Translating BPMN to BPEL*

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

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

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

Abstract. The Business Process Modelling Notation (BPMN) is a graph-oriented language in which control and action nodes can be connected almost arbitrarily. It is supported by various modelling tools but so far no systems can directly execute BPMN models. The Business Process Execution Language for Web Services (BPEL) on the other hand is a mainly block-structured language supported by several execution platforms. In the current setting, mapping BPMN models to BPEL code is a necessary step towards unified and standards-based business process development environments. It turns out that this mapping is challenging from a scientific viewpoint as BPMN and BPEL represent two fundamentally different classes of languages. Existing methods for mapping BPMN to BPEL impose limitations on the structure of the source model, especially with respect to cycles. This report proposes a technique that overcomes these limitations. Beyond its direct relevance in the context of BPMN and BPEL, this technique addresses difficult problems that arise generally when translating between flow-based languages with parallelism.

1 Introduction

The *Business Process Execution Language for Web Services* (BPEL) [4] is emerging as a de-facto standard for implementing business processes on top of web services technology. Numerous platforms support the execution of BPEL processes (see `http://en.wikipedia.org/wiki/BPEL`). Some of these platforms also provide graphical editing tools for defining BPEL processes. However, these tools directly follow the syntax of BPEL without elevating the level of abstraction to make them usable during the analysis and design phases of the development cycle. On the other hand, the *Business Process Modelling Notation* (BPMN) [10] has attained some level of adoption among business analysts and system architects as a language for defining business process blueprints for subsequent implementation. Despite being a recent proposal, BPMN is already supported by

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

more than 30 tools (see `www.bpmn.org`). Consistent with the level of abstraction targeted by BPMN, none of these tools supports the execution of BPMN models directly. Instead, some of them support the translation of BPMN to BPEL.

Close inspection of existing translations from BPMN to BPEL, e.g. the one sketched in [10], shows that these translations fail to fulfill the following key requirements: (i) completeness, i.e. applicable to any BPMN model; (ii) automation, i.e. capable of producing target code without requiring human intervention to identify patterns in the source model; and (iii) readability, i.e. consistently producing target code that is understandable by humans. The latter requirement is important since the BPEL definitions produced by the translation are likely to require refinement (e.g. to specify partner links and data manipulation expressions) as well as testing and debugging. If BPEL was only intended as a language for machine consumption and not for human use, it could be replaced by mainstream programming languages or even (virtual) machine languages, but this would defeat the purpose of BPEL as a domain-specific language for service composition.

The limitations of existing BPMN-to-BPEL translations are not surprising given that BPMN and BPEL belong to two fundamentally different classes of languages. BPMN is graph-oriented while BPEL is mainly block-structured (albeit providing graph-oriented constructs with syntactical limitations). Mapping between graph-oriented and block-structured process definition languages is notoriously challenging. In the case of flowcharts, mapping unstructured charts to structured ones is a well-understood problem. However, graph-oriented process definition languages extend flowcharts with parallelism (i.e. AND-splits and AND-joins) and other constructs such as deferred choice [1].

This paper addresses the challenge of proposing a translation from BPMN to BPEL addressing all three requirements above. This is a first step towards model-driven, standards-based tools for developing process-oriented web services. Beyond its direct relevance in this context, the proposed technique addresses difficult problems that arise when translating from graph-oriented process languages (e.g. UML Activity Diagrams, YAWL, or Petri nets) to block-structured ones.

The remainder of the report is structured as follows. Section 2 gives an overview of BPMN and BPEL and reviews related work. Section 3 presents an algorithm for translating BPMN into BPEL. The translation algorithm is then illustrated through a case study in Section 4. Finally, Section 5 concludes and outlines future work.

2 Background and Related Work

2.1 BPEL and BPMN

BPEL [4] is essentially an extension of imperative programming languages with constructs specific to web service implementations. A BPEL process definition relates a number of *activities*. An activity is either a basic or a structured activity. *Basic activities* correspond to atomic actions such as: *invoke*, invoking

an operation on a web service; *receive*, waiting for a message from a partner; *exit*, terminating the entire service instance; *empty*, doing nothing; and etc. To enable the presentation of complex structures the following *structured activities* are defined: *sequence*, for defining an execution order; *flow*, for parallel routing; *switch*, for conditional routing; *pick*, for race conditions based on timing or external triggers; *while*, for structured looping; and *scope*, for grouping activities into blocks to which event, fault and compensation handlers may be attached. In particular, an event handler is an *event-action rule* associated with a scope. It is enabled when the associated scope is under execution and may execute concurrently with the main activity of the scope. When an occurrence of the event associated with an enabled event handler is registered (and this may be a message receipt or a timeout), the body of the handler is executed. The completion of the scope as a whole is delayed until all active event handlers have completed. *Fault* and *compensation handlers* are designed for exception handling and are not used further in this report.

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

BPMN essentially provides a graphical notation for business process modelling, with an emphasis on control-flow. It defines a *Business Process Diagram* (BPD), which is a kind of flowchart incorporating constructs tailored to business process modelling, such as AND-split, AND-join, XOR-split, XOR-join, and deferred (event-based) choice. We describe BPMN in more detail when we introduce the mapping.

2.2 Related Work

White [9,10] informally sketches a translation from BPMN to BPEL. However, as acknowledged in [10] this translation is fundamentally limited, e.g. it excludes diagrams with arbitrary topologies. Also, several steps in White's translation require human input to identify patterns in the source model.

Research into structured programming in the 60s and 70s led to techniques for translating unstructured flowcharts into structured ones. However, these techniques are not applicable when AND-splits and AND-joins are introduced. An identification of situations where unstructured process diagrams cannot be translated into equivalent structured ones (under weak bisimulation equivalence) can be found in [5, 7], while an approach to overcome some of these limitations for processes without parallelism is sketched in [6]. However, these related work only address a piece of the puzzle of translating from graph-oriented process modelling languages to BPEL.

This paper combines insights from two of our previous publications. In [2], we describe a case study where the requirements of a bank system are captured as Coloured Workflow nets (a subclass of Coloured Petri nets) and the system is then implemented in BPEL. In this study we use a semi-automated mapping from (Coloured) Petri nets to BPEL [3] that has commonalities with a subset of the translation discussed in this paper. In [8], we present a mapping from a graph-oriented language supporting AND-splits, AND-joins, XOR-splits, and XOR-joins, into BPEL. In the following section, we extend this previous mapping to cover a broader set of BPMN constructs and to improve the readability of the generated code. Whereas in [8] the generated code relies heavily on BPEL event handlers, in this paper we make greater use of BPEL's block-structured constructs.

3 Mapping BPMN onto BPEL

This section presents a mapping from BPMN to BPEL. The mapping focuses on the control-flow perspective. First, we define the syntax of a *core* subset of BPDs used for mapping. Second, we discuss the transformation of a BPD from a graph structure to a block structure. We use the term "components" to refer to subsets of a BPD. A component may be *well-structured* so that it can be directly mapped onto BPEL structured activities, whereas a component that does not preserve such property can be translated into BPEL via *event-action rules*. We identify these two categories of components and introduce the corresponding translation approaches in two separate subsections. Finally, we propose the algorithm for mapping an entire BPD onto BPEL.

3.1 Business Process Diagram (BPD)

BPMN uses BPDs to describe business processes. A BPD is made up of BPMN elements. We consider a core subset of BPMN elements shown in Figure 1. There are *objects* and *sequence flow*. The 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*), the immediate termination of a process (*end terminate 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. A gateway is a routing construct used to control the divergence and convergence of sequence flow. There are: *parallel fork gateways* for creating concurrent sequence flows, *parallel join gateways* for synchronizing concurrent sequence flows, *data/event-based XOR decision gateways* for selecting one out of a set of mutually exclusive alternative sequence flows where the choice is based on either the process data (data-based) or external event (event-based), and *XOR merge gateways* for joining a set of mutually exclusive alternative sequence flows into one sequence flow.

A BPD, which is made up of the above core subset of BPMN elements, is hereafter referred to as a *core BPD*.

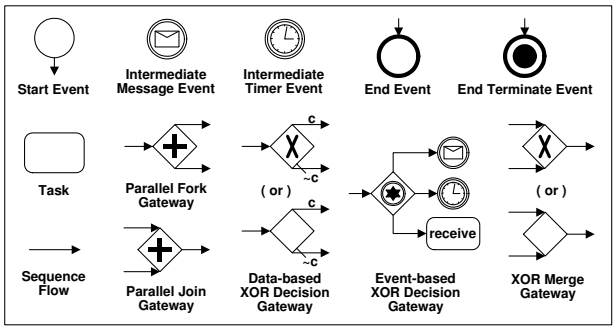

Figure. 1. A core subset of BPMN elements.

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

- $\mathcal{O}$ *is a set of objects which can be partitioned into disjoint sets of tasks* $\mathcal{T}$*, events* $\mathcal{E}$ *and gateways* $\mathcal{G}$*,*
- $\mathcal{T}^R \subseteq \mathcal{T}$ *is a set of receive tasks,*
- $\mathcal{E}$ *can be partitioned into disjoint sets of start events* $\mathcal{E}^S$*, intermediate events* $\mathcal{E}^I$ *and end events* $\mathcal{E}^E$*,*
- $\mathcal{E}^I$ *can be partitioned into disjoint sets of intermediate message events* $\mathcal{E}^I_M$ *and timer events* $\mathcal{E}^I_T$*,*
- $\mathcal{E}^E_T \subseteq \mathcal{E}^E$ *is a set of end terminate events,*
- $\mathcal{G}$ *can be partitioned into disjoint sets of parallel fork gateways* $\mathcal{G}^F$*, parallel join gateways* $\mathcal{G}^J$*, data-based XOR decision gateways* $\mathcal{G}^D$*, event-based XOR decision gateways* $\mathcal{G}^V$*, and XOR merge gateways* $\mathcal{G}^M$*,*
- $\mathcal{F} \subseteq \mathcal{O} \times \mathcal{O}$ *is the control flow relation,*
- Cond*:* $\mathcal{F} \nrightarrow \mathcal{C}$ *is a function mapping sequence flows within* $dom(\mathsf{Cond}) = \mathcal{F} \cap (\mathcal{G}^D \times \mathcal{O})$ *to conditions.*[3]

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

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

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

- $\forall\ s \in \mathcal{E}^S$*,* $\mathsf{in}(s) = \varnothing \wedge |\mathsf{out}(s)| = 1$*, i.e. start events have an indegree of zero and an outdegree of one,*

[3] A condition is a boolean function operating over a set of propositional variables that can be abstracted out of the control flow definition. The condition may evaluate to true or false, which determines whether or not the associated sequence flow is taken during the process execution.

- $\forall\ e \in \mathcal{E}^E$, $\mathsf{out}(e) = \varnothing \wedge |\mathsf{in}(e)| = 1$, *i.e., end events have an outdegree of zero and an indegree of one,*
- $\forall\ x \in \mathcal{T} \cup \mathcal{E}^I$, $|\mathsf{in}(x)| = 1$ *and* $|\mathsf{out}(x)| = 1$, *i.e. tasks and intermediate events have an indegree of one and an outdegree of one,*
- $\forall\ g \in \mathcal{G}^F \cup \mathcal{G}^D \cup \mathcal{G}^V$: $|\mathsf{in}(g)| = 1 \wedge |\mathsf{out}(g)| > 1$, *i.e. fork or decision gateways have an indegree of one and an outdegree of more than one,*
- $\forall\ g \in \mathcal{G}^J \cup \mathcal{G}^M$, $|\mathsf{out}(g)| = 1 \wedge |\mathsf{in}(g)| > 1$, *i.e. join or merge gateways have an outdegree of one and an indegree of more than one,*
- $\forall\ g \in \mathcal{G}^V$, $\mathsf{out}(g) \subseteq \mathcal{E}^I \cup \mathcal{T}^R$, *i.e. event-based XOR decision gateways must be followed by intermediate events or receive tasks,*
- $\forall g \in \mathcal{G}^D$, $\exists x \in \mathsf{out}(g)$, $\mathsf{Cond}((g,x)) = \neg \bigwedge_{y \in \mathsf{out}(g) \setminus \{x\}} \mathsf{Cond}((g,y))$, *i.e.* (g,x) *is the default flow among all the outgoing flows from* g,
- $\forall\ x \in \mathcal{O}$, $\exists\ (s,e) \in \mathcal{E}^S \times \mathcal{E}^E$, $s\mathcal{F}^* x \wedge x\mathcal{F}^* e$,[4] *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, without losing generality we assume that both $\mathcal{E}^S$ and $\mathcal{E}^E$ are singletons, i.e. $\mathcal{E}^S = \{s\}$ and $\mathcal{E}^E = \{e\}$.

3.2 Decomposing a BPD into components

We would like to achieve two goals when mapping BPMN onto BPEL. One is to define an algorithm which allows us to translate each well-formed core BPD into a valid BPEL process, the other is to generate readable and compact BPEL code. To map a BPD onto (readable) BPEL code, we need to transform a graph structure into a block structure. For this purpose, we decompose a BPD into *components*. A component is a subset of the BPD that has one entry point and one exit point. We then try to map components onto suitable "BPEL blocks". For example, a component holding a purely sequential structure should be mapped onto a BPEL *sequence* construct while a component holding a parallel structure should be mapped onto a *flow* construct.

The next two subsections describe how to map components of a BPD onto BPEL constructs. Before describing the mapping, this section first formalizes the notion of components. To facilitate the definitions, we specify an auxiliary function elt over a domain of singletons, i.e., if $X = \{x\}$, then $\mathsf{elt}(X) = x$.

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

- $\mathcal{O}_c \subseteq \mathcal{O} \setminus (\mathcal{E}^S \cup \mathcal{E}^E)$,
- $|(\bigcup_{x \in \mathcal{O}_c} \mathsf{in}(x)) \setminus \mathcal{O}_c| = 1$, *i.e., there is a single entry point outside the component, which can be denoted as* $\mathsf{entry}(\mathcal{C}) = \mathsf{elt}((\bigcup_{x \in \mathcal{O}_c} \mathsf{in}(x)) \setminus \mathcal{O}_c)$,
- $|(\bigcup_{x \in \mathcal{O}_c} \mathsf{out}(x)) \setminus \mathcal{O}_c| = 1$, *i.e., there is a single exit point outside the component, which can be denoted as* $\mathsf{exit}(\mathcal{C}) = \mathsf{elt}((\bigcup_{x \in \mathcal{O}_c} \mathsf{out}(x)) \setminus \mathcal{O}_c)$,

[4] $\mathcal{F}^*$ is a reflexive transitive closure of $\mathcal{F}$, i.e. $x\mathcal{F}^* y$ if there is a path from x to y in $\mathcal{BPD}$.

– *there exists a unique source object* $i_c \in \mathcal{O}_c$ *and a unique sink object* $o_c \in \mathcal{O}_c$ *and* $i_c \neq o_c$*, such that* $\mathsf{entry}(\mathcal{C}) \in \mathsf{in}(i_c)$ *and* $\mathsf{exit}(\mathcal{C}) \in \mathsf{out}(o_c)$*,*
– $\mathcal{F}_c = \mathcal{F} \cap (\mathcal{O}_c \times \mathcal{O}_c)$*,*
– 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 included in the final translation algorithm in Section 3.5.

The decomposition of a BPD helps to define an iterative approach which allows us to incrementally transform a "componentized" BPD to a block-structured BPEL process. Below, we define function Fold that replaces a component by a single task object in a BPD. This function can be used to perform iterative reduction of a componentized BPD until no component is left in the BPD. The function will play a crucial role in the final translation algorithm where we incrementally replace BPD components by BPEL constructs.

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

– $\mathcal{O}' = (\mathcal{O} \backslash \mathcal{O}_c) \cup \{t_c\}$*,*
– *if* $\mathcal{T}_c$ *denotes the set of tasks in* $\mathcal{C}$ *and* $\mathcal{T}'$ *denotes the set of tasks in* $\mathsf{Fold}(\mathcal{BPD}, \mathcal{C}, t_c)$*, then* $\mathcal{T}_c \subseteq \mathcal{O}_c$*,* $\mathcal{T}' \subseteq \mathcal{O}'$*, and* $\mathcal{T}' = (\mathcal{T} \backslash \mathcal{T}_c) \cup \{t_c\}$*,*
– $\mathcal{F}' = (\mathcal{F} \cap (\mathcal{O} \backslash \mathcal{O}_c \times \mathcal{O} \backslash \mathcal{O}_c)) \cup \{(\mathsf{entry}(\mathcal{C}), t_c), (t_c, \mathsf{exit}(\mathcal{C})\}$*,*
– $\mathsf{Cond}' = \begin{cases} \mathsf{Cond}[\mathcal{F}'] & \text{if } \mathsf{entry}(\mathcal{C}) \notin \mathcal{G}^D \\ \mathsf{Cond}[\mathcal{F}'] \cup \{((\mathsf{entry}(\mathcal{C}), t_c), (\mathsf{entry}(\mathcal{C}), i_c))\} & \text{otherwise} \end{cases}$

3.3 Structured activity-based translation

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

Definition 5 (Well-structured components). *Let* $\mathcal{BPD} = (\mathcal{O}, \mathcal{F}, \mathsf{Cond})$ *be a well-formed core BPD and* $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$ *be a component of* $\mathcal{BPD}$*.* i_c *is the source object of* $\mathcal{C}$ *and* o_c *is the sink object of* $\mathcal{C}$*. The following components are well-structured:*

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

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

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

(*d*) $\mathcal{C}$ *is a PICK-component if*
- $i_c \in \mathcal{G}^V \wedge o_c \in \mathcal{G}^M$,
- $\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I \cup \{i_c, o_c\}$,
- $\forall\ x \in \mathit{out}(i_c)$, $\exists\ y \in \mathcal{O}_c \backslash (\{i_c\} \cup \mathit{out}(i_c))$, $\mathit{in}(y) = \{x\} \wedge \mathit{in}(x) = \{y\}$,
- $\forall\ y' \in \mathcal{O}_c \backslash (\{i_c, o_c\} \cup \mathit{out}(i_c))$, $\mathit{out}(y')=\{o_c\}$.

(*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 depicts examples of mapping each of the components mentioned above onto the corresponding BPEL structured activities. Using function Fold in Definition 4, a component $\mathcal{C}$ is replaced by a single task t_c attached with the corresponding BPEL translation of $\mathcal{C}$. For simplicity, we assume that an initial task object (t_1, ..., or t_n) in component $\mathcal{C}$ is mapped onto an *invoke* activity. However, it should be noted that based on the nature of these task objects they may be mapped onto any types of BPEL activities.

In Figure 2(a) to (e), the mappings of the five components, SEQUENCE, FLOW, SWITCH, PICK and WHILE, are straightforward. Note that in a PICK-component (Figure 2(d), an event-based XOR decision gateway must be followed by receive tasks or intermediate message or timer events. For this reason, a SEQUENCE-component (Figure 2(a)) cannot be preceded by an event-based XOR decision gateway.

In Figure 2(f) and (g), the two components, REPEAT and REPEAT+WHILE, represent repeat loops which are also structured loops. Repeat loops are the opposite of while loops. A while loop (see Figure 2(e)) evaluates the loop condition *before* the body of the loop is executed, so that the loop is never executed if the condition is initially false. In a repeat loop, the condition is checked *after* the body of the loop is executed, so that the loop is always executed at least once. Accordingly, it is not difficult to transform a repeat loop into a while loop. In

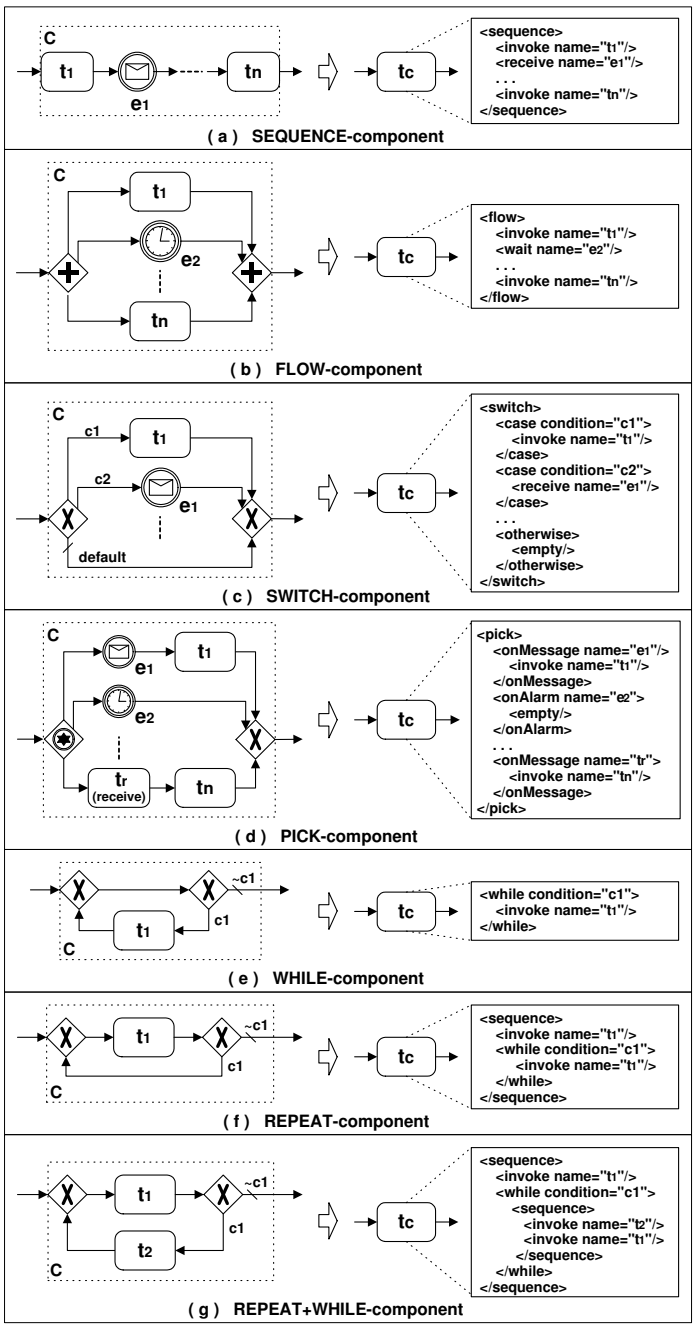

Figure. 2. Examples of folding a well-structured component $\mathcal{C}$ into a single task object t_c attached with the corresponding BPEL translation of $\mathcal{C}$.

Figure 2(f), the semantics of a repeat loop of task t_1 is equivalent to a single execution of t_1 followed by a while loop of t_1. In Figure 2(g), a repeat loop of task t_1 is combined with a while loop of task t_2, and both loops share one loop condition. In this case, task t_1 is always executed once before the initial evaluation of the condition, which is then followed by a while loop of sequential execution of t_2 and t_1.

3.4 Event-action rule-based translation

A well-formed core BPD may contain components that are not well-structured, e.g. components capturing multi-merge patterns [1] or unstructured loops. To map these components onto BPEL, the structured activity-based approach mentioned above is no longer applicable.

Definition 6 (Non-well-structured components). *Let* $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$ *be a component of a well-formed core BPD.* $\mathcal{C}$ *is not well structured if it does not match any of the "patterns" given in Definition 5.* $\mathcal{C}$ *is a minimal non-well-structured component if* $\mathcal{C}$ *is not well structured and there is no other component* $\mathcal{C}' = (\mathcal{O}'_c, \mathcal{F}'_c, \mathsf{Cond}_c')$ *such that* $\mathcal{O}'_c \subset \mathcal{O}_c$.

In the following, We present an approach that can be used to translate a non-well-structured component into a *scope* activity by exploiting the "event handler" construct in BPEL. An event handler is an *event-action rule* associated with a scope, and thus the corresponding translation approach is based on event-action rules. It should be mentioned that the so-called event-action rule-based approach can be applied to translating any component to BPEL. However, this approach produces less readable BPEL code and hence we resort only to this approach when there are only non-well-structured components left in the BPD.

As the first step in the event-action rule-based translation, we generate a set of *preconditions* for each object within a component. The term "precondition" is used to capture one possible way that leads to the execution of an object, and thus a set of preconditions associated with the object encodes all possible ways of reaching that object.

Figure 3 shows an algorithm for generating all precondition sets for a component. The algorithm is sketched using a functional programming notation. It defines three functions. The first one, namely AllPreCondSets, generates a set of precondition sets for all objects (given by the set Objects($\mathcal{C}$)) in component $\mathcal{C}$ by relying on a second function named PreCondSet. PreCondSet computes the set of preconditions for an object by relying on a third function named EventOnFlow.

Function EventOnFlow takes as input a sequence flow f and produces a single event[5] resulting from the execution of the source object of the flow (denoted as Source(f)). If the flow's source object xs is outside the component to which the flow's target object (Target(f)) belongs, it implies that the flow's target object x is the source object of the above component (Component(x)). In this

[5] Note that these are events within the context of event-action rules, and are different from BPMN event objects.

case, the function returns an event (Start(Component(x))) signalling to start the execution of this component. Otherwise, the function operates based on the type of xs (ObjectType(xs)). If xs is one of the objects (tasks, events, or join or merge gateways) that have only one *outgoing* flow, an event (end(xs)) is returned indicating the end of the execution of xs. Intuitively, this means that the flow in question (f) may be taken when the object xs has completed its execution. Otherwise, xs could be one of the objects (fork or decision gateways) with multiple *outgoing* flows. Assume that x is an output object of xs. If xs is a fork gateway, an event (flow(xs,x)) is returned indicating the control flow is splitting from xs to x. Next, if xs is a data-based decision gateway, an event (switch(xs,x,c)) is returned indicating the control flow is leaving from xs to x given that condition c holds. Finally, if xs is an event-based decision gateway, an event (pick(xs,x)) is returned indicating the control flow is leaving from xs to x on the occurrence of a trigger that leads to the execution of x.

Now we look closer into the second function PreCondSet. It operates based on the object type of the input parameter x. If x is one of the objects (tasks, events, or decision or fork gateways) that have a single *incoming* flow (InFlow(x)), the function returns a precondition set comprising just one precondition capturing a single event. Otherwise, if x is a merge gateway (resp. a join gateway), there exist multiple *incoming* flows (given by the set InFlows(x)), and the resulting precondition set contains a number of single events (resp. a conjunction of single events) from these incoming flows, to capture the fact that when any (resp. all) of these events occurs (resp. occur) the corresponding merge (resp. join) gateway may be executed.

In the second step, we transform the above precondition sets with their associated objects into a set of event-action rules. An event-action rule can be written in the form of $E\{A\}$: E is a single event or a conjunction of single events (namely a *composite event*) causing the rule to be triggered, and A is a list of actions being executed when the rule is triggered. The list of actions can be executed in sequence ($a_1; a_2$) or in parallel ($a_1||a_2$). If an event-action rule allows the use of single events only, it is called a *simple* event-action rule; otherwise, it is a *composite* event-action rule.

Figure 4 lists the event-action rules translated from the precondition sets related to different types of objects. There are three new notations: Flow(fg), Switch(dg) and Pick(eg).

Let out(fg) = {x_1, ..., x_n}, out(dg) = {y_1, ..., y_n} and c_i = Cond((dg,y_i)) ($i = 1, ..., n$), and out(eg) = {z_1, ..., z_n}, then

- Flow(fg) = {flow(fg, x_1), ..., flow(fg, x_n)},
- Switch(dg) = {switch(dg, y_1, c_1), ..., switch(dg, y_n, c_n)},
- Pick(eg) = {pick(eg, z_1), ..., pick(eg, z_n)}.

In Figure 4, most of the translations are straightforward except the following issue that is worth mentioning. The precondition set for a join gateway comprises just one precondition capturing a composite event, and in the general case, this leads to a composite event-action rule. However, BPEL only supports simple

```
AllPreCondSets(C: Component):
   let {x1, ..., xn} = Objects(C) in
      return {PreCondSet(x1), ..., PreCondSet(xn)}

PreCondSet(x: Object):
    if ObjectType(x) ∈ {"task", "event", "data-based decision",
                        "event-based decision", "fork"}
      return {EventOnFlow(InFlow(x))}
    else if ObjectType(x) = "merge"
      let {f1, ..., fn} = InFlows(x) in
         return {EventOnFlow(f1), ..., EventOnFlow(fn)}
    else if ObjectType(x) = "join"
      let {f1, ..., fn} = InFlows(x) in
         return {EventOnFlow(f1) ∧ ... ∧ EventOnFlow(fn)}

EventOnFlow(f: Flow):
    let xs = Source(f) and x = Target(f)
       if xs ∉ Objects(Component(x))
         return Start(Component(x))
       else if ObjectType(xs) ∈ {"task", "event", "join", "merge"}
              return end(xs)
       else if ObjectType(xs) = "fork"
              return flow(xs, x)
       else if ObjectType(xs) = "data-based decision"
              let c = Cond(f) in
                 return switch(xs, x, c)
       else if ObjectType(x) = "event-based decision"
              return pick(xs, x)
```

Figure. 3. Algorithm for deriving precondition sets from a component of a well-formed core BPD.

event-action rules. To address this issue, we translate the above composite event-action rule to a simple event-action rule, by separating the first single event (e_1) from the rest in the initial composite event. Although the resulting rule can be triggered by event e_1, the action "invoke end(jg)" will not be performed until occurrences of all the remaining events e_2 to e_n have been registered.

Object	Precondition Set	Event-Action Rule
task/event (a)	{ e }	e { do a ; invoke end(a) }
fork gateway (fg)	{ e }	e { execute Flow(fg) }
data-based decision gateway (dg)	{ e }	e { execute Switch(dg) }
event-based decision gateway (eg)	{ e }	e { execute Pick(eg) }
merge gateway (mg)	{ e_1, . . ., e_n }	e_1 { invoke end(mg) } . . . e_n { invoke end(mg) }
join gateway (jg)	{ $e_1 \wedge \ldots \wedge e_n$ }	e_1 { on e_2 || . . . || on e_n ; invoke end(jg) }

Figure. 4. Event-action rules translated from precondition sets for different types of objects.

As the last step, we translate event-action rules to BPEL. A simple event-action rule $e\{A\}$ is realised by a BPEL event handler (`onEvent`) encoded as:

```
<onEvent e>
   <!-- BPEL translation of A -->
</onEvent>
```

The list of actions A can be mapped to BPEL as shown in Figure 5. Based on this, we translate the set of event-action rules derived from a component $\mathcal{C}$ to a BPEL scope. Let $m+1$ be the number of event-action rules derived from $\mathcal{C}$, $\{\mathsf{Start}(\mathcal{C}), \mathsf{e}_1, ..., \mathsf{e}_m\}$ be the set of events for triggering each of these rules, $\mathcal{C}$ can be mapped onto a scope encoded as:

```
<scope name="t_c">
   <onEvent e_1> . . . </onEvent>
   . . .
   <onEvent e_m> . . . </onEvent>
   <invoke Start(C)/>
</scope>
```

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

Action	BPEL Translation
do a	*an appropriate activity named "a"*
on e	`<receive e/>`
invoke end(x)	`<invoke end(x)/>`

execute Flow(fg)	`<flow name="fg">`
	`  <invoke flow(fg, x1)/>`
	`  ...`
	`  <invoke flow(fg, xn)/>`
	`</flow>`

execute Switch(dg)	`<switch name="dg">`
	`  <case condition="c1">`
	`  <invoke switch(dg, y1, c1)/>`
	`  </case>`
	`  ...`
	`  <case condition="cn">`
	`  <invoke switch(dg, yn, cn)/>`
	`  </case>`
	`</switch>`

execute Pick(eg)	`<pick name="eg">`
	`  <onEvent name="z1">`
	`  <invoke pick(eg, z1)/>`
	`  </onEvent>`
	`  ...`
	`  <onEvent name="zn">`
	`  <invoke pick(eg, zn)/>`
	`  </onEvent>`
	`</pick>`

;	`<sequence> ... </sequence>`
||	`<flow> ... </flow>`

Figure. 5. BPEL translation of actions.

Finally, it should be mentioned that the above events for triggering event-action rules are performed by a BPEL invoke activity via a *local* partner link between the final BPEL process (i.e. the mapping of the BPD to which the component $\mathcal{C}$ belongs) and itself. The interested reader may refer to [8] for definitions of a local partner link and an event being invoked or consumed via a local partner link.

3.5 Translation algorithm

Based on the mapping of each of the components aforementioned, we now define the algorithm to translate a well-formed core BPD into BPEL. The basic idea behind this algorithm is to select a component in the BPD, provide its BPEL translation, and fold the component. This is repeated until no component is left in the BPD. Below, the set of components of a BPD (X) is denoted as $[X]_c$.

Definition 7 (Algorithm). *Let $\mathcal{BPD}$ be a well-formed core BPD with one start event and one end event.*

(1) $X := \mathcal{BPD}$
(2) *if $[X]_c = \varnothing$ (i.e., X is initially a trivial BPD), output the BPEL translation of the single task or event object between the start and end events in X, and goto* (5).
(3) *while $[X]_c \neq \varnothing$ (i.e., X is a non-trivial BPD)*
  (3-a) *if there is a maximal SEQUENCE-component $\mathcal{C} \in [X]_c$, selected it and goto* (3-d).
  (3-b) *if there is a well-structured (non-sequence) component $\mathcal{C} \in [X]_c$, select it and goto* (3-d).
  (3-c) *if there is a minimal non-well-structured component $\mathcal{C} \in [X]_c$, select it.*
  (3-d) *Attach BPEL translation of $\mathcal{C}$ to task object t_c.*
  (3-e) $X := \mathsf{Fold}(X, \mathcal{C}, t_c)$ *and return to* (3).
(4) *Output the BPEL code attached to the task object t_c.*
(5) *Start event and end event are translated into a pair of* `<process>` *and* `</process>` *tags to enclose the BPEL code generated in steps* (2) *or* (4). *In addition, for an end terminate event, add* `<exit/>` *activity after the above BPEL code (and before* `</process>`*).*

In the above algorithm, the translation of components is done in step (3-d) followed by the folding in step (3-e). The component to be translated is selected in steps (3-a) to (3-c). In order to keep the translation as compact as possible, the selection always starts from a maximal SEQUENCE-component after each folding. Only if there are no sequences left in the BPD, other well-structured components are considered. Since all well-structured non-sequence components are disjoint, the order of selecting these components is irrelevant. Finally, the minimal non-well-structured components are considered.

4 Case Study

Consider the complaint handling process model shown in Figure 6. 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 within two weeks (event *returned-questionnaire*), task *process questionnaire* is executed. Otherwise, the result of the questionnaire is discarded (event *time-out*). In parallel the complaint is evaluated (task *evaluate*). Based on the evaluation result, the processing is either done or continues to task *check processing*. If the check result is not ok, the complaint requires re-processing. Finally task *archive* is executed. Note that labels *DONE*, *CONT*, *OK* and *NOK* on the outgoing flows of each data-based XOR decision gateway, are abstract representations of conditions on these flows.

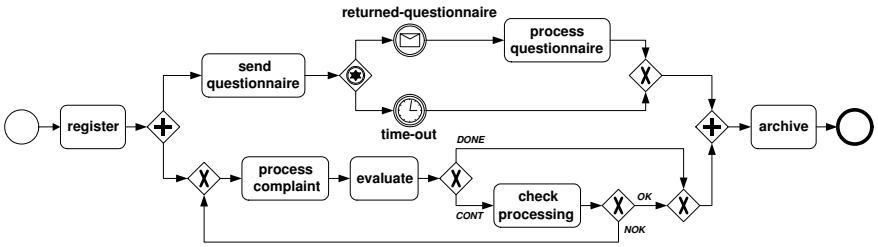

Figure. 6. A complaint handling process model.

Following the algorithm in Section 3, we now translate the above BPD to a BPEL specification. Figure 7 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 in the initial BPD, and 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 specification will be presented in simplified BPEL syntax which defines the control flow for the process but omits all details related to data definitions such as partners, messages and variables.

1st Translation. The algorithm first tries to locate SEQUENCE-components. In the initial BPD shown in Figure 6, the component C_1 consisting of tasks a_6 and a_7 is the only SEQUENCE-component that can be identified. Hence, C_1 is folded into a task t_c^1 attached with the BPEL translation sketched as:

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

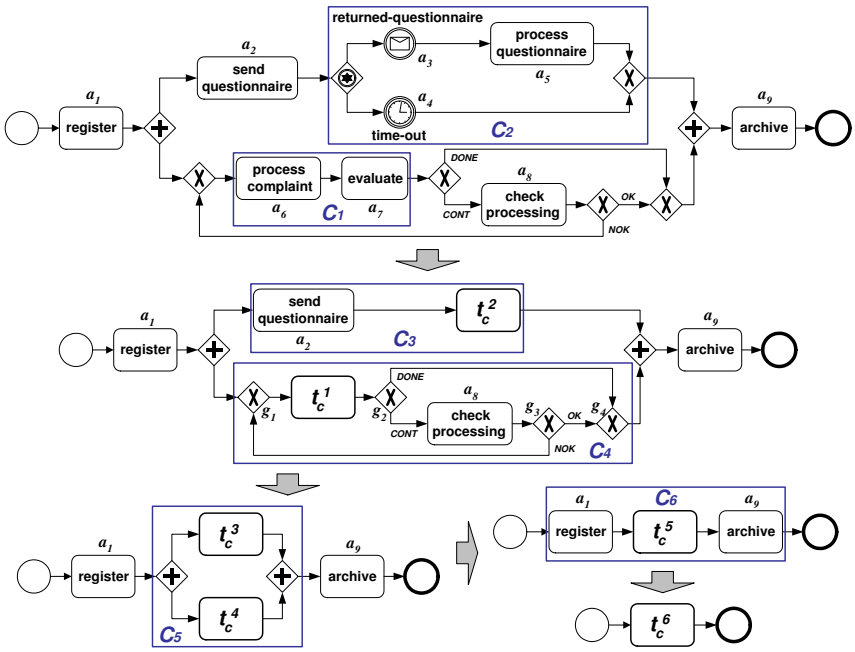

Figure. 7. Translating the complaint handling process model in Figure 6 into BPEL.

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

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

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

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

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

4th Translation. After the above three components C_1 to C_3 have been folded into the corresponding tasks t_c^1 to t_c^3, there is no well-structured components left in the BPD. The algorithm continues to identify any minimal non-well-structured component. As a result, the component C_4 is selected. It is translated into a scope with a number of event handlers through the following steps:

Step 1: Generating Precondition Sets. The component C_4 consists of a set of six objects $\{g_1, t_c^1, g_2, a_8, g_3, g_4\}$. The source object is g_1 and the sink object is g_4. The precondition sets for each of the six objects are:

$$
\begin{aligned}
\mathsf{PreCondSet}(g_1) &= \{\mathsf{Start}(C_4),\ \mathsf{switch}(g_3,\ g_1,\ NOK)\} \\
\mathsf{PreCondSet}(t_c^1) &= \{\mathsf{end}(g_1)\} \\
\mathsf{PreCondSet}(g_2) &= \{\mathsf{end}(t_c^1)\} \\
\mathsf{PreCondSet}(a_8) &= \{\mathsf{switch}(g_2,\ a_8,\ CONT)\} \\
\mathsf{PreCondSet}(g_3) &= \{\mathsf{end}(a_8)\} \\
\mathsf{PreCondSet}(g_4) &= \{\mathsf{switch}(g_2,\ a_4,\ DONE),\ \mathsf{switch}(g_3,\ a_4,\ OK)\}
\end{aligned}
$$

Step 2: Generating Event-Action Rules. All the precondition sets above can be translated into the set of (simple) event-action rules listed below, where for "for x_i" is a shortened form of "for execution of object x_i".

For g_1: $\mathsf{Start}(C_4)\{\text{invoke } \mathsf{end}(g_1)\}$
$\quad\quad$ $\mathsf{switch}(g_3,\ g_1,\ NOK)\{\text{invoke } \mathsf{end}(g_1)\}$
For t_c^1: $\mathsf{end}(g_1)\{\text{do } t_c^1\text{; invoke } \mathsf{end}(t_c^1)\}$
For g_2: $\mathsf{end}(t_c^1)\{\text{execute } \mathsf{Switch}(g_2)\}$
For a_8: $\mathsf{switch}(g_3,\ a_8,\ CONT)\{\text{do } a_8\text{; invoke } \mathsf{end}(a_8)\}$
For g_3: $\mathsf{end}(a_8)\{\text{execute } \mathsf{Switch}(g_3)\}$
For g_4: $\mathsf{switch}(g_2,\ g_4,\ DONE)\{\text{invoke } \mathsf{end}(g_4)\}$
$\quad\quad$ $\mathsf{switch}(g_3,\ g_4,\ OK)\{\text{invoke } \mathsf{end}(g_4)\}$

Step 3: Deriving the BPEL Code. The above event-action rules can be translated into a BPEL scope activity of which the XML code is sketched below. Note that the rule triggered by event $\mathsf{Start}(C_4)$ is mapped to the main activity of the scope, while the rest of the rules are mapped to event handlers.

```
<scope name="t_c^4">
   <onEvent Start(C_4)>
      <invoke end(g_1)/>
   </onEvent>
   <onEvent switch(g_3,g_1,NOK)>
      <invoke end(g_1)/>
   </onEvent>
   <onEvent end(g_1)>
      <sequence>
         <sequence name="t_c^1"> ... </sequence>
         <invoke end(t_c^1)/>
      </sequence>
   </onEvent>
```

```
<onEvent end(t_c^1)>
   <switch name="g_2">
      <case condition="DONE">
         <invoke switch(g_2,g_4,DONE)/>
      </case>
      <case condition="CONT">
         <invoke switch(g_2,a_8,CONT)/>
      </case>
   </switch>
</onEvent>
<onEvent switch(g_2,a_8,CONT)>
   <sequence>
      <invoke name="check processing".../>
      <invoke end(a_8)/>
   </sequence>
</onEvent>
<onEvent end(a_8)>
   <switch name="g_3">
      <case condition="OK">
         <invoke switch(g_3,g_4,OK)/>
      </case>
      <case condition="NOK">
         <invoke switch(g_3,g_1,NOK)/>
      </case>
   </switch>
</onEvent>
<onEvent switch(g_2,g_4,DONE)>
   <invoke end(g_4)/>
</onEvent>
<onEvent switch(g_3,g_4,OK)>
   <invoke end(g_4)/>
</onEvent>
<invoke Start(C_4)/>
</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 specification 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>
```

For clarity, a complete listing of the XML code for the BPEL specification of the complaint handling process shown in Figure 6 is given in the appendix.

5 Conclusions

In this paper, we presented an algorithm to translate models captured in a core subset of BPMN into BPEL. The translation algorithm is capable of generating readable BPEL code by discovering "patterns" in the BPMN models that can be mapped onto BPEL structured constructs. The algorithm also exploits BPEL event handlers for unstructured subsets of the BPMN models. As a result, the algorithm can handle any BPMN model composed of tasks, events, parallel gateways, and XOR gateways (both data-based and event-based) connected in arbitrary topologies.

Implementation of the algorithm is ongoing. A first version of the implementation supports the translation of a smaller subset of BPMN models (called *Standard Process Models*) into BPEL event handlers and is available at `http://www.bpm.fit.qut.edu.au/projects/babel/tools`. We are now extending this tool with the ability to detect patterns in the BPMN models and to map these onto BPEL structured constructs.

Other ongoing work aims at exploring the use of BPEL's non-structured constructs called *control links* in the translation. Control links support the definition of precedence, synchronization and conditional dependencies on top of those captured by the structured activity constructs. They allow the definition of directed graphs but with syntactical limitations. Our previous work [3] shows that a larger class of "patterns" could be mapped onto BPEL's flow construct by making greater use of control links. However, since control links enable dead path elimination, such an extension, if not performed carefully, may hide errors such as deadlocks and livelocks in the source model during the translation, thus requiring verification technology for detection.

References

1. 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.
2. 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 R. Meersman and Z. Tari et al., editors, *On the Move to*

Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005, volume 3760 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 2005.
3. W.M.P. van der Aalst and K.B. Lassen. Translating Workflow Nets to BPEL4WS. BETA Working Paper Series, WP 145, Eindhoven University of Technology, Eindhoven, 2005.
4. A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu, editors. *Web Services Business Process Execution Language Version 2.0*. Working Draft. WS-BPEL TC OASIS, May 2005.
5. 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.
6. 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.
7. 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.
8. C. Ouyang, M. Dumas, S. Breutel, and A.H.M. ter Hofstede. Translating standard process models to BPEL. Technical Report BPM-05-27, BPMcenter.org, November 2005. Available via `http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-27.pdf`.
9. S. White. Using BPMN to Model a BPEL Process. *BPTrends*, 3(3):1–18, 2005.
10. S. A. White. *Business Process Modeling Notation (BPMN) Version 1.0*. Business Process Management Initiative, BPMI.org, May 2004.

Appendix

This appendix provides a complete listing of the XML code for the BPEL specification of the complaint handling process model shown in Figure 6.

```
<process name="complaint handling">
   <sequence name="t_c^6">
      <invoke name="register">
      <flow name="t_c^5">
         <sequence name="t_c^3">
            <invoke name="send questionnaire".../>
            <pick name="t_c^2">
               <onMessage operation="returned-questionnaire"...>
                  <invoke name="process questionnaire".../>
               </onMessage>
               <onAlarm for='P14DT'>
                  <empty/>
               </onAlarm>
```

```
   </pick>
</sequence>
<scope name="t_c^4">
   <onEvent Start(C_4)>
      <invoke end(g_1)/>
   </onEvent>
   <onEvent switch(g_3,g_1,NOK)>
      <invoke end(g_1)/>
   </onEvent>
   <onEvent end(g_1)>
      <sequence>
         <sequence name="t_c^1">
            <invoke name="process complaint".../>
            <invoke name="evaluation".../>
         </sequence>
         <invoke end(t_c^1)/>
      </sequence>
   </onEvent>
   <onEvent end(t_c^1)>
      <switch name="g_2">
         <case condition="DONE">
            <invoke switch(g_2,g_4,DONE)/>
         </case>
         <case condition="CONT">
            <invoke switch(g_2,a_8,CONT)/>
         </case>
      </switch>
   </onEvent>
   <onEvent switch(g_2,a_8,CONT)>
      <sequence>
         <invoke name="check processing".../>
         <invoke end(a_8)/>
      </sequence>
   </onEvent>
   <onEvent end(a_8)>
      <switch name="g_3">
         <case condition="OK">
            <invoke switch(g_3,g_4,OK)/>
         </case>
         <case condition="NOK">
            <invoke switch(g_3,g_1,NOK)/>
         </case>
      </switch>
   </onEvent>
   <onEvent switch(g_2,g_4,DONE)>
      <invoke end(g_4)/>
   </onEvent>
```

```
            <onEvent switch(g3,g4,OK)>
               <invoke end(g4)/>
            </onEvent>
            <invoke Start(C4)/>
         </scope>
      </flow>
      <invoke name="archive">
   </sequence>
</process>
```