Translating Unstructured Workflow Processes to Readable BPEL: Theory and Implementation

Wil M.P. van der Aalst[1,2] and Kristian Bisgaard Lassen[2]

[1] Department of Computer Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands. w.m.p.v.d.aalst@tue.nl
[2] Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark. k.b.lassen@daimi.au.dk

Abstract. The *Business Process Execution Language for Web Services* (BPEL) has emerged as the de-facto standard for implementing processes. Although intended as a language for connecting web services, its application is not limited to cross-organizational processes. It is expected that in the near future a wide variety of process-aware information systems will be realized using BPEL. While being a powerful language, BPEL is difficult to use. Its XML representation is very verbose and only readable for the trained eye. It offers many constructs and typically things can be implemented in many ways, e.g., using links and the flow construct or using sequences and switches. As a result only experienced users are able to select the right construct. Several vendors offer a graphical interface that generates BPEL code. However, the graphical representations are a direct reflection of the BPEL code and not easy to use by end-users. Therefore, we provide a mapping from Workflow Nets (WF-nets) to BPEL. This mapping builds on the rich theory of Petri nets and can also be used to map other languages (e.g., UML, EPC, BPMN, etc.) onto BPEL. In addition to this we have implemented the algorithm in a tool called WorkflowNet2BPEL4WS.

Keywords: BPEL4WS, Petri nets, workflow management, business process management.

1 Introduction

After more than a decade of attempts to standardize workflow languages (cf. [6, 51]), it seems that the Business Process Execution Language for Web Services (BPEL4WS or BPEL for short) [13] is emerging as the de-facto standard for executable process specification. Systems such as Oracle BPEL Process Manager, IBM WebSphere Application Server Enterprise, IBM WebSphere Studio Application Developer Integration Edition, and Microsoft BizTalk Server 2004 support BPEL, thus illustrating the practical relevance of this language.

Interestingly, BPEL was intended initially for cross-organizational processes in a web services context: "BPEL4WS provides a language for the formal specification of business processes and business interaction protocols. By doing so, it extends the Web Services interaction model and enables it to support business transactions." (see page 1 in [13]). However, it can also be used to support intra-organizational processes. The authors of BPEL [13] envision two possible uses of the language: "Business processes can be described in two ways. Executable business processes model actual behavior of a participant in a business interaction. Business protocols, in contrast, use process descriptions that specify the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal behavior. The process descriptions for business protocols are called abstract processes. BPEL4WS is meant to be used to model the behavior of both executable and abstract processes." In this paper we focus on the use of BPEL as an execution language.

BPEL is an expressive language [67] (i.e., it can specify highly complex processes) and is supported by many systems. Unfortunately, BPEL is not a very intuitive language. Its XML representation is very verbose and there are many, rather advanced, constructs. Clearly, it is at another level than the graphical languages used by the traditional workflow management systems (e.g., Staffware, FileNet, COSA, Lotus Domino Workflow, SAP Workflow, etc.). This is the primary motivation of this paper. How to generate BPEL code from a graphical workflow language?

The modeling languages of traditional workflow management systems are executable but at the same time they appeal to managers and business analysts. Clearly, managers and business analysts will have problems understanding BPEL code. As a Turing complete[3] language BPEL can do, well, anything, but to do this it uses two styles of modeling: graph-based and structured. This can be explained by looking at its history: BPEL builds on IBM's WSFL (Web Services Flow Language) [45] and Microsoft's XLANG (Web Services for Business Process Design) [60] and combines accordingly the features of a block structured language inherited from XLANG with those for directed graphs originating from WSFL. As a result simple things can be implemented in two ways. For example a sequence can be realized using the `sequence` or `flow` elements, a choice based on certain data values can be realized using the `switch` or `flow` elements, etc. However, for certain constructs one is forced to use the block structured part of the language, e.g., a *deferred choice* [8] can only be modeled using the `pick` construct. For other constructs one is forced to use the links, i.e., the more graph-based oriented part of the language, e.g., two parallel processes with a one-way synchronization require a `link` inside a `flow`. In addition, there are very subtle restrictions on the use of links: "A link MUST NOT cross the boundary of a while activity, a serializable scope, an event handler or a compensation handler... In addition, a link that crosses a fault-handler boundary MUST be outbound, that is, it MUST have its source activity within the fault handler and its target activity within a scope that encloses the scope associated with the fault handler. Finally, a link MUST NOT create a control cycle, that is, the source activity must not have the target activity as a logically preceding activity, where an activity A logically precedes an activity B if the initiation of B semantically requires the completion of A. Therefore, directed graphs created by links are always acyclic." (see page 64 in [13]). All of this makes the language complex for end-users. Therefore, there is a need for a "higher level" language for which one can generate *intuitive* and *maintainable* BPEL code.

Such a "higher level" language will not describe certain implementation details, e.g., particularities of a given legacy application. This needs to be added to the generated BPEL code. Therefore, it is important that the generated BPEL code is intuitive and

[3] Since BPEL offers the typical constructs one also encounters in programming languages (e.g., loops and if-then-else constructs) and XML data types, it is easy to show that BPEL is Turing complete.

maintainable. If the generated BPEL code is unnecessary complex or counter-intuitive, it cannot be extended or customized.

Note that tools such as Oracle BPEL Process Manager and IBM WebSphere Studio offer graphical modeling tools. However, these tools reflect directly the BPEL code, i.e., the designer needs to be aware of the structure of the XML document and required BPEL constructs. For example, to model a *deferred choice* in the context of a parallel process [8] the user needs to add a level to the hierarchy (i.e., a `pick` defined at a lower level than the `flow`). Moreover, subtle requirements such as links not creating a cycle still need to be respected in the graphical representation. Therefore, it is interesting to look at a truly graph-based language with no technological-oriented syntactical restrictions and see whether it is possible to generate BPEL code.

In this paper we use a specific class of Petri nets, named *WorkFlow nets* (WF-nets) [1–3], as a *source language* to be mapped onto the *target language* BPEL. There are several reasons for selecting Petri nets as a source language. It is a simple graphical language with a strong theoretical foundation. Petri nets can express all the routing constructs present in existing workflow languages [4, 24, 64] and enforce no technological-oriented syntactical restrictions (e.g., no loops). Note that WF-nets are classical Petri nets without data, hierarchy, time and other extensions. Therefore, their applicability is limited. However, we do *not* propose WF-nets as the language to be used by end-users; we merely use it as the theoretical foundation. It can capture the control-flow structures present in other graphical languages, but it abstracts from other aspects such as data flow, work distribution, etc. Using a real-life example, we will show that the mapping from WF-nets to BPEL presented in this paper can also be used to map Colored Petri Nets (CPNs) onto BPEL, i.e., in the case study we have used CPN Tools as an editor and simulation tool. Similarly, the mapping can be used as a basis for translations from other source languages such as UML activity diagrams [32], Event-driven Process Chains (EPCs) [38, 57], and the Business Process Modeling Notation (BPMN) [66]. Moreover, the basic ideas can also be used to map graph-based languages onto other (partly) block-structured languages.

To support the approach described in this paper, we implemented the tool *WorkflowNet2BPEL4WS*. This tool automatically translates CPNs (respecting the structure of a WF-net) into BPEL code. The code generated by WorkflowNet2BPEL4WS can be imported into, e.g., WebSphere Studio [68].

The remainder of this paper is organized as follows. First, we provide an overview of related work. Then, we present some preliminaries including the BPEL language (Section 3.1), Petri nets (Section 3.2), WF-nets (Section 3.3), and soundness (Section 3.4). Then, in Section 4, we show the how and when WF-nets can be decomposed into components. These decomposition results are used in Section 5 to map WF-nets onto BPEL. The WorkflowNet2BPEL4WS tool is presented in Section 6. In Section 7, we present a case study. Finally, in Section 8 we conclude the paper.

2 Related Work

Since the early nineties workflow technology has matured [30] and several textbooks have been published, e.g., [7, 18, 35, 46]. During this period many languages for modeling workflows have been proposed, i.e., languages ranging from generic Petri-net-based languages to tailor-made domain-specific languages. The Workflow Management Coalition (WfMC) has tried to standardize workflow languages since 1994 but failed to do so [24]. XPDL, the language proposed by the WfMC, has semantic problems [4] and is rarely used. In a way BPEL [13] succeeded in doing what the WfMC was aiming at. However, BPEL is really at the implementation level rather than the workflow modeling level or the requirements level (thus providing the motivation for this paper). For a detailed analysis of BPEL based on the workflow patterns [8] we refer to [67].

Recently, several groups have been working on providing formal semantics for BPEL with the goal to provide some form of analysis [21, 23, 25–29, 34, 44, 47, 49, 50, 55, 59]. It is impossible to discuss all of these approaches in detail. Therefore, we focus on four of the most relevant tools in this area: *WofBPEL*, *WSAT*, *LTSA-WS/BPEL4WS*, and *Tools4BPEL*.

WofBPEL [55] uses a detailed mapping from BPEL to Petri nets and unlike many other approaches this mapping is feature complete, i.e., control links, joins conditions, event handling, fault handling, scopes, compensation, etc. are taking into account in this translation. WofBPEL uses a Petri-net-based analysis tool dedicated to the analysis of workflow processes. This tool (Woflan) was developed over the last decade [63] and is tailored towards the analysis of WF-nets.

WSAT (Web Service Analysis Tool) [28, 29] is a formal specification, verification, and analysis tool for web service compositions based on so-called Guarded Automata (GA). The tool is developed at the University of California at Santa Barbara. BPEL specifications are translated to guarded automata. These are then mapped onto Promela, the input language of the well-know model-checker SPIN. Using SPIN a variety of properties can be checked as long as the mapping yields a finite system. The authors have focussed on interacting BPEL web services [28, 29] using concepts such as synchronizability (i.e., when can asynchronous communication be replaced by synchronous communication).

LTSA-WS/BPEL4WS [26, 27] (also known as the LTSA WS-Engineer plug-in for Eclipse) is an extension to the Labelled Transition System Analyser (LTSA) which allows models to be described by translation of the BPEL4WS implementations and WS-CDL descriptions. It has been developed at the Imperial College London. LTSA-WS/BPEL4WS is able to map BPEL specifications onto labelled transition systems and perform various checks including deadlock freedom, safety, and progress properties (all provided by LTSA). Since model checking is used, also other properties can be investigated. Interestingly, the tool also allows for the synthesis of Message Sequence Charts (MSCs) and compare the synthesized result with the BPEL specification [27].

Tools4BPEL is a tool set which consists of *BPEL2oWFM* and *Fiona*. It has been developed at the Humboldt-Universität zu Berlin. BPEL2oWFM maps BPEL specifications onto Petri nets and is a successor of the BPEL2PN tool based on the translation described in [34, 59]. The focus of the work is twofold. On the one hand, different properties are being verified using model checking techniques (through LOLA) [34]. On the other hand, the tool Fiona can check for controllability (i.e., is there an environment that can interact properly) and, if so, generate the operating guidelines (i.e., instructions on how to use the service) [47].

Note that none of the above approaches aims at the mapping of models onto BPEL, i.e., they all map BPEL onto some other language (e.g., Petri nets).

The work reported in this paper is also related to the various tools and mappings used to generate BPEL code being developed in industry. Tools such as the IBM WebSphere Choreographer and the Oracle BPEL Process Manager offer a graphical notation for BPEL. However, this notation directly reflects the code and there is no intelligent mapping as shown in this paper. This implies that users have to think in terms of BPEL constructs (e.g., blocks, syntactical restrictions on links, etc.). More related is the work of Steven White that discusses the mapping of BPMN onto BPEL [65] and the work by Jana Koehler and Rainer Hauser on removing loops in the context of BPEL [43]. Note that none of these publications provides a mapping of some (graphical) process modeling language onto BPEL: [65] merely presents the problem and discusses some issues using examples and [43] focusses on only one piece of the puzzle.

The work presented in this paper is related to [9] where we describe a case study where for a new bank system requirement are mapped onto Colored Workflow Nets (a subclass of Colored Petri Nets) which are then implemented using BPEL in the IBM WebSphere environment (cf. Section 7).

Finally, we would like to mention that some of the ideas presented in this paper have been applied to the translation from BPMN to BPEL [53]. This translation adopts the ideas presented in this paper to BPMN and combines this with the mapping presented in [54].

3 Preliminaries

This section provides the preliminaries used to map WF-nets onto BPEL.

3.1 Business Process Execution Language for Web Services (BPEL)

As indicated in the introduction, BPEL [13] intends to support the modeling of two types of processes: executable and abstract processes. An *abstract*, (not executable) *process* is a business protocol, specifying the message exchange behavior between different parties without revealing the internal behavior for anyone of them. An *executable process*, specifies the execution order between a number of *activities* constituting the process, the *partners* involved in the process, the *messages* exchanged between these partners, and the *fault* and *exception handling* specifying the behavior in cases of errors and exceptions. The approach described in this paper can be used for both executable and abstract processes. However, in our examples we will mainly refer to executable BPEL processes.

A BPEL process itself is a kind of flow-chart, where each element in the process is called an *activity*. An activity is either a primitive or a structured activity. The set of *primitive activities* contains: `invoke`, invoking an operation on some web service; `receive`, waiting for a message from an external source; `reply`, replying to an external source; `wait`, waiting for some time; `assign`, copying data from one place to another; `throw`, indicating errors in the execution; `terminate`, terminating the entire service instance; and `empty`, doing nothing.

To enable the presentation of complex structures the following *structured activities* are defined: `sequence`, for defining an execution order; `switch`, for conditional routing; `while`, for looping; `pick`, for race conditions based on timing or external triggers; `flow`, for parallel routing; and `scope`, for grouping activities to be treated by the same fault-handler. Structured activities can be nested and combined in arbitrary ways. Within activities executed in parallel the execution order can further be controlled by the usage of `links` (sometimes also called control links, or guarded links), which allows the definition of directed graphs. The graphs too can be nested but must be acyclic and satisfy other subtle requirements as indicated in the introduction.

A detailed or complete description of BPEL is out-of-the-scope of this paper. For more details, the reader is referred to [13] and various web sites such as: `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.

3.2 Petri nets

This section introduces the basic Petri net terminology and notations. Readers familiar with Petri nets can skip this section.

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

Definition 1 (Petri net). *A Petri net is a triple (P, T, F):*

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

Note that we do not consider multiple arcs from one node to another. In the context of workflow procedures it makes no sense to have other weights, because places correspond to conditions.

Elements of $P \cup T$ are called *nodes*. A node x is an *input node* of another node y iff there is a directed arc from x to y (i.e., $(x, y) \in F$). Node x is an *output node* of y iff $(y, x) \in F$. For any $x \in P \cup T$, $\overset{N}{\bullet} x = \{y \mid (y, x) \in F\}$ and $x \overset{N}{\bullet} = \{y \mid (x, y) \in F\}$; the superscript N may be omitted if clear from the context.

The *projection* and *union* of a Petri net are defined as follows.

Definition 2 (Projection). *Let $PN = (P, T, F)$ and $PN' = (P', T', F')$ be Petri nets and $X \subseteq P \cup T$ a set of nodes. $PN|_X = (P \cap X, T \cap X, F \cap (X \times X))$ is the projection of PN onto X. $PN \cup PN' = (P \cup P', T \cup T', F \cup F')$ is the union of PN and PN'.*

At any time a place contains zero or more *tokens*, drawn as black dots. The *state*, often referred to as *marking*, is the distribution of tokens over places, i.e., $M \in P \to \mathbb{N}$. We will represent a state as follows: $1p_1 + 2p_2 + 1p_3 + 0p_4$ is the state with one token in place p_1, two tokens in p_2, one token in p_3 and no tokens in p_4. We can also represent this state as follows: $p_1 + 2p_2 + p_3$. To compare states we define a partial ordering. For any two states M_1 and M_2, $M_1 \leq M_2$ iff for all $p \in P$: $M_1(p) \leq M_2(p)$.

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

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

Given a Petri net (P, T, F) and a state M_1, we have the following notations:

- $M_1 \xrightarrow{t} M_2$: transition t is enabled in state M_1 and firing t in M_1 results in state M_2
- $M_1 \rightarrow M_2$: there is a transition t such that $M_1 \xrightarrow{t} M_2$
- $M_1 \xrightarrow{\sigma} M_n$: the firing sequence $\sigma = t_1t_2t_3 \dots t_{n-1}$ leads from state M_1 to state M_n via a (possibly empty) set of intermediate states $M_2, ...M_{n-1}$, i.e., $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} ... \xrightarrow{t_{n-1}} M_n$

A state M_n is called *reachable* from M_1 (notation $M_1 \xrightarrow{*} M_n$) iff there is a firing sequence σ so that $M_1 \xrightarrow{\sigma} M_n$. Note that the empty firing sequence is also allowed, i.e., $M_1 \xrightarrow{*} M_1$.

We use (PN, M) to denote a Petri net PN with an initial state M. A state M' is a *reachable state* of (PN, M) iff $M \xrightarrow{*} M'$.

Let us define some standard properties for Petri nets. First, we define properties related to the dynamics of a Petri net, then we give some structural properties.

Definition 3 (Live). *A Petri net (PN, M) is live iff, for every reachable state M' and every transition $t \in T$ there is a state M'' reachable from M' which enables t.*

A Petri net is *structurally live* if there exists an initial state such that the net is live.

Definition 4 (Bounded, safe). *A Petri net (PN, M) is bounded iff for each place $p \in P$ there is a natural number n so that for every reachable state the number of tokens in p is less than n. The net is safe iff for each place the maximum number of tokens does not exceed 1.*

A Petri net is *structurally bounded* if the net is bounded for any initial state.

For $PN = (P, T, F)$ we also define some standard structural properties.

Definition 5 (Strongly connected). *A Petri net is strongly connected iff, for every pair of nodes (i.e., places and transitions) x and y, there is a path leading from x to y.*

Definition 6 (Free-choice). *A Petri net is a free-choice Petri net iff, for every two transitions $t_1 \in T$ and $t_2 \in T$, $\bullet t_1 \cap \bullet t_2 \neq \emptyset$ implies $\bullet t_1 = \bullet t_2$.*

Definition 7 (State machine). *A Petri net is state machine iff each transition has at most one input place and at most one output place, i.e., for all $t \in T$: $|\bullet\, t| \leq 1$ and $|t \bullet| \leq 1$.*

Definition 8 (Marked graph). *A Petri net is marked graph iff each place has at most one input transition and at most one output transition, i.e., for all $p \in P$: $|\bullet\, p| \leq 1$ and $|p \bullet| \leq 1$.*

See [17, 56] for a more elaborate introduction to these standard notions.

3.3 WF-nets

A Petri net which models the control-flow dimension of a workflow, is called a *Work-Flow net* (WF-net, [1]). In WF-net the transitions correspond to activities. Some of the transitions represent "real activities" while others are added for routing purposes (i.e., similar to the structured activities in BPEL). Places correspond to pre- and post-conditions of these activities. It should be noted that a WF-net specifies the dynamic behavior of a single *case* (i.e., *process instance* in BPEL terms) in isolation.

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

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

A WF-net has one input place (i) and one output place (o) because any case handled by the procedure represented by the WF-net is created when it enters the WFM system and is deleted once it is completely handled by the system, i.e., the WF-net specifies the life-cycle of a case. The third requirement in Definition 9 has been added to avoid "dangling activities and/or conditions", i.e., activities and conditions which do not contribute to the processing of cases.

Given the definition of a WF-net it is easy to derive the following properties [3].

Proposition 1 (Properties of WF-nets). *Let $PN = (P, T, F)$ be Petri net.*

- *If PN is a WF-net with source place i, then for any place $p \in P$: $\bullet p \neq \emptyset$ or $p = i$, i.e., i is the only source place.*
- *If PN is a WF-net with sink place o, then for any place $p \in P$: $p\bullet \neq \emptyset$ or $p = o$, i.e., o is the only sink place.*
- *If PN is a WF-net and we add a transition t^* to PN which connects sink place o with source place i (i.e., $\bullet t^* = \{o\}$ and $t^*\bullet = \{i\}$), then the resulting Petri net is strongly connected.*
- *If PN has a source place i and a sink place o and adding a transition t^* which connects sink place o with source place i yields a strongly connected net, then every node $x \in P \cup T$ is on a path from i to o in PN and PN is a WF-net.*

3.4 Soundness

In this section we summarize some of the basic results for WF-nets presented in [1–3].

The three requirements stated in Definition 9 can be verified statically, i.e., they only relate to the structure of the Petri net. However, there is another requirement which should be satisfied:

> *For any case, the procedure will terminate eventually and the moment the procedure terminates there is a token in place o and all the other places are empty.*

Moreover, there should be no dead activities, i.e., it should be possible to execute an arbitrary transition by following the appropriate route through the WF-net. These two additional requirements correspond to the so-called *soundness property* [2].

Definition 10 (Sound). *A procedure modeled by a WF-net $PN = (P, T, F)$ is sound if and only if:*

(i) For every state M reachable from state i, there exists a firing sequence leading from state M to state o. Formally:[4]

$$\forall_M (i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} o)$$

(ii) State o is the only state reachable from state i with at least one token in place o. Formally:

$$\forall_M (i \xrightarrow{*} M \ \wedge \ M \geq o) \Rightarrow (M = o)$$

(iii) There are no dead transitions in (PN, i). Formally:

$$\forall_{t \in T} \ \exists_{M,M'} \ i \xrightarrow{*} M \xrightarrow{t} M'$$

Note that the soundness property relates to the dynamics of a WF-net. The first requirement in Definition 10 states that starting from the initial state (state *i*), it is always possible to reach the state with one token in place *o* (state *o*). If we assume a strong notion of fairness, then the first requirement implies that eventually state *o* is reached. Strong fairness means that in every infinite firing sequence, each transition fires infinitely often. The fairness assumption is reasonable in the context of WFM: All choices are made (implicitly or explicitly) by applications, humans or external actors. Clearly, they should not introduce an infinite loop. Note that the traditional notions of fairness (i.e., weaker forms of fairness with just local conditions, e.g., if a transition is enabled infinitely often, it will fire eventually) are not sufficient. See [2, 41] for more details. The second requirement states that the moment a token is put in place *o*, all the other places should be empty. The last requirement states that there are no dead transitions (activities) in the initial state *i*.

Given a WF-net $PN = (P, T, F)$, we want to decide whether PN is sound. In [1] we have shown that soundness corresponds to liveness and boundedness. To link soundness to liveness and boundedness, we define an extended net $\overline{PN} = (\overline{P}, \overline{T}, \overline{F})$. $\overline{PN}$ is the Petri net obtained by adding an extra transition t^* which connects o and i. The extended Petri net $\overline{PN} = (\overline{P}, \overline{T}, \overline{F})$ is defined as follows: $\overline{P} = P$, $\overline{T} = T \cup \{t^*\}$, and $\overline{F} = F \cup \{(o, t^*), (t^*, i)\}$. In the remainder we will call such an extended net the *short-circuited* net of PN. The short-circuited net allows for the formulation of the following theorem.

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

Proof. See [1]. □

This theorem shows that standard Petri-net-based analysis techniques can be used to verify soundness.

Sometimes we require a WF-net to be safe, i.e., no marking reachable from (PN, i) marks a place twice. Although safeness is defined with respect to some initial marking, we extend it to WF-nets and simply state that a WF-net is safe or not.

[4] Note that there is an overloading of notation: the symbol *i* is used to denote both the *place i* and the *state* with only one token in place *i* (see Section 3.2).

In literature there exist many variants of the "classical" notion of soundness used here. Juliane Dehnert uses the notion of relaxed soundness where proper termination is possible but not guaranteed [16, 19]. The main idea is that the scheduler of the workflow system should avoid problems like deadlocks etc. In [42] Ekkart Kindler et al. define variants of soundness tailored towards interorganizational workflows. Kees van Hee et al. [33] define a notion of soundness where multiple tokens in the source place are considered. A WF-net is k-sound if it "behaves well" when there are k tokens in place i, i.e., no deadlocks and in the end there are k tokens in place o. Robert van der Toorn uses the same concept in [61]. In [11, 5] stronger notions of soundness are used and places have to be safe. Another notion of soundness is used in [39, 40] where there is not a single sink place but potentially multiple sink transitions. See [61] for the relation between these variants of the same concept. Other references using (variants of) the soundness property include [31, 48]. For simplicity we restrict ourselves to the classical notion of soundness described in Definition 10.

4 Decomposing a WF-net into Components

After introducing the preliminaries we focus on the actual problem: mapping WF-nets onto BPEL. As indicated in the introduction, it is important that the generated BPEL code is intuitive and maintainable. If the generated BPEL code is unnecessary complex or counter-intuitive, it cannot be extended or customized. Therefore, we try to map parts of the WF-net onto BPEL constructs that fit the best. For example, a sequence of transitions connected through places should be mapped onto a BPEL `sequence`. We aim at recognizing "sequences", "switches", "picks", "while's", and "flows" where the most specific construct has our preference, e.g., for a sequence we prefer to use the `sequence` element over the `flow` element even though both are possible. We aim at an iterative approach where the WF-net is reduced by packaging parts of the network into suitable BPEL constructs.

We would like to stress that our goal is not to provide just any mapping of WF-nets onto BPEL. Note that a large class of WF-nets can be mapped directly onto a BPEL `flow` construct. However, such a translation results in unreadable BPEL code. Instead we would like to map a graph-based language like WF-nets onto a hierarchical decomposition of specific BPEL constructs. For example, if the WF-net contains a sequence of transitions (i.e., activities) this should be mapped onto the more specific `sequence` construct rather than the more general (and more verbose) `flow` construct. Hence, our goal is to *generate readable and compact code*. Note that when we talk about readable code, we primarily refer to the structure and not the syntactical representation. BPEL-based XML code will always remain verbose. However, BPEL editors tend to reflect the structure and therefore it is most important that the designer can understand the structure, i.e., the nesting structured activities.[5] Note that typically the generated code only serves as template code. Hence, designers need to understand the structure.

[5] Consider for example the mapping of "standard process models" onto BPEL described in [54]. This mapping is quite generic, but maps all concepts onto event handlers, thus resulting in less readable code that is using only a very small subset of BPEL.

To map WF-nets onto (readable) BPEL code, we need to transform a graph structure to a block structure. For this purpose we use *components*. A component should be seen as a selected part of the WF-net that has a clear start and end. One can think of it as subnet satisfying properties similar to a WF-net. However, unlike a WF-net, a component may start and/or end with a transition, i.e., WF-nets are "place bordered" while components may be "place and/or transition bordered". The goal is to map components onto "BPEL blocks". For example, a component holding a purely sequential structure should be mapped onto a BPEL `sequence` while a component holding a parallel structure should be mapped onto a `flow`.

Section 5 describes the mapping of components in the WF-net to BPEL constructs. However, before describing the mapping, this section formalizes the notion of components and analyzes some of their properties.

Definition 11 (Component). *Let $PN = (P, T, F)$ be a WF-net. C is a component of PN if and only if*

(i) $C \subseteq P \cup T$,
(ii) there exists different source and sink nodes $i_C, o_C \in C$ such that
- $\bullet(C \setminus \{i_C\}) \subseteq C \setminus \{o_C\}$,
- $(C \setminus \{o_C\})\bullet \subseteq C \setminus \{i_C\}$, *and*
- $(o_C, i_C) \notin F$.

Note that any component contains at least a place and a transition. A component is *trivial* if it only contains one transition (and one or two places). Note that trivial components contain two or three nodes.

As indicated above components may be "place and/or transition bordered". The following definition provides some notations and terminology to deal with components having a transition as source or sink node.

Definition 12. *Let $PN = (P, T, F)$ be a WF-net and let C be a component of PN with source i_C and sink o_C. We introduce the following notations and terminology:*

– *C is a PP-component if $i_C \in P$ and $o_C \in P$,*
– *C is a TT-component if $i_C \in T$ and $o_C \in T$,*
– *C is a PT-component if $i_C \in P$ and $o_C \in T$,*
– *C is a TP-component if $i_C \in T$ and $o_C \in P$,*
– $\overline{C} = C \setminus \{i_C, o_C\}$,
– $PN||_C =$
  - $PN|_C$ *if* $i_C \in P$ *and* $o_C \in P$,
  - $PN|_C \cup (\{p_{(i,C)}\}, \{i_C\}, \{p_{(i,C)}, i_C\}) \cup (\{p_{(o,C)}\}, \{o_C\}, \{o_C, p_{(o,C)}\})$ *if* $i_C \in T$ *and* $o_C \in T$,[6]
  - $PN|_C \cup (\{p_{(o,C)}\}, \{o_C\}, \{o_C, p_{(o,C)}\})$ *if* $i_C \in P$ *and* $o_C \in T$,
  - $PN|_C \cup (\{p_{(i,C)}\}, \{i_C\}, \{p_{(i,C)}, i_C\})$ *if* $i_C \in T$ *and* $o_C \in P$.
– *$[PN]$ is the set of non-trivial components of PN, i.e., all components containing two or more transitions.*

[6] Note that $p_{(i,C)}$ and $p_{(o,C)}$ are the (fresh) identifiers of the places added to make a transition bordered component into a place bordered component.

$PN||_C$ transforms a component into a place-bordered component, i.e., a classical WF-net. In case of a TP-component or PT-component one place needs to be added. In case of a TT-component two places need to be added: $p_{(i,C)}$ are $p_{(o,C)}$. Using the following lemma we will show that the result is indeed a WF-net.

Lemma 1. *Let $PN = (P, T, F)$ be a WF-net. Components of PN are uniquely defined by their source and sink nodes, i.e., for any two components C_1, C_2: $C_1 = C_2$ if and only if $i_{C_1} = i_{C_2}$ and $o_{C_1} = o_{C_2}$.*

Proof. Clearly, $C_1 = C_2$ implies $i_{C_1} = i_{C_2}$ and $o_{C_1} = o_{C_2}$. Now assume that $i_{C_1} = i_{C_2}$ and $o_{C_1} = o_{C_2}$ but there is a node $x \in C_1 \setminus C_2$. This is not possible because x must be on a path from i_{C_1} to o_{C_1} and therefore also on a path from i_{C_2} to o_{C_2} and in C_2. Similarly, there cannot be a node in $x \in C_2 \setminus C_1$, and therefore, C_1 and C_2 coincide. □

The next theorem not only shows that $PN||_C$ results in a WF-net, but that, provided the initial WF-net is safe and sound, the component is also safe a sound. This result will be used to prove the compositional nature of safe and sound WF-nets and, consequently, allow us to incrementally transform a "componentized" WF-net into a block-structured BPEL specification.

Theorem 2. *Let $PN = (P, T, F)$ be a WF-net and C is a component of PN.*

- *$PN||_C$ is a WF-net.*
- *If PN is safe and sound, then $PN||_C$ is safe and sound.*

Proof. First, we prove that $PN||_C$ is a WF-net. Assume C is a PP-component. i_C is the source place of $PN||_C$ because $(C \setminus \{o_C\})\bullet \subseteq C \setminus \{i_C\}$. i_C is not the output node of any node x in C because if $x \in (C \setminus \{o_C\})$ then $x\bullet \subseteq C \setminus \{i_C\}$ and if $x = o_C$ then $(o_C, i_C) \notin F$. Similarly, o_C can be shown to be a sink place of $PN||_C$. Every node $x \in C$ is on a path from i_C to o_C in $PN||_C$. Since PN is a WF-net there is a path from the source node to the sink node visiting x in PN. Clearly, this path also visits i_C and o_C. If C is not a PP-component, the same argumentation can be used. However, a dummy node is added before the source node i_C and/or after the sink node o_C.

Second, we prove that $PN||_C$ is safe and sound if PN is safe and sound. This result can be obtained by applying Theorem 3.4 in [3]. Let $PN = (P, T, F)$ be safe and sound. Assume C is a PP-component. Consider the subnet $PN||_C = (P_1, T_1, F_1)$. Create another Petri net $PN' = (P_2, T_2, F_2)$ resulting from replacing the nodes in $\overline{C}$ (cf. Definition 12) by a transition t^+. Note that the only overlap between $PN||_C$ and PN' is $\{i_C, o_C\}$. If in PN a transition in $\bullet i_C$ fires, then it should be possible to fire a transition in $o_C\bullet$ because of the liveness of the original net. If a transition in $o_C\bullet$ fires, the places in $\overline{C}$ should become empty. If the places in $\overline{C}$ are not empty after firing a transition in $o_C\bullet$, then there are two possibilities: (1) it is possible to move the subnet to a state such that a transition in $o_C\bullet$ can fire (without firing transitions in $T \setminus C$) or (2) it is not possible to move to such a state. In the first case, the place o_C in PN is not safe. In the second case, a token is trapped in the subnet or the subnet is not safe the moment a transition in $\bullet i_C$ fires. Hence, $PN||_C$ is sound and safe. If C is not a PP-component, the same argumentations can be used. □

Soundness and safeness are desirable properties. Theorem 2 shows that these desirable properties are propagated to any component in the net. A similar result holds in the other direction. To prove this we define function $fold$ that replaces a component by a single transition. This function will play a crucial role in the next section where we incrementally replace components by BPEL code.

Definition 13 (Fold). *Let $PN = (P, T, F)$ be a WF-net and let C be a non trivial component of PN (i.e., $C \in [PN]$). Function fold replaces C in PN by a single transition t_C, i.e., $fold(PN, C) = (P', T', F')$ with:*

- $P' = P \setminus \overline{C}$,
- $T' = (T \setminus C) \cup \{t_C\}$,
- $F' = (F \cap ((P' \times T') \cup (T' \times P'))) \cup \{(p, t_C) | p \in P \cap (\{i_C\} \cup \bullet i_C)\} \cup \{(t_C, p) | p \in P \cap (\{o_C\} \cup o_C \bullet)\}$.

Note that folding can also be defined for trivial components. However, in the result $fold(PN, C)$ the only transition in C is renamed to t_C without changing the net structure. Figure 1 illustrates the basic idea of function $fold$. Fort the moment, please ignore the illustration behind each of the transitions t_C for the moment: It symbolizes the BPEL code attached to t_C describing component C (as will be explained later). Note that Figure 1 shows each of the four possible component types: PP, TP, PT, and TT.

Now we can formulate the main result of this section. Using the $fold$ operator it is possible to replace a component by a single transition. The resulting Petri net is again a WF-net. Moreover, desirable properties such as soundness and safeness are not affected by folding or unfolding the component.

Theorem 3. *Let $PN = (P, T, F)$ be a WF-net and let $C \in [PN]$ be a non-trivial component.*

- *$fold(PN, C)$ is a WF-net.*
- *PN is safe and sound if and only if both $PN||_C$ and $fold(PN, C)$ are safe and sound.*

Proof. It is easy to see that $fold(PN, C)$ is indeed a WF-net. Folding C does not remove source place i or sink place o of PN. Moreover, folding does not introduce any new source of sink nodes. It also does not disable any paths from i to o: any path in PN through C is still possible in $fold(PN, C)$ by going through transition t_C.

Assume PN is safe and sound. Theorem 2 already showed that in this case $PN||_C$ is safe and sound. Remains to prove that $fold(PN, C)$ is safe and sound. Here we can use the same line of reasoning as in the proof of Theorem 2. Component C in PN becomes "active" if one of its transitions fires. Then there may be several internal steps in C executed in parallel with the rest of PN but eventually o_C (if o_C is a place) or the places in $o_C\bullet$ (if o_C is a transition) get marked. Since PN is safe, C can be activated only once. Hence, if one abstracts from the internal states of C, PN and $fold(PN, C)$ have identical behaviors and clearly $fold(PN, C)$ is safe and sound. In other words: since the subnet which corresponds to t_C behaves like a transition which may postpone the

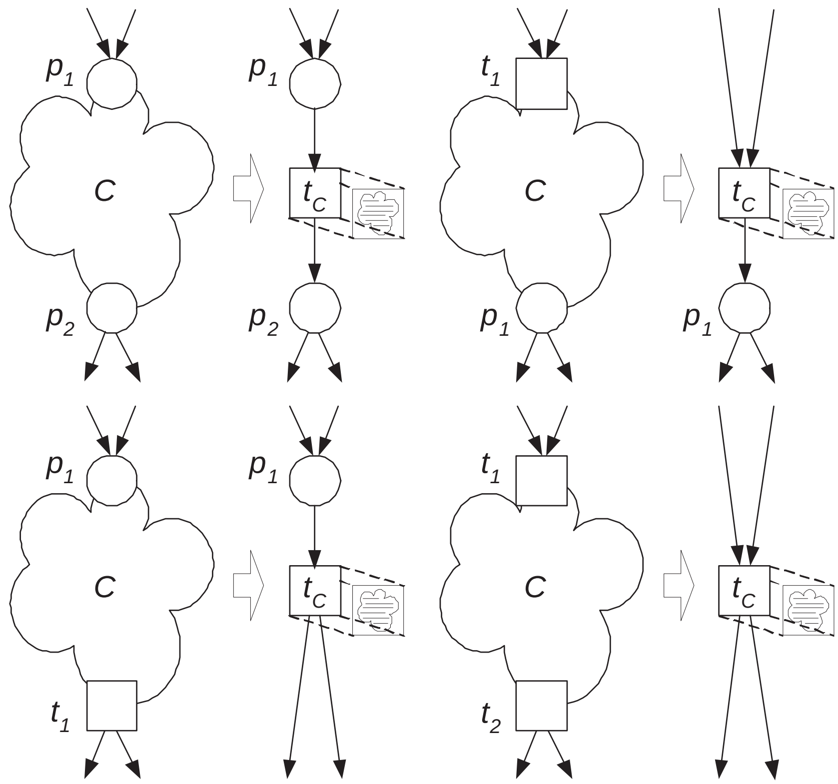


Figure 1. Folding a component C into a single transition t_C using the four types of components identified in Definition 12 (PP, TT, PT, and TP).

production of tokens, we can replace the subnet by t_C without changing dynamic properties such as safeness and soundness. This can be shown more formally by establishing a relation between the markings of (PN, i) and $(\mathit{fold}(PN, C), i)$. Let M_1 be a marking of (PN, i) and let M_2 be a marking of $(\mathit{fold}(PN, C), i)$. M_1 corresponds to M_2 if and only if (1) $M_1 = M_2$ (i.e., the component is not activated) or (2) $M_1(p) = M_2(p)$ for all $p \notin P \cap (\{i_C\} \cup \bullet i_C)$ and $M_1(p) = M_2(p) + \mathit{if}\,(M_1 \cap \overline{C}) \neq \emptyset\ \mathit{then}\ 1\ \mathit{else}\ 0$ for all $p \in P \cap (\{i_C\} \cup \bullet i_C)$.

Assume both $PN||_C$ and $\mathit{fold}(PN, C)$ are safe and sound. We can use a similar approach to show that PN is safe and sound. Again the states in PN can be related to states in $PN||_C$ and $\mathit{fold}(PN, C)$. The subnet of PN corresponding to t_C can only postpone the production of tokens on the output places of t_C and therefore cannot invalidate properties such as safeness and soundness. □

Theorem 3 shows that desirable properties such as safeness and soundness are not affected by the folding or unfolding of components. Assume we have a component C having not only a Petri net representation $PN||_C$ but also an equivalent BPEL representation. If we associate the BPEL representation to some activity t_C, we can replace C by t_C without changing safeness and soundness.

The focus of Theorem 3 is on safeness and soundness. However, as the proof of this theorem suggest, it is possible to make a much more direct relationship between the states of the folded and unfolded net. As is shown in [11] notions such as branching bisimulation can be used to reason about the observational equivalence of the folded and unfolded net. However, we do not show this here because it only makes sense to formalize this if there is a manageable formalization of BPEL. At this point in time such as formalization does not exist. There have been several approaches using finite state machines [25, 28], process algebras [23, 44], abstract state machines [20, 22], and Petri nets [52, 50, 58, 62]. However, these formalizations are either incomplete or very complicated. Since we are not attempting to provide a formal BPEL semantic, we can take a more pragmatic approach. We simply map WF-nets onto BPEL and restrict ourselves to a simple subset of BPEL.

5 Mapping WF-nets onto BPEL

This section introduces the mapping from WF-nets onto BPEL. First, we discuss possible annotations of transitions to refer to primitive BPEL activities. Second, we describe the algorithm used to generate BPEL code. Finally, we present an example.

The basic idea of the approach was already shown in Figure 1. The idea is to start with an annotated WF-net where each transition is labeled with references to primitive activities such as `invoke` (invoking an operation on some web service), `receive` (waiting for a message from an external source), `reply` (replying to an external source), `wait` (waiting for some time), `assign` (copying data from one place to another), `throw` (indicating errors in the execution), and `empty` (doing nothing). Taking this as starting point, a component in the annotated WF-net is mapped onto BPEL code. The component C is replaced by transition t_C whose inscription (cf. Figure 1) describes the BPEL code associated to the whole component. This process is repeated until there is just a single transition whose inscription corresponds to the BPEL specification of the entire process. How this can be done is detailed in the remainder.

5.1 Annotating WF-nets

For the translation to BPEL the nature of choices is important, i.e., a place with multiple output transitions can be mapped onto a `pick` or a `switch`. Similarly, it is important to state what primitive activity a transition refer to. Therefore, we annotate places with information on the nature of choices and transitions with references to primitive activities such as `invoke`, `receive`, `reply`, `wait`, `assign`, `throw`, and `empty`.

Definition 14 (Annotated WF-net). $PN = (P, T, F, \tau_P, \tau_G, \tau_{MA}, \tau_T)$ *is annotated WF-net if and only if:*

(i) (P, T, F) *is a WF-net.*
(ii) τ_P *is a function with domain* $dom(\tau_P) = \{p \in P \mid |p\bullet| \geq 2\}$ *such that for all* $p \in dom(\tau_P)$: $\tau_P(p) \in \{explicit, implicit\}$,
(iii) τ_G *is a function with domain* $dom(\tau_G) = \{t \in T \mid \exists_{p \in \bullet t}\, p \in dom(\tau_P) \wedge \tau_P(p) = explicit\}$ *such that for all* $t \in dom(\tau_G)$: $\tau_G(t)$ *is a boolean expression (i.e. the guard of one of the alternatives in an explicit choice),*
(iv) τ_{MA} *is a function with domain* $dom(\tau_{MA}) = \{t \in T \mid \exists_{p \in \bullet t}\, p \in dom(\tau_P) \wedge \tau_P(p) = implicit\}$ *such that for all* $t \in dom(\tau_{MA})$: $\tau_{MA}(t)$ *is a string describing a message (in case of a message trigger, i.e., the BPEL* `onMessage` *construct) or a time trigger (i.e., the BPEL* `onAlarm` *construct). Note that a time trigger has a* `for` *attribute (to specify a timeout) and/or an* `until` *attribute (to specify a deadline).*
(v) τ_T *is a function with domain* T *such that for all* $t \in T$: $\tau_T(t) \in \{receive, reply, wait, empty, \ldots\}$.

We do not associate any semantics to these annotations but just use them to guide the generation of BPEL code. Moreover, we assume that for any $p \in dom(\tau_P)$, the output transitions $t \in p\bullet$ have guards, where $\tau_P(p) = explicit$, that are mutually exclusive and together cover all possibilities for guards. In any "context": $\forall_{t_1, t_2 \in p\bullet}\ (\tau_G(t_1) \wedge \tau_G(t_2)) \Rightarrow (t_1 = t_2)$ and $\exists_{t \in p\bullet}\ \tau_G(t)$. In the rest of the paper we will use the shorthand *expr* on an arc from place p, where $\tau_P(p) = explicit$, to transition t for the annotation $\tau_G(t) = expr$.

Output transitions $t \in p\bullet$ where $\tau_P(p) = implicit$ have a label $\tau_{MA}(t)$ describing the trigger to select this specific branch in a choice situation. The `pick` construct in BPEL allows for two types of triggers: message triggers and time triggers. In case of a message trigger, $\tau_{MA}(t)$ denotes the message expected (e.g., the corresponding `operation` attribute). In case of a time trigger, $\tau_{MA}(t)$ denotes the attributes expected by the `onAlarm` construct. There may be a `for` attribute denoting the timeout (i.e., a duration expression) and/or an `until` attribute denoting the deadline (i.e., an absolute time). In the rest of the paper we will use the shorthand `msg1`, `msg2`, ..., `to1`, `to2`, ..., `dl1`, `dl2`, ..., `(to1,dl1)`, etc. to denote $\tau_{MA}(t)$ in the figures.

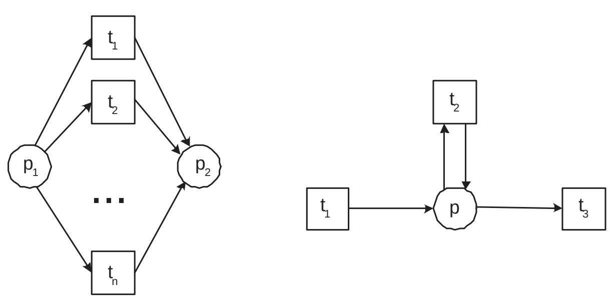
We assume all definitions for WF-nets to be defined for annotated WF-nets.

Note that we abstract from the actual BPEL syntax here. One can think of the range of function τ_T as the set of possible primitive activities; the default τ_T of a transition t is `invoke`. The algorithm does not depend on the precise syntax of these activities and therefore we consider this just to be a label (e.g., a piece of text). The focus is on the control-flow, e.g., we do not do any checking on variables. This focus is justified by our starting point: an annotated WF-net.

5.2 Algorithm

As indicated before and partly illustrated by Figure 1, our approach iteratively folds an annotated WF-net into a WF-net with just one transition whose label contains the complete BPEL specification. The algorithm starts with the complete WF-net and in each step part of this net (i.e., a component) is replaced by a single transition whose label captures the behavior of the component in terms of BPEL. This is repeated until no further folding is possible, i.e., there is just one activity left. Initially transitions correspond to primitive BPEL activities. However, while folding the WF-net, the "composite transitions" refer to structured BPEL activities (i.e., containing primitive activities).

Again we would like to stress that our goal is not to provide just any mapping of WF-nets onto BPEL (cf. Section 4). The mapping should be compact and intuitive. In a way, the mapping should try to select the BPEL constructs an experienced BPEL designer would use. Therefore, we do a kind of "pattern matching", i.e., we scan the WF-net for components that can be mapped onto a suitable structured BPEL activity. The structured activities BPEL relevant for this paper are: `sequence`, `switch` (for conditional routing), `while` (for looping), `pick` (for race conditions based on timing or external triggers), and `flow` (for parallel subprocesses). The `flow` construct can also be used to model sequential and conditional processes. Therefore, it is possible to use the `flow` construct instead of the `sequence` and `switch` constructs. However, as indicated before, we try to use the most compact and intuitive construct. Clearly, a sequence in the WF-net should be mapped onto a `sequence` rather than a `flow`.



(a) *choice net* (b) *while net*

Figure 2. Two subclasses (while net and choice net) inspired by the `switch`, `pick` and `while` in BPEL.

To be able to automate the "pattern matching" required to spot components that can be easily mapped onto a suitable BPEL component, we define three subclasses of Petri nets. The first subclass we define is the *choice net*, i.e., a fragment corresponding to `switch` or `pick` in BPEL.

Definition 15. *A Petri net $PN = (P, T, F)$ is a choice net if and only if P contains two places, say p_1 and p_2, such that $F = \{(p_1, t)|t \in T\} \cup \{(t, p_2)|t \in T\}$.*

In a choice net there is a source place and sink place and an arbitrary number of transitions connecting these places. The second subclass is the *while net*. As the name suggests, this will be mapped onto the BPEL `while` construct. This is the only type of looping BPEL supports.

Definition 16. *A Petri net $PN = (P, T, F)$ is a while net if and only if P contains one place, say p, and T contains three transitions, say t_1, t_2, and t_3, such that $F = \{(t_1, p), (p, t_2), (t_2, p), (p, t_3)\}$.*

Both the while net and choice net are shown in Figure 2. The third subclass is the class of *well-structured* nets. This class is inspired by the class of SWF-nets defined in [12]. Components corresponding to this class will be mapped onto the BPEL `flow` construct.

Definition 17. *A Petri net $PN = (P, T, F)$ is* well-structured *if and only if the following three properties hold:*

- *For all $p \in P$ and $t \in T$ with $(p, t) \in F$: $|p \bullet| > 1$ implies $|\bullet t| = 1$.*
- *For all $p \in P$ and $t \in T$ with $(p, t) \in F$: $|\bullet t| > 1$ implies $|\bullet p| = 1$.*
- *There are no cycles (i.e., for all $x \in P \cup T$: $(x, x) \notin F^*$).*[7]

This class is characterized by Figure 3. This figure shows two constructs not allowed in a well-structured Petri net. The left one corresponds to the requirement that for all $(p, t) \in F$: $|p \bullet| > 1$ implies $|\bullet t| = 1$. This corresponds to a restricted form of *free-choice nets* [17] (cf. Definition 6). The right hand construct corresponds to the requirement that for all $(p, t) \in F$: $|\bullet t| > 1$ implies $|\bullet p| = 1$. This requirement can be seen as complementary to the free-choice requirement. It enforces a fixed set of predecessors that needs to be synchronized. The third requirement of Definition 17 (i.e., no cycles) is not shown graphically. The three requirements are triggered by the limitations of the BPEL `flow` construct, this construct does not allow for loops, non-free choice behavior or a variable set of predecessors that needs to be synchronized.

Using the three classes just defined and standard notions such as *state machines* (Definition 7) and *marked graphs* (Definition 8), we classify different types of components corresponding to the `sequence`, `flow`, `switch`, `pick`, and `while` constructs in BPEL.

Definition 18. *Let $PN = (P, T, F, \tau_P, \tau_G, \tau_{MA}, \tau_T)$ be an annotated WF-net and let C be a component of PN.*

[7] F^* is the transitive closure of F, i.e., $(x, y) \in F^*$ if there is a path from x to y in the net.

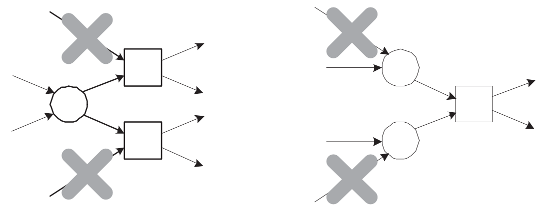


Figure 3. Two constructs not allowed in a well-structured Petri net.

- *C is a SEQUENCE-component if $PN||_C$ is both a state machine, marked graph and not trivial,*
- *C is a maximal SEQUENCE-component if C is a SEQUENCE-component and there is no other SEQUENCE-component C' such that $C \subset C'$,*
- *C is a SWITCH-component if $PN||_C$ is a choice net and $\tau_P(i_C) =$ explicit,*
- *C is a PICK-component if $PN||_C$ is a choice net and $\tau_P(i_C) =$ implicit,*
- *C is a WHILE-component if $PN||_C$ is a while net and $\tau_P(p) =$ explicit (where $p \in P \cap C$),*
- *C is a FLOW-component if $PN||_C$ is well-structured and for all $p \in C \cap dom(\tau_P)$ is $\tau_P(p) = explicit$, and*
- *C is a maximal FLOW-component if C is a FLOW-component and there is no other FLOW-component C' such that $C \subset C'$.*

Before presenting the algorithm to transform an annotated WF-net into a BPEL specification, we illustrate the mapping for each of the components mentioned in Definition 18 using simple examples.

Clearly, the SEQUENCE-component allows for the most straightforward mapping onto BPEL. Since it is both a state machine and marked graph there are no choices and there is no parallelism, i.e., things are executed in a fixed order. Figure 4 shows two examples where a sequence of two transitions is mapped onto a single transition t_c bearing the BPEL. The example on the left is a place-bordered component (a PP-component) while the example on the left is a transition-bordered component (a TT-component). The other two possible cases, i.e., a PT-component or a TP-component, can be mapped in a similar way. Note we assume the initial transitions are mapped onto an `invoke` activity. However, based on the annotation this could be any primitive BPEL activity.

Next we focus on the mapping of a FLOW-component to the BPEL `flow` construct. Figure 5 and 6 shows two examples. The first example shown in Figure 5 depicts a transition bordered component with one initial and one final activity and two parallel activities. The second example in Figure 6 shows a FLOW-component bordered by places. People familiar with the BPEL construct will be able to see that this mapping is indeed correct.

In the two FLOW-component examples we use the shorthands `Any` and `All` for the boolean expression specified in the `joinCondition` of an activity joining multiple links. Assume an activity is target for the links `link1`, ..., `linkN` then the expression `Any` is shorthand for `bpws:getLinkStatus('link1') or ... or`

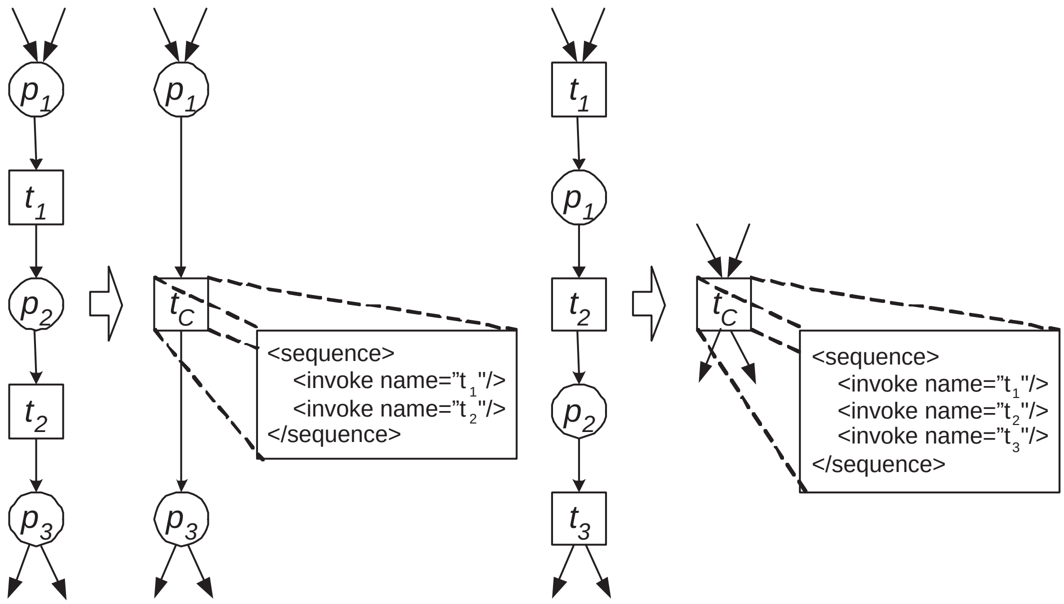


Figure 4. Examples of the SEQUENCE-component and its corresponding BPEL expression.

`bpws:getLinkStatus('linkN')` and the expression `And` is shorthand for `bpws:getLinkStatus('link1') and ... and bpws:getLinkStatus('linkN')`. `Any` correspond to a logical OR and `All` to a logical AND, both of the incoming links.

The two examples shown in Figure 5 and 6 trigger the question of how to map any FLOW-component onto a `flow`? To explain how this works we need to revisit Definition 17. Let T be the set of transitions (i.e., activities) in the component. These are all inserted in the `flow` construct. Then we need to add links, specify the link conditions (if needed), and specify the join condition in case of multiple target links. Let $L \subseteq T \times T$ be the set of links and $join \in T \not\rightarrow \{\text{All}, \text{Any}\}$ the join condition. To correctly map a FLOW-component onto a `flow` activity, $L = \{(t_1, t_2) \in T \times T \mid t_1 \bullet \cap \bullet\, t_2 \neq \emptyset\}$. $dom(join) = \{t \in T \mid \exists_{t_1, t_2 \in T}\ (t_1, t) \in L \ \wedge \ (t_2, t) \in L \ \wedge \ t_1 \neq t_2\}$, i.e., only activities with multiple incoming links have a join condition. Transitions with multiple input places correspond to activities with a join condition set to `All`, i.e., $join(t) = \text{All}$ if $|\bullet\, t| \geq 2$. All other transitions in $dom(join)$ correspond to activities with a join condition set to `Any`. The conditions on the transitions involved in an explicit choice are mapped onto link conditions, i.e., a link $(t_1, t_2) \in L$ has a condition $\tau_G(t_2)$ if and only if $t_2 \in dom(\tau_G)$ (cf. Definition 14).

In the case of a PP- or PT-component of type FLOW where $|i_C \bullet| > 1$ we need to add an additional place and transition before mapping it onto a BPEL `flow`. For a FLOW-component C of type PP or TP with source place i_C add a new place p and a new transition t such that $\bullet t = \{i_C\}$ and $t \bullet = \{p\}$. The output transitions of i_C in the original net are now the output transitions of p. This transformation preserves the WF-net properties of the original C. The reason for this transformation is that i_C in

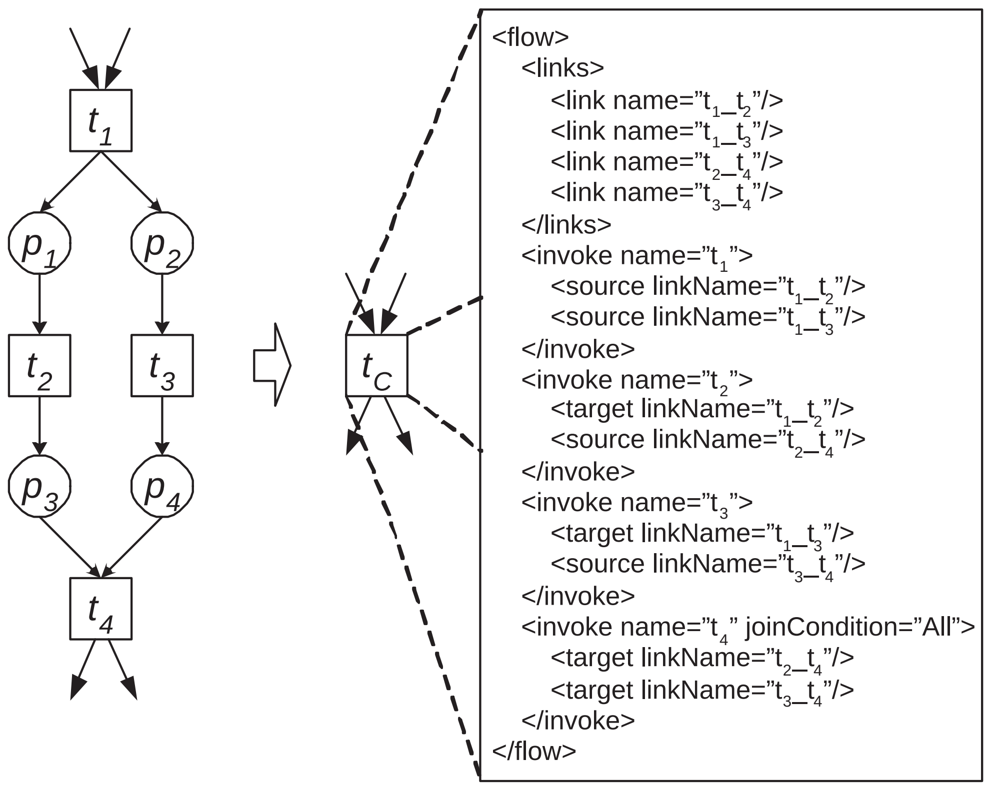


Figure 5. Examples of the FLOW-component and its corresponding BPEL expression.

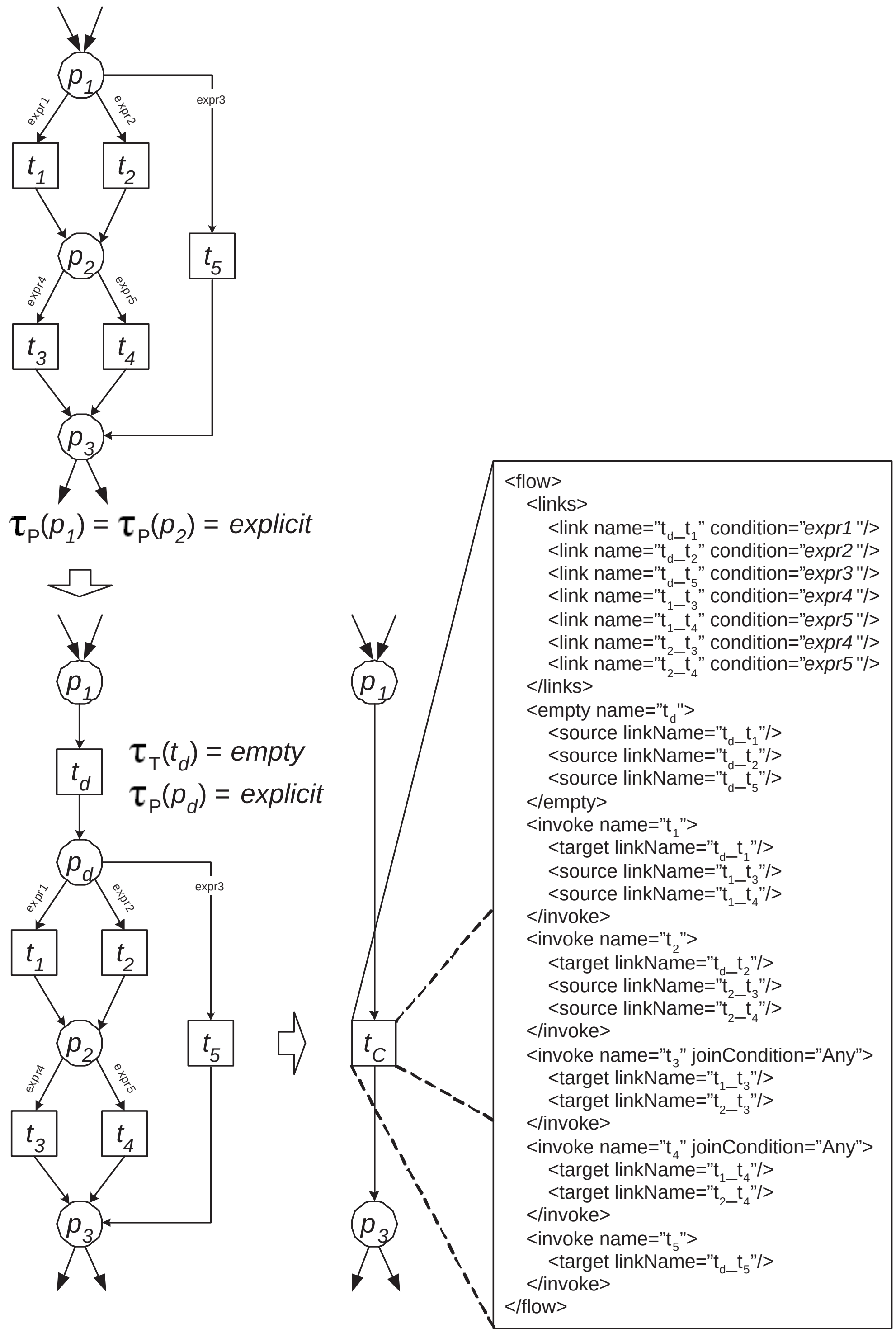


Figure 6. Example of a FLOW-component and its corresponding BPEL expression.

the original C may have had multiple input arcs, but activities translated from transitions in $i_C\bullet$ could not be guarded in the BPEL specification since guards are specified from a source to a target. Since there is no preceding activity from any of the translated transitions in $i_C\bullet$ such guards can not be specified in BPEL. By injecting a transition with the empty annotation, a place and arcs as described previously, we can now guard the activities of the translated transitions of $i_C\bullet$. In Figure 6 we show how this transformation works. Note that the activities t_1, t_2 and t_5 could not be guarded against the expressions of the arcs from i_C if the empty task t_d had not been added.

It is easy to verify that the mapping just described is correct (assuming the FLOW-component is safe and sound, i.e., the conditions of Theorem 3 are satisfied). First all, the graph structure of a FLOW-component is acyclic. Second, a closer inspection of Definition 17 shows that if there is a place p connecting t_1 to t_2 (i.e., $(t_1, t_2) \in L$), then p is the only place connecting t_1 to t_2, and it is the only input place of t_2 or $|\bullet p| = |p\bullet| = 1$. If t_2 has multiple input places, then the join condition is set to `All`. This is correct since in the FLOW-component all preceding activities (i.e., $\bullet(\bullet t_2)$) need to complete because $|\bullet p| = |p\bullet| = 1$. If t_2 has only one input place p, but p has multiple output transitions (i.e., $|\bullet p| \geq 2$), then the join condition is set to `Any`. This is correct since in the FLOW-component only one of the preceding activities needs to complete. Note that it is important that the FLOW-component is safe and sound. As a result precisely one of the preceding activities will and can complete. It is interesting to see that the class of well-structured Petri net can be mapped onto the `flow` construct in BPEL. However, we will only advocate the mapping of such FLOW-components onto the `flow` construct if it is not possible to map it onto one of the other structured activities (e.g., `sequence` or `switch`).

Figure 7 illustrates the mapping of choice nets, i.e., a SWITCH-component is mapped onto a `switch` and a PICK-component is mapped onto a `pick`. Given the above, the mapping is fairly straightforward. In a choice net there is one source place and one sink place. These are connected by transitions each representing an alternative activity. If the nature of the choice is explicit (i.e., a SWITCH-component), the net fragment is mapped onto a `switch` where $\tau_G(t)$ is the condition of the case corresponding to transition t. If the nature of the choice is implicit (i.e., a PICK-component), the net fragment is mapped onto a `pick` construct where τ_{MA} specifies the messages, timeouts and deadlines for the PICK-component (if present). In Figure 7 there are two possible messages that may arrive (`msg1` and `msg2`) and there is one `onAlarm` with a deadline (`dl1`) and timeout (`to1`).

Finally, we show the mapping of the WHILE-component onto the BPEL `while` construct (cf. Figure 8). Note that the real loop is only formed by place p and transition t_2. However, to model the entry and exit of the loop we need to consider t_1 and t_3. This is reflected in the BPEL translation. The `while` construct is embedded in a `sequence`. The guards of t_2 and t_3 (i.e., $\tau_G(t_2) \wedge \neg\tau_G(t_3)$) are in the condition used in the `while` construct. (Note that the $\neg\tau_G(t_3)$ part is not needed if the guards, i.e., $\tau_G(t_2)$ and $\tau_G(t_3)$, are indeed mutually exclusive.)

After showing the mapping of each of the components mentioned in Definition 18, we can present the algorithm to translate an annotated WF-net onto BPEL. The basic idea of the algorithm is to take a component, provide the BPEL translation, and fold

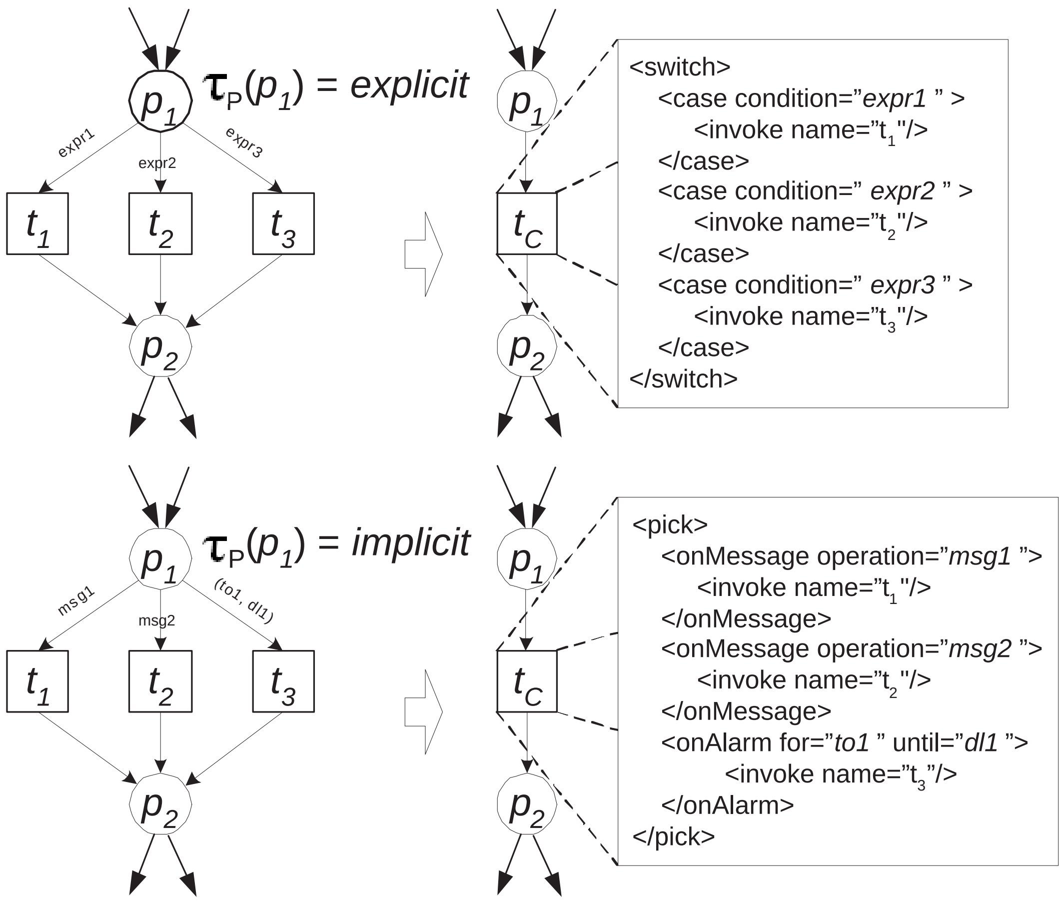


Figure 7. Examples of the SWITCH-component and PICK-component and their corresponding BPEL expression.

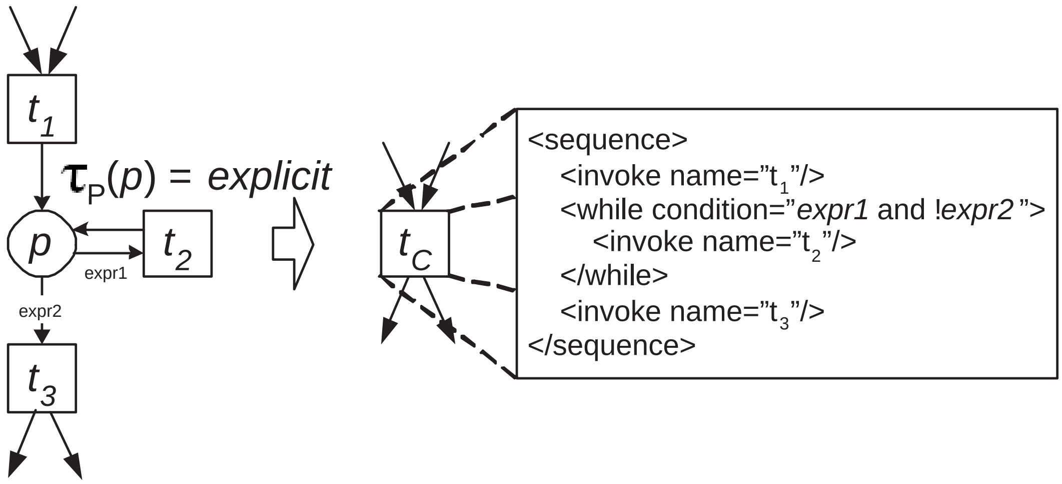


Figure 8. Example of the WHILE-component and its corresponding BPEL expression.

the net. This is repeated until a WF-net with just one transition is obtained. Since we only reduce standard components (sequence etc.) as defined in this paper, we leave the algorithm extensible so that it is possible to adapt the algorithm for translation of more elaborate components that can be found in WF-nets.

Definition 19 (Algorithm). *Let* $PN = (P, T, F, \tau_P, \tau_G, \tau_{MA}, \tau_T)$ *be a safe and sound annotated WF-net.*

(i) $X := PN$
(ii) while $[X] \neq \emptyset$ *(i.e.,* X *contains a non-trivial component)*[8]
(iii-a) If there is a maximal SEQUENCE-component $C \in [X]$*, select it and go to (vi).*
(iii-b) If there is a SWITCH-component $C \in [X]$*, select it and go to (vi).*
(iii-c) If there is a PICK-component $C \in [X]$*, select it and go to (vi).*
(iii-d) If there is a WHILE-component $C \in [X]$*, select it and go to (vi).*
(iii-e) If there is a maximal FLOW-component $C \in [X]$*, select it and go to (vi).*
(iv) If there is a component $C \in [X]$ *that appears in the component library, select it and go to (vi).*
(v) Select a component $C \in [X]$ *to be manually mapped onto BPEL and add it to the component library.*
(vi) Attach the BPEL translation of C *to* t_C *as illustrated in Figure 1.*
(vii) $X := fold(PN, C)$ *and return to (ii).*
(viii) Output the BPEL code attached to the transition in X*.*

The actual translation of components is done in step *(vi)* followed by the folding in step *(vii)*. The component to be translated/folded is selected in steps *(iii)*. If there is still a sequence remaining in the net, this is selected. A maximal sequence is selected to keep

[8] Note that this is the case as long as X is not reduced to a WF-net with just a single transition.

the translation as compact and simple as possible. Only if there are no sequences left in the WF-net, other components are considered. The next one in line is the SWITCH-component followed by the PICK-component and the WHILE-component. Given the fact that SWITCH-, PICK- and WHILE-components are disjoint, the order of steps *(iii-b)*, *(iii-c)*, and *(iii-d)* is irrelevant. Finally, maximal FLOW-components are considered.

Not every net can be reduced into SEQUENCE-, SWITCH-, PICK-, WHILE- and FLOW-components. (We will show an example to illustrate this later in this section.) Therefore, steps *(iv)* and *(v)* have been added. The basic idea is to allow ad-hoc translations. These translations are stored in a component library. If the WF-net cannot be reduced any further using the standard SEQUENCE-, SWITCH-, PICK-, WHILE- and FLOW-components, the algorithm searches the component library (note that it only has to consider the network structure and not the specific names and annotations). If the search is successful, the stored BPEL mapping can be applied (modulo renaming of nodes and annotations). If there is not a matching component, a manual translation can be provided and stored in the component library for future use.

Note that the algorithm described in Definition 19 uses function *fold* defined in Section 4. We will not give a formal proof of the correctness of the algorithm. However, the results in Section 4 (Theorem 3 in particular) provide insight into the correctness of the approach. Note that we do not give a formal proof because manageable formal semantics of BPEL are missing (cf. discussion in Section 4).

The appendix describes an elaborate example illustrating the approach. It shows how in a step-by-step fashion a complex WF-net can be transformed into a BPEL specification. Note that this example also illustrates steps *(iv)* and *(v)* by presenting a component which cannot be reduced using the standard rules. See more in [10].

6 Implementation: WorkflowNet2BPEL4WS

To support the approach reported in the previous section, we developed a tool called *WorkflowNet2BPEL4WS*. The tool is written in Java and takes as input a Colored Petri net created using CPN Tools [15]. CPN Tools support Petri nets with data, time, and hierarchy, also known as *Colored Petri Nets* (CPNs). To implement the algorithm that we have presented we ignore all reference to the data, time, and hierarchy parts of the net and look only at the structure of the net. This means that the nets that the tool looks at are Petri nets, since CPNs without these parts are essentially Petri nets. The reason why we chose the export format of CPN Tools as the input format, and not e.g. PNML [37, 14], is that one of the authors of this paper have been part of the implementation of CPN Tools and is therefore very familiar with the format. After parsing the input file, the tool translates the Petri net model using the following steps:

1. The net is checked for conformance to Definition 9.
2. The net is reduced using the algorithm described in Definition 19, with the exception that step (iv) and (v) results in an exception (i.e. the program halts). (The possibility of creating and/or selecting components in a library is not yet implemented.)
3. The BPEL annotation of the transition in the reduced net, the trivial net, is written in a file and can be loaded into a variety of BPEL tools.

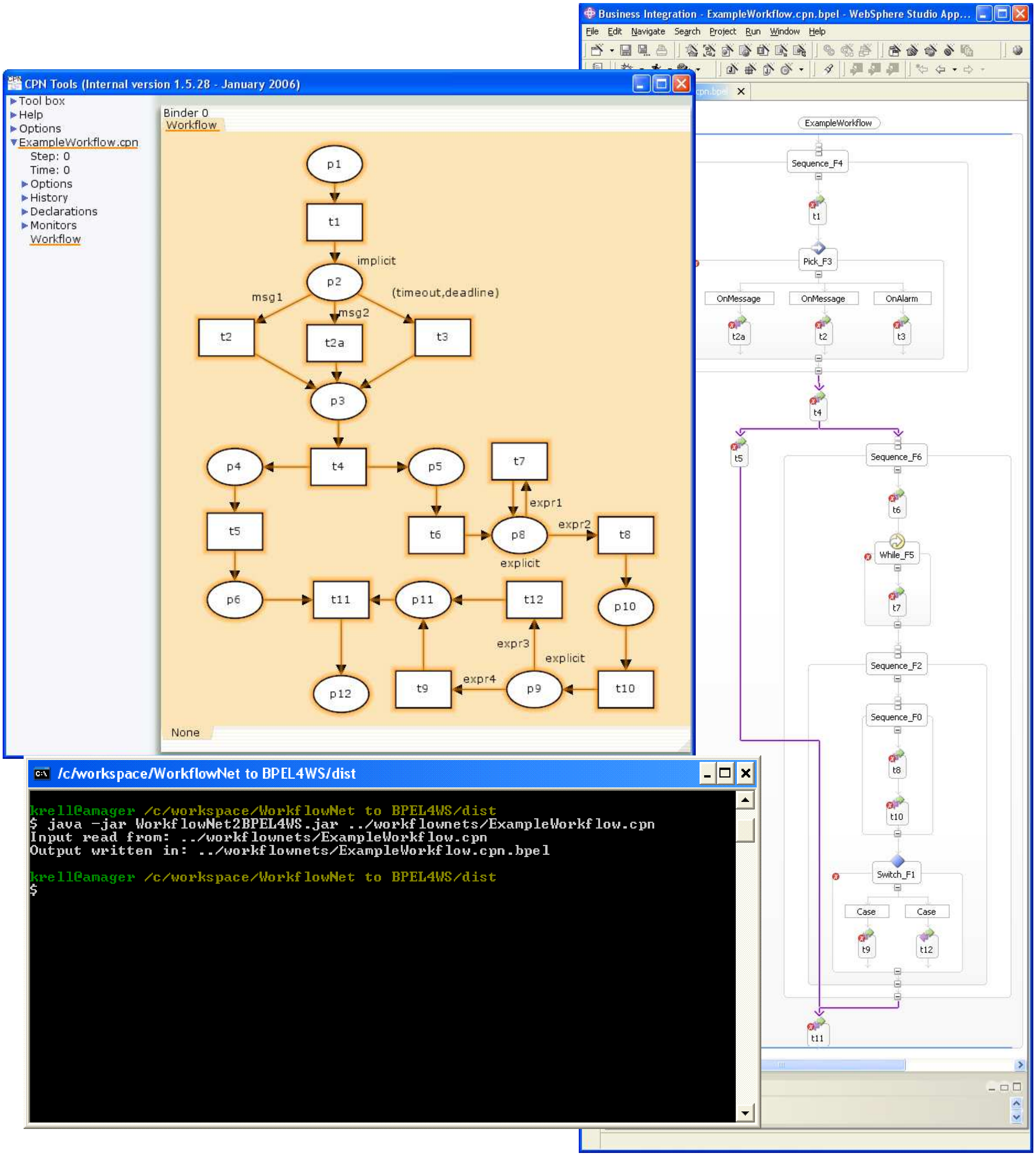


Figure 9. WorkflowNet2BPEL4WS allows for the translation of CPN abstractions (i.e., WF-nets) to BPEL template code.

In Figure 9 we see how the implementation is used. The upper left part shows a screenshot of CPN Tools while displaying a process model that can be mapped onto a WF-net by abstraction, i.e., there are no color sets defined, but we do annotate the WF-net; we have added information about the nature of a choice and expression/messages on arcs from explicit/implicit choices. Figure 9 also shows the command prompt where WorkflowNet2BPEL4WS is started. Finally, to the right we have the result of the translation, as shown in WebSphere Studio [68]. Note all the small red error balls scattered across the process. These are places where the programmer has to define things such as partner links, variables, port types, correlations sets, etc. to make the process valid. This shows that one should really think of the BPEL code that is generated as template code. Therefore, it is realistic to abstract from data, time, and hierarchy in CPN Tools.

WorkflowNet2BPEL4WS and the instruction on how to use it, can be downloaded from `http://www.daimi.au.dk/~krell/WorkflowNet2BPEL4WS.zip`. CPN Tools can be downloaded from `http://wiki.daimi.au.dk/cpntools/`

7 Case Study: Generating BPEL template code for a Bank System

To illustrate our approach (and the possible use of WorkflowNet2BPEL4WS), we describe a case study where *Colored Petri Nets* (CPNs) are used in the development of a bank system [9]. The bank system is named the *Adviser Portal (AP)* and is being developed by Bankdata (a Danish company providing software solutions for banks). AP has been bought by 15 Danish banks and will be used by thousands of bank advisers in hundreds of bank branches. Its main goal is to increase the efficiency and quality of bank advisers' work. In the context of the development of AP we mapped Petri nets onto BPEL using the algorithm presented in this paper.

In [9] it is shown that one can use a two step approach to go from a requirements model to an implementation of the new system. The initial requirements model is presented as an executable CPN [36] whose sole purpose is to specify, validate, and elicit user requirements (independent of the target language). In the first translation step, a *workflow model* is derived from the requirements model. This model is represented in terms of a so-called *Colored Workflow Net (CWN)*, which is a generalization of the classical workflow nets to CPN. In the second translation step, the CWN is translated into implementation code by first mapping the CWN model onto a WF-net and then using our algorithm to translate the WF-net onto BPEL.

The focus of [9] is on the case study and the two step approach rather than on the mapping. In fact the mapping of Petri nets to BPEL plays only a minor role in the paper and only the result is mentioned. Therefore, we do not elaborate on details of the case study here and focus on the actual mapping from WF-net onto BPEL. We also do not show the requirements CPN model or the CWN model.

Figure 10 shows the resulting WF-net which is a result of a well-defined procedure to map a CWN onto BPEL [9].

Using the algorithm presented in Section 5, we can automatically generate the BPEL code. Again we first look for maximal SEQUENCE-components. Figure 10 shows the maximal SEQUENCE-components. Each of these components is folded into a single

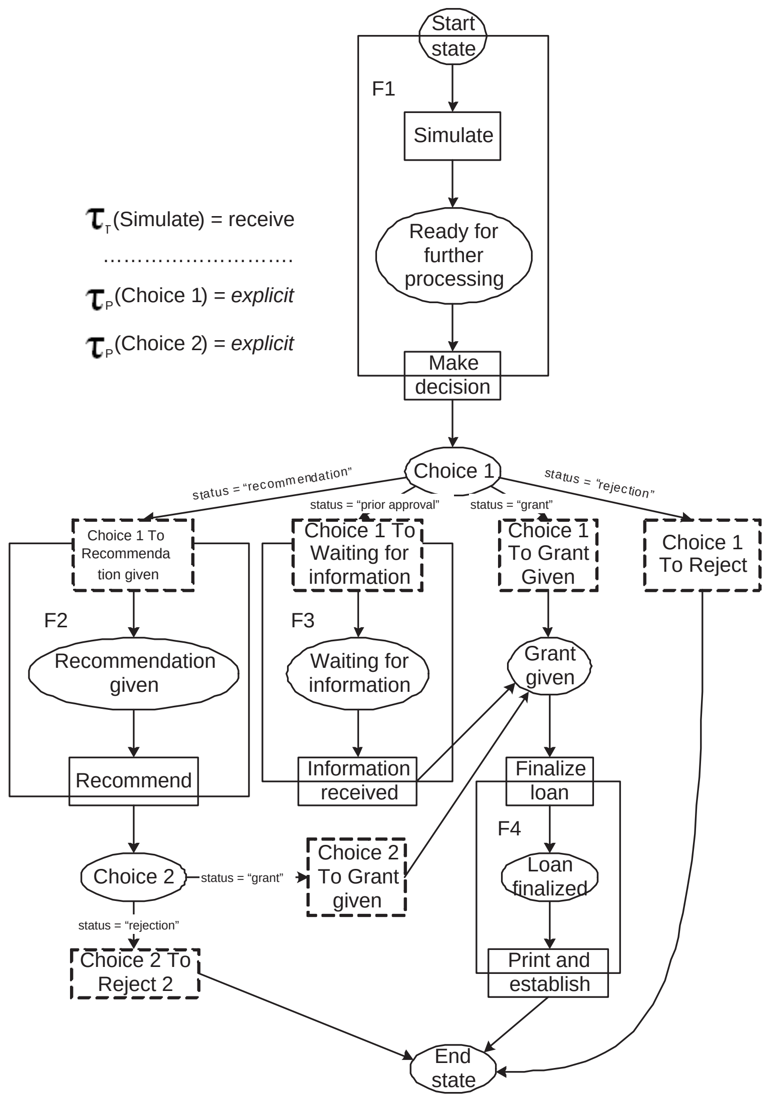


Figure 10. WF model of a blanc loan process.

transition labeled with the appropriate BPEL code (i.e., a `sequence` activity). The snippets of BPEL code can be seen in listings 1, 2, 3 and 4.

Listing 1. Fragment F1

```
<sequence>
  <receive name="Receive"/>
  <invoke name="Simulate"/>
  <invoke name="MakeDecision"/>
</sequence>
```

Listing 2. Fragment F2

```
<sequence>
  <empty name="Choice_1_To_Recommendation_given"/>
  <invoke name="Recommend"/>
</sequence>
```

Listing 3. Fragment F3

```
<sequence>
  <empty name="Choice_1_To_Waiting_for_information"/>
  <invoke name="Get information"/>
</sequence>
```

Listing 4. Fragment F4

```
<sequence>
  <invoke name="Finalize_loan"/>
  <invoke name="Print_and_Establish"/>
</sequence>
```

The WF-net after folding the SEQUENCE-components is well-structured, i.e., the whole net is a maximal FLOW-component and can be represented using the BPEL `flow` construct. The resulting BPEL specification is shown in Listing 5.

Listing 5. Complete BPEL specification of case study example

```
<flow>
  <links>
```

```
    <link name="F1_F2"/>
    <link name="F1_F3"/>
    <link name="F3_F4"/>
    <link name="F1_Choice_1_To_Refusal"/>
    <link name="F1_Choice_1_To_Grant_given"/>
    <link name="Choice_1_To_Grant_given_F4"/>
    <link name="F2_Choice_2_To_Refusal_2"/>
    <link name="F2_Choice_2_To_Grant_given"/>
    <link name="Choice_2_To_Grant_given_F4"/>
  </links>
  <sequence name="F1">
    <source linkName="F1_Choice_1_To_Grant_given"/>
    <source linkName="F1_F2"/>
    <source linkName="F1_F3"/>
    <source linkName="F1_Choice_1_To_Refusal"/>
    <<F1>>
  </sequence>
  <sequence name="F2">
    <target linkName="F1_F2"/>
    <source linkName="F2_Choice_2_To_Grant_given"/>
    <source linkName="F2_Choice_2_To_Refusal 2"/>
    <<F2>>
  </sequence>
  <sequence name="F3">
    <target linkName="F1_F3"/>
    <source linkName="F3_F4"/>
    <<F3>>
  </sequence>
  <sequence name="F4" joinCondition="Any">
    <target linkName="Choice_2_To_Grant_given_F4"/>
    <target linkName="Choice_1_To_Grant_given_F1"/>
    <target linkName="F3_F4"/>
    <<F4>>
  </sequence>
  <empty name="Choice_1_To_Refusal">
    <target linkName="F1_Choice_1_To_Refusal"/>
  </empty>
  <empty name="Choice_1_To_Grant_given">
    <target linkName="F1_Choice 1 To Grant given"/>
    <source linkName="Choice_1_To_Grant_given_F4"/>
  </empty>
  <empty name="Choice_2_To_Refusal_2">
    <target linkName="F2_Choice_2_To_Refusal_2"/>
  </empty>
  <empty name="Choice_2_To_Grant_given">
```

```
    <target linkName="F2_Choice_2_To_Grant_given"/>
    <source linkName="Choice_2_To_Grant_given_F4"/>
  </empty>
</flow>
```

In this section, we have used a real-life example (the new AP system of Bankdata) to illustrate the algorithm presented in Section 5. The resulting BPEL code has been used to implement the process in IBM WebSphere. The use of CPN models (both the requirements CPN and the CWN) was supported by CPN Tools, i.e., the models have been simulated and end-users could interact with the model through the animation facilities of CPN Tools. Using a CWN rather than an arbitrary CPN enables the automatic generation of template BPEL code, since a well-defined procedure exists to translate a CWN to a WF-net [9], which we can map onto BPEL using our translation. Therefore, we described in [9] an approach to migrate a requirements CPN to a CWN. However, [9] does not describe the translation to BPEL code in any detail.

In this section we provided small but realistic example showing that our approach and tools can indeed be used to generate readable BPEL. For another example of how the algorithm works we refer to a technical report [10] describing the translation process step-by-step.

8 Conclusion

In this paper we presented an algorithm to generate BPEL specifications from WF-nets. While most researchers have been working on translations from BPEL to formal models like Petri nets, we argued that a translation from Petri nets to BPEL is probably more relevant. Designers prefer a graphical language without all kinds of syntactical restrictions. However, the graphical editors of systems supporting BPEL tend to directly visualize the structure of the BPEL code. Therefore, it is not possible to have arbitrary splits and joins, loops, etc. WF-nets, a subclass of Petri nets, do not have these restrictions and therefore the mapping is relevant and challenging. The goal of our translation is to generate compact and readable BPEL template code, i.e., we carefully try to discover patterns in the WF-nets that fit well with specific BPEL constructs. This way the BPEL specification remains readable and maintainable.

Using a small case study (the blanc loan process to be supported by the AP system of Bankdata) we showed the applicability of our approach. Here we used a “colored” variant of WF-nets (adding the data and resource perspectives to the control perspective) named Colored Workflow Nets (CWN). CPN Tools allows for the definition, execution, and analysis of such models. Using the algorithm presented in this paper, we translated the CWN into BPEL code and the result has been implemented using IBM WebSphere. Although we implement a tool that can read models from CPN Tools, our approach is not limited to (colored) Petri nets. Good candidates for applying the mapping are UML activity diagrams [32], Event-driven Process Chains (EPCs) [38, 57], the Business Process Modeling Notation (BPMN) [66]. In fact, we already successfully applied the ideas presented in this paper to BPMN. Also our approach is not limited to BPEL as a target language. Note that the “patterns” represented by SEQUENCE-, SWITCH-, PICK-, WHILE- and FLOW-components also exist in many other graphical languages. Hence it is relatively easy to provide a mapping from these languages to BPEL using a variant of the algorithm presented in this paper. Similarly, elements of our approach can be used for other target languages. Like BPEL, most workflow management systems use languages imposing all kind of restrictions on the structure of the process model.

References

1. W.M.P. van der Aalst. Verification of Workflow Nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer-Verlag, Berlin, 1997.
2. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
3. W.M.P. van der Aalst. Workflow Verification: Finding Control-Flow Errors using Petri-net-based Techniques. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer-Verlag, Berlin, 2000.
4. W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer-Verlag, Berlin, 2004.
5. W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. *Theoretical Computer Science*, 270(1-2):125–203, 2002.
6. W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Web Service Composition Languages: Old Wine in New Bottles? In G. Chroust and C. Hofer, editors, *Proceeding of the 29th EUROMICRO Conference: New Waves in System Architecture*, pages 298–305. IEEE Computer Society, Los Alamitos, CA, 2003.
7. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
8. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
9. 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 Paper. 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, Berlin, 2005.
10. 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.
11. W.M.P. van der Aalst, K.M. van Hee, and R.A. van der Toorn. Component-Based Software Architectures: A Framework Based on Inheritance of Behavior. *Science of Computer Programming*, 42(2-3):129–171, 2002.
12. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
13. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, 2003.

14. J. Billington and et. al. The Petri Net Markup Language: Concepts, Technology, and Tools. In W.M.P. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–506. Springer-Verlag, Berlin, 2003.
15. CPN Group, University of Aarhus, Denmark. CPN Tools Home Page. http://wiki.daimi.au.dk/cpntools/.
16. J. Dehnert. *A Methodology for Workflow Modeling: From Business Process Modeling Towards Sound Workflow Specification*. PhD thesis, TU Berlin, Berlin, Germany, 2003.
17. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.
18. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley & Sons, 2005.
19. R. Eshuis and J. Dehnert. Reactive Petri nets for Workflow Modeling. In W.M.P. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 295–314. Springer-Verlag, Berlin, 2003.
20. D. Fahland and W. Reisig. ASM-based semantics for BPEL: The negative control flow. In D. Beauquier and E. Börger and A. Slissenko, editor, *Proc. 12th International Workshop on Abstract State Machines*, pages 131–151, Paris, France, March 2005.
21. R. Farahbod, U. Glässer, and M. Vajihollahi. Abstract Operational Semantics of the Business Process Execution Language for Web Services. Technical Report SFU-CMPT-TR-2004-03, School of Computer Science, Simon Fraser University, Burnaby B.C. Canada, April 2004.
22. R. Farahbod, U. Glässer, and M. Vajihollahi. Specification and validation of the business process execution language for web services. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 79–94, Lutherstadt Wittenberg, Germany, May 2004. Springer-Verlag, Berlin.
23. A. Ferrara. Web services: A process algebra approach. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM Press.
24. L. Fischer, editor. *Workflow Handbook 2003, Workflow Management Coalition*. Future Strategies, Lighthouse Point, Florida, 2003.
25. J.A. Fisteus, L.S. Fernández, and C.D. Kloos. Formal verification of BPEL4WS business collaborations. In K. Bauknecht, M. Bichler, and B. Proll, editors, *Proceedings of the 5th International Conference on Electronic Commerce and Web Technologies (EC-Web '04)*, volume 3182 of *Lecture Notes in Computer Science*, pages 79–94, Zaragoza, Spain, August 2004. Springer-Verlag, Berlin.
26. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Composition. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 152–161, Montreal, Canada, October 2003.
27. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Tool Support for Model-Based Engineering of Web Service Compositions. In *Proceedings of 2005 IEEE International Conference on Web Services (ICWS 2005)*, pages 95–102, Orlando, FL, USA, July 2005. IEEE Computer Society.
28. X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *International World Wide Web Conference: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM Press.
29. X. Fu, T. Bultan, and J. Su. WSAT: A tool for formal analysis of Web services. In *Proceedings of 16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 510–514. Springer-Verlag, Berlin, 2004.
30. D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

31. R.J. van Glabbeek and D.G. Stork. Query Nets: Interacting Workflow Modules that Ensure Global Termination. In W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske, editors, *International Conference on Business Process Management (BPM 2003)*, volume 2678 of *Lecture Notes in Computer Science*, pages 184–199. Springer-Verlag, Berlin, 2003.
32. Object Management Group. *OMG Unified Modeling Language 2.0 Proposal, Revised submission to OMG RFPs ad/00-09-01 and ad/00-09-02, Version 0.671*. OMG, http://www.omg.com/uml/, 2002.
33. K. van Hee, N. Sidorova, and M. Voorhoeve. Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In W.M.P. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 335–354. Springer-Verlag, Berlin, 2003.
34. S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske, editors, *International Conference on Business Process Management (BPM 2003)*, volume 2678 of *Lecture Notes in Computer Science*, pages 220–235. Springer-Verlag, Berlin, 2003.
35. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
36. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
37. M. Jungel, E. Kindler, and M. Weber. The Petri Net Markup Language. In S. Philippi, editor, *Proceedings of AWPN 2000 - 7thWorkshop Algorithmen und Werkzeuge für Petrinetze*, pages 47–52. Research Report 7/2000, Institute for Computer Science, University of Koblenz, Germany, 2000.
38. G. Keller, M. Nüttgens, and A.W. Scheer. Semantische Processmodellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, 1992.
39. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2003. Available via http://www.workflowpatterns.com.
40. B. Kiepuszewski, A.H.M. ter Hofstede, and W.M.P. van der Aalst. Fundamentals of Control Flow in Workflows. *Acta Informatica*, 39(3):143–209, 2003.
41. E. Kindler and W.M.P. van der Aalst. Liveness, Fairness, and Recurrence. *Information Processing Letters*, 70(6):269–274, June 1999.
42. E. Kindler, A. Martens, and W. Reisig. Inter-Operability of Workflow Applications: Local Criteria for Global Soundness. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 235–253. Springer-Verlag, Berlin, 2000.
43. 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.
44. M. Koshkina and F. van Breugel. Verification of Business Processes for Web Services. Technical report CS-2003-11, York University, October 2003. Available from: http://www.cs.yorku.ca/techreports/2003/.
45. F. Leymann. Web Services Flow Language, Version 1.0, 2001.
46. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.

47. N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing Interacting BPEL Processes. In S. Dustdar, J.L. Faideiro, and A. Sheth, editors, *International Conference on Business Process Management (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, Berlin, 2006.
48. A. Martens. On Compatibility of Web Services. *Petri Net Newsletter*, 65:12–20, 2003.
49. A. Martens. *Verteilte Geschäftsprozesse - Modellierung und Verifikation mit Hilfe von Web Services (In German)*. PhD thesis, Institut für Informatik, Humboldt-Universität zu Berlin, Berlin, Germany, 2003.
50. A. Martens. Analyzing Web Service Based Business Processes. In M. Cerioli, editor, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *Lecture Notes in Computer Science*, pages 19–33. Springer-Verlag, Berlin, 2005.
51. M. zur Muehlen. *Workflow-based Process Controlling: Foundation, Design and Application of workflow-driven Process Information Systems*. Logos, Berlin, 2004.
52. C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, and H.M.W. Verbeek. Formal Semantics and Analysis of Control Flow in WS-BPEL. BPM Center Report BPM-05-15, BPMcenter.org, 2005.
53. C. Ouyang, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. From BPMN Process Models to BPEL Web Services. In *Proceedings of the 4th International Conference on Web Services (ICWS), Chicago IL, USA*, pages 285–292. IEEE Computer Society, September 2006.
54. C. Ouyang, M. Dumas, S. Breutel, and A.H.M. ter Hofstede. Translating Standard Process Models to BPEL. BPM Center Report BPM-05-27, BPMcenter.org, 2005.
55. C. Ouyang, E. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, and A.H.M. ter Hofstede. WofBPEL: A Tool for Automated Analysis of BPEL Processes. In B. Benatallah, F. Casati, and P. Traverso, editors, *Proceedings of Service-Oriented Computing (ICSOC 2005)*, volume 3826 of *Lecture Notes in Computer Science*, pages 484–489. Springer-Verlag, Berlin, 2005.
56. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.
57. A.W. Scheer. *ARIS: Business Process Modelling*. Springer-Verlag, Berlin, 2000.
58. C. Stahl. Transformation von BPEL4WS in Petrinetze (In German). Master's thesis, Humboldt University, Berlin, Germany, 2004.
59. C. Stahl. A Petri net semantics for BPEL. Technical Report 188, Humboldt-Universität zu Berlin, June 2005.
60. S. Thatte. XLANG Web Services for Business Process Design, 2001.
61. R. van der Toorn. *Component-Based Software Design with Petri nets: An Approach Based on Inheritance of Behavior*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2004.
62. H.M.W. Verbeek and W.M.P. van der Aalst. Analyzing BPEL Processes using Petri Nets. In D. Marinescu, editor, *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 59–78. Florida International University, Miami, Florida, USA, 2005.
63. H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.
64. WFMC. Workflow Management Coalition Workflow Standard: Workflow Process Definition Interface – XML Process Definition Language (XPDL) (WFMC-TC-1025). Technical report, Workflow Management Coalition, Lighthouse Point, Florida, USA, 2002.
65. S. White. Using BPMN to Model a BPEL Process. *BPTrends*, 3(3):1–18, March 2005.
66. S.A. White et al. Business Process Modeling Notation (BPML), Version 1.0, 2004.

67. P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In I.Y. Song, S.W. Liddle, T.W. Ling, and P. Scheuermann, editors, *22nd International Conference on Conceptual Modeling (ER 2003)*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215. Springer-Verlag, Berlin, 2003.
68. IBM WebSphere Studio. www-306.ibm.com/software/integration/wsadie/.