation between BPMN and BPEL: Conceptual mismatch between process modeling languages. In T. Latour and M. Petit, editors, *Proceedings of Workshops and Doctoral Consortium for the 18th International Conference on Advanced Information Systems Engineering.* Namur University Press, Luxembourg, Grand-Duchy of Luxembourg, 2006.
25. M. Rosemann. Preparation of Process Modeling. In J. Becker, M. Kugeler, and M. Rosemann, editors, *Process Management. A Guide for the Design of Business Processes*, pages 41–78. Springer-Verlag, 2003.
26. S. Thatte. *XLANG Web Services for Business Process Design*, 2001.
27. S. White. Using BPMN to Model a BPEL Process. *BPTrends*, 3(3):1–18, March 2005.