From Business Process Models to Process-oriented Software Systems*

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

[1] Faculty of Information Technology, Queensland University of Technology,
GPO Box 2434, Brisbane QLD 4001, Australia
{c.ouyang,m.dumas,a.terhofstede,j.mendling}@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. Several methods for enterprise systems analysis rely on flow-oriented representations of business operations, otherwise known as business process models. The Business Process Modeling Notation (BPMN) is a standard for capturing such models. BPMN models facilitate communication between domain experts and analysts and provide input to software development projects. Meanwhile, there is an emergence of methods for enterprise software development that rely on detailed process definitions that are executed by process engines. These process definitions refine their counterpart BPMN models by introducing data manipulation, application binding and other implementation details. The de facto standard for defining executable processes is the Business Process Execution Language (BPEL). Accordingly, a standards-based method for developing process-oriented systems is to start with BPMN models and to translate these models into BPEL definitions for subsequent refinement. However, instrumenting this method is challenging because BPMN models and BPEL definitions are structurally very different. Existing techniques for translating BPMN to BPEL only work for limited classes of BPMN models. This paper proposes a translation technique that does not impose structural restrictions on the source BPMN model. At the same time, the technique emphasizes the generation of readable (block-structured) BPEL code. An empirical evaluation conducted over a large collection of process models shows that the resulting BPEL definitions are largely block-structured. Beyond its direct relevance in the context of BPMN and BPEL, the technique presented in this paper addresses issues that arise when translating from graph-oriented to block-structure flow definition languages.

Keywords: Business process modeling, business process execution, BPMN, BPEL

1 Introduction

Business Process Management (BPM) is an established discipline for building, maintaining and evolving large enterprise systems on the basis of business process models [6]. A business process

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

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.

The *Business Process Modeling Notation* (BPMN) [27] is gaining adoption as a standard notation for capturing business processes [34]. The main purpose of business process models generally, and BPMN models in particular, is to facilitate communication between domain analysts and to support decision-making based on techniques such as cost analysis, scenario analysis and simulation [34,36]. However, BPMN models are also used as a basis for specifying software system requirements, and in such cases, they are handed over to software developers. In this setting, the motivating question of this paper is: How can developers fully exploit BPMN process models produced by domain analysts?

Meanwhile, the *Business Process Execution Language* (BPEL) [15] is emerging as a de facto standard for implementing business processes on top of Web service technology. More than a dozen platforms, such as Oracle BPEL, IBM WebSphere, and Microsoft BizTalk, support the execution of BPEL process definitions (see `http://en.wikipedia.org/wiki/BPEL` for a list). BPEL process definitions are more detailed than BPMN ones. For example, they include elements related to data manipulation, Web service bindings and other implementation aspects that are not present in their counterpart BPMN models.

In this setting, a standards-based approach to process-oriented systems development is to take BPMN models as input and to translate these models into templates of BPEL process definitions for subsequent manipulation by software developers. However, the instrumentation of this method is hindered by a fundamental mismatch between BPMN and BPEL [35]. A BPMN model consists of nodes that can be connected through control flow arcs in arbitrary ways. Meanwhile, BPEL offers block-structured constructs to capture control flow, plus a notion of "control link" to connect a collection of activities in an acyclic graph. In other words, BPMN supports arbitrary control-flow structures, whereas BPEL supports only restricted control-flow structures. As a result, existing mappings between BPMN and BPEL [23,27] 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.

The ensuing problem is to some extent similar to that of translating unstructured flowcharts into structured ones (or GOTO programs into WHILE programs) [28]. A major difference though is that process modeling languages include constructs for capturing parallel execution and 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 these additional features, the class of structured process models is strictly contained in the class of unstructured process models as discussed in [18]. This raises the question:

Can every BPMN model be translated into a BPEL model?

This paper shows that 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. Specifically, the original BPMN process model is decomposed into a collection of event-action rules that trigger one another to encode the underlying control flow logic. Arguably, the resulting BPEL code is not readable and thus difficult to modify and to maintain. For the generated BPEL code to be readable, the control flow logic should be captured using BPEL's block-structured control flow constructs and control links, as opposed to a construct intended for event handling. But since BPEL's control flow constructs are syntactically restricted, it is not always possible to generate BPEL code satisfying these readability criterion. Therefore, the paper also addresses the question:

Are there classes of BPMN models that can be translated into "readable" BPEL process definitions, i.e. process definitions in which control flow dependencies in the source model are not encoded as event handlers?

This paper identifies two such classes of BPMN models. The first one corresponds to the class of structured process models as defined in [18]. Such models can be mapped onto the structured control flow constructs of BPEL. The second class corresponds to the class of synchronising process models as defined in [17], which can be mapped onto BPEL control links. An acyclic BPMN model, or an acyclic fragment of a BPMN model, falls under this class if it satisfies a number of semantic conditions such as absence of deadlock. We apply Petri net analysis techniques to statically check these semantic conditions on the source BPMN model.

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.

It is beyond the scope of this paper to discuss every detail of a translation from BPMN to BPEL. Many of these details, such as how to map tasks and events into BPEL, are discussed in an appendix of the BPMN standard specification [27]. Instead, this paper concentrates on open issues arising from the mismatch between BPMN and BPEL discussed above.

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 to automate a mapping from UML Activity Diagrams [26] to BPEL.

The rest of the paper is structured as follows: Section 2 overviews BPMN and BPEL, and defines an abstract syntax for each of them. Section 3 presents three approaches which comprise an overall algorithm for translating BPMN into BPEL. The translation algorithm is then illustrated through two examples in Section 4. Section 5 discusses the tool support for our translation approach and uses a set of 568 business process models from practice to test whether the approach really yields readable BPEL models. Finally, Section 6 compares the proposal with related work while Section 7 concludes and outlines future work. In addition, a formal semantics of BPMN in terms of Petri nets is given in Appendix A.

2 Background: BPMN and BPEL

2.1 Business Process Execution Language for Web Services (BPEL)

BPEL [15] combines features found in classical imperative programming languages with constructs for capturing concurrent execution and constructs specific to Web service implementation. A BPEL process definition consists of a set of inter-related 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; *empty*, doing nothing; etc. To enable the presentation of complex structures the following *structured activities* are defined: *sequence*, for defining an execution order; *flow*, for parallel routing; *if*, for conditional routing; *pick*, for race conditions based on timing or external triggers; *while* and *repeatUntil*, 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 while 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 paper.

In addition to these block-structured constructs, BPEL provides a construct known as *control links* which, together with the associated notions of *join condition* and *transition condition*, allow the definition of directed acyclic graphs of activities. 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 iff 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 an *if* 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*. A control link is always defined inside a *flow* activity. In the definition of our mapping, it is convenient to differentiate between *flow* activities that have control links attached to them, from those that do not. Accordingly, we use the term *link-based flow* (or *link-flow* for short) to refer to a *flow* activity that has at least one control link directly attached to it.

Below is an abstract syntax of BPEL used in the rest of the paper. Since BPEL process definitions consist primarily of nested activities, we choose to represent this abstract syntax using a functional notation. Note that we use a superscript *seq* for specifying an ordered list of elements, and *set* for a normal set of elements. A BPEL process is a (top-level) *scope* activity.

Definition 1 (Abstract syntax of BPEL).

$$
\begin{aligned}
&\mathit{event} = \mathit{msgReceipt}\text{: } \mathit{messageType} \mid \mathit{alarm}\text{: } \mathit{timeSpec}\\
&\mathit{cond} = \mathit{bool\_expression}\\
&\mathit{activity} = \texttt{invoke}\text{: } \mathit{messageType} \mid \texttt{empty} \mid\\
&\qquad\qquad \texttt{receive}\text{: } \mathit{messageType} \mid \texttt{reply} \mid \texttt{wait} \mid \texttt{assign} \mid \texttt{exit} \mid\\
&\qquad\qquad \texttt{sequence}\text{: } \mathit{activity}^{seq} \mid\\
&\qquad\qquad \texttt{if}\text{: } (\mathit{cond} \times \mathit{activity})^{seq} \mid \texttt{pick}\text{: } (\mathit{event} \times \mathit{activity})^{set} \mid\\
&\qquad\qquad \texttt{while}\text{: } \mathit{cond} \times \mathit{activity} \mid \texttt{repeatUntil}\text{: } \mathit{cond} \times \mathit{activity} \mid\\
&\qquad\qquad \texttt{flow}\text{: } \mathit{activity}^{set} \mid \texttt{link-flow}\text{: } \mathit{linksInfo} \times \mathit{activity}^{set} \mid\\
&\qquad\qquad \texttt{scope}\text{: } (\mathit{event} \times \mathit{activity})^{set} \times \mathit{activity}\\
&\mathit{linksInfo} = \textsc{Struct}(\texttt{Links}\text{: } (\mathit{activity} \times \mathit{activity})^{set},\\
&\qquad\qquad\qquad\qquad \texttt{TransCond}\text{: } (\mathit{link} \times \mathit{cond})^{set},\\
&\qquad\qquad\qquad\qquad \texttt{JoinCond}\text{: } (\mathit{activity} \times \mathit{cond})^{set})
\end{aligned}
$$

The abstract syntax introduces an abstract datatype for BPEL activities and defines a number of constructors for this type (one per type of activity). Some of these constructors are parameterised. For example, the `sequence` constructor takes as parameter a sequence of activities. The abstract syntax does not cover all constructs, but only those that are used in the rest of the paper. Also, the syntax does not capture some syntactic constraints such as the fact that the set of control links in a process definition can not form cycles. A more comprehensive abstract syntax for BPEL can be found in [31].

2.2 Business Process modeling Notation (BPMN)

BPMN [27] essentially provides a graphical notation for business process modeling, 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 modeling, such as AND-split, AND-join, XOR-split, XOR-join, and deferred (event-based) choice.

A BPD is made up of BPMN elements as 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 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 events (event-based), and *XOR merge gateways* for joining a set of mutually exclusive alternative sequence flows into one sequence flow. 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). This restriction is not imposed for data-based decision gateways. On the other hand, the outgoing flows of a data-based XOR decision gateway are labelled with conditional expressions, except for one of them which acts as a default flow (depicted by an arrow with a backslash). The default flow is taken if the conditions associated with all other outgoing conditional flows evaluate to false at run time. This ensures that exactly one outgoing flow is taken.

The BPMN elements shown in Figure 1 cover what we call the *core* subset of BPMN. BPMN defines several other control-flow constructs besides these "core" ones. These include: (1) *task looping*, (2) *multi-instance task*, (3) *exception flow*, (4) *sub-process invocation*, (5) *inclusive OR decision gateway* (also called OR-split), and (6) *inclusive OR merge gateway* (also called OR-join). The mapping of the first five of these "non-core" constructs onto BPEL does not entail additional challenges. Task looping, which corresponds to structured loops can be easily

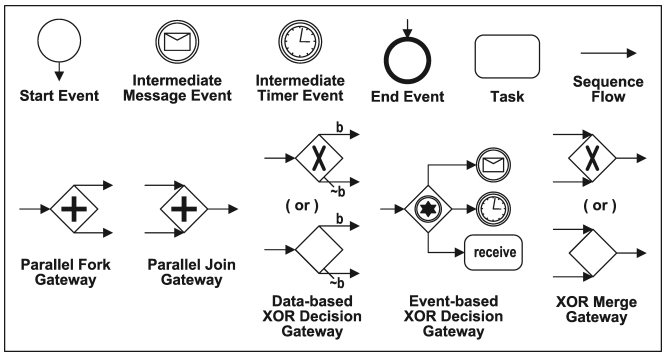

Figure 1. A set of BPMN elements covering the fundamental control flows in BPMN.

mapped to BPEL "while" activities. Similarly, a multi-instance task can be directly mapped to a "parallel foreach" activity. Sub-processes can be mapped onto separate BPEL processes which call one another. Any OR-split gateway can be expanded into a combination of AND-split and XOR-split gateways [2]. Hence, it does not require a separate mapping rule. On the other hand, the mapping of OR-joins requires a special treatment that falls outside the scope of this work (see Section 7). In the rest of the paper we focus on BPDs composed only of core constructs as per the following definition.

Definition 2 (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 is divided 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,*
- $\mathcal{E}$ *is divided into disjoint sets of start events* $\mathcal{E}^S$*, intermediate events* $\mathcal{E}^I$*, and end events* $\mathcal{E}^E$*,*
- $\mathcal{E}^I$ *is divided into disjoint sets of intermediate message events* $\mathcal{E}^I_M$ *and timer events* $\mathcal{E}^I_T$*,*
- $\mathcal{G}$ *is divided into disjoint sets of parallel fork gateways* $\mathcal{G}^F$*, join gateways* $\mathcal{G}^J$*, data-based XOR decision gateways* $\mathcal{G}^D$*, event-based 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, i.e., a set of sequence flows connecting objects,*
- Cond*:* $\mathcal{F} \nrightarrow \mathcal{B}$ *is a function mapping sequence flows emanating from data-based XOR decision gateways to conditions,*[1] *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 given 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 $\mathsf{out}(x) = \{y \in \mathcal{O} \mid x\mathcal{F}y\}$.

[1] $\mathcal{B}$ 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.

Definition 2 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 3 (Well-formed core BPD). *A core BPD as defined in Definition 2 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 and both types of 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 and 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,*
- $\forall\ g \in \mathcal{G}^D$, $\exists$ *an order* $<$ *which is a strict total order over the set of outgoing flows of* g *(i.e.* $\{g\}\times\mathsf{out}(g)$*), and for* $x \in \mathsf{out}(g)$ *such that* $\neg\ \exists_{f \in \{g\}\times\mathsf{out}(g)}(f<(g,x))$, (g,x) *is the default flow among all the outgoing flows from* g,
- $\forall\ x \in \mathcal{O}$, $\exists\, s \in \mathcal{E}^S$, $\exists\, e \in \mathcal{E}^E$, $s\mathcal{F}^*x \wedge x\mathcal{F}^*e$,[2] *i.e. every object is on a path from a start event to an end event.*

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 $\mathcal{E}^E = \{e\}$.[3]

3 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

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

[3] 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.

three different approaches to the mapping of components.[4] 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.

3.1 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 is mapped onto a BPEL *sequence* construct while a component holding a parallel structure is 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 4 (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,*
- $\mid (\bigcup_{x \in \mathcal{O}_c} \mathsf{in}(x)) \backslash \mathcal{O}_c \mid = 1$*, i.e., there is a single entry point outside the component,*[5] *which can be denoted as* $\mathsf{entry}(\mathcal{C}) = \mathsf{elt}((\bigcup_{x \in \mathcal{O}_c} \mathsf{in}(x)) \backslash \mathcal{O}_c)$*,*
- $\mid (\bigcup_{x \in \mathcal{O}_c} \mathsf{out}(x)) \backslash \mathcal{O}_c \mid = 1$*, i.e., there is a single exit point outside the component, which can be denoted as* $\mathsf{exit}(\mathcal{C}) = \mathsf{elt}((\bigcup_{x \in \mathcal{O}_c} \mathsf{out}(x)) \backslash \mathcal{O}_c)$*,*
- $\mid \mathsf{out}(\mathsf{entry}(\mathcal{C})) \cap \mathcal{O}_c \mid = 1$*, i.e., there is a unique source object* $i_c = \mathsf{elt}(\mathsf{out}(\mathsf{entry}(\mathcal{C})) \cap \mathcal{O}_c)$*,*
- $\mid \mathsf{in}(\mathsf{exit}(\mathcal{C})) \cap \mathcal{O}_c \mid = 1$*, i.e., there is a unique sink object* $o_c = \mathsf{elt}(\mathsf{in}(\mathsf{exit}(\mathcal{C})) \cap \mathcal{O}_c)$*,*
- $i_c \neq o_c$*,*

[4] It should be noted that the first two approaches are inspired by the mapping from Petri nets to BPEL as described in [4].

[5] Note that $\mathsf{in}(x)$ is not defined with respect to the component but refers to the whole BPD. This also applies to $\mathsf{out}(x)$.

– $\mathcal{F}_c = \mathcal{F} \cap (\mathcal{O}_c \times \mathcal{O}_c)$,

– $\mathsf{Cond}_c = \mathsf{Cond}[\mathcal{F}_c]$, *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 3.5).

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 the function Fold that replaces a component by a single (blank) task object in a BPD. This function can be used to iteratively reduce 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 5 (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:*

– $\mathcal{O}' = (\mathcal{O} \backslash \mathcal{O}_c) \cup \{t_c\}$,

– $\mathcal{T}' = (\mathcal{T} \backslash \mathcal{O}_c) \cup \{t_c\}$ *is the set of tasks in* $\mathsf{Fold}(\mathcal{BPD}, \mathcal{C}, t_c)$,

– $\mathcal{T}^{R'} = (\mathcal{T}^R \backslash \mathcal{O}_c)$ *is the set of receive tasks in* $\mathsf{Fold}(\mathcal{BPD}, \mathcal{C}, t_c)$,

– $\mathcal{F}' = (\mathcal{F} \cap ((\mathcal{O} \backslash \mathcal{O}_c) \times (\mathcal{O} \backslash \mathcal{O}_c))) \cup \{(\mathsf{entry}(\mathcal{C}), t_c), (t_c, \mathsf{exit}(\mathcal{C}))\}$,

– $\mathsf{Cond}' = \begin{cases} \mathsf{Cond}[\mathcal{F}'] & \text{if } \mathsf{entry}(\mathcal{C}) \notin \mathcal{G}^D \\ \mathsf{Cond}[\mathcal{F}'] \cup \{((\mathsf{entry}(\mathcal{C}), t_c), \mathsf{Cond}(\mathsf{entry}(\mathcal{C}), i_c))\} & \text{otherwise} \end{cases}$

3.2 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*, *if*, *pick*, *while* and *repeatUntil*, have the first preference if the corresponding structures appear in the BPD. Components that have a direct and intuitive correspondence to one of these six structured constructs are considered as *well-structured components* (WSCs). Below, we classify different types of WSCs resembling these six structured constructs.

Definition 6 (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 WSCs:*

(*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 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* IF*-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*

- $i_c \in \mathcal{G}^V \wedge o_c \in \mathcal{G}^M$,
- $\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I \cup \{i_c, o_c\}$,
- $\forall\, x \in \mathcal{O}_c \backslash (\{i_c, o_c\} \cup \mathsf{out}(i_c))$, $\mathsf{in}(x) \subseteq \mathsf{out}(i_c) \wedge \mathsf{out}(x) = \{o_c\}$.[6]

(*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 2 illustrates how to map each of the above WSCs onto the corresponding BPEL structured activities. Using function Fold in Definition 5, 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 $Mapping(t_i)$. Based on the nature of these task objects they can be mapped onto the proper types of BPEL basic activities. For example, a

[6] 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, i.e. for any $x \in \mathcal{T}^R \cup \mathcal{E}^I$: $|\ \mathsf{in}(x)\ | = |\ \mathsf{out}(x)\ | = 1$. Moreover, 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.

service task is mapped onto an invoke activity, a *receive* task (like t_r in Figure 2(d)) is mapped onto a receive activity, and a *user* task may be mapped onto an invoke activity followed by a receive activity. 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. The interested reader may refer to [27] for some guidelines on mapping BPMN tasks into BPEL activities. Finally, the task t_i may result from the folding of a previous component $\mathcal{C}'$, in which case, $Mapping(t_i)$ is the code for the mapping of component $\mathcal{C}'$.

With the examples shown in Figure 2, we now generalise the mapping of a given WSC $\mathcal{C}$ using the BPEL syntax defined in Definition 1, as follows:

- A SEQUENCE-component is mapped to $\texttt{sequence}([Mapping(x) \mid x \leftarrow [\mathcal{O}_c]^<])$. The notation $[\mathcal{O}_c]^<$ represents the (sequentially) ordered list of objects in $\mathcal{C}$, and $x \leftarrow [\mathcal{O}_c]^<$ indicates that each object x is taken in the order as they appear in the list $[\mathcal{O}_c]^<$;
- A FLOW-component to $\texttt{flow}(\{Mapping(x) \mid x \in \mathcal{T}_c \cup \mathcal{E}_c\})$;
- An IF-component to $\texttt{if}([(\mathsf{Cond}_c(i_c, x), Mapping(x)) \mid x \leftarrow [\mathsf{out}(i_c)]^<])$. The source object i_c is the data-based decision gateway in the component, and $[\mathsf{out}(i_c)]^<$ captures the order of the outgoing (conditional) flows of the gateway so that the conditional branches in the resulting *if* construct are evaluated in the same order;
- In a PICK-component, an event-based decision gateway must be followed by receive tasks or intermediate message or timer events[7]. We use function $\mathsf{Map2Event}$ to capture the fact that the above receive task or intermediate message will be mapped to a BPEL *msgReceipt* event (`<onMessage>` in BPEL concrete syntax) and the timer event to a BPEL *alarm* event (`<onAlarm>`). Therefore, the mapping of a PICK-component can be written as $\texttt{pick}(\{(\mathsf{Map2Event}(x), Mapping(\mathsf{succ}(x))) \mid x \in \mathsf{out}(i_c)\})$. Note that for any object x that has an outdegree of one, $\mathsf{succ}(x)$ refers to the only output object of x;
- A REPEAT-component to $\texttt{repeatUntil}(\mathsf{Cond}_c(o_c, \mathsf{exit}(\mathcal{C})), Mapping(\mathsf{succ}(i_c)))$;
- A WHILE-component is mapped to a *while* construct of $\texttt{while}(\mathsf{Cond}_c(o_c, x), Mapping(x))$ where $x = \mathsf{elt}(\mathsf{in}(i_c) \cap \mathsf{out}(o_c))$; and
- A REPEAT+WHILE-component to a *while* construct of the above being nested within a construct of $\texttt{repeatUntil}(\mathsf{Cond}_c(o_c, \mathsf{exit}(\mathcal{C})), \texttt{sequence}([Mapping(\mathsf{succ}(i_c)), while]))$.

3.3 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

[7] For this reason, a SEQUENCE-component cannot be preceded by an event-based XOR decision gateway (as defined by $\mathsf{entry}(\mathcal{C}) \notin \mathcal{G}^V$ in Definition 6(a)).

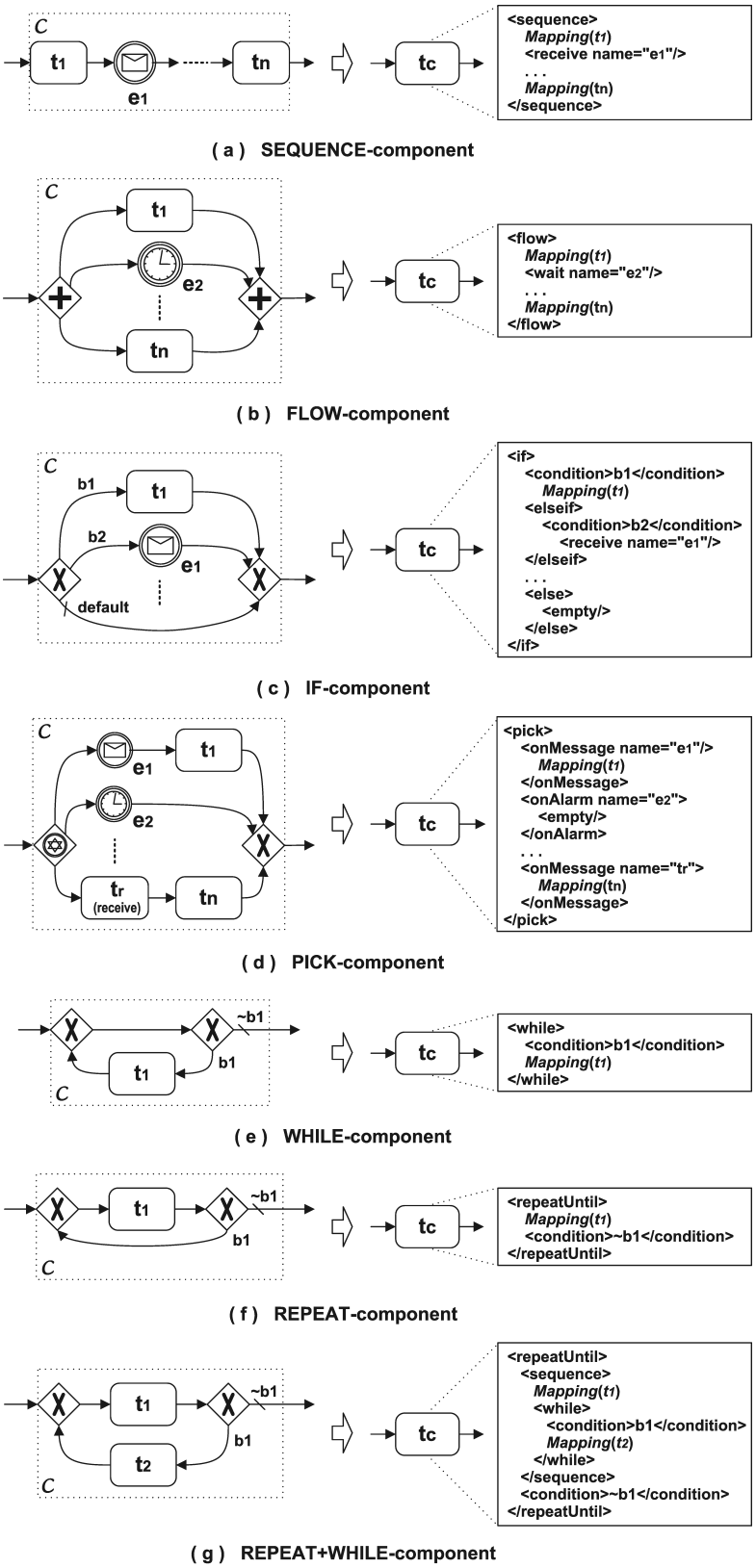

Figure 2. Mapping a WSC $\mathcal{C}$ onto a BPEL structured activity and folding $\mathcal{C}$ into a single task object t_c attached with the BPEL code for mapping.

the "patterns" as defined in Definition 6. 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 BPD components. As mentioned in Section 2.1, the term "link-based flow construct" is used to refer to a flow activity in which all sub-activities are connected through links to form directed acyclic graphs.

It is important to emphasize two issues related to link semantics. First, the use of control links may hide errors such as deadlocks. This means that the designer makes a modeling error that in other languages would result in a deadlock, however, given the dead-path elimination semantics of BPEL the error is masked. As an example, the use of control links can lead to models where one or several actions are "unreachable", i.e., these actions will never be executed. The interested reader may refer to [31] for examples of such undesirable models. Second, since an activity cannot "start until the status of all its incoming links has been determined and the, implicit or explicit, join condition has been evaluated" (Section 11.6.2 of [15]), each execution of an activity can trigger at most one execution of any subsequent activity to which it is connected via a control link.

Consider, as shown in Figure 3, the two acyclic components that are not well-structured (in the sense of Definition 6). The one in (a) can be mapped onto a link-based flow construct without any problem.[8] 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 behavior that cannot be represented using control links which only trigger the target activity at most once.

In Petri net terminology, a component like the one shown in Figure 3 cannot be qualified as being "sound" and "safe". Intuitively, if a BPD component is sound, it is free from deadlocks and dead tasks (i.e. tasks that can never be executed), and once the component is executed, the execution always starts at the source object and will always reach the sink object. If a BPD component is safe, it implies that any object in the component that is already activated cannot receive another activation signal unless the current activation of the object is completed. Further discussions on soundness and safeness properties within the context of Petri nets are given in Appendix A.3.

[8] A model with similar topology as Figure 3(a) has been used in the proof of the existence of "arbitrary, well-behaved, workflow models that cannot be modelled as structured workflow models" on page 438 of [18].

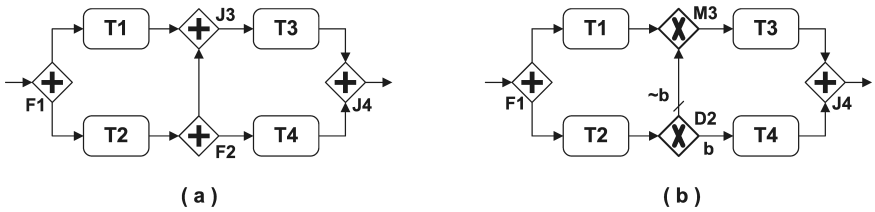

Figure 3. Two non-well-structured acyclic components: (a) can be mapped onto a link-based flow construct, whereas (b) is not sound and safe and therefore cannot be translated.

3.3.1 Components for Control Link-based Translation

We use the term *Synchronising Process Component* (SPC) to refer to an acyclic component that is sound and safe and is free from event-based gateways (see Definition 7). Each SPC can be mapped to a link-based flow construct preserving the same semantics. The name SPC is inspired by the concept of *synchronising process models*, in which "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" [17]. This way the semantics of control links are well captured and thus the activities can be viewed as being connected via control links.

Definition 7 (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 an SPC if it satisfies the following three conditions:*

(*a*) *There are no cycles (i.e., $\forall\, x \in \mathcal{O}_c$, $(x, x) \notin \mathcal{F}_c^*$);*

(*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*) *$\mathcal{C}$ is sound and safe (this can be determined based on a Petri net semantics of $\mathcal{C}$).*

It is worth noting that some WSCs such as SEQUENCE-component, FLOW-component and IF-component are also SPCs. When mapping a BPD onto BPEL we will always try to use the structured activity-based translation described in Section 3.2, until there are no WSCs left in the BPD. Therefore, the control link-based translation only applies to a subset of SPCs that are not well-structured. In addition, we will always try to detect a *minimal* SPC for translation. An SPC $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$ is *minimal* if there is no 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 WSCs left (they have been iteratively removed).

3.3.2 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 an SPC into a set of control links. Before translation, it is necessary to pre-process the component using the following two steps. First, as aforementioned, a minimal SPC always has a gateway as its source or sink object. Since control links connects only task or 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 8 (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 $\mathcal{C}_w = (\mathcal{O}_w, \mathcal{F}_w, \mathsf{Cond}_w)$ where:*

- $\mathcal{O}_w = \mathcal{O}_c \cup \{a_h, a_t\}$,
- $\mathcal{F}_w = \mathcal{F}_c \cup \{(a_h, i_c), (o_c, a_t)\}$,
- $\mathsf{Cond}_w = \mathsf{Cond}_c$

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 [27]). 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 3). 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

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

It is easy to prove that the above pre-processing will not change the behavior of an SPC.

We now derive from the structure of a wrapped SPC, 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 or event objects that are directly or indirectly (via gateways) connected within the component. To this end, we define two functions as shown in Figure 4. One named PreTEC-Sets (lines 1-7), takes an object x and generates the set of sets of pairs each containing a preceding task or event and a boolean expression for x. If any data-based decision gateways are involved, the boolean expression is used to capture the conditions specified on the outgoing flows of these gateways; otherwise, it has a boolean value of TRUE by default. Also, for any object x that has an indegree of one, pred(x) refers to the only input object of x.

Except for the source object (line 4), PreTEC-Sets relies on the second function named PreTEC-SetsFlow (lines 9-30) to compute the results for all the other objects in the component. The function PreTEC-SetsFlow 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 (line 21), a set is returned containing a singleton set of a pair comprising that task or event and a default boolean value of TRUE since there is no data-based decision gateway on the flow. 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 (line 14), the (refined) condition on the flow is added via conjunction to each of boolean expressions in the resulting set. 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 (line 23), the union of the resulting sets captures the fact that when *any* of these flows is taken, the gateway may be executed. For a join gateway (line 24), the combination[9] of the resulting sets captures the fact that when *all* of these flows are taken, the gateway may be executed.

Based on the above, Figure 5 defines an algorithm which derives from a wrapped SPC $\mathcal{C}_w$, the set of control links for connecting the tasks and events in $\mathcal{C}_w$, the set of transition conditions associated with each of the links, and the set of join conditions associated with each of the tasks and events. The algorithm consists of four functions. The main function Map2Links (lines 1-9) returns the final result by calling three other functions, namely GetLinks (lines 11-14), GetTransCond (lines 16-24), and GetJoinCond (lines 26-30). A control link is defined as a pair comprising the source and the target objects. A transition condition associated with a control link is a boolean expression that functions as a guard on the link. For a given task or event x, the task or event object in each of the pairs (i.e. (taskevent, boolexpr)) in PreTEC-Sets(x) rep-

[9] This is performed by first calculating the cartesian product of a number of n sets and then converting each element in the resulting set from a tuple of cardinality n to a set of union of the n elements in the tuple.

```
1:  function PreTEC-Sets(x ∈ T_w ∪ E_w: a task/event object in C_w)
2:     : SET of (SET of ((T_w ∪ E_w)×Bool-Expr))
3:  begin
4:     if x = a_h // the source object of C_w
5:        return ∅
6:     else return PreTEC-SetsFlow(pred(x), x)
7:  end
8:
9:  function PreTEC-SetsFlow((x, y) ∈ F_w: a flow relation in C_w)
10:    : SET of (SET of ((T_w ∪ E_w)×Bool-Expr))
11: begin
12:    case x ∈ T_w ∪ E_w: return {{(x, TRUE)}}
13:    case x ∈ G_w^F: return PreTEC-SetsFlow(pred(x), x)
14:    case x ∈ G_w^D:
15:       let c_{x,y} = RefinedCond(x, y);
16:       let {{(a_1, cond_1), ... , (a_i, cond_i)},
17:              ...
18:            {(a_{i+j}, cond_{i+j}), ... , (a_n, cond_n)}} =
19:            PreTEC-SetsFlow(pred(x), x)
20:       return {{(a_1, cond_1 ∧ c_{x,y}), ... , (a_i, cond_i ∧ c_{x,y})},
21:                  ...
22:               {(a_{i+j}, cond_{i+j} ∧ c_{x,y}), ... , (a_n, cond_n ∧ c_{x,y})}}
23:    case x ∈ G_w^M: return ⋃_{z∈in(x)} PreTEC-SetsFlow(z, x)
24:    case x ∈ G_w^J:
25:       let {(s_{1,1}, ... , s_{1,n}), ... , (s_{m,1}, ... , s_{m,n})} =
26:          PreTEC-SetsFlow(z_1, x) × ... × PreTEC-SetsFlow(z_n, x)
27:       where {z_1, ... , z_n} = in(x)
28:       return {s_{1,1} ∪ ... ∪ s_{1,n}, ... , s_{m,1} ∪ ... ∪ s_{m,n}}
29:    end case
30: end
```

Figure 4. Algorithm for deriving the set of preceding tasks or events with conditions sets for any object in a wrapped SPC $\mathcal{C}_w = (\mathcal{O}_w, \mathcal{F}_w, \mathsf{Cond}_w)$.

resents the source object of a link; and for this link, its transition condition can be derived as a disjunction of all the boolean expressions in the pairs that share the same task or event as their first element. Next, a join condition associated with a task or event object is specified as a boolean expression over the linkstatus of each of the links leading to that object. For a given task or event x, the join condition of all the incoming links to x can be derived capturing the fact that a combination (which implies a conjunction) of the links obtained from each set of the pairs in PreTEC-Sets(x), represent an alternative way (which implies a disjunction) to reach x.

Now, with the set of links returnd by Map2Links($\mathcal{C}_w$), we can map the entire component $\mathcal{C}_w$ onto a link-based flow construct. The mapping of each task or event object x in $\mathcal{C}_w$ is denoted as *Mapping*(x). Using the BPEL syntax given in Definition 1, the resulting link-based flow construct can be written as `link-flow`(Map2Links($\mathcal{C}_w$), {*Mapping*(x) | $x \in \mathcal{T}_w \cup \mathcal{E}_w$}).

Finally, it is important to mention the interplay between the structured activity-based approach (Section 3.2) 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

```
1:  function Map2Links(Wrapped SPC C_w)
2:     : STRUCT(Links: SET of (TE_w×TE_w),
3:                 TransCond: SET of ((TE_w×TE_w)×Bool-Expr),
4:                 JoinCond: SET of (TE_w×(Bool-Fun: (SET of linkstatus(TE_w×TE_w))→Bool)))
5:  begin
6:     return STRUCT(Links: GetLinks(C_w),
7:                   TransCond: GetTransCond(C_w),
8:                   JoinCond: GetJoinCond(C_w))
9:  end
10:
11: function GetLinks(Wrapped SPC C_w): SET of (TE_w×TE_w)
12: begin
13:    return ⋃_{x∈TE_w}(⋃_{s∈PreTEC-Sets(x)}(⋃_{p∈s}{(taskevent(p), x)}))
14: end
15:
16: function GetTransCond(Wrapped SPC C_w): SET of ((TE_w×TE_w)×Bool-Expr)
17: begin
18:    transCond := {}
19:    for all l ∈ GetLinks(C_w) do
20:       cond_l := ⋁_{s∈PreTEC-Sets(taskevent(l))}(⋁_{p∈s∧taskevent(p)=taskevent(l)} boolexpr(p))
21:       transCond := transCond ∪ {(l, cond_l)}
22:    end for
23:    return transCond
24: end
25:
26: function GetJoinCond(Wrapped SPC C_w)
27:    : SET of (TE_w×(Bool-Fun: (SET of linkstatus(TE_w×TE_w))→Bool))
28: begin
29:    return ⋃_{x∈TE_w}{(x, ⋁_{s∈PreTEC-Sets(x)}(⋀_{p∈s}(taskevent(p), x)))}
30: end
```

Figure 5. Algorithm for deriving the set of control links, the set of transition conditions, and the set of join conditions from a wrapped SPC.

WSCs, 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, i.e., it always works but may lead to less readable models.

3.4 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. For example, a component capturing a multi-merge pattern, which allows each incoming branch to continue independently of the others thus enabling multiple threads of execution on the subsequent branch [2], or an unstructured loop, i.e., a loop with more than one entry point and/or more than one exit point. 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 to this approach only 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 "event" of receiving a message which triggers the corresponding event handler. The actions taken by the event handler must ensure to invoke (i.e. send) message(s) signaling the completion of the object execution. Note that the word "event" we mention here refers to the event within the context of event-action rules, and therefore is different from BPMN event objects.

Figure 6 illustrates how to map each type of BPMN objects onto BPEL event handlers. We use $m_{(y,x)}$ to denote a message that is created once the sequence flow connecting object y to object x is taken. This message signals the completion of y so that the execution of x may start. The event of receiving a message $m_{(y,x)}$ can be written as $msgReceipt(m_{(y,x)})$. Each task, event, fork gateway, or decision gateway object is mapped onto one event handler, which is triggered upon the receipt of the message from the only incoming flow of the object. For a task or event object x, let z denote the only output object of x, the resulting event handler first executes x (whose mapping is denoted as $Mapping(x)$), and then invokes the message $m_{(x,z)}$ once the sequence flow from object x to object z is taken, signaling the completion of the execution of x. For a fork or decision gateway, the resulting event handler invokes a number of messages to capture the outgoing flows in order as defined by the gateway. To this end, the BPEL *flow* activity is used for the mapping of a fork gateway, the *if* activity is used for a data-based decision gateway, and the *pick* activity is used for 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 receipt of the message 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 6, a join gateway x can be mapped onto one event handler by separating, for example, the receipt of the message ($m_{(y_1,x)}$) on the first incoming flow, from those ($m_{(y_2,x)}$, ..., $m_{(y_n,x)}$) on the rest of the incoming flows. Although the resulting event handler can be triggered by the receipt of message $m_{(y_1,x)}$,

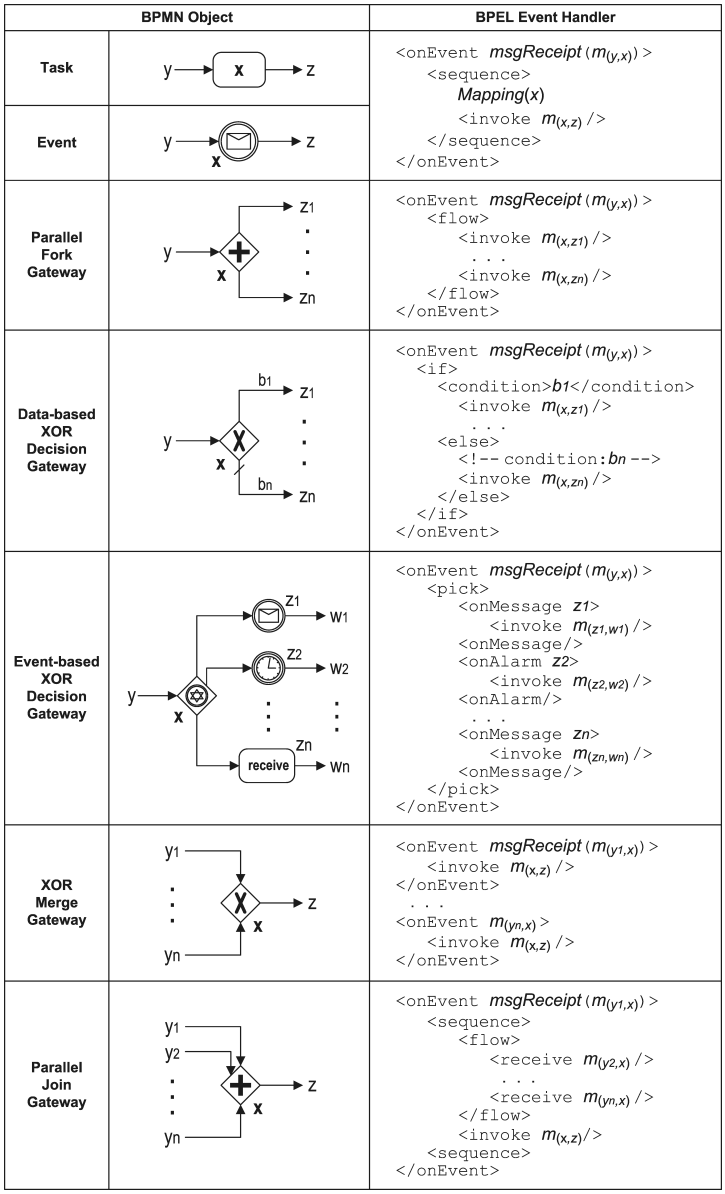

Figure 6. Mapping BPMN objects onto BPEL event handlers.

the real action, i.e. invoking the message $m_{(x,z)}$, will not be performed until all the remaining messages $m_{(y_2,x)}$ to $m_{(y_n,x)}$ have been received.

Based on the above, Figure 7 defines a function named Map2EHs which takes as input a component $\mathcal{C}$ in a well-formed core BPD, and returns the set of event handlers as the mappings of all the objects in $\mathcal{C}$. The resulting code is written in BPEL syntax as defined in Definition 1. With the set of event handlers returned by $\mathsf{Map2EHs}(\mathcal{C})$, we can then map the entire component $\mathcal{C}$ onto a BPEL scope construct. Let $m_{(\mathsf{entry}(C),i_c)}$ denote the message being created once the sequence flow is taken which connects from the entry point $\mathsf{entry}(\mathcal{C})$ (outside the component $\mathcal{C}$) to the source object i_c. Using the BPEL syntax definition, the resulting scope can be written as $\mathtt{scope}(\mathsf{Map2EHs}(\mathcal{C}), \mathtt{invoke}(m_{(\mathsf{entry}(\mathcal{C}),i_c)}))$. Here, the main activity of the scope is to invoke message $m_{(\mathsf{entry}(\mathcal{C}),i_c)}$. Upon receiving this message, the event handler for the source

```
1:  function Map2EHs(C: a component of a well-formed BPD)
2:     : BPEL's (event × activity)^set
3:  begin
4:     EHs := {}
5:     for all x ∈ O_c do
6:        case x ∈ T_c ∪ E_c:
7:           EH := {(msgReceipt(m_(pred(x),x)), sequence([Mapping(x), invoke(m_(x,succ(x)))]))}
8:        case x ∈ G_c^F:
9:           EH := {(msgReceipt(m_(pred(x),x)), flow({invoke(m_(x,y)) | y ∈ out(x)}))}
10:       case x ∈ G_c^D:
11:          EH := {(msgReceipt(m_(pred(x),x)), if([(Cond_c(x, y), invoke(m_(x,y))) | y ← [out(x)]^<]))}
12:       case x ∈ G_c^V:
13:          EH := {(msgReceipt(m_(pred(x),x)), pick({(Map2Event(y), invoke(m_(y,succ(y)))) | y ∈ out(x)}))}
14:       case x ∈ G_c^M:
15:          EH := {(msgReceipt(m_(y,x)), invoke(m_(x,succ(x)))) | y ∈ in(x)}
16:       case x ∈ G_c^J:
17:          select any y_i ∈ in(x) do
18:             EH := {(msgReceipt(m_(y_i,x)),
19:                     sequence([flow({receive(m_(y,x)) | z∈in(x)\y_i}), invoke(m_(x,succ(x)))]))}
20:          end select
21:       end case
22:       EHs := EHs ∪ EH
23:    end for
24:    return EHs
25: end
```

Figure 7. Algorithm for deriving the set of event handlers from a well-formed BPD component.

object i_c will be triggered. Then, the event handlers for the remaining objects will be executed according to the execution order specified in $\mathcal{C}$. Finally, the entire scope will complete after the executions of its main activity and all active event handlers are completed.

3.5 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, namely BPD2BPEL, which takes a well-formed core BPD Q with one start event s and one end event e, and produces the resulting mapping to a BPEL process. Let $\mathcal{O}_Q$ denote the set of objects in Q, and Components(Q) the set of components in Q. The translation procedure starts with mapping each primitive task and each event that is not immediately preceded by an event-based decision gateway onto BPEL basic activities. This is denoted by function Map2BasicAct. Also, for a given object x, the mapping of x to BPEL is given by *Mapping*(x). Next, if Q contains at least one component (i.e. Q is a non-trivial BPD), the basic idea is to select a component in Q, provide its BPEL translation, and fold the component (lines 12-39). This is repeated until no component is left in Q (i.e. Q is a trivial BPD). The translation of a trivial BPD is straightforward, which involves the mapping of the only object (*root*) between the start and

1: **function** BPD2BPEL(Well-formed core BPD Q): BPEL process
2: **begin**
3: $s :=$ start event of Q
4: $e :=$ end event of Q
5: $Mapping := \{\}$
6: **for all** $x \in \mathcal{T} \cup \mathcal{E} \setminus (\{s, e\} \cup \bigcup_{y \in \mathcal{G}^V} \mathsf{out}(y))$ **do**
7: $Mapping := Mapping \cup \{(x, \mathsf{Map2BasicAct}(x))\}$
8: **end for**
9: **if** $\mathsf{Components}(Q) \neq \varnothing$
10: **then** // *mapping of a non-trivial BPD*
11: **repeat**
12: **if exists** a maximal SEQUENCE-component $\mathcal{C} \in \mathsf{Components}(Q)$
13: **then** $Bpel\_frag :=$ `sequence`$([Mapping(x) \mid x \leftarrow [\mathcal{O}_c]^<])$
14: **else if exists** a (non-sequence) WSC $\mathcal{C} \in \mathsf{Components}(Q)$
15: **case** $\mathcal{C}$ of a FLOW-component: $Bpel\_frag :=$ `flow`$(\{Mapping(x) \mid x \in \mathcal{T}_c \cup \mathcal{E}_c\})$
16: **case** $\mathcal{C}$ of a IF-component:
17: $Bpel\_frag :=$ `if`$([(\mathsf{Cond}_c(i_c, x), Mapping(x)) \mid x \leftarrow [\mathsf{out}(i_c)]^<])$
18: **case** $\mathcal{C}$ of a PICK-component:
19: $Bpel\_frag :=$ `pick`$(\{(\mathsf{Map2Event}(x), Mapping(\mathsf{succ}(x))) \mid x \in \mathsf{out}(i_c)\})$
20: **case** $\mathcal{C}$ of a WHILE-component:
21: $Bpel\_frag :=$ `while`$(\mathsf{Cond}_c(o_c, x), Mapping(x))$ where $x = \mathsf{elt}(\mathsf{in}(i_c) \cap \mathsf{out}(o_c))$
22: **case** $\mathcal{C}$ of a REPEAT-component:
23: $Bpel\_frag :=$ `repeatUntil`$(\mathsf{Cond}_c(o_c, \mathsf{exit}(\mathcal{C})), Mapping(\mathsf{succ}(i_c)))$
24: **case** $\mathcal{C}$ of a REPEAT+WHILE-component:
25: $Bpel\_frag :=$ `while`$(\mathsf{Cond}_c(o_c, x), Mapping(x))$ where $x = \mathsf{elt}(\mathsf{in}(i_c) \cap \mathsf{out}(o_c))$;
26: $Bpel\_frag :=$ `sequence`$([Mapping(\mathsf{succ}(i_c)), Bpel\_frag])$;
27: $Bpel\_frag :=$ `repeatUntil`$(\mathsf{Cond}_c(o_c, \mathsf{exit}(\mathcal{C})), Bpel\_frag)$
28: **end case**
29: **else** select a minimal non-WSC $\mathcal{C} \in \mathsf{Components}(Q)$
30: **if** $\mathcal{C}$ is a synchronising process component
31: **then** $\mathcal{C}_w := \mathsf{Wrapped}(\mathcal{C})$;
32: $Bpel\_frag :=$ `link-flow`$(\mathsf{Map2Links}(\mathcal{C}_w), \{Mapping(x) \mid x \in \mathcal{T}_w \cup \mathcal{E}_w\})$
33: **else** $Bpel\_frag :=$ `scope`$(\mathsf{Map2EHs}(\mathcal{C}), \mathtt{invoke}(e_{(\mathsf{entry}(\mathcal{C}), i_c)}))$
34: **end if**
35: **end if**
36: $t_c :=$ a new blank task object
37: $Mapping := Mapping \cup \{(t_c, Bpel\_frag)\}$
38: $Q := \mathsf{Fold}(Q, \mathcal{C}, t_c)$
39: **until** $\mathsf{Components}(Q) = \varnothing$
40: **else** $root := \mathsf{elt}(\mathcal{O}_Q \setminus \{s, e\})$ // Mapping of a trivial BPD
41: **end if**
42: **return** $Mapping(root)$
43: **end**

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

the end events in the BPD (line 40). The resulting BPEL process can be then retrieved from *Mapping*(*root*) (line 42).

In more details, for a non-trivial BPD, the component mapping always starts from a maximal SEQUENCE-component after each folding (lines 12 to 13). When there are no sequences left in the BPD, other WSCs are processed (lines 14 to 28). Since all non-sequence WSCs are disjoint, the order of mapping these components is irrelevant. Next, when no WSCs are left, the algorithm selects a *minimal* non-WSC for translation (line 19). Note that $\mathcal{C}$ is a *minimal* non-WSC, 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-WSC $\mathcal{C}$ and not a maximal one to avoid missing any "potential" WSC 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 SPC so that the control link-based translation approach can be applied (lines 30-32). Note that as part of the SPC identification, one needs to check if the component is sound and safe. This can be done by mapping the component into a Petri net and then checking the soundness and safeness properties of this Petri net (see Appendix A). Also, for any well-formed BPD component, the function Wrapped returns the correspondingly wrapped component as defined in Definition 8. Next, if $\mathcal{C}$ is not a SPC, the event-action rule-based translation approach is used as a last resort (line 33). 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 two examples in the next section and by the empirical evaluation presented in Section 5.

With the above algorithm, a non-trivial well-formed core BPD can always be componentized and each component is either a WSC or a non-WSC. Every WSC or non-WSC can be mapped onto a certain BPEL construct or a combination of BPEL constructs according to the algorithm. By following-up every mapping operation with a corresponding folding operation the source BPD is gradually simplified and ultimately reduced to a trivial BPD. As part of this process, the target BPEL code is gradually excluded and finalized when the trivial BPD is mapped.

4 Examples

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

4.1 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.

Following the algorithm in Section 3, 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 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 is presented in simplified BPEL syntax which defines the control flow for the process and omits the 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, 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:

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

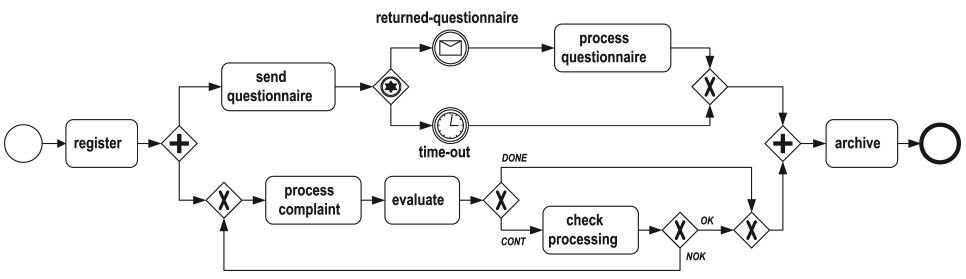

Figure 9. A complaint handling process model.

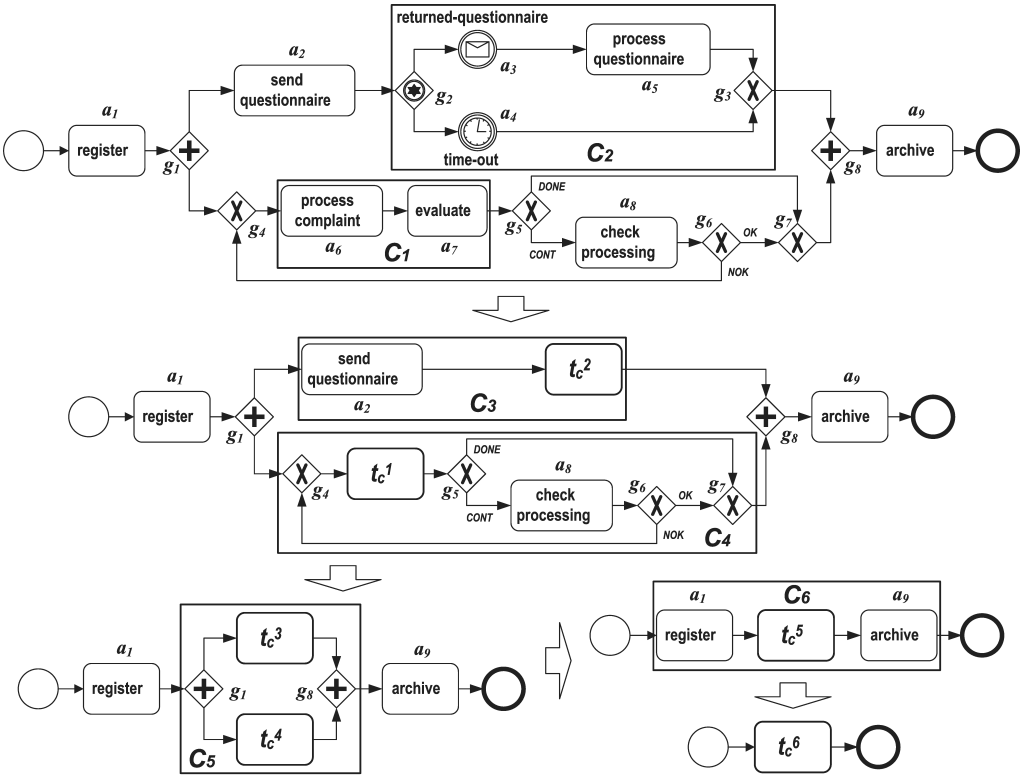

Figure 10. Translating the complaint handling process model in Figure 9 into BPEL.

2nd Translation. When no SEQUENCE-components can be identified, the algorithm tries to discover any non-sequence WSC. As a result, component C_2 is selected. It is a PICK-component and is folded into a task t_c^2 attached with the BPEL code 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 WSCs left in the BPD. The algorithm continues to identify any minimal non-WSC. As a result, the component C_4 is selected. Since C_4 contains cycles, it is not a SPC. Below, we map C_4 onto a scope with event handlers.

```
<scope name="t_c^4">
   <!-- mapping of g_4 -->
   <onEvent msgReceipt(m_(g1,g4))>
      <invoke m_(g4,t_c^1)/>
   </onEvent>
   <onEvent msgReceipt(m_(g6,g4))>
      <invoke m_(g4,t_c^1)/>
   </onEvent>
   <!-- mapping of t_c^1 -->
   <onEvent msgReceipt(m_(g4,t_c^1))>
      <sequence>
         <sequence name="t_c^1"> ... </sequence>
         <invoke m_(t_c^1,g5)/>
      </sequence>
   </onEvent>
   <!-- mapping of g_5 -->
   <onEvent msgReceipt(m_(t_c^1,g5))>
      <if>
         <case condition="branchVar='DONE'">
            <invoke m_(g5,g7)/>
         </case>
         <case condition="branchVar='CONT'">
            <invoke m_(g5,a8)/>
         </case>
      </if>
   </onEvent>
   <!-- mapping of a_8 -->
   <onEvent msgReceipt(m_(g5,a8))>
      <sequence>
         <invoke name="check processing".../>
         <invoke m_(a8,g6)/>
      </sequence>
   </onEvent>
   <!-- mapping of g_6 -->
   <onEvent msgReceipt(m_(a8,g6))>
      <if>
         <case condition="branchVar='OK'">
            <invoke m_(g6,g7)/>
         </case>
         <case condition="branchVar='NOK'">
```

```
            <invoke m_(g6,g4)/>
          </case>
        </if>
    </onEvent>
    <!-- mapping of g7 -->
    <onEvent msgReceipt(m_(g5,g7))>
        <invoke m_(g7,g8)/>
    </onEvent>
    <onEvent msgReceipt(m_(g6,g7))>
        <invoke m_(g7,g8)/>
    </onEvent>
    <!-- to trigger source object g4 -->
    <invoke m_(g1)/>
</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>
```

4.2 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 then the process completes.

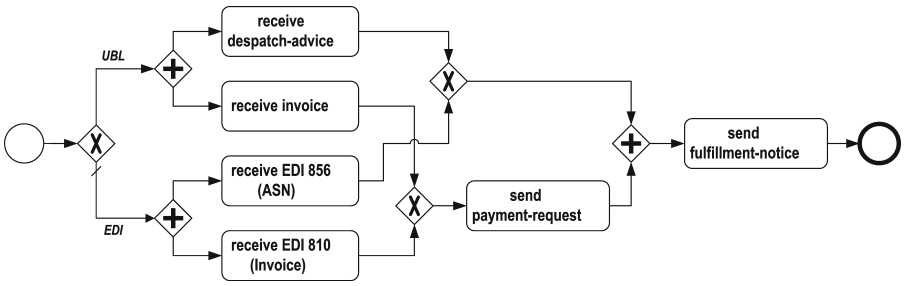

Figure 11. An order fulfillment process model.

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

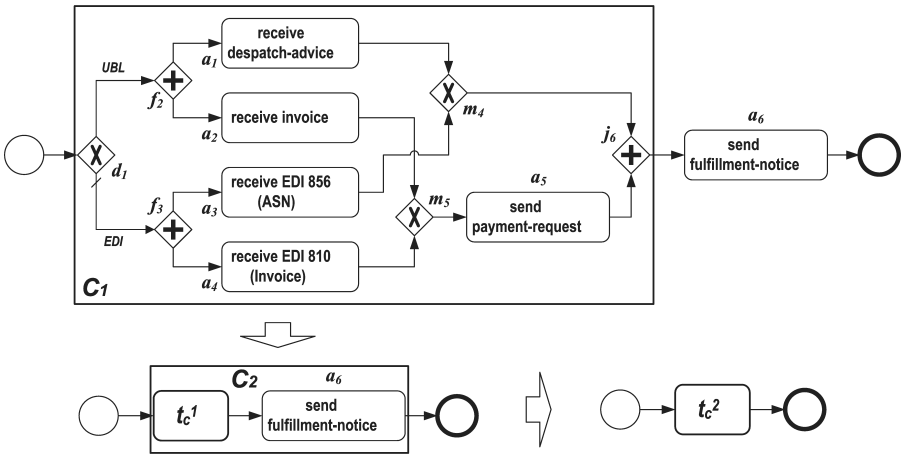

Figure 12. Translating the order fulfillment process model in Figure 11 into BPEL.

1st Translation. Initially, no WSCs can be detected in the BPD shown in Figure 11. Component C_1 consisting of tasks a_1 to a_5 is the only minimal non-WSC identified. Since it is acyclic, has no event-based gateway, and is also proven to be sound and safe (see Appendix A), the component C_1 is a SPC and can be mapped to a link-based flow construct.

First, we pre-process the component C_1 as illustrated in Figure 13. 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). This way we obtain a wrapped component of C_1. 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'.

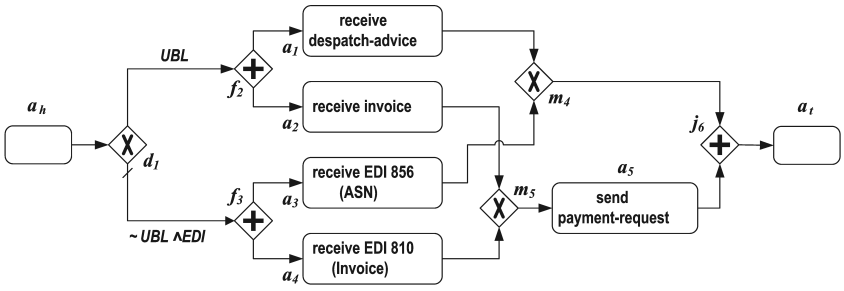

Figure 13. Pre-processing the component C_1 shown in Figure 12.

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

$$\begin{aligned}
&\mathsf{PreTEC\text{-}Sets}(a_h) = \varnothing,\\
&\mathsf{PreTEC\text{-}Sets}(a_1) = \mathsf{PreTEC\text{-}Sets}(a_2) = \{\{(a_h, UBL)\}\},\\
&\mathsf{PreTEC\text{-}Sets}(a_3) = \mathsf{PreTEC\text{-}Sets}(a_4) = \{\{(a_h, \neg UBL \wedge EDI)\}\},\\
&\mathsf{PreTEC\text{-}Sets}(a_5) = \{\{(a_2, \texttt{TRUE})\}, \{(a_4, \texttt{TRUE})\}\}, \text{ and}\\
&\mathsf{PreTEC\text{-}Sets}(a_t) = \{\{(a_1, \texttt{TRUE}), (a_5, \texttt{TRUE})\}, \{(a_3, \texttt{TRUE}), (a_5, \texttt{TRUE})\}\}
\end{aligned}$$

Third, we derive the set of control links for connecting all the tasks in C_1', the set of transition conditions associated with the links, and the join conditions for each of the tasks.

$$\begin{aligned}
&\mathsf{Map2Links}(C_1') =\\
&\quad \textsc{Struct}(\texttt{Links}:\{(a_h, a_1), (a_h, a_2), (a_h, a_3), (a_h, a_4),\\
&\qquad\qquad (a_2, a_5), (a_4, a_5), (a_1, a_t), (a_3, a_t), (a_5, a_t)\})\\
&\qquad \texttt{TransCond}:\{((a_h, a_1), UBL), ((a_h, a_2), UBL),\\
&\qquad\qquad ((a_h, a_3), \neg UBL \wedge EDI), ((a_h, a_4), \neg UBL \wedge EDI),\\
&\qquad\qquad ((a_2, a_5), \texttt{TRUE}), ((a_4, a_5), \texttt{TRUE}),\\
&\qquad\qquad ((a_1, a_t), \texttt{TRUE}), ((a_3, a_t), \texttt{TRUE}), ((a_5, a_t), \texttt{TRUE}))\\
&\qquad \texttt{JoinCond}:\{(a_h, \texttt{TRUE}), (a_1, \mathsf{linkstatus}(a_h, a_1)), (a_2, \mathsf{linkstatus}(a_h, a_2)),\\
&\qquad\qquad (a_3, \mathsf{linkstatus}(a_h, a_3)), (a_4, \mathsf{linkstatus}(a_h, a_4)),\\
&\qquad\qquad (a_5, \mathsf{linkstatus}(a_2, a_5) \vee \mathsf{linkstatus}(a_4, a_5)),\\
&\qquad\qquad (a_t, (\mathsf{linkstatus}(a_1, a_t) \vee \mathsf{linkstatus}(a_3, a_t)) \wedge \mathsf{linkstatus}(a_5, a_t))\}
\end{aligned}$$

Note that task a_h is the source object of component C_1' and has no incoming links. Hence, $\texttt{JoinCond}(a_h) = \texttt{TRUE}$ 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="UBL"/>
   <link name="l_(a_h,a_2)" condition="UBL"/>
   <link name="l_(a_h,a_3)" condition="¬UBL∧EDI"/>
   <link name="l_(a_h,a_4)" condition="¬UBL∧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_(a3,at)) and
                               bpws:getLinkStatus(l_(a5,at)))">
        <target linkName="l_(a1,at)"/>
        <target linkName="l_(a3,at)"/>
        <target linkName="l_(a5,at)"/>
      </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>
```

5 Tool Support and Evaluation

The combined translation technique has been implemented as an open-source tool, namely BPMN2BPEL. It is available at `http://www.bpm.fit.qut.edu.au/projects/babel/tools`. The tool takes as input a BPD represented in XML format and produces a template of a BPEL skeleton. The source XML format follows simple conventions, with elements for different types of nodes (tasks, gateways, events) and for arcs (flows). The generated BPEL skeleton includes structured activities and placeholders for basic activities. The tool provides an extensibility feature to enrich the source BPMN model with implementation details. Specifically, XML elements containing BPEL code can be inserted in the source model and these BPEL code fragments are then copied at the appropriate places in the generated BPEL process definition. Thus, if a BPMN modeling tool is able to produce BPEL code for each individual task and event (e.g. by using additional metadata provided by a developer), the BPMN2BPEL tool can then assemble these BPEL code fragments to produce executable BPEL code. To validate the correctness of the generated BPEL process definitions we created a sample of BPMN models that required a Structured Activity-based, a Control Link-based, or Event-Action Rule-based Translation with their respective variants. This way we could assure that all control paths of the algorithm were taken. We then loaded the generated BPEL code into the Oracle BPEL Process Manager (version 10.1.2)[10] to check for syntactical correctness. Finally, we navigated through

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

both the original BPMN and the generated BPEL to verify that the respective behaviors coincide.

Below, we report the results of an evaluation of our approach and its respective implementation BPMN2BPEL. We use a set of 568 BPMN models from practice and transform them automatically using the BPMN2BPEL tool and analyze the generated BPEL models. This way we aim to get insight into the readability of the resulting models. The sample of the 568 BPMN models was drawn from a larger set of process models that was collected from four real-world modeling projects and described in [24]. The first collection is the SAP Reference Model [16]. It includes a documentation of processes supported by the SAP system. The second collection stems from a German process reengineering project in the service sector. The project was conducted in the late 1990s. The third model collection contains the models of a process documentation project in the Austrian financial industry. The fourth collection covers process models from three different consulting companies. The sample basically includes those models of this larger collection that had exactly one start and one end event, no OR-splits or OR-joins, and that were sound. The number of nodes in these models ranges from three to 59 with an average of ten nodes. We used a script program to trigger the transformation for each of these models and generated a table showing which BPEL elements were created for each of the input models. Finally, we examined the occurrence of structured activities in the BPEL files that were created by the BPMN2BPEL tool.

In a first step we investigated how an increasing number of nodes influences the identification of structured activities by the transformation algorithm. In particular, we aimed to rule out the hypothesis that structure identification would be only applicable for small models. Indeed, this is not the case. Figure 14 illustrated the connection between number of nodes and the number of structured activities that are identified. It can be seen that an increase in nodes results in more structured activities. All models take a position in a corridor from the left bottom to the right top of the diagram. Furthermore, we used inferential statistics to support this observation. Firstly, we found a strong correlation of 0.885 (significance level: 0.001) between both parameters. Secondly, we calculated a linear regression confirming that 87.4% of the variability in the number of structured activities can be explained by the number of nodes (measured as adjusted R^2). Based on an estimated coefficient of 0.312 (significance level: 0.001), we can summarize that the addition of three nodes to a process model led to one additional structured activity in our sample, no matter if the model was large of small. To put it different, the identification of structured activities really paid off since apparently large parts of process models from practice can be mapped to structured activities.

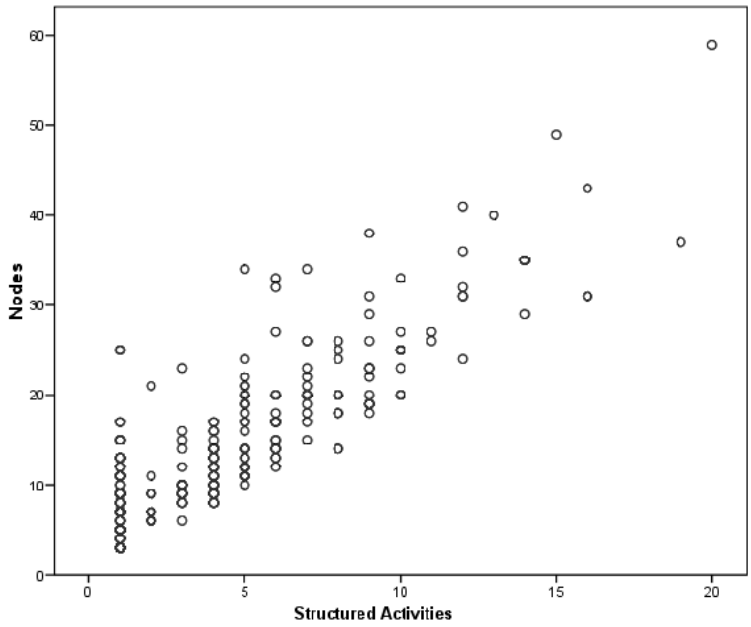

Figure 14. Number of nodes versus number of generated structured activities in the 568 process models.

In a second step, we analyzed how often certain transformation rules were applied. Figure 15 illustrates the results. Please note the logarithmic scale on the x-axis. The statistics strongly support our claim that our algorithm presented in Figure 8 tends to create readable BPEL code with structured activities. In the sample we used, *only three out of 568 models required a mapping to a BPEL flow with control links* and *three models with cycles had to be mapped to BPEL event handlers*. All the remaining control flow could be represented by 1678 structured activities including sequence, if, flow without links, and while.

6 Related Work

White et al. [27, 38] informally outline a translation from BPMN to BPEL. However, this translation does not cover BPDs with arbitrary topologies. Specifically, [27] states that acyclic

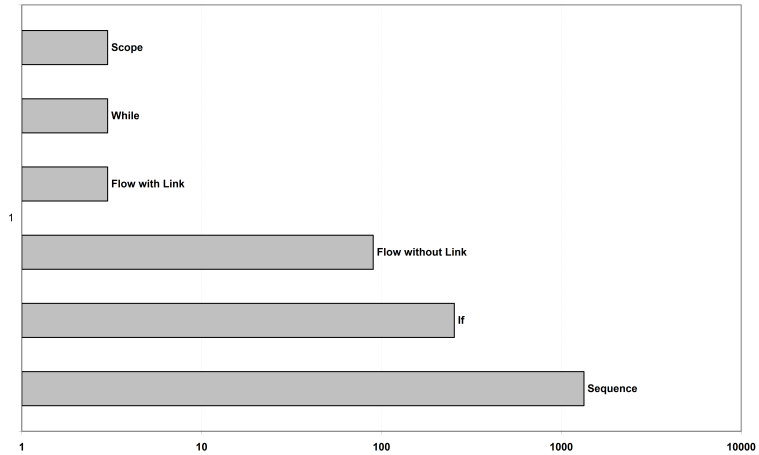

Figure 15. Structured activities and their occurrence in the 568 generated BPEL models.

graphs with arbitrary topologies could be translated to BPEL flow activities with control links, but the details of this translation are left as future work. A method for translating some classes of unstructured cycles is outlined, but no automated and general method is given. Also, several steps in White's translation require human input to identify patterns in the source model. A number of commercial tools can generate BPEL code from BPMN models using methods similar to those outlined by White et al. However, these tools impose intricate syntactic restrictions on the source BPMN model.[11]

Research into structured programming has led to techniques for translating unstructured flowcharts into structured ones [5,10,28,33]. However, these techniques are not applicable when AND-splits and AND-joins are introduced. An identification of situations where unstructured process models can not be translated into equivalent structured ones (under weak bisimulation equivalence) can be found in [18, 21], while an approach to overcome some of these limitations for processes without parallelism is sketched in [19, 39].

Our translation technique relies on the identification of Single-Entry, Single-Exit (SESE) components in the process diagram. Johnson et al. [14] outline an algorithm that identifies all SESE regions in a control flow graph. These regions are arranged in a so-called Program Structure Tree (PST) in which each node corresponds to a SESE region and the SESE region corresponding to a node contains every SESE region corresponding to its children. While our tool implementation currently relies on an algorithm that identifies SESE components one by one, a more efficient implementation could be achieved by exploiting the concept of PST.

In [23] we review a number of techniques for translating graph-oriented process models to BPEL. We classify these translations into four strategies. Firstly, the so-called *Structure-Identification Strategy* works like the *structured activity-based translation approach* presented in this paper. This strategy is applied in many commercial tools that implement BPMN-to-BPEL translations. However, this strategy can not deal with process models with arbitrary topologies. Next, the *Element-Preservation Strategy* aims at translating acyclic graph-oriented models into BPEL process definitions with control links. In the element-preservation strategy, each BPMN gateway is mapped to an empty activity in BPEL, in order to explicitly capture the structure of the source model. This strategy is adopted for example in the mappings from UML Activity Diagrams to BPEL by Gardner [11] and by Mantell [22] as well as in a mapping from Petri nets to BPEL [20]. The *Element-Minimization Strategy* is similar to the element-preservation strategy, except that it tries to reduce the number of empty activities that are introduced in the resulting BPEL definition. In this paper, we have formally characterized the set of acyclic process models that can be mapped to BPEL process definitions with control links, and we have

[11] For a discussion on such restrictions, refer to: `http://www.webcitation.org/5SsMacscU`

presented an algorithm that implements this translation according to an element-minimization strategy. 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. This strategy is applied in [37], but the authors use a simple version of this strategy assuming that the process model is sequential. The technique presented in this paper follows the structure-maximization strategy (combined with the element-minimization one) but without imposing restrictions on the structure of the source model.

This paper combines insights from our previous studies. In [3], we describe a case study where the requirements of a bank system are captured as Coloured Petri nets and the system is then implemented in BPEL. In this study, we used 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 is implemented by a tool called Workflownet2BPEL4WS [4] and is also supported by recent versions of ProM[12]. 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 process models created in student projects [4].

Finally, in [29] 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, and in [30] we take into account the mapping to BPEL block-structured constructs. In this paper, we have extended the previous mappings to cover a broader set of BPMN constructs and to exploit the use of BPEL control links. To improve the readability of the generated BPEL code to a maximum extent, we propose an overall translation algorithm which makes greater use of BPEL's block-structured constructs and control links. Also, in parallel work, Giner et al. [12] have defined and implemented a model transformation from the meta-model used by the the BPMN editor provided by the SOA Tools Platform (STP) to a BPMN meta-model corresponding to our BPMN2BPEL tool, thus providing a seamless bridge between the two tools.

Our work contributes to the broader area of model-driven software development since BPMN can be regarded as a platform-independent model and BPEL as a platform-specific model. Our transformation algorithm clearly shows that mapping between both languages is not always straightforward and requires a trade-off between different design considerations, in our case the benefit of readable structured activities versus the burden of analyzing the BPMN

[12] The ProM framework is available from `http://www.processmining.org`

graph. An alternative approach to generating executable code from process models would be to adapt concepts from domain-specific language design [7, 13, 32]. Our approach is framed in the context of the co-existence of BPMN and BPEL and it aims at bridging these two languages, and more generally, graph-oriented and block-structured process modeling languages.

7 Conclusion

This paper presented an integrated set of techniques to translate models captured using BPMN into BPEL. The proposed techniques are capable of 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) and navigating through both the original BPMN and the generated BPEL to verify that the corresponding behaviors are the same. Furthermore, our evaluation confirms that large parts of process models from practice can be mapped to structured activities.

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 standardization 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 instru-

menting 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.

Overall, our work shows that the co-existence of BPMN and BPEL is not optimal due to their significant mismatch. Market and standardization forces, possibly underpinned by conflicting demands and preferences from different types of stakeholders (e.g. business analysts and software developers), have led to this co-existence. Our contribution bridges these two languages, but still, it does not erase all the disadvantages of their co-existence. We suggest that future standardization efforts should aim at more closely aligning BPMN and BPEL and to reduce the mismatches exposed in this work.

Acknowledgments. The authors are extremely grateful to Stephan Breutel for his efforts in implementing the BPMN2BPEL tool. We also 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. Finally, we thank all the reviewers for their valuable comments which have led to substantial improvements to this paper.

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, A.H.M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.
3. 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 *OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, volume 3760 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 2005.
4. W.M.P. van der Aalst and K.B. Lassen. Translating unstructured workflow processes to readable BPEL: Theory and implementation. Accepted for publication in the *International Journal of Information and Software Technology*, 2007.

5. Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Trans. Software Eng.*, 18(3):237–251, 1992.
6. J. Becker, M. Kugeler, and M. Rosemann, editors. *Process Management. A Guide for the Design of Business Processes*. Springer-Verlag, 2003.
7. M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process modeling in web applications. *ACM Trans. Softw. Eng. Methodol.*, 15(4):360–409, 2006.
8. 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.
9. R. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in BPMN. Accepted for publication in *Information and Software Technology*. Elsevier Science Publications. A technical report version is available via `http://eprints.qut.edu.au/archive/00007115/`.
10. A.M. Erosa and L.J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In H.E. Bal, editor, *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France*, pages 229–240. IEEE Computer Society, 1994.
11. T. Gardner. UML Modelling of Automated Business Processes with a Mapping to BPEL4WS. In *Proceedings of the First European Workshop on Object Orientation and Web Services*. Springer, 2003.
12. P. Giner, V. Torres, and V. Pelechano. Bridging the Gap between BPMN and WS-BPEL: M2M Transformations in Practice. In *Proceedings of the 3rd International Workshop on Model-Driven Web Engineering (MDWE)*, volume 261 of *CEUR Workshop Proceedings*, July 2007.
13. S. Hong, J. Lee, A. Athalye P.M. Djuric, and W.-D. Cho. Design methodology for domain specific parameterizable particle filter realizations. *IEEE Transactions on Circuits and Systems*, 54(9):1987–2000, 2007.
14. R. Johnson, D. Pearson, and K. Pingali. The Program Structure Tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 171–185, Orlando FL, USA, June 1994. ACM Press.
15. D. Jordan and J. Evdemon. *Web Services Business Process Execution Language Version 2.0*. Committee Specification. OASIS WS-BPEL TC, January 2007. Available via `http://www.oasis-open.org/committees/download.php/22475/wsbpel-v2.0-CS01.pdf`.
16. G. Keller and T. Teufel. *SAP(R) R/3 Process Oriented Implementation: Iterative Process Prototyping*. Addison-Wesley, 1998.
17. 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.
18. 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.
19. 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, *OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004*, volume 3290 of *Lecture Notes in Computer Science*, pages 121–138, 2004.
20. A. Koschmider and M. von Mevius. A Petri net based approach for process model driven deduction of BPEL code. In R. Meersman, Z. Tari, and P. Herrero, editors, *OTM Workshops*, volume 3762 of *Lecture Notes in Computer Science*, pages 495–505. Springer, 2005.

21. 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.
22. K. Mantell. From UML to BPEL. URL: `http://www.ibm.com/developerworks/webservices/library/ws-uml2bpel`, September 2005.
23. 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.
24. J. Mendling, G. Neumann, and W.M.P. van der Aalst. Understanding the Occurrence of Errors in Process Models Based on Metrics. In F. Curbera, F. Leymann, and M. Weske, editors, *Proceedings of the OTM Conference on Cooperative information Systems (CoopIS 2007)*, volume 4803 of *Lecture Notes in Computer Science*, pages 113–130. Springer-Verlag, 2007.
25. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
26. 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`.
27. OMG. *Business Process Modeling Notation (BPMN) Version 1.0.* OMG Final Adopted Specification. OMG, February 2006. Available via `http://www.bpmn.org/`.
28. G. Oulsnam. Unravelling unstructured programs. *Computer Journal*, 25(3):379–387, 1982.
29. 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`.
30. C. Ouyang, M. Dumas, A.H.M. ter Hofstede, and W.M.P. van der Aalst. From BPMN process models to BPEL Web services. In *Proceedings of 4th International Conference on Web Services (ICWS'06)*, pages 285–292, Chicargo, Illinois, USA, September 2006. IEEE Computer Society.
31. 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. *Science of Computer Programming*, 67(2-3):162–198, 2007. A technical report version is available via `http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-15.pdf`.
32. Scott Pakin. The design and implementation of a domain-specific language for network performance testing. *IEEE Trans. Parallel Distrib. Syst.*, 18(10):1436–1449, 2007.
33. L. Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM*, 35(4):893–920, 1988.
34. 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.
35. 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.

36. 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.
37. D. Skogan, R. Grønmo, and I. Solheim. Web service composition in UML. In *8th International Enterprise Distributed Object Computing Conference (EDOC 2004), 20-24 September 2004, Monterey, California, USA, Proceedings*, pages 47–57. IEEE Computer Society, 2004.
38. S. White. Using BPMN to Model a BPEL Process. *BPTrends*, 3(3):1–18, March 2005.
39. W. Zhao, R. Hauser, K. Bhattacharya, B.R. Bryant, and F. Cao. Compiling business processes: untangling unstructured loops in irreducible flow graphs. *Int. Journal of Web and Grid Services*, 2(1):68–91, 2006.

Appendix A Petri-net Semantics of BPMN

We use Petri nets [25] to define formal semantics for BPMN.[13] Below, we first introduce the basic Petri net terminology and notations. Readers familiar with Petri nets can skip this introduction. Then, we specify a mapping from a BPD component to Petri nets. Based on this, we finally define the notions of soundness and safeness of a BPD component.

A.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 represented by circles and transitions by rectangles.

Definition 9 (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. 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 active components in a Petri net: they change the state of the net based on 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.

[13] The current BPMN specification [27] describes BPMN in natural language and does not contain a formal semantics of BPMN.

(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 10 (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 11 (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.*

A.2 Semantics of BPD Components

Figure 16 depicts the mapping from core BPMN objects to Petri-net modules. A task or an intermediate event is transformed into a transition with one input place and one output place. The occurrence of the transition models the execution of that task or event. Gateways, except event-based decision gateways, are mapped onto small Petri-net modules where transitions are used to capture their routing behavior. These transitions are considered as "silent" transitions [25]. In the case of a fork or join gateway, only one transition is used, which is uniquely identified by that gateway. For a data-based decision gateway, multiple transitions are used and each of them is identified by 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 the gateway and one of the gateway's input objects. For an event-based gateway, the race condition between events or receive tasks is captured in a way that all the corresponding event/task transitions share the same output place from the gateway's input object. Finally, places are used to link the Petri net modules of two connecting BPMN objects and therefore is 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 "E1" is followed by task "T1", the output place of transition E1 becomes the input place of transition T1. A formal definition of the mapping from BPD components to Petri nets is given as follows.

Definition 12 (Petri net semantics of BPD 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*

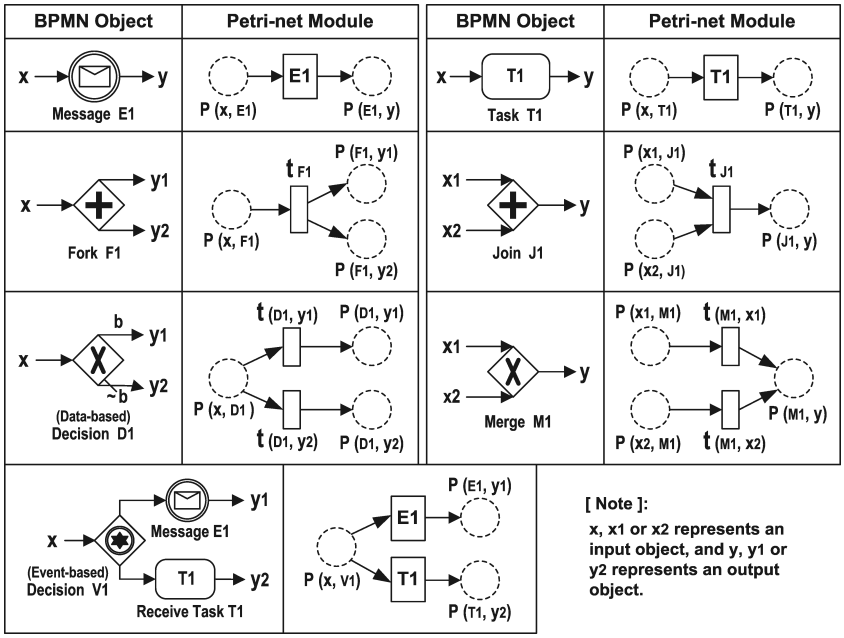

Figure 16. Mapping BPMN objects onto Petri-net modules.

a BPD (see Definition 2), 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:*

$$
\begin{aligned}
P' = {} & \{p_{(\mathsf{entry}(\mathcal{C}),i_c)}, p_{(o_c,\mathsf{exit}(\mathcal{C}))}\} \cup && \text{– } \textit{source/sink place} \\
& \{p_{(x,y)} \mid x \mathcal{F}_c y \wedge x \notin \mathcal{G}_c^V\} && \text{– } \textit{other places} \\
\\
T' = {} & \mathcal{T}_c \cup \mathcal{E}_c \cup && \text{– } \textit{task/event} \\
& \{t_x \mid x \in \mathcal{G}_c^F \cup \mathcal{G}_c^J\} \cup && \text{– } \textit{fork/join} \\
& \{t_{(x,y)} \mid x \in \mathcal{G}_c^D \wedge y \in \mathsf{out}(x)\} \cup && \text{– } \textit{data decision} \\
& \{t_{(x,y)} \mid x \in \mathcal{G}_c^M \wedge y \in \mathsf{in}(x)\} \cup && \text{– } \textit{merge} \\
\\
F' = {} & \{(p_{(x,y)}, y) \mid y \in \mathcal{T}_c \cup \mathcal{E}_c \wedge x \in \mathsf{in}(y) \wedge x \notin \mathcal{G}_c^V\} \cup \\
& \{(y, p_{(y,z)})\} \mid y \in \mathcal{T}_c \cup \mathcal{E}_c \wedge z \in \mathsf{out}(y)\} \cup && \text{– } \textit{task/event} \\
& \{(p_{(x,y)}, t_y)\} \mid y \in \mathcal{G}_c^F \cup \mathcal{G}_c^J \wedge x \in \mathsf{in}(y)\} \cup \\
& \{(t_y, p_{(y,z)})\} \mid y \in \mathcal{G}_c^F \cup \mathcal{G}_c^J \wedge z \in \mathsf{out}(y)\} \cup && \text{– } \textit{fork/join} \\
& \{(p_{(x,y)}, t_{(y,z)}) \mid y \in \mathcal{G}_c^D \wedge x \in \mathsf{in}(y) \wedge z \in \mathsf{out}(y)\} \cup \\
& \{(t_{(y,z)}, p_{(y,z)}) \mid y \in \mathcal{G}_c^D \wedge z \in \mathsf{out}(y)\} \cup && \text{– } \textit{data decision} \\
& \{(p_{(x,y)}, t_{(y,x)}) \mid y \in \mathcal{G}_c^M \wedge x \in \mathsf{in}(y)\} \cup \\
& \{(t_{(y,x)}, p_{(y,z)}) \mid y \in \mathcal{G}_c^M \wedge x \in \mathsf{in}(y) \wedge z \in \mathsf{out}(y)\} \cup && \text{– } \textit{merge} \\
& \{(p_{(x,y)}, z) \mid y \in \mathcal{G}_c^V \wedge x \in \mathsf{in}(y) \wedge z \in \mathsf{out}(y)\} && \text{– } \textit{event decision}
\end{aligned}
$$

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 modeling 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.

A.3 Soundness and Safeness of BPD Components

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 12 to a BPD component results in a Petri net satisfying some desirable properties which make automated analysis easier. In particular, we are interested in the soundness and safeness properties which are used for identifying a SPC for control link-based translation approach in Section 3.3. To capture these properties, it is necessary to introduce the concepts of *WorkFlow nets* (WF-net) [1] and *free-choice nets* [8]. 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 13 (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*;[14] and (2) *there should be no dead tasks, i.e., it should be possible to execute an arbitrary task*

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

by following the appropriate route through the WF-net. In [1] we have shown that soundness corresponds to liveness and boundedness.

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 [1].

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 12, 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 12.

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 abvove two theorems demonstrate that results related to safeness and boundedness of free-choice nets can be used to check the soundness of a BPD component 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). Next, as a free-choice WF-net, PN_c is guaranteed to be safe if it is proven to be sound.

Example. During the translation of the BPMN model of the order fulfillment process into BPEL in Section 4.2, we map component C_1 shown in Figure 12 onto the Petri net shown in Figure 17 which is a free-choice WF-net. We can prove that this free-choice WF-net is sound and safe and therefore the component C_1 in Figure 12 is sound and safe.

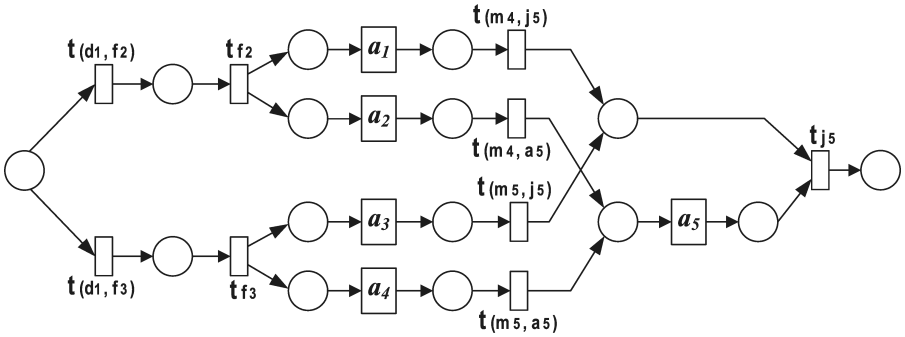

Figure 17. The Petri net mapping of component C_1 shown in Figure 12.

Finally, a more comprehensive mapping from BPMN to Petri nets can be found in [9]. The tool implementation, namely BPMN2PNML, is freely available at `http://is.tm.tue.nl/staff/rdijkman/cbd.html#transformer`. Petri nets that result from the application of BPMN2PNML can be subjected to a subsequent analysis in ProM and Woflan. For example, it can be checked whether they are sound and safe.

Appendix B An XML Format of BPMN

The BPMN2BPEL tool takes as input a BPD represented in XML format. This XML format follows simple conventions, with elements for different types of nodes (tasks, gateways, events) and for arcs (flows). For example, the BPD of the complaint handling process shown in Figure 9 can be written as an XML document shown below. Note that the node type is "task" by default.

```
<bpmn>
  <process id="ComplaintHandling">
    <variables>
      <variable name="DONE"/>
      <variable name="CONT"/>
      <variable name="OK"/>
      <variable name="NOK"/>
    </variables>
    <nodes>
```

```
    <node id="a0" name="start" type="StartEvent"/>
    <node id="a1" name="register"/>
    <node id="a2" name="sendQuestionnaire"/>
    <node id="a3" name="returnedQuestionaire" type="MessageEvent"/>
    <node id="a4" name="timeOut" type="TimerEvent"/>
    <node id="a5" name="processQuestionnaire"/>
    <node id="a6" name="processComplaint"/>
    <node id="a7" name="evaluate"/>
    <node id="a8" name="checkProcessing"/>
    <node id="a9" name="archive"/>
    <node id="a10" name="end" type="EndEvent"/>
    <node id="g1" type="AND-Split"/>
    <node id="g2" type="EB-XOR-Split"/>
    <node id="g3" type="XOR-Join"/>
    <node id="g4" type="XOR-Join"/>
    <node id="g5" type="XOR-Split"/>
    <node id="g6" type="XOR-Split"/>
    <node id="g7" type="XOR-Join"/>
    <node id="g8" type="AND-Join"/>
</nodes>
<arcs>
    <arc id="f0" source="a0" target="a1"/>
    <arc id="f1" source="a1" target="g1"/>
    <arc id="f2" source="g1" target="a2"/>
    <arc id="f3" source="g1" target="g4"/>
    <arc id="f4" source="a2" target="g2"/>
    <arc id="f5" source="g2" target="a3"/>
    <arc id="f6" source="g2" target="a4"/>
    <arc id="f7" source="a3" target="a5"/>
    <arc id="f8" source="a5" target="g3"/>
    <arc id="f9" source="a4" target="g3"/>
    <arc id="f10" source="g3" target="g8"/>
    <arc id="f11" source="g4" target="a6"/>
    <arc id="f12" source="a6" target="a7"/>
```

```
        <arc id="f13" source="a7" target="g5" />
        <arc id="f14" source="g5" target="g7" guard="DONE" />
        <arc id="f15" source="g5" target="a8" guard="CONT" />
        <arc id="f16" source="a8" target="g6" />
        <arc id="f17" source="g6" target="g7" guard="OK" />
        <arc id="f18" source="g6" target="g4" guard="NOK" />
        <arc id="f19" source="g7" target="g8" />
        <arc id="f20" source="g8" target="a9" />
        <arc id="f21" source="a9" target="a10" />
      </arcs>
    </process>
</bpmn>
```