Pattern-based Translation of BPMN Process Models to BPEL Web Services*

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

[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 Mathematics and Computer Science, Eindhoven University of Technology,
GPO Box 513, NL-5600 MB, The Netherlands
w.m.p.v.d.aalst@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 primarily targeted at domain analysts and is supported by many modelling tools, but in its current form, it lacks the semantic precision required to capture fully executable business processes. The Business Process Execution Language for Web Services (BPEL) on the other hand is a mainly block-structured language, targeted at software developers and supported by several execution platforms. In the current setting, translating BPMN models into BPEL code is a necessary step towards standards-based business process development environments. This translation is challenging since BPMN and BPEL represent two fundamentally different classes of languages. Existing BPMN-to-BPEL translations rely on the identification of block-structured patterns in BPMN models that are mapped into block-structured BPEL constructs. This paper advances the state of the art in BPMN-to-BPEL translation by defining methods for identifying not only perfectly block-structured fragments in BPMN models, but also quasi-structured fragments that can be turned into perfectly structured ones and flow-based acyclic fragments that can be mapped into a combination of block-structured constructs and control links. Beyond its direct relevance in the context of BPMN and BPEL, this paper addresses issues that arise generally when translating between graph-oriented and block-structured flow definition languages.

1 Introduction

The *Business Process Execution Language for Web Services* (BPEL) [2] 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.[3] 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) [5] 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 more than 30 tools.[4] 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 [5], shows that these translations fail to fulfil the following key requirements: (i) completeness, i.e., applicable to BPMN model with arbitrary topology; (ii) automation, i.e., capable

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

[3] See http://en.wikipedia.org/wiki/BPEL

[4] See http://www.bpmn.org

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 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].

In prior work [15], we proposed a translation that achieves the completeness and automation requirements outlined above for a core subset of BPMN models. However, the code produced by this translation lacks readability. Essentially, the BPMN process model is translated into a set of event-condition-action rules, and these rules are then encoded using BPEL event handlers. Thus, the translation does not exploit the block-structured constructs of BPEL, which would clearly lead to more readable code.

This paper presents a complementary technique to translate BPMN to BPEL that emphasises the readability requirement. The proposal is based on the identification of structural patterns of BPMN models which can be translated into block-structured BPEL code. The patterns are divided into: (i) well-structured patterns, which can be directly mapped onto block-structured BPEL constructs; (ii) quasi-structured patterns, which can be re-written into perfectly structured ones and then mapped onto block-structured BPEL constructs; and (iii) flow-based acyclic fragments which can be mapped onto combinations of block-structured BPEL constructs and additional control links to capture dependencies between activities located in different blocks.

Beyond their direct relevance in the context of BPMN-to-BPEL mapping, the translation patterns and algorithm presented in this paper address issues that arise generally when translating from graph-oriented process languages (e.g., UML Activity Diagrams, EPCs, YAWL, or Petri nets) to block-structured ones.

The rest of the paper is structured as follows. Section 2 and 3 provide an overview of BPMN and BPEL respectively. Next, Section 4 presents the identification of patterns in a given BPMN process model and their mapping onto BPEL. The overall translation approach is then illustrated through a case study in Section 5. Finally, Section 6 discusses related work while Section 7 concludes and outlines future work.

2 Business Process Execution Language for Web Services (BPEL)

BPEL [2] can be seen as an extension of imperative programming languages with constructs specific to web service implementation. These extensions are mainly inspired from business process modelling languages. Accordingly, the top-level concept of BPEL is that of *process definition*. A BPEL process definition relates a number of *activities* that need to be performed by a Web service. 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; 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*, *repeat* and *sequential foreach*, for structured iteration; 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* that may fire at any time during the execution of a scope, while *fault* and *compensation handlers* are designed for catching and handling exceptions. Finally, the *parallel foreach* construct enables multiple instances of a given scope to be executed multiple times concurrently.

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

There are over 20 execution engines supporting BPEL. 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 onto BPEL.

BPEL process definitions can be either fully executable or they can be left underspecified. Executable BPEL process definitions are intended to be deployed into an execution engine, while underspecified BPEL definitions, also called *abstract processes*, capture a non-executable set of interactions between a given service and several other "partner services". Our BPMN-to-BPEL mapping focuses on the control flow perspective, and does not deal with data manipulation and other implementation details. Accordingly, the BPEL definitions generated by the proposed mapping correspond to abstract processes, which can be used as templates and enriched with additional details to obtain executable processes. This approach is in line with the difference in abstraction between BPMN and BPEL: BPMN is intended as a modelling language for analysis and design, while BPEL is an implementation language. Thus, one way or another BPMN models need to undergo some form of refinement to yield executable BPEL processes. This refinement is outside the scope of this paper.

3 Business Process Modelling Notation (BPMN)

BPMN [5] 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. Below, we first introduce BPDs and then define an abstract syntax for it.

3.1 Business Process Diagrams (BPD)

BPMN uses BPDs to describe business processes. A BPD is made up of BPMN elements. We consider a core subset of BPMN elements that can be used to build BPDs covering the fundamental control flows in BPMN. These elements are shown in Figure 1. There are *objects* and *sequence flows*. A sequence flow links two objects in a BPD and shows the control flow relation (i.e., execution order). An object can be an *event*, a *task* or a *gateway*. An event may signal the start of a process (*start event*), the end of a process (*end event*), a message that arrives, or a specific time-date being reached during a process (*intermediate message/timer event*). A task is an atomic activity and stands for work to be performed within a process. There are seven task types: *service*, *receive*, *send*, *user*, *script*, *manual*, and *reference*. For example, a receive task is used when the process waits for a message to arrive from an external partner. Also, a task may be none of the above types, which we refer to as a *blank* task. A gateway is a routing construct used to control the divergence and convergence of sequence flow. There are: *parallel fork gateways* (i.e., AND-splits) for creating concurrent sequence flows, *parallel join gateways* (i.e., AND-joins) for synchronising 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, i.e., XOR-splits) or external events (event-based, i.e., deferred choice), and *XOR merge gateways* (i.e., XOR-joins) for joining a set of mutually exclusive alternative sequence flows into one sequence flow. It is important to note that 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).

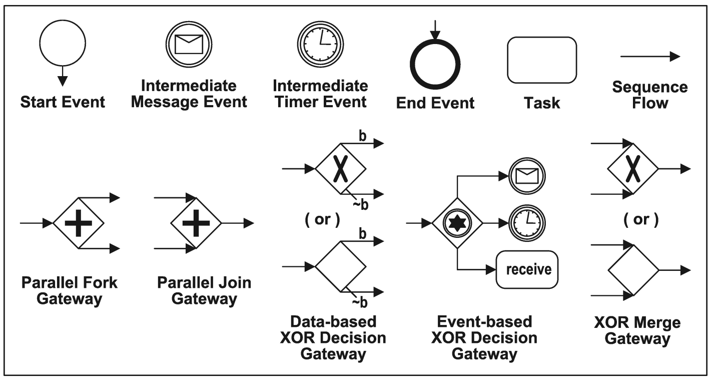


Figure 1. A core subset of BPMN elements.

BPMN defines several other control-flow constructs. 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 "while-do" and "repeat-until" structured loops, can be directly mapped onto the corresponding BPEL structured activities. Similarly, a multi-instance task can be directly mapped onto 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 [1]. Hence, it does not require a separate mapping rule. On the other hand, the mapping of OR-joins requires a special treatment and will be briefly discussed in Section 7.

3.2 Abstract syntax of a BPD

A BPD, which is made up of the core subset of BPMN elements shown in Figure 1, is hereafter referred to as a *core BPD*. Below, we define the syntax of a core BPD.

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

- $\mathcal{O}$ *is a set of objects which 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* $\mathcal{G}^F$*, 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, i.e., a set of sequence flows connecting objects,*
- $\mathsf{Cond}\colon \mathcal{F} \cap (\mathcal{G}^D \times \mathcal{O}) \rightarrow \mathcal{B}$ *is a function mapping sequence flows emanating from data-based XOR decision gateways to the set of all possible conditions* $(\mathcal{B})$.[5]

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, it is necessary to restrict the definition to *well-formed core BPDs*. Note that these restrictions are without loss of generality and are to facilitate the definition of the mapping of BPMN to BPEL.

Definition 2 (Well-formed core BPD). *A core BPD is well formed iff 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,*
- $\forall\ e \in \mathcal{E}^E,\ \mathsf{out}(e) = \varnothing \wedge |\mathsf{in}(e)| = 1$*, 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,*

[5] 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 process execution.

– $\forall\, g \in \mathcal{G}^D$, $\exists$ *an order* $<_g$ *which is a strict total order over the set of flows* $\{g\} \times \mathsf{out}(g)$, *and for* $x \in \mathsf{out}(g)$ *such that* $\neg\, \exists_{f \in \{g\} \times \mathsf{out}(g)}((g, x) <_g f)$, (g, x) *is the default flow among all the outgoing flows from* g,[6]
– $\forall\, x \in \mathcal{O}$, $\exists\, s \in \mathcal{E}^S$, $\exists\, e \in \mathcal{E}^E$, $s\mathcal{F}^*x \wedge x\mathcal{F}^*e$,[7] *i.e., every object is on a path from a start event to an end event.*

For the translation of BPMN to BPEL in this paper, 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, 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, while a BPD with multiple end events can be transformed into a BPD with a unique end event by using an inclusive OR merge gateway (i.e., OR-join, see discussion in Section 7). Therefore, without loss of generality, we assume that both $\mathcal{E}^S$ and $\mathcal{E}^E$ are singletons, i.e., $\mathcal{E}^S = \{s\}$ and $\mathcal{E}^E = \{e\}$.

4 Identification and Translation of BPMN patterns

We would like to achieve two goals when mapping BPMN onto BPEL. One is to define an algorithm which allows us to translate any well-formed core BPD into a valid BPEL process; the other is to generate readable and compact BPEL code. In prior work [15], we proposed a complete translation from well-formed core BPDs to BPEL. However, the code produced by this translation lacks readability. In the following, we exploit the block-structured constructs of BPEL to address the readability requirement. The translation is based on the identification of three categories of patterns of BPMN fragments which can be mapped onto block-structured BPEL code: (i) *well-structured patterns*, (ii) *quasi-structured patterns*, and (iii) *generalised* FLOW*-patterns*. The well-structured pattern-based translation (Section 4.2) is intended to cover the class of BPMN corresponding to structured workflow models as defined in [7]. Quasi-structured patterns (Section 4.3) introduce some level of unstructuredness, but still, they can be expanded into well-structured components. Finally, the generalised FLOW-pattern-based translation (Section 4.4) caters for the class of acyclic BPMN models consisting of tasks, events, sequence flows, and parallel gateways only. Models in this latter class do not contain decision points (whether event-driven or data-driven) but they can be unstructured. The translations of these three categories of patterns can be combined with our prior work [15]. The resulting combined translation approach, as discussed in [14], can deal with BPMN models containing all parallel gateways and XOR (event-based and data-based) gateways and with arbitrary topology.

4.1 Decomposing a BPD into components

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” which refer to subsets of the BPD. We then try to identify “patterns” which resemble different groups of components that can be mapped onto suitable “BPEL blocks” in a systematic way. 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.

[6] The total order defined over the set of outgoing flows of an XOR decision gateway is to capture the fact that these flows are evaluated in order and the “default flow” is the one evaluated at last. This is part of the semantics of XOR decision gateways as defined in the BPMN specification.
[7] $\mathcal{F}^*$ is the reflexive transitive closure of $\mathcal{F}$, i.e., $x\mathcal{F}^*y$ iff there is a path from x to y in $\mathcal{BPD}$.

A component is a subset of a BPD that has one entry point and one exit point. Before identifying patterns in the remaining subsections, it is necessary to formalize the notion of components in a BPD. Note that we adopt similar notations for representing an entire BPD and a component thereof. Also, to formulate 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. A subset of $\mathcal{BPD}$, as given by $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$, is a component iff:*

- *$\mathcal{O}_c \subseteq \mathcal{O} \backslash (\mathcal{E}^S \cup \mathcal{E}^E)$, i.e., a component does not contain any start or end event,*
- *$|(\bigcup_{x \in \mathcal{O}_c} \mathsf{in}(x)) \backslash \mathcal{O}_c| = 1$, i.e., there is a single entry point into the component,[8] which can be denoted as $\mathsf{entry}(\mathcal{C}) = \mathsf{elt}((\bigcup_{x \in \mathcal{O}_c} \mathsf{in}(x)) \backslash \mathcal{O}_c)$,*
- *$|(\bigcup_{x \in \mathcal{O}_c} \mathsf{out}(x)) \backslash \mathcal{O}_c| = 1$, i.e., there is a single exit point out of the component, which can be denoted as $\mathsf{exit}(\mathcal{C}) = \mathsf{elt}((\bigcup_{x \in \mathcal{O}_c} \mathsf{out}(x)) \backslash \mathcal{O}_c)$,*
- *there exists a unique source object $i_c \in \mathcal{O}_c$ and a unique sink object $o_c \in \mathcal{O}_c$ and $i_c \neq o_c$, such that $\mathsf{entry}(\mathcal{C}) \in \mathsf{in}(i_c)$ and $\mathsf{exit}(\mathcal{C}) \in \mathsf{out}(o_c)$,*
- *$\mathcal{F}_c = \mathcal{F} \cap (\mathcal{O}_c \times \mathcal{O}_c)$,*
- *$\mathsf{Cond}_c = \mathsf{Cond}[\mathcal{F}_c]$, i.e., the Cond function where the domain is restricted to $\mathcal{F}_c$.*

Note that all event objects in a component are intermediate events. Also, a component contains at least two objects: the source object and the sink object. A BPD without any component, which is referred to as a *trivial BPD*, has only a single task or intermediate event between the start event and the end event. Hence, translating a trivial BPD into BPEL is straightforward.

The decomposition of a BPD helps to define an iterative approach which allows us to incrementally transform a "componentised" BPD to a block-structured BPEL process. Below, we define the 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 componentised BPD until no component is left in the BPD. It will play a crucial role in the mapping 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 $\mathcal{BPD}$. The function Fold replaces $\mathcal{C}$ in $\mathcal{BPD}$ by a blank task object $t_c \notin \mathcal{O}$, i.e., $\mathsf{Fold}(\mathcal{BPD}, \mathcal{C}, t_c) = (\mathcal{O}', \mathcal{F}', \mathsf{Cond}')$ with:*

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

4.2 Well-structured pattern-based translation

Since one of our goals for mapping BPMN onto BPEL is to generate readable BPEL code, 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 any of these five structured constructs are identified as *well-structured patterns*. Below, we classify different types of well-structured patterns resembling these five structured constructs.

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

Definition 5 (Well-structured patterns). *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 identified as well-structured patterns:*

(*a*) *$\mathcal{C}$ exhibits a* SEQUENCE*-pattern iff $\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I$ (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*-pattern iff $\mathcal{C}$ is a* SEQUENCE*-pattern and there is no other* SEQUENCE*-pattern $\mathcal{C}'$ 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 identified as a* FLOW*-pattern iff*
- $i_c \in \mathcal{G}^F \wedge o_c \in \mathcal{G}^J$,
- $\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I \cup \{i_c, o_c\}$,
- $\forall\ x \in \mathcal{O}_c \backslash \{i_c, o_c\}$, $\mathsf{in}(x) = \{i_c\} \wedge \mathsf{out}(x) = \{o_c\}$.

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

(*d*) *$\mathcal{C}$ is identified as a* PICK*-pattern iff*
- $i_c \in \mathcal{G}^V \wedge o_c \in \mathcal{G}^M$,
- $\mathcal{O}_c \subseteq \mathcal{T} \cup \mathcal{E}^I \cup \{i_c, o_c\}$,
- $\forall\ x \in \mathcal{O}_c \backslash (\{i_c, o_c\} \cup \mathsf{out}(i_c))$, $\mathsf{in}(x) \subset \mathsf{out}(i_c) \wedge \mathsf{out}(x) = \{o_c\}$.[9]

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

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

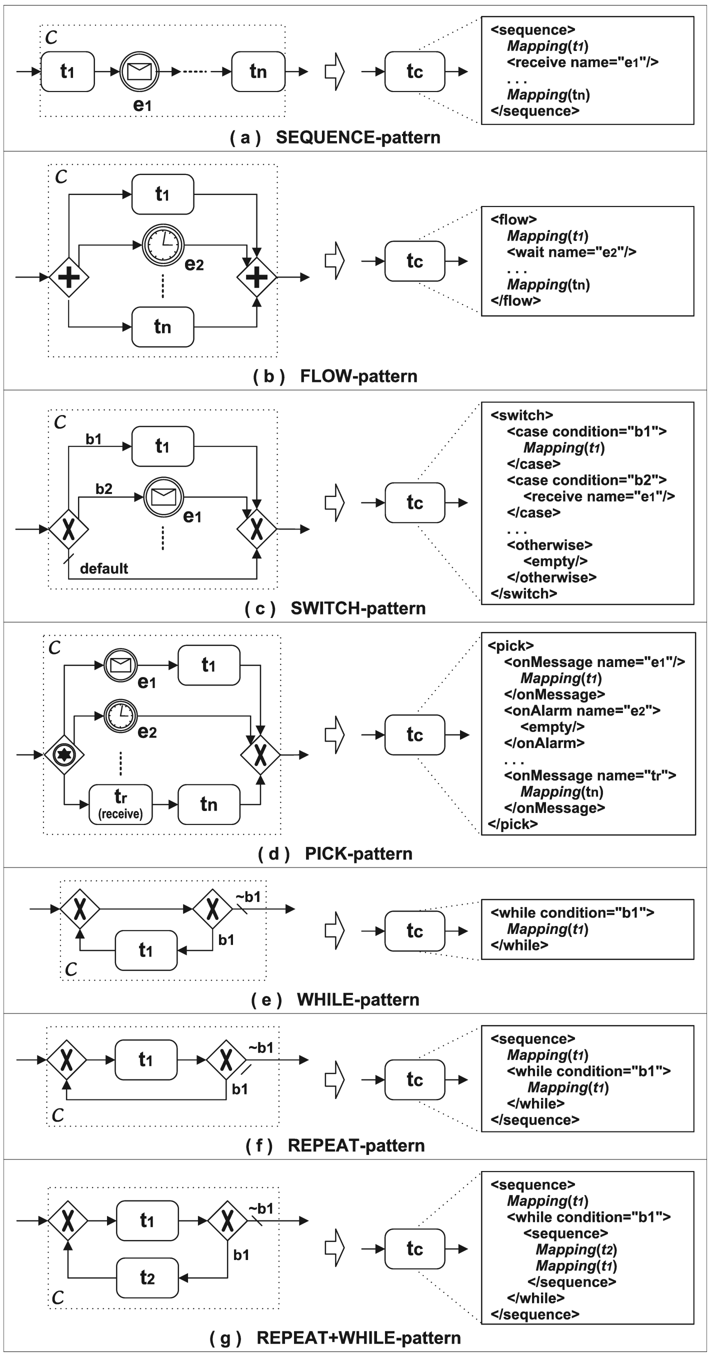
(*g*) *$\mathcal{C}$ is identified as a* REPEAT+WHILE*-pattern iff*
- $i_c \in \mathcal{G}^M \wedge o_c \in \mathcal{G}^D \wedge x_1, x_2 \in \mathcal{T} \cup \mathcal{E}^I \wedge x_1 \neq x_2$,
- $\mathcal{O}_c = \{i_c, o_c, x_1, x_2\}$,
- $\mathcal{F}_c = \{(i_c, x_1), (x_1, o_c), (o_c, x_2), (x_2,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

Figure 2 illustrates how to map each of the above patterns onto the corresponding BPEL structured activities. Using the function Fold of Definition 4, a component $\mathcal{C}$ is replaced by a single task t_c attached with the BPEL translation of $\mathcal{C}$. Note that the BPEL code for the mapping of each task t_i $(i = 1, ..., n)$ is denoted as $\mathsf{Mapping}(t_i)$. Based on the nature of these task objects they are mapped onto the corresponding types of BPEL activities. For example, a *service* task is mapped to an invoke activity, a *receive* task (like t_r in Figure 2(d)) is mapped to a receive activity, and a *user* task may be mapped to an invoke activity followed by a receive activity[10]. Also, a task t_i may result from the folding of a previous component $\mathcal{C}'$, in which case, $\mathsf{Mapping}(t_i)$ is the code for the mapping of component $\mathcal{C}'$.

In Figure 2(a) to (e), the mappings of the five patterns, SEQUENCE, FLOW, SWITCH, PICK and WHILE, are straightforward. In a PICK-pattern (Figure 2(d)), an event-based XOR decision gateway must be followed by receive tasks or intermediate message or timer events.

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

[10] Since the goal of this paper is to define an approach for translating BPDs to BPEL processes, we do not discuss further how to map simple tasks in BPMN to BPEL. Interested readers may refer to [5] for some guidelines on mapping BPMN tasks into BPEL activities.



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

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

```
<switch>
   <case condition="b1">
      Mapping(t1)
   </case>
   <case condition="b2">
      <receive name="e1"/>
   </case>
   . . .
   <otherwise>
      <empty/>
   </otherwise>
</switch>
```

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

```
<while condition="b1">
   Mapping(t1)
</while>
```

```
<sequence>
   Mapping(t1)
   <while condition="b1">
      Mapping(t1)
   </while>
</sequence>
```

```
<sequence>
   Mapping(t1)
   <while condition="b1">
      <sequence>
         Mapping(t2)
         Mapping(t1)
      </sequence>
   </while>
</sequence>
```

Figure 2. Mapping a well-structured pattern $\mathcal{C}$ onto a BPEL structured activity and folding $\mathcal{C}$ into a single task object t_c attached with the resulting BPEL code.

For this reason, a SEQUENCE-pattern (Figure 2(a)) cannot be preceded by an event-based XOR decision gateway.

In Figure 2(f) and (g), two patterns, REPEAT and REPEAT+WHILE, represent *repeat* loops. *Repeat* loops are the opposite of *while* loops. A *while* loop (see WHILE-pattern in 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. In Figure 2(f), 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 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.

4.3 Quasi-structured pattern-based translation

It can be observed from the previous subsection that well-structured patterns impose strict structural restrictions on BPDs. Again, to achieve the goal of producing readable BPEL code for the mapping of BPMN, we would like to exploit patterns with potential well-structuredness even if they do not strictly satisfy the restrictions captured in the previous patterns. To this end, we start to identify some of those components for which it is easy to temporarily extend them (without changing semantics) to allow for further reductions, e.g., split a gateway into two gateways to separate the incompatible parts. We call this category of components *quasi-structured patterns*. In the following, we classify three types of quasi-structured patterns for FLOW, SWITCH, and PICK, respectively. Also, we specify how to refine them to construct a well-structured pattern within them. To illustrate this definition, Figure 3 depicts examples of three types of quasi-structured patterns and the corresponding refinements. Note that for a given object x that has an outdegree of one (e.g., a join gateway), $\mathsf{succ}(x)$ provides the object that follows x. Similarly, for any object y that has an indegree of one (e.g., a fork gateway), $\mathsf{pred}(x)$ provides the object that precedes y (this will be used in Section 4.4).

Definition 6 (Quasi-structured patterns). *Let $\mathcal{BPD} = (\mathcal{O}, \mathcal{F}, \mathit{Cond})$ be a well-formed core BPD, and $\mathcal{TE}=\mathcal{T}\cup\mathcal{E}$. We may identify three types of quasi-structured patterns as follows:*

(*a*) *Let $x \in \mathcal{G}^F$, $y \in \mathcal{G}^J$, $X = \mathit{out}(x)$, $Y = \mathit{in}(y)$, and $Z = X \cap Y$. If $X \neq Y$ and $|Z \cap \mathcal{TE}| > 1$, we can identify a subset of $\mathcal{BPD}$ as $\mathcal{Q} = (\mathcal{O}_q, \mathcal{F}_q, \varnothing)$ where $\mathcal{O}_q = \{x, y\} \cup X \cup Y$ and $\mathcal{F}_q = \mathcal{F} \cap (\mathcal{O}_q \times \mathcal{O}_q)$. $\mathcal{Q}$ is called a quasi-*FLOW*-pattern, and can be converted to $\mathcal{Q}' = (\mathcal{O}'_q, \mathcal{F}'_q, \varnothing)$ where*
- $\mathcal{O}'_q = \mathcal{O}_q \cup \{x', y'\}$,
- $\mathcal{F}'_q = (\mathcal{F}_q \backslash ((\{x\} \times Z) \cup (Z \times \{y\}))) \cup \{(x, x'), (y', y)\} \cup (\{x'\} \times Z) \cup (Z \times \{y'\})$, *and*
- $\mathcal{Q}'$ *contains a* FLOW-*pattern* $\mathcal{C} = (Z \cup \{x', y'\}, (\{x'\} \times Z) \cup (Z \times \{y'\}), \varnothing)$.

(*b*) *Let $x \in \mathcal{G}^D$, $y \in \mathcal{G}^M$, $X = \mathit{out}(x)\backslash\{y\}$, $Y = \mathit{in}(y)$, and $Z = X \cap Y$. If $X \subset Y$ and $|Z \cap \mathcal{TE}| + |Y \cap \{x\}| > 1$, we can identify a subset of $\mathcal{BPD}$ as $\mathcal{Q} = (\mathcal{O}_q, \mathcal{F}_q, \mathit{Cond}_q)$ where $\mathcal{O}_q = \{x, y\} \cup X \cup Y$, $\mathcal{F}_q = \mathcal{F} \cap (\mathcal{O}_q \times \mathcal{O}_q)$, and $\mathit{Cond}_q = \mathit{Cond}[\mathcal{F}_q]$. Q is called a quasi-*SWITCH*-pattern, and can be converted to $\mathcal{Q}' = (\mathcal{O}'_q, \mathcal{F}'_q, \mathit{Cond}_q)$ where*
- $\mathcal{O}'_q = \mathcal{O}_q \cup \{y'\}$,
- $\mathcal{F}'_q = (\mathcal{F}_q \backslash (Z \times \{y\})) \cup \{(y', y)\} \cup (Z \times \{y'\})$, *and*
- $\mathcal{Q}'$ *contains a* SWITCH-*pattern* $\mathcal{C} = (Z \cup \{x, y'\}, (\{x\} \times Z) \cup (Z \times \{y'\}), \mathit{Cond}_q)$.

(*c*) *Let $x \in \mathcal{G}^V$, $y \in \mathcal{G}^M$, $X = \bigcup_{z \in \mathit{out}(x)}\{\mathit{succ}(z)\}\backslash\{y\}$, $Y = \mathit{in}(y)$, and $Z = X \cap Y$. If $X \subset Y$ and $|Z \cap \mathcal{TE}| + |Y \cap \bigcup_{z \in \mathit{out}(x)}\{z\}| > 1$, we can identify a subset of $\mathcal{BPD}$ as $\mathcal{Q} = (\mathcal{O}_q, \mathcal{F}_q, \varnothing)$ where $\mathcal{O}_q = \{x, y\} \cup X \cup Y$ and $\mathcal{F}_q = \mathcal{F} \cap (\mathcal{O}_q \times \mathcal{O}_q)$. Q is called a quasi-*PICK*-pattern, and can be converted to $\mathcal{Q}' = (\mathcal{O}'_q, \mathcal{F}'_q, \varnothing)$ where*
- $\mathcal{O}'_q = \mathcal{O}_q \cup \{y'\}$,
- $\mathcal{F}'_q = (\mathcal{F}_q \backslash (Z \times \{y\})) \cup \{(y', y)\} \cup (Z \times \{y'\})$, *and*
- $\mathcal{Q}'$ *contains a* PICK-*pattern* $\mathcal{C} = (Z \cup \{x, y'\}, (\{x\} \times Z) \cup (Z \times \{y'\}), \varnothing)$.

From the above definition, it should be mentioned that for quasi-SWITCH and quasi-PICK-patterns, we decided not to consider the situation when there are additional outgoing flows from decision gateways. For quasi-SWITCH-patterns, due to the fact that the conditional flows emanating from an XOR-decision gateway are evaluated in order, it would become quite complicate and error-prone to refine the conditions on the outgoing flows when splitting XOR-decision gateways. For quasi-PICK-patterns, since we cannot decompose race conditions, it is not possible to split an event-based decision gateway.

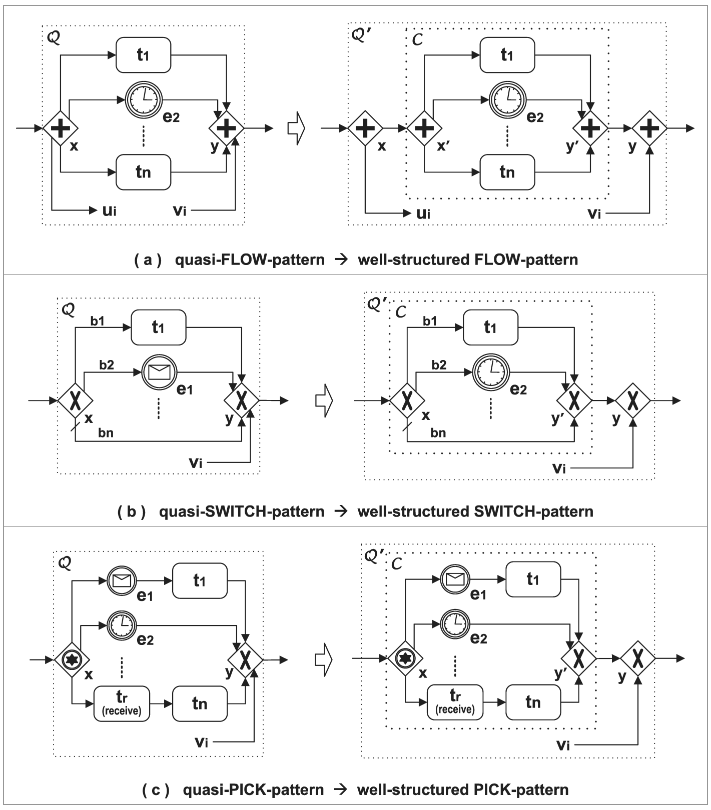

Q
t1
e2
tn
x
y
ui
vi
Q'
C
x'
y'
(a) quasi-FLOW-pattern → well-structured FLOW-pattern
b1
b2
e1
bn
(b) quasi-SWITCH-pattern → well-structured SWITCH-pattern
tr
(receive)
(c) quasi-PICK-pattern → well-structured PICK-pattern

Figure 3. Refining a quasi-structured pattern $\mathcal{Q}$ to construct a well-structured pattern $\mathcal{C}$ in $\mathcal{Q}'$.

4.4 Generalised FLOW-pattern-based translation

As mentioned before, we are interested in exploring more patterns which preserve structuredness to a certain extent such that they can always be mapped onto block-structured BPEL code. In this subsection, we look into another group of components which are acyclic and contain parallelism only. We name this group of components *generalised* FLOW*-patterns*.

Definition 7 (Generalised FLOW-pattern). *Let $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$ be a component of a well-formed core BPD. $\mathcal{C}$ is a generalised* FLOW*-pattern iff:*

- *$\mathcal{C}$ contains no cycles,*
- *all the gateways in $\mathcal{C}$ are either parallel fork or join gateways (i.e., $\mathcal{G}_c = \mathcal{G}_c^F \cup \mathcal{G}_c^J$), and*
- *there is no other component $\mathcal{C}' = (\mathcal{O}'_c, \mathcal{F}'_c, \mathsf{Cond}')$ such that $\mathcal{O}'_c \subset \mathcal{O}_c$.*

The reason for restricting this pattern to acyclic fragments is that we intend to map occurrences of this pattern using BPEL control links. BPEL control links can be used to define directed graphs of activities provided that these graphs are acyclic. In addition, the graphs in question must be *sound* and *safe*, which means that it should not have deadlocks (*sound*) and that there should be no cases where multiple instances of the same activity are executed concurrently (*safe*). Generalised FLOW-patterns as defined above satisfy these two properties. Indeed, these patterns contain only AND-gateways, and AND-gateways are such that every time the nodes that precede the gateway are executed once (each), then the nodes that follow it are also executed once (each).

Translation Algorithm We could easily map a generalised FLOW-pattern into BPEL using control links. However, control links lead to BPEL code that is arguably not as readable as equivalent code using BPEL structured activities. Many BPEL tools do not even offer a way of visualising control links. Therefore, given a generalised FLOW-pattern, we aim at mapping it onto a combination of BPEL structured activities (only *flow* and *sequence* activities will be used) with as few control links as possible. In other words, during the mapping of such a pattern, we need to preserve as much as possible the well-structuredness within the pattern, by incrementally deriving and removing the links from the pattern. Based on this, Figure 4 defines an algorithm for mapping generalised FLOW-patterns to BPEL. To illustrate the algorithm, Figure 5 provides an example of applying the algorithm to the mapping of a generalised FLOW-pattern.

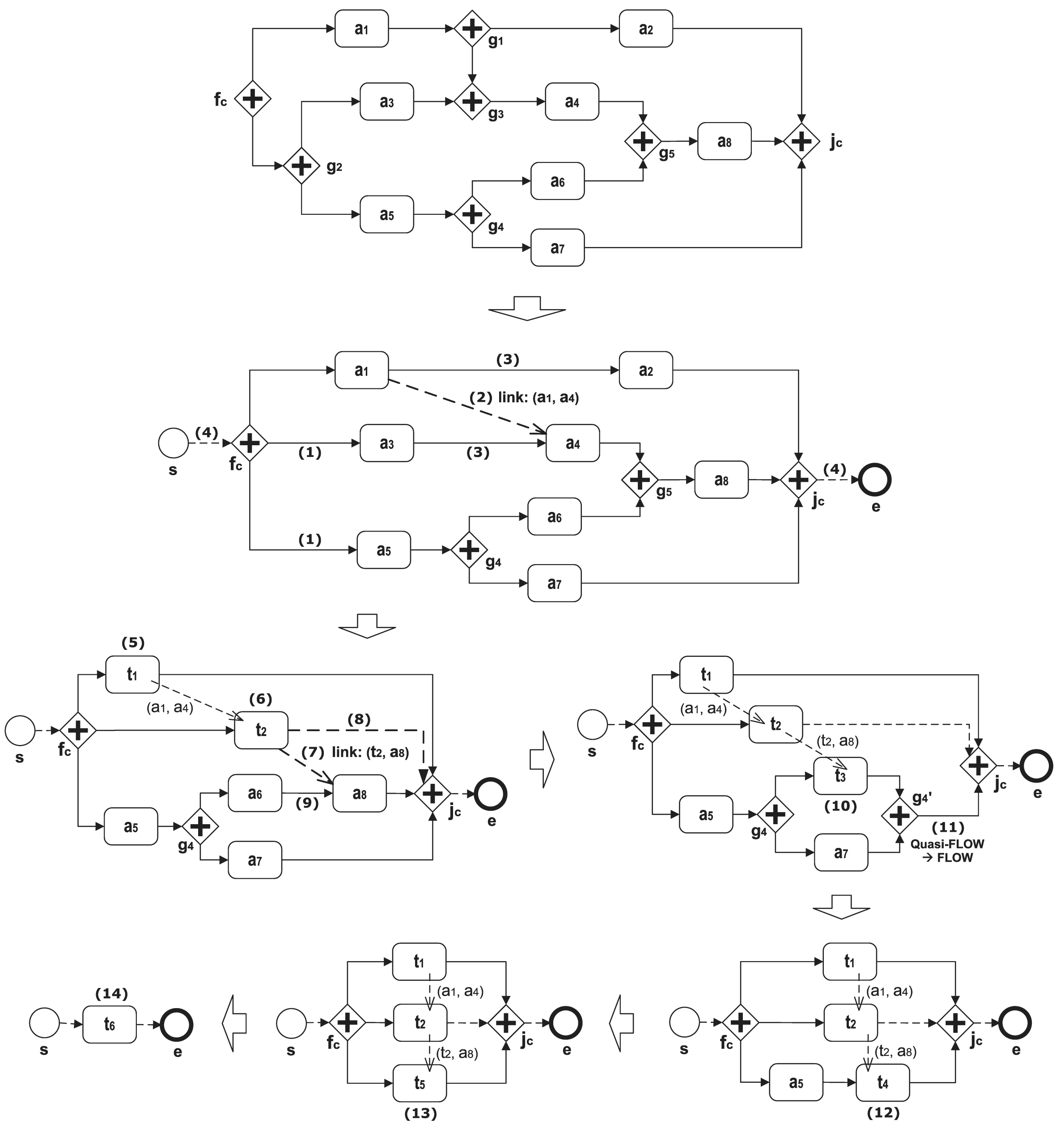
First, we introduce a couple of functions used in the algorithm. The function STRUCTUREDMAP translates a well-structured pattern into the corresponding BPEL structured construct according to Figure 2 in Section 4.2. It outputs a string of BPEL code. The function QUASI2STRUCTURED takes as input a well-formed BPD $\mathcal{W}$ and a quasi-structured pattern $\mathcal{Q}$ in $\mathcal{W}$, refines $\mathcal{Q}$ to construct the corresponding well-structure pattern $\mathcal{C}$ according to Figure 3 in Section 4.3, and finally produces as output the updated BPD with the refined pattern $\mathcal{Q}'$. Also, apart from pred and succ, we apply another auxiliary function CompIn. For any well-formed BPD $\mathcal{W}$, $\mathsf{CompIn}(\mathcal{W})$ returns the set of components contained in $\mathcal{W}$.

For our algorithm defined in Figure 4, the key issue is the identification of control links from a generalised FLOW-pattern. The control links are derived at two different stages. First, before it starts to search for the well-structured patterns in the component, the algorithm checks for all the arcs connecting a fork gateway x directly to a join gateway g (lines 19-28). If such an arc does not lead from a *source* fork gateway nor lead to a *sink* join gateway, it can be explicitly viewed as a link which imposes causal dependencies between the input object of x ($\mathsf{pred}(x)$) and the output object of g ($\mathsf{succ}(g)$) on two parallel threads (lines 22-24). Note that after the reduction of sequentially connected fork/join gateways at the beginning of the algorithm (lines 9-17), it is always true that a gateway is preceded/followed by a task/event object. Therefore, we can derive from the above arc connecting x to g (written as (x, g)), a link identified by its source object $\mathsf{pred}(x)$ and target object $\mathsf{succ}(g)$ (written as $(\mathsf{pred}(x), \mathsf{succ}(g))$). Then, each arc connecting a fork gateway to a join gateway is removed from the component (line 25). After the removal of these arcs, there may appear gateways with an indegree of one and an outdegree of one. These gateways do not perform any routing functions, and thus can be further reduced from the component (lines 30-33). After the above reductions, the component is now ready for the next stage of the mapping.

In a second stage, to facilitate the mapping of the entire component $\mathcal{C}$, we construct a well-formed BPD $\mathcal{W}$ by simply adding a start event and an end event to wrap $\mathcal{C}$ (lines 35-36). $\mathcal{W}$ is a non-trivial BPD unless $\mathcal{C}$ is folded into a task object. Thus, the mapping of $\mathcal{W}$ is repeated until the whole component $\mathcal{C}$ is translated into BPEL (lines 38-61). For a non-trivial BPD $\mathcal{W}$, the mapping always starts from a well-structured SEQUENCE-pattern or FLOW-pattern after each iteration. Such a pattern $\mathcal{C}'$ will be mapped to BPEL via the function STRUCTUREDMAP and then be folded into a new task object t_c (lines 39-42). Next, when there are no well-structured patterns, the algorithm tries to search for a quasi-structured pattern and then re-writes it into a well-structured one via the function QUASI2STRUCTURED (lines 43-44). Finally, when none of the above patterns can be identified, the algorithm starts to derive an additional link from any arc connecting a task/event object x to a join gateway g

```
1:  input: $\mathcal{C} = (\mathcal{O}_c, \mathcal{F}_c, \mathsf{Cond}_c)$: a generalised FLOW-pattern
2:  output: $\mathit{Blocks}$: $\{(t_c$: task object, $\mathit{bpelcode}$: string$)\}$;
3:          $\mathit{Links}$: $\{(s_l$: task/event object, $t_l$: task/event object$)\}$
4:  begin
5:     let $\mathcal{O}_c = \mathcal{T}_c \cup \mathcal{E}_c \cup \mathcal{G}_c$;  $\mathcal{G}_c = \mathcal{G}_c^F \cup \mathcal{G}_c^J$
6:        $f_c \in \mathcal{G}_c^F$ is the source (fork gateway) object of $C$
7:        $j_c \in \mathcal{G}_c^J$ is the sink (join gateway) object of $C$
8:     // reduction of two sequentially connected fork gateways into one fork gateway
9:     while $\exists\ g \in \mathcal{G}_c^F \backslash \{f_c\}$ such that $\mathsf{pred}(g) \in \mathcal{G}_c^F$ do
10:       $\mathcal{G}_c^F := \mathcal{G}_c^F \backslash \{g\}$
11:       $\mathcal{F}_c := \mathcal{F}_c \cup (\{\mathsf{pred}(g)\} \times \mathsf{out}(g)) \setminus (\{(\mathsf{pred}(g), g)\} \cup (\{g\} \times \mathsf{out}(g)))$
12:    end while
13:    // reduction of two sequentially connected join gateways into one join gateway
14:    while $g \in \mathcal{G}_c^J \backslash \{j_c\}$ such that $\mathsf{succ}(g) \in \mathcal{G}_c^J$ do
15:       $\mathcal{G}_c^J := \mathcal{G}_c^J \backslash \{g\}$
16:       $\mathcal{F}_c := \mathcal{F}_c \cup (\mathsf{in}(g) \times \{\mathsf{succ}(g)\}) \setminus (\{(g, \mathsf{succ}(g))\} \cup (\mathsf{in}(g) \times \{g\}))$
17:    end while
18:    // deriving links from arcs connecting a fork gateway directly to a join gateway
19:    for all $g \in \mathcal{G}_c^J$ such that $\mathsf{in}(g) \cap \mathcal{G}_c^F \neq \varnothing$
20:       while $|\mathsf{in}(g)| > 1$ do
21:          select any $x \in \mathsf{in}(g) \cap \mathcal{G}_c^F$ such that $|\mathsf{out}(x)| > 1$
22:             if $x \neq f_c$ and $g \neq j_c$
23:             then $\mathit{Links} := \mathit{Links} \cup \{(\mathsf{pred}(x), \mathsf{succ}(g))\}$
24:             end if
25:             $\mathcal{F}_c := \mathcal{F}_c \backslash \{(x, g)\}$
26:          end select
27:       end while
28:    end for
29:    // reduction of gateways with an indegree and an outdegree of one
30:    while $\exists\ g \in \mathcal{G}_c \backslash \{f_c, j_c\}$ such that $|\mathsf{in}(g)| = |\mathsf{out}(g)| = 1$ do
31:       $\mathcal{G}_c := \mathcal{G}_c \backslash \{g\}$
32:       $\mathcal{F}_c := (\mathcal{F}_c \backslash \{(\mathsf{pred}(g), g), (g, \mathsf{succ}(g))\}) \cup \{(\mathsf{pred}(g), \mathsf{succ}(g))\}$
33:    end while
34:    // construction of a BPD $\mathcal{W}$ containing only component $\mathcal{C}$
35:    let $s$ be a start event and $e$ be an end event
36:    $\mathcal{W} := (\mathcal{O}_c \cup \{s, e\}, \mathcal{F}_c \cup \{(s, f_c), (j_c, e)\}, \mathsf{Cond}[\mathcal{F}_c])$
37:    // mapping of component $\mathcal{C}$
38:    while $\mathsf{CompIn}(\mathcal{W}) \neq \varnothing$ do
39:       if $\exists$ a maximal SEQUENCE-pattern or a FLOW-pattern $\mathcal{C}' \in \mathsf{CompIn}(\mathcal{W})$
40:       then $t :=$ a new task object
41:            $\mathit{Blocks} := \mathit{Blocks} \cup \{(t, \textsc{StructuredMap}(\mathcal{C}'))\}$
42:            $\mathcal{W} := \mathsf{Fold}(\mathcal{W}, \mathcal{C}', t)$
43:       else if $\exists$ a quasi-FLOW-pattern $\mathcal{Q}$ in $\mathcal{W}$
44:            then $\mathcal{W} := \textsc{Quasi2Structured}(\mathcal{W}, \mathcal{Q})$
45:            else // deriving an additional link from any arc connecting a task/event to a join gateway
46:                 begin
47:                    select any $g \in \mathcal{G}_c^J \backslash \{j_c\}$
48:                       select any $x \in \mathsf{in}(g)$
49:                          $\mathit{Links} := \mathit{Links} \cup \{(x, \mathsf{succ}(g))\}$
50:                          $\mathcal{F}_c := (\mathcal{F}_c \backslash \{(x, g)\}) \cup \{(x, j_c)\}$
51:                       end select
52:                       if $|\mathsf{in}(g)| = 1$
53:                       then $\mathcal{G}_c^J := \mathcal{G}_c^J \backslash \{g\}$
54:                            $\mathcal{F}_c := (\mathcal{F}_c \backslash \{(\mathsf{pred}(g), g), (g, \mathsf{succ}(g))\}) \cup \{(\mathsf{pred}(g), \mathsf{succ}(g))\}$
55:                       end if
56:                    end select
57:                 end
58:            end if
59:       end if
60:    end while
61: end
```

Figure 4. Algorithm for translating a generalised FLOW-pattern into a BPEL *flow* construct with control links.



Steps in the above translation procedure:

(1) reduction of fork gateway g_2 which immediately follows source fork gateway f_c
(2) deriving a link from tasks a_1 to a_4, and deleting arc from fork gateway g_1 to join gateway g_3
(3) reduction of gateways g_2 and g_3, both having an indegree of one and an outdegree of one
(4) adding a start event **s** and an end event **e** to construct a BPD for mapping of the whole component
(5) identifying a SEQUENCE-pattern comprising tasks a_1 and a_2, and folding it into task t_1
(6) identifying a SEQUENCE-pattern comprising tasks a_3 and a_4, and folding it into task t_2
(7) deriving a link from tasks t_2 to a_8, and deleting the arc from task t_2 to join gateway g_5
(8) adding arc from task t_2 to sink join gateway j_c to maintain a well-formed BPD
(9) reduction of gateway g_5 which has an indegree of one and an outdegree of one
(10) identifying a SEQUENCE-pattern comprising tasks a_6 and a_8, and folding it into task t_3
(11) re-writing the quasi-FLOW-pattern (enclosed within gateways g_4 and j_c) to a well-structured FLOW-pattern by inserting a new join gateway g_4'
(12) identifying a FLOW-pattern comprising tasks t_3 and a_7, and folding it into task t_4
(13) identifying a SEQUENCE-pattern comprising tasks a_5 and t_4, and folding it into task t_5
(14) identifying a FLOW-pattern comprising tasks t_1, t_2 and t_5, and folding it into task t_6

Figure 5. Example of applying the algorithm shown in Figure 4 to the translation of a generalised FLOW-pattern.

(except the sink join gateway) in $\mathcal{W}$ (lines 46-57). Similarly, after the arc (x, g) is selected and mapped onto a link $(x, \mathsf{succ}(g))$, it is then removed from the component. However, in this case, the source task/event object x will lose its outgoing flow and thereby the resulting BPD will not be well-formed any more. In order to maintain the well-formedness of the BPD without changing its behaviour, the algorithm adds an arc from the task/event x to the end join gateway j_c. Next, if the join gateway g has only one incoming arc left, it can be deleted. Note that in the above procedure, conducting the identification of additional control links (one at a time) only as a last resort, reflects the desire to produce structured BPEL code with as few control links as possible.

Complexity Analysis We now analyse the complexity of the above algorithm. In this analysis, we use the following notations. Symbol $k_{n,m}$ denotes the worst-case complexity of the fragment of the algorithm contained between lines n and m. Symbols ind or outd denote functions that take as input an object x and return the indegree and the outdegree of x respectively, i.e., $\mathsf{ind}(x) = |\mathsf{in}(x)|$, $\mathsf{outd}(x) = |\mathsf{out}(x)|$. Meanwhile, app is a higher-level function that applies a function given as first parameter (e.g., ind or outd) to each element of a set $\mathcal{S} = \{x_1, ..., x_n\}$ given as second parameter, e.g. $\mathsf{app}(\mathsf{ind}, \mathcal{S}) = \{\mathsf{ind}(x_1), ..., \mathsf{ind}(x_n)\}$. Finally, function max takes as input a set of integers, and returns the integer with the maximal value among the set. We use $\mathsf{ind}_{max}(S)$ as a shorthand notation for $\mathsf{max}(\mathsf{app}(\mathsf{ind}, S))$, and $\mathsf{outd}_{max}(S)$ for $\mathsf{max}(\mathsf{app}(\mathsf{outd}, S))$.

Consider the body of the algorithm (lines 5 to 60) in Figure 4. Lines 5 to 7 do not contribute to the asymptotic worst-case complexity. In lines 9 to 12, a reduction is performed for every two sequentially connected fork gateways in $\mathcal{C}$. For each reduction, the set operations (lines 10 to 11) are performed, leading to a complexity of $O(\mathsf{outd}_{max}(\mathcal{G}_c^F))$. In the extreme case, all fork gateways in $\mathcal{G}_c^F$ are sequentially connected, and to reduce them into one fork gateway, $|\mathcal{G}_c^F| - 1$ reductions need to be performed. Thus, the complexity of lines 9 to 12 is bounded by $k_{9,12} = O(|\mathcal{G}_c^F| * \mathsf{outd}_{max}(\mathcal{G}_c^F))$. Similarly, the complexity of lines 14 to 17 for reducing join gateways is $k_{14,17} = O(|\mathcal{G}_c^J| * \mathsf{ind}_{max}(\mathcal{G}_c^J))$. In lines 19 to 28, the number of join gateways that directly follow fork gateways is bounded by $|\mathcal{G}_c^J|$, and for each of these join gateways, the number of its preceding fork gateways is bounded by $O(\mathsf{ind}_{max}(\mathcal{G}_c^J))$. Given that mapping an arc between a join gateway and its preceding fork gateway onto a control link is trivial, the complexity of lines 19 to 28 is given by $k_{19,28} = O(|\mathcal{G}_c^J| * \mathsf{ind}_{max}(\mathcal{G}_c^J))$. In lines 30 to 33, gateways with only one incoming arc and one outgoing arc are removed, hence a complexity of $k_{30,33} = O(|\mathcal{G}_c|)$.

The construction of a BPD $\mathcal{W}$ around component $\mathcal{C}$ in lines 35 and 36 is trivial and does not contribute to the asymptotic worst-case complexity. Within $\mathcal{W}$, the entire component $\mathcal{C}$ is then gradually mapped onto BPEL block constructs in lines 38 to 60. First, in the *if*-clause (line 39), to identify a maximal SEQUENCE-pattern, one may need to explore all tasks and events in linear time, and to identify a FLOW-pattern, one may need to go through all parallel fork gateways. As a result, the complexity of line 39 is bounded by $k_{39} = O(|\mathcal{T}_c \cup \mathcal{E}_c^I| + |\mathcal{G}_c^F| * \mathsf{outd}_{max}(\mathcal{G}_c^F))$. Next, in the *then*-clause (lines 40 to 42), both the BPEL code translation (line 41) and the pattern folding operation (line 42) can be performed in linear time and thus the complexity of this *then*-clause is dominated by the pattern identification in the *if*-clause. The *else*-clause contains a nested *if-then-else* structure. Again, the complexity of the pattern identification in the nested *if*-clause (line 43) dominates the complexity of the pattern refinement in the nested *then*-clause (line 44) as well as the complexity of the link derivation in the nested *else*-clause (lines 46 to 57). Thus, we can focus on analysing the complexity of the *if*-clause. A quasi-FLOW-pattern is similar to a well-structured FLOW-pattern but it allows additional outputs from the starting (fork) gateway and additional inputs to the closing (join) gateway. When identifying such a pattern, an additional step is required to check that at least two output task/event objects of the fork gateway share one same output join gateway. Given a fork gateway $g \in \mathcal{G}_c^F$, this additional step can be implemented by using an algorithm that detects if there is a duplicate in the set of second-degree successors of g, i.e., $\{\mathsf{out}(x) | x \in \mathsf{out}(g)\}$. This duplicate detection can be achieved for example using a sorting algorithm. Hence, the complexity of identifying a quasi-FLOW-pattern is $O(|\mathcal{G}_c^F| * \mathsf{outd}_{max}(\mathcal{G}_c^F) * log(\mathsf{outd}_{max}(\mathcal{G}_c^F)))$. This is also the value of $k_{43,58}$. Next, based on the fact that a maximum of $|\mathcal{O}_c|/2$ patterns may be identified in component $\mathcal{C}$, the complexity of lines 38 to 60 is $k_{38,60} = O(|\mathcal{O}_c| * (k_{39} + k_{43,58}))$. Finally, the complexity of the algorithm shown in Figure 5 can be obtained as a sum of the above worst-case complexity bounds, i.e., $k_{9,12} + k_{14,17} + k_{19,28} + k_{30,33} + k_{38,60}$.

5 Overall Translation Approach: An Example

In this section, we describe and illustrate the overall translation approach. Given a well-formed BPD, the basic idea of each step of the approach is to identify an occurrence of a pattern, to generate its BPEL translation, and then to fold the component or fragment of the model matched by the pattern into a single task object. This is repeated until no pattern is left in the BPD. The process of identifying which pattern to apply next always starts from a maximal SEQUENCE-pattern after each folding. When there are no occurrences of the sequence pattern left in the BPD, occurrences of other well-structured patterns are searched for. If one such occurrence is found, it is processed, leading the folding of a component or fragment of the model. The search for patterns then starts again with the sequence pattern. Since all well-structured non-sequence components are disjoint, the order of identifying these patterns is irrelevant. Next, when no well-structured patterns are left, any quasi-structured patterns are processed – again, the identification of an occurrence of any of these pattern leads to part of the model being folded and the search re-starts again from the sequence pattern. Finally, if no well-structured nor quasi-structured patterns are left, the approach searches for a generalised FLOW-pattern as a last resort. Note that a BPD comprising only the patterns defined in Section 4 can be fully translated into a BPEL process. For other BPDs, we can use the general translation approach described in our prior work [15].

Consider the complaint handling process model shown in Figure 6 which is 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 evaluated (task *evaluate*). If the complainant returns the questionnaire within two weeks (event *returned-questionnaire*), task *process questionnaire* is executed. Otherwise (event *time-out*), the result of the questionnaire is discarded. After either the questionnaire is processed or a time-out has occurred, the result needs to be archived (task *archive*), and in parallel, if the complaint

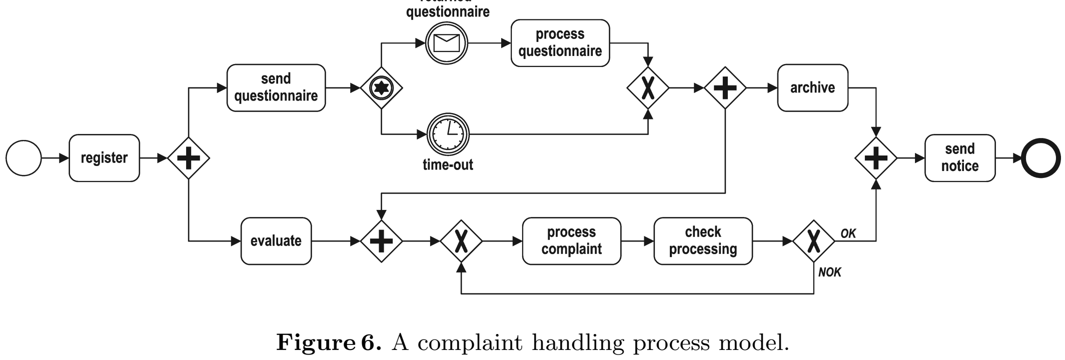


Figure 6. A complaint handling process model.

evaluation has been completed, the actual processing of the complaint (task *process complaint*) can start. Next, the processing of the complaint is checked via task *check processing.* If the check result is NOT OK, the complaint requires re-processing. Otherwise, if the check result is OK and also the questionnaire has been archived, a notice will be sent to inform the complainant about the completion of the complaint handling (task *send notice*). Note that labels OK and NOK on the outgoing flows of a data-based XOR decision gateway, are abstract representations of conditions on these flows.

Following the pattern-based translation approach in Section 4, we map the above BPD onto a BPEL process. Figure 7 sketches the translation procedure which shows how this BPD can be reduced to a trivial BPD. Each component is named $\mathcal{C}_i$ where i specifies in what order the components are processed, and $\mathcal{C}_i$ is folded into a task object named t_c^i. In Figure 7, seven components are identified. All these components except $\mathcal{C}_5$ capture well-structured patterns. In particular, $\mathcal{C}_1$, $\mathcal{C}_3$, $\mathcal{C}_6$, $\mathcal{C}_7$ and $\mathcal{C}_8$ are identified as SEQUENCE-patterns and are folded into *sequence* activities t_c^1, t_c^3, t_c^6, t_c^7 and t_c^8, respectively; $\mathcal{C}_2$ exhibits a PICK-pattern and is folded into a *pick* activity t_c^2; and $\mathcal{C}_4$ exhibits a REPEAT-pattern and is folded into a sequence of activity t_c^1 followed by a *while* activity t_c^4. C_5 exhibits a generalised FLOW-pattern, and is folded into t_c^5 which is a *flow* activity with a control link connecting t_c^3 to t_c^4. The resulting BPEL process is sketched as:

```
<process>
   <links>
      <link name="t3TOt4"/>
   </links>
   <sequence name="$t_c^8$">
      <invoke name="register"/>
      <flow name="$t_c^5$">
         <sequence name="$t_c^6$">
            <sequence name="$t_c^3$">
               <source linkName="t3TOt4"/>
               <invoke name="send questionnaire"/>
               <pick name="$t_c^2$">
                  <onMessage name="returned-questionnaire">
                     <invoke name="process questionnaire"/>
                  </onMessage>
                  <onAlarm name="time-out">
                     <empty/>
                  </onAlarm>
               </pick>
            </sequence>
            <invoke name="archive"/>
         </sequence>
         <sequence name="$t_c^7$">
            <invoke name="evaluate"/>
            <sequence name="$t_c^4$">
               <target linkName="t3TOt4"/>
               <sequence name="$t_c^1$">
                  <invoke name="process complaint"/>
                  <invoke name="check processing"/>
               </sequence>
```

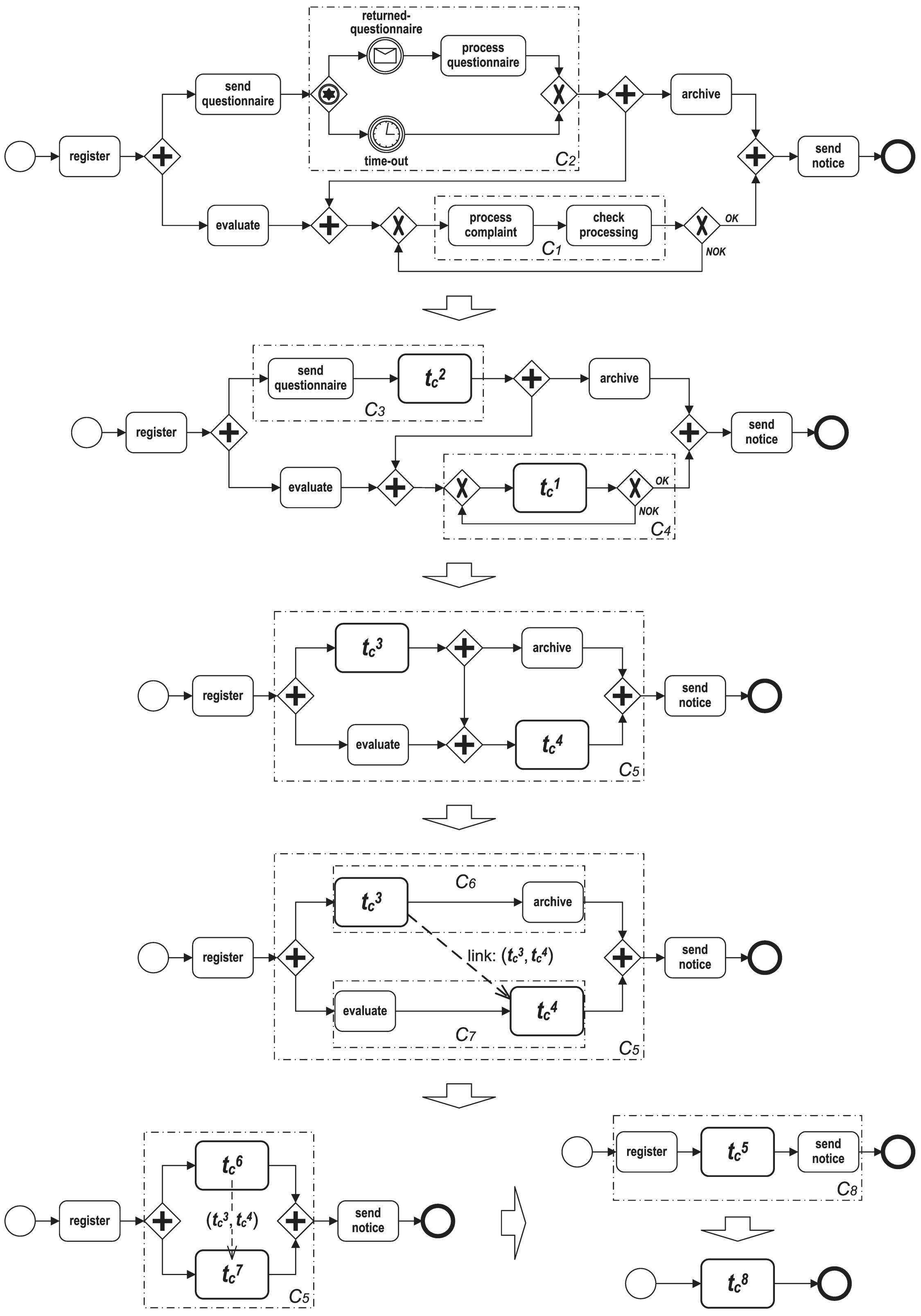


Figure 7. Translating the complaint handling process model in Figure 6 into BPEL.

```
                <while condition="NOK">
                   <sequence name="$t_c^1$">
                      <invoke name="process complaint"/>
                      <invoke name="check processing"/>
                   </sequence>
                </while>
             </sequence>
          </sequence>
       </flow>
       <invoke name="send notice"/>
    </sequence>
</process>
```

6 Related Work

White [5, 18] informally sketches a translation from BPMN to BPEL. However, as acknowledged in [5] this translation is fundamentally limited, e.g. it excludes diagrams with arbitrary topologies and several steps in the translation require human input to identify patterns in the source model. Several tool vendors have integrated variants of this translation into their BPMN modelling tools. Not surprisingly however, these tools are only able to export BPEL code for restricted classes of BPMN models [4, 17].

Research into structured programming in the 60s and 70s led to techniques for translating unstructured flowcharts into structured ones. However, these techniques are no longer 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 [7, 11], while an approach to overcome some of these limitations for processes without parallelism is sketched in [9]. However, these related work only address a piece of the puzzle of translating from graph-oriented process modelling languages to BPEL.

This paper is an extended version of [16] which in turn draws upon insights from one of our previous publications [10]. In [10], we implement a semi-automated mapping from Coloured Petri nets to BPEL. This semi-automated mapping is based on the identification of structural patterns that have commonalities with the set of well-structured translation patterns discussed in the present paper.

In other complementary work [15], we presented a mapping from a graph-oriented language supporting AND-splits, AND-joins, XOR-splits, and XOR-joins, into BPEL. This mapping can deal with any BPMN model composed of elementary activities and these four types of gateways. However, the generated BPEL code relies heavily on BPEL event handlers, while in this paper we make use of BPEL's block-structured constructs, thus obtaining more readable code. It is possible to combine these two methods by iteratively applying first the method presented in this paper, and when this method can not be applied to any component of the model, applying the method in [15] on a minimal-size component, and so on until the process model is reduced to one single component. The technical details of this combined translation procedure are discussed in reference [14] which also introduces another translation approach that can deal with any acyclic BPMN graph containing AND and XOR gateways.

In parallel with our work, Mendling et al. [13] have developed four strategies to translate from graph-oriented process modelling languages (such as BPMN) to BPEL. The first of these strategies, namely *Structure-Identification Strategy*, works by identifying perfectly well-structured components and folding them incrementally, as discussed in Section 4.2. The second and third strategies, namely the *Element-Preservation Strategy* and the *Element-Minimisation Strategy* translate acyclic graph-oriented models into BPEL process definitions that rely intensively on control links, as opposed to relying on BPEL structured activities. These strategies are similar to one of the translation procedures formalised in [14]. Finally, the *Structure-Maximisation Strategy* tries to derive a BPEL process with as many structured activities as possible and for the remaining unstructured fragments, it tries to apply the strategies that rely on control links. None of the translation strategies by Mendling et al. are able to identify quasi-structured and generalised flow-patterns as discussed in this paper.

Finally, it is interesting to mention the work conducted on the translations in the opposite direction, i.e., from BPEL to graph-oriented process modelling languages. One typical example is a toolset, namely *Tools4BPEL*, which consists of two main tools: *BPEL2oWFN* and *Fiona*. BPEL2oWFN maps BPEL specifications onto open Workflow nets using an algorithm described in [6]. The resulting nets can be loaded into a tool called Fiona that can check for various properties both for standalone and for inter-connected processes [12]. Driven by the same motivation of enabling static analysis of BPEL processes, Brogi and Popescu [3] present a translation from BPEL to YAWL (a graph-oriented process definition language inspired by Petri nets). Arguably, this latter BPEL-to-YAWL translation could be adapted to yield a BPEL-to-BPMN translation.

7 Conclusion

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 and structured BPEL code by discovering structural patterns in the BPMN models. The proposal advances the state of the art in BPMN-to-BPEL translations by formally defining not only perfectly block-structured patterns of BPMN models, but also quasi-structured patterns and flow-based acyclic patterns. We have shown how well-structured and quasi-structured model fragments can be mapped into block-structured BPEL constructs, while flow-based acyclic fragments can be mapped into block-structured BPEL constructs with some additional control links to capture dependencies between activities located in different blocks.

An implementation of the proposed pattern-based translation algorithm is available as an open-source tool called BPMN2BPEL[11]. The current tool implementation covers the well-structured patterns presented in this paper as well as other complementary translation algorithms presented in [14, 15]. Ongoing work aims at extending the tool with the ability to detect quasi-structured patterns and flow-based acyclic patterns.

The translation technique described in this paper focuses on mapping a core subset of BPMN's control-flow constructs. For the translation to be complete, it needs to be extended to cover: (i) mappings of individual tasks and events such as send and receive tasks, user tasks, message events, timer events, etc.; and (ii) mappings of other constructs. The first point is covered in some details by [5, 18] and by other ongoing work in the context of the BPMN standardisation process. The second point brings up some challenges which require special attention. In particular, mapping the OR-join gateway is likely to prove challenging especially for BPMN models containing arbitrary cycles. Indeed, this construct has a non-local semantics [8], meaning that the firing behaviour of an OR-join gateway may depend on other gateways in the model located far away from the OR-join gateway, as opposed to only depending on the tokens available on its input arcs. In future, we plan to investigate

[11] BPMN2BPEL is available via `http://www.bpm.fit.qut.edu.au/projects/babel/tools`.

under which conditions can a BPMN model containing OR-join gateways be transformed into an equivalent BPMN model containing only AND and XOR gateways, so that the resulting models can be translated into BPEL using the algorithms described in this and related papers.

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. A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu, editors. *Web Services Business Process Execution Language Version 2.0.* Committee Draft. WS-BPEL TC OASIS, 2005.
3. A. Brogi and R. Popescu. From BPEL processes to YAWL workflows. In *Proceedings of the 3rd International Workshop on Web Services and Formal Methods (WS-FM'2006)*, volume 4184 of *Lecture Notes in Computer Science*, pages 107–122. Springer-Verlag, 2006.
4. Y. Gao. BPMN-BPEL transformation and round trip engineering. URL: `http://www.eclarus.com/pdf/BPMN_BPEL_Mapping.pdf`, March 2006. eClarus Software.
5. Object Management Group. *Business Process Modeling Notation (BPMN) Version 1.0.* OMG Final Adopted Specification. Object Management Group, 2006.
6. S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri nets. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the International Conference on Business Process Management (BPM2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235, Nancy, France, September 2005. Springer-Verlag.
7. 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.
8. Ekkart Kindler. On the semantics of EPCs: Resolving the vicious circle. *Data & Knowledge Engineering*, 56(1):23–40, 2006.
9. J. Koehler and R. Hauser. Untangling Unstructured Cyclic Flows - A Solution Based on Continuations. In R. Meersman, Z. Tari, W.M.P. van der Aalst, C. Bussler, and A. Gal et al., editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004*, volume 3290 of *Lecture Notes in Computer Science*, pages 121–138, 2004.
10. K.B. Lassen and W.M.P. van der Aalst. WorkflowNet2BPEL4WS: A tool for translating unstructured workflow processes to readable BPEL. In *On the Move to Meaningful Internet Systems 2006, OTM Confederated International Conferences, 14th International Conference on Cooperative Information Systems (CoopIS 2006)*, volume 4275 of *Lecture Notes in Computer Science*, pages 127–144. Springer-Verlag, 2006.
11. 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.
12. N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing interacting BPEL processes. In S. Dustdar, J. L. Fiadeiro, and A. Sheth, editors, *Proceedings of the 4th International Conference on Business Process Management (BPM2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2006.
13. 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.
14. C. Ouyang, M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. From business process models to process-oriented software systems: The BPMN to BPEL way. Technical Report BPM-06-27, BPMcenter.org, 2006. Available via `http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-27.pdf`.
15. 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.
16. 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 the 4th International Conference on Web Services (ICWS'06)*, pages 285–292, Chicago, Illinois, USA, September 2006. IEEE Computer Society.
17. B. Silver. The next step in process modelling. URL: `http://www.brsilver.com/wordpress/2006/02/03/the-next-step-in-process-modeling/`, February 2006. BPMSWatch.
18. S. White. Using BPMN to Model a BPEL Process. *BPTrends*, 3(3):1–18, March 2005.