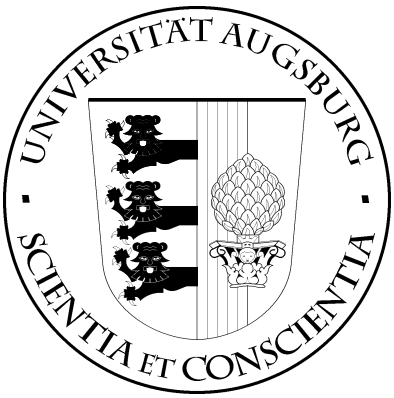
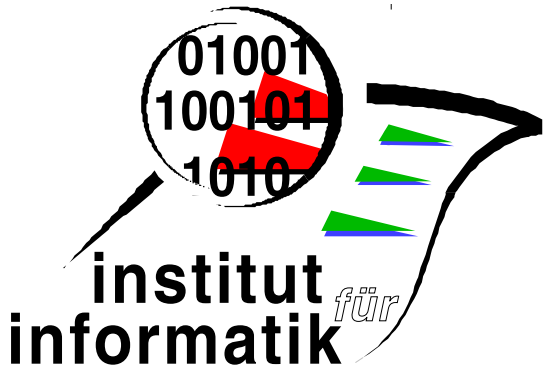
Universität Augsburg



Using Token Analysis to Transform Graph-Oriented Process Models to BPEL

Götz, Roser, Lautenbacher & Bauer

Report 2008-08 Juni 2008



Institut für Informatik

D-86135 Augsburg

In Business Process Management, graph-based models facilitate convenient process modelling. Current workflow engines are commonly based on mainly block-structured languages, such as WS-BPEL, that differ structurally and semantically from process graphs. Recent work has accomplished elaborate mappings between both representations. Although most mappings strongly depend on the segmentation of the graph-model into components, the necessary graph-decomposition itself is not described in these works. This article presents an approach based on *Token Analysis* to automatically identify components. The technique enables simple integration of further improvements steps in the translation of process graphs to executable workflows and yields general results in graph-theory, that might also be of interest in related fields, such as workflow analysis for compilers and multi-threaded processors.

Contents

1 Introduction

Graph-based models – commonly expressed in the *Business Process Modelling Notation* (BPMN, OMG (2006)) – enable intuitive modelling of processes: they are readily understandable without the need for learning abstract programming languages. A process is described as a state-transition system by vertex elements representing activities, events, gateways for decisions or parallel execution, and the connecting flow between them, represented by arcs. Implementation concerns are covered in-depth in Jablonski & Bussler (1996). The expressive power of graph-based languages, however, also has some disadvantages. Graphs can be inconsistent or have no defined meaning. The integrity of the model cannot be ensured on a syntactic level, as done in imperative programming, but requires further analysis.

Current workflow engines are very often based on imperative block-structured languages. The *Business Process Execution Language for Web Services* (WS-BPEL, OASIS (2003)) is a block-oriented language, but also includes graph-based control links. Language elements, like *invoke*, *sequence* or *while*, are composed in a tree-structure that differs essentially from arbitrary, possibly cyclic graphs; while providing better control over semantics and misuse of syntax, the block structure requires experience in programming. This is not suitable for business analysts and designers.

Recker & Mendling (2006) discusses the conceptual discrepancy between BPMN and BPEL. Originating from different backgrounds, the languages differ in their semantic expressiveness because of the different paradigms in technical and business analysis. Second, BPMN is relevant at an early stage of BPM design, and BPEL in the late stage of execution.

To close the gap between ease of description and execution, recent work presents transformation strategies. Mendling et al. (2006) and Ouyang, Dumas, ter Hofstede & van der Aalst (2006*a*) provide mappings from sub-graphs of BPMN models – henceforth called *components*[1] – to equivalent BPEL code. Automatic translation requires to determine all components and chose the best mapping strategies for them. For large scaled input, a structured and efficient approach is needed.

OMG (2006) informally outlines in its BPMN specification a translation from BPMN to BPEL based on a technique called *Token Analysis*. Conceptual tokens flowing along the arcs

[1] In some fields of Compiler Theory and Parallel Programming, these subgraphs are referred to as *hammock graphs*, see Sect. 2.2

are used to identify the boundaries of structured components which are then mapped to BPEL elements. The method depends on manual interaction for the identification of the structured components, so it is not suitable for an automatic export filter.

Existing techniques for automatic graph decomposition into components from related fields, in workflow analysis for compilers and multi-threaded processor design, are complex and not well adjusted to producing BPEL code. The objective of this work is to develop the basic Token Analysis idea into a convenient segmentation algorithm, customised to business process diagrams (BPDs, see OMG (2006)), that does not require human input.

This aims at building a bridge between the ideas proposed in the articles on Token Flow by OMG (2006) and the mapping routines by Ouyang, Dumas, ter Hofstede & van der Aalst (2006*a*). From the given informal description of the Token concept, we derive a machine executable algorithm. This involves investigating cyclic graph structures and their meaning to the analysis.

The overall result is an automatic and efficient algorithm for the segmentation of (process) graphs into components. The results further improve the translation procedure: we introduce a new method called *partial token convergence* to detect additional components that only become available through gateways splitting. This enhances the readability of the produced code and is one major advantage of the algorithm introduced in this work, that is currently not supported by related techniques.

The benefits of the extended Token Analysis approach are:

- it works for arbitrary graphs,
- improvements in the translations procedure,
- enhanced readability of generated code.

Furthermore, our implementation of the Token Analysis algorithm, including automatic mapping to BPEL code, has proven that the algorithm successfully integrates in the overall transformation. The developed Java framework can be used to enhance existing modelling tools with BPEL export. Furthermore, the algorithm and theoretical results convey insights into cyclic structures and their reduction to acyclic flow.

This article is structured as follows: Chapter 2 gives an overview of transformation strategies from graph to block structured models. The extended token analysis algorithm in Chapter 3 is presented as pseudo code and backed up by a proof-of-concept implementation, which is applied to an example graph in Chapter 4. Finally, Chapter 5 concludes and discusses future work.

2 Background and Related Work

Recent work explores transformation strategies from graph structured process models to block structure. The mapping concepts involve a trade-off between *readability* of the output code and *completeness*, that is the ability to transform arbitrary input models.

The mapping approaches described in 2.1.1 and 2.1.2 are the most specific to BPEL. 2.1.3 introduces the Token Flow idea. Sect. 2.2 lists some alternate graph segmentations procedures from compiler construction and parallel programming. Finally, 2.3 tries to capture the state of BPEL export in current workflow modelling tools.

2.1 BPEL Related Mappings

The following mapping strategies are customised to produce BPEL code.

2.1.1 Mendling et al.

A *process graph PG* is represented in an EPC-like notation, consisting of a directed graph that has special vertex types (called functions and connectors). Depending on the type, there are restrictions on input and output cardinalities, and some connector types have guard expressions. For the transformation from PGs to BPEL we are concerned with here,

Mendling et al. (2006) uses an *annotated graph*, that is annotations at special vertices that refer to portions of successively translated BPEL code. In succeeding steps, subgraphs translate to BPEL. The article describes four different mapping strategies: *Element-Preservation* translates the whole graph to one BPEL <flow> activity. The graph must be acyclic because of BPEL's dead path elimination mechanism. Activities and start events map to BPEL basic activities, connectors (gateways) map to empty activities. Flow is uniformly modeled by control-links. This strategy produces less readable code and many undesired nodes which are reduced by *Element-Minimization*. Superfluous empty activities created by the former method are replaced by direct links. *Structure-Identification* iteratively identifies and translates structured sub-graphs of the process graph to BPEL code and replaces sub-graphs with annotated vertices, carrying the corresponding translation. Structured sub-graphs have

a direct BPEL translation including sequences, pairs of connectors matching <flow> and <switch> activities and while and repeat loops. This method covers graphs that can be iteratively reduced to one annotated vertex. Finally, *Structure-Maximization* combines all strategies by applying Structure-Identification until all possible structured activities have been translated, then ElementPreservation and Minimization for the remaining graph. This improves readability while still working on a broader class of input graphs. The mappings are described algorithmically, but in Structure-Maximization the best strategy has to be chosen manually. Implementation details are not given, and many steps are described informally. The strategies do not work for arbitrary cyclic graphs.

2.1.2 Ouyang et al. and the BABEL Project

The BABEL group, that some of the authors of Ouyang, Dumas, ter Hofstede & van der Aalst (2006*a*) belong to, is aimed at language heterogeneity and coherence, and has produced reference-implementations of their algorithms. The described techniques offer the most comprehensive capabilities. van der Aalst & Lassen (2007) describes a semi-automatic pattern-based approach which is based on Petri nets. Ouyang, Dumas, Breutel & ter Hofstede (2006) instead shows a fully automatic approach based on event-condition-action rules which is applied to BPMN as input producing BPEL code. The article describes *event-action* rules, using BPEL event handlers that represent unbounded parallelism by an unbounded number of threads within a <scope> running simultaneously. <invoke> operations provide goto-like structures. The strategy is further improved in Ouyang, Dumas, ter Hofstede & van der Aalst (2006*a*) to a compact representation that can be directly implemented. Although the mappings provide translations for arbitrary graphs, the produced code severely lacks readability. As a remedy, Ouyang, Dumas, ter Hofstede & van der Aalst (2006*a*) suggests the combination with another translation type for well-structured sub-graphs on the same basis as Structure-Identification in Mendling et al. (2006). Sub-graphs having one single entry-point and one exit-point are called *components*. The reduction of components to a single vertex is called *folding* and formalised in detail. Based on the decomposition of a process graph into components, the article presents an incremental bottom-up overall translation.

The algorithm chooses the next component and transforms is using the best possible transformation strategy. Maximal sequence components are preferred to well-structured components. Unstructured (possibly cyclic) components are only transformed if no other option is available using Event-Condition-Action-rules. This can enable the identification of new well-structured components in the next iteration. The approach selects the best trade-off between readability and completeness for the two algorithms. Although the procedures are given algorithmically, no details are given on how to decompose a graph into its components. Ouyang et al. (2007) further extends the translation by mappings for components that are

not well-structured but can be translated without the use of event-handlers. *Quasi-structured patterns* describes a method to temporarily extend the graph structure by splitting gateways. This creates further components for pairs of gateways that make them match to one <switch>, <flow> or <pick> activity. (Ouyang et al. 2007, p.10) argues that the refinement of the guard conditions might become complicated and error-prone. In this thesis, we propose a solution on how the idea of gateway-splitting can be extended to work for XOR gateways. In Sect. 3.6.2 we show how to use Token Analysis to identify and create those components. *Generalised flow-pattern* translation uses one <flow> activity together with control links and works for acyclic components that have no sub-components, only contain AND gateways and are *safe and sound*, that is guarantee not to cause any deadlocks during execution. First, several simple reduction steps are applied. Next, if sequence patterns or quasi-flow patterns exist, they are translated and folded *remaining within the <flow>*. If not, links through AND gateways are replaced with direct control links inside the flow. Again, the process continues iteratively, until everything has been translated.

The overall translation is similar as in Ouyang, Dumas, ter Hofstede & van der Aalst (2006*a*) but has an extended priority list:

1. maximal sequence components
2. well-structured components
3. quasi-structured components
4. generalised flow pattern
5. unstructured components

Furthermore, a technical report Ouyang, Dumas, ter Hofstede & van der Aalst (2006*b*) from the same group exists, in which another translation method is presented, called *control link-based translation algorithm*. It is similar to the generalised flow-pattern, but can in addition contain XOR gateways. Unfortunately, there exist some inconsistencies within the algorithm.

It works for arbitrary cyclic sub-graphs, using event-handlers by *event-action rule-based translation*. The work also includes a bottom-up procedure, called *folding* to automatically translate process graphs. In succeeding steps the next component enabling the best trade-off is translated and folded to a single task object for the next iteration.

In Koehler & Hauser (2004), a control flow normalisation algorithm (described by Ammarguellat (1992)) is used for process-graph to BPEL transformation. The graph is transformed into a set of *continuation equations* consisting of a label, an instruction (invoke or if-branch) or goto-like references to labels; they can be concatenated via a sequence operator. A set of rules iteratively transforms the equations until only one remains. The remaining construct then has

a straightforward mapping to BPEL using <while>, <sequence> and <switch> activities and variable assignments. Two issues within the transformation exist: first, the transformation is *not confluent*, meaning a different order in rule applications result in output of differing, and second, *non-reducible* input models result in duplicate BPEL code fragments. Concerning confluence, a heuristic transformation strategy is proposed, that picks a transformation rule according to rule priorities, and how often each goto destination occurs on the right hand side of the equations. Code duplicates, however, are unavoidable by this method. The article considers *non-concurrent business process models* only, parallel flow is unsupported.

2.1.3 The Token Flow Algorithm

OMG (2006) describe a mapping based on Token Analysis. The token concept is related to Petri Nets but rather different in its realisation. The tokens are not indistinguishable but have individual markings. Furthermore, Token Flow is not used to express parallel execution but is used for the analysis of graph properties. Conceptual tokens traverse along the flow of the process. They are used to identify the boundaries of fragments of the graph that map to BPEL activities. The segmentation of the graph is a main aspect of the strategy and the technique is described by a set of informal rules. Tokens are created at vertices with multiple out-flow and then propagate downstream along the arcs of the process graph, carrying information on their origin and the number of paths being traced. They recombine with tokens from the same origin. This serves to determine the boundaries of components. The beginning of an activity is "... usually a gateway..." and the end is at the object where "... all the Tokens [...] can be recombined."

For loops, the section of the graph is taken from where "... the loops first merge back (upstream) into the flow until all the paths have merged back to normal flow." (OMG 2006, pp. 202ff). Simple loops with an out-degree of two map to while activities. Multiple out-degree is matched to a <while>/<switch> combination. For interleaved loops, the whole block containing the loops is mapped to a new process. Nested gateways translate to switch activities where branches connecting upstream are handled by goto-like <invoke> operations. The idea has similarities with Ouyang et al.'s Event-action rules and also with Mendling et al.'s Element preservation. Infinite loops translate to <while> statements with a *false* loop condition.The article and the additional example White (2005) make it clear, that the approach relies on human interaction.

Token flow is different to control flow: is does not describe the course of events but rather produces a static labelling at the arcs that enables the identification of components. Sect. 3 extends this concept to work automatically. Unlike in OMG (2006), the method does not stop whenever a component can be identified, but calculates the labelling in one run (token flow) from which component boundaries are deduced (token analysis). Quasi-structured patterns

in Ouyang et al. (2007) can be identified by introducing *Partial Token Convergence*. All concepts in this article are oriented towards a seamless integration of the mappings described in Ouyang, Dumas, ter Hofstede & van der Aalst (2006*a*), Ouyang et al. (2007).

2.2 Workflow Analysis: Relations to Compiler Construction and Parallel Programming

The problem of finding *single-entry single-exit (SESE)* graphs in workflows is a concern in compiler construction and parallel programming. Control regions facilitate instruction scheduling for pipelined machines (Gupta & Soffa (1978)). In the field of interval analysis, irreducible loops are characterised by single entry regions. Zhang & D'Hollander (2004) describe a method of transforming control flow into a hammock[1] graph. This is done by a threefold process: single branches (reducible loops) are replaced by block structured elements, the remaining code around backward and forward branches is converted into loops where the branch is replaced by an exit statement. The loop is then followed by a conditional branch. The technique assumes that the input control graph be already in block-structure. A lexical order of the operations is required, that exists for sequential code, but not for workflow graphs. The proposed algorithm is therefore not suitable for the problems addressed by this work. Also the algorithm runs on connected structures *for each branch*, meaning if this were applied on general workflow, each edge in the graph (apart from simple sequences) would have to be considered a branch.

Havlak (1997) present an algorithm to build a loop nesting tree for arbitrary control flow (for the notions of loop nesting trees and reducibility, also refer to Ramalingam (2000)). from which reducible and irreducible regions can be identified. For building the loop tree, the article uses an extended version of Tarjan's algorithm Tarjan (1974), that does not stop at unstructured regions, but rather marks them and continues processing. The algorithm uses the depth-first search tree for computation. As the dfs-tree is not unique, results depend on the tree structure and thus some reducible portions can be missed. To mend this problem, the article introduces a preparation step, based on the dominator tree: reducible loops can be found from the dominator tree, then empty nodes are inserted into the input graph (preserving semantics), so that reducible loops are guaranteed to be found. The combination of the three algorithms provides a completely automated way of graph decomposition. It is rather complex, however, as different methods need to be combined. Johnson et al. (1994) presents another algorithm for SESE-decomposition that "runs faster than Lengauer and Tarjan's algorithm" using cycle equivalence. In this work we develop an alternative and new approach for the same

[1] In the literature, different definitions for hammock graphs exist; they are subgraphs with a single entry and exit point, in analogy of what is called *components* in this work.

problem. Furthermore, our approach can detect splitted gateways, which is not possible by simple hammock decomposition based on the observation that *edges marking the source and sink of an SESE region are cycle equivalent in the undirected Graph which has an additional connection from the end-to the start-node.* Cycle equivalent means that two edges either both belong to any given cycle, or both do not belong to it. The article provides an efficient algorithm for testing cycle equivalence, based on modified list-operations *create*, *size*, *push*, *top*, *delete*, and *concat*.

2.3 Modelling tools with BPEL export

A variety of workflow modelling tools exists, but support for automatic model transformations is rare. Still, efforts are being made to provide export functionality in the future.

The *IBM Tool Suite* allows modelling arbitrary processes with the WebSphere Business Modeler which can be exported to BPEL using the WebSphere Process Server. Certain BPEL export restrictions prevent all models from being fully transformed. (Wahli et al. 2006, p.109). Also, the results are not necessarily compatible with BPEL; there are restrictions of the types of modeling elements that can be used. Items such as for-loops, while-loops, timers or notification receivers cannot be added to the process ((IBM 2006, p.18)). Additionally, export to BPEL requires the designer of the business process to be technically proficient.

Both, the *Oracle BPEL Process Manager* (Oracle (n.d.)) and ActiveEndpoint's *ActiveBPEL* only have a block-structured model representation. On the contrary, *TIBCO Business Studio* and also the Eclipse version and Process Workbench of *CARNOT* use a graph-structured editor, but do not provide BPEL export. *Bull FR*'s provide modelling tools for BPEL (block-structured) and for graph-based workflow models, but no conversion.

The *BONAPART BPEL-Modeler* provides graphical representation of both models, but with manual conversion. *BOC's ADONIS* provides support for the conversion by keeping track of made transformations, so that only changes in the graph model have to be applied to the BPEL model.

The company argues in their whitepaper that "fully automated transformation, as proclaimed in the MDA (Model Driven Architecture) vision, can neither be realised between CIM (Computation Independent Model) and PSM (Platform Specific Model), nor between PIM-(Platform Independent Model) and PSM level."(BOC 2007, p.16 from German)

MID's Innovator supports BPEL code generation together with its *Innovator Object eXcellence* extension. The tool facilitates orchestration of web-services and transformation to BPEL code, which can be layered into several levels of analysis and design ((Oliver Pera 2007, p.2)).

microTOOL has announced support for automatic model transformation in their upcoming version of *objectiF* Nagy (n.d.), but apart from support for some structured elements (flow and sequence) no details on the transformation are given there.

Telelogic's System Architect supports BPEL code generation from process diagrams, which again is limited to a subset of BPMN (Recker & Mendling (2006)).

EMC has recently made the step into process management and now offers the *Documentum Process Suite* with the Process Analyzer (Java based). They also provide a free tool called *Whiteboards Modeler*, that offers BPEL export functionality to some extent. Only simple sequences can be transformed, and the software is still very unstable and not well documented.

3 Algorithms

The following presents a formal description of the token-flow algorithm (Sect. 3.2) that grounds on the concepts in OMG (2006). Sect. 3.1 delivers the mathematical foundation. The problem of cycles for the algorithm is described in Sect. 3.3. Implications are investigated in Sect. 3.4, and *contraction* is proposed as a solution. The developed techniques to handle arbitrary cyclic graphs are formalised in Sect. 3.5. Finally, Sect. 3.6 shows how the components are derived from the results of the token flow covering maximum sequences and partial convergence.

3.1 Terminology

Much of the used notation is adapted from Ouyang, Dumas, ter Hofstede & van der Aalst (2006*a*), except for some structural differences: the discussion is not restricted to BPDs, but rather based upon directed graphs. This renders more general results that can be applied to different problems. Vertices, for instance, can have multiple in and multiple out degree at the same time.

3.1.1 Set Theory.

Let S be some set. $\mathcal{P}$ is the set of all subsets: $\mathcal{P}(S) = \{s \mid s \subseteq S\}$. The element function elt selects the only element from a singleton set: $\mathsf{elt}(\{x\}) := x$. Similarly $\mathsf{first}((a, b)) := a$. For a function $f : S \longrightarrow X$ we lift the function symbol to apply to subsets of S by element-wise application, that is: define $f : \mathcal{P}(S) \longrightarrow \mathcal{P}(X)$ for some subset $S' \subseteq S$ as $f(S') := \{f(s) \mid s \in S'\}$.

A *partial* function $g : S \longrightarrow X \cup \{\perp\}$ is indirectly referred to as a set containing the elements that have a defined mapping:

$$s \in dom(g) :\Leftrightarrow g(s) \neq \perp$$

3.1.2 Graph Theory.

A *directed graph* is a tuple (V, A). V contains the *vertices*, $A \subseteq V \times V$ the *arcs*. We refer to the start- and end-vertex of an arc via the functions $\mathsf{from}((v,w)) = v$ and $\mathsf{to}((v,w)) = w$. The incoming and leaving arcs of a vertex v are addressed by the mappings

$$\mathsf{in}(v) = \{(w,v) \in A \mid w \in V\},$$

$$\mathsf{out}(v) = \{(v,w) \in A \mid w \in V\} \ .$$

Now let $p \in V^*, p = v_1, \ldots, v_n$. p is called a *path*, if $\forall\ i \in 1, ..., n-1 :\ (v_i, v_{i+1}) \in A$. Furthermore, if $i \neq j \Longrightarrow v_i \neq v_j$, we call p a *simple path*. Also define $\mathsf{vertices}(p) := \{v_i \mid i \in 1, ..., n\}$ and $\mathsf{arcs}(p) := \{(v_i, v_{i+1}) \mid i \in 1, ..., n-1\}$.

We assume graphs to have *exactly one start and one end vertex*, each with *exactly one entering/leaving arc*. Multiple start events in a BPD can be connected to one single start vertex through a fork gateway. For multiple end vertices a join gateway is used.

Definition 3.1.1 (Cycle)**.** A (simple) path $c = v_1, \ldots, v_n$ forms a *(simple) cycle*, if and only if $(v_n, v_1) \in A$. For cycles, let $\mathsf{arcs}(c) := \{(v_i, v_{i+1}) \mid i \in 1, ..., n-1\} \cup \{(v_n, v_1)\}$.

A cycle s is called *strongly connected component* (SCC), if it is maximal, i.e. if for all cycles $c : \mathsf{vertices}(c) \cap \mathsf{vertices}(s) \neq \emptyset \Rightarrow \mathsf{vertices}(c) \subseteq \mathsf{vertices}(s)$.

In this article we *do not consider graphs that contain unreachable cycles.*

Components are connected subsets with the following properties.

Definition 3.1.2 (Component)**.** A subgraph $\mathcal{C} = (V_\mathcal{C}, A_\mathcal{C})$ of a graph (V, A) with $V_\mathcal{C} \subseteq V$, $A_\mathcal{C} = A \cap (V_\mathcal{C} \times V_\mathcal{C})$ is called *component*, if and only if all of the following conditions hold:

$$\begin{array}{ll} \mathcal{C}i) & |\mathsf{in}(V_\mathcal{C}) \backslash A_\mathcal{C}| = 1 \ , \\ \mathcal{C}ii) & |\mathsf{out}(V_\mathcal{C}) \backslash A_\mathcal{C}| = 1 \ , \\ \mathcal{C}iii) & \forall\ v \in V_\mathcal{C} : |\mathsf{in}(v)| > 0 \wedge |\mathsf{out}(v)| > 0 \ . \end{array}$$

Let $\mathsf{source}(\mathcal{C}) := \mathsf{elt}(\mathsf{in}(V_\mathcal{C}) \backslash A_\mathcal{C})$ and $\mathsf{sink}(\mathcal{C}) := \mathsf{elt}(\mathsf{out}(V_\mathcal{C}) \backslash A_\mathcal{C})$. If $\mathsf{to}(\mathsf{source}) = \mathsf{from}(\mathsf{sink})$ then the component is *trivial*.

All flow must enter a component through its source arc and leave it through its sink arc:

Lemma 3.1.1. *For a component $\mathcal{C}$ and for all $v \in V_\mathcal{C}$:*

$$\mathsf{from}(\mathsf{in}(v) \backslash \{\mathsf{source}(\mathcal{C})\}) \cup \mathsf{to}(\mathsf{out}(v) \backslash \{\mathsf{sink}(\mathcal{C})\}) \subseteq V_\mathcal{C} \ .$$

Let $v \in V_{\mathcal{C}}$. From $\mathcal{C}i$ and $\mathcal{C}ii$ follows, that v's entering arcs $\mathsf{in}(v)\backslash\{\mathsf{source}\}$ and leaving arcs $\mathsf{out}(v)\backslash\{\mathsf{sink}\}$ must also be included in $A_{\mathcal{C}}$. By construction of $A_{\mathcal{C}}$, this means that all vertices connected to these arcs also belong to the component. □

Furthermore, $\mathcal{C}iii$ states that no start or end arcs (having in or out degree of one) may be contained in components. (otherwise, for instance, the graph in Fig. 3.1 would contain a component with $\mathsf{source} = a_1$ and $\mathsf{sink} = a_2$).

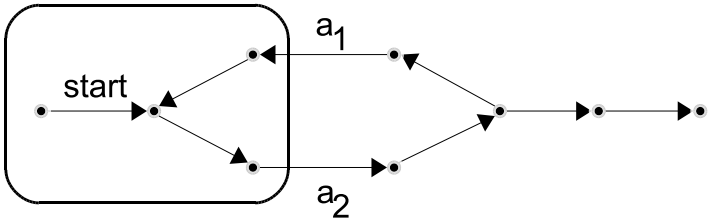


Figure 3.1: Components do not contain start events.

3.2 Token-flow Algorithm

Based on the notation above, we now give a precise, formal description of the token *propagation* mechanism described in OMG (2006).

Token flow is calculated in two steps: first the single tokens propagate through the graph and second tokens from the same origin re-combine.

In the first step, tokens are created at the out-flow of splitting gateways carrying information on their origin. They propagate along the flow and can compound with other tokens. Each arc is assigned a subset of tokens called *token labelling*.

For a single token, the propagation through the graph is calculated by tracking its route along the arcs. When tokens arrive at a gateway with several out arcs, *all of the gateway's out-arcs* are labelled with the same token: At vertices with out degree > 1 , new tokens are being created. The out-arcs are labelled with the union of the arriving token sets and the newly generated tokens. At merging gateways, the out-arc is labelled with *the union of all incoming tokens*.

Calculating the flow for each token separately is inefficient because arcs have to be visited several times, once for each token. It becomes more efficient when handling complete sets of tokens, by successively calculating the out-flow at nodes where all entering flow has been labelled, thus the leaving flow can be determined. This strategy performs a blocking wait, and it does not work for arbitrary graphs. Deadlocks occur at cycles, but the problem can be avoided by separating them from the graph. The required techniques are developed in Sect. 3.4.

In the second step, the recombination of tokens is calculated. When all tokens belonging to the same gateway have arrived at one arc, they are removed from the labelling (recombination). Components can be derived by matching pairs of arcs with equal token sets.

Different to OMG (2006), the process does not stop whenever a component is encountered but continues until all the arcs have been labelled. The procedure does not have to start again, and also enables for the recognition of more advanced, interleaved structures.

We now define the set of *tokens* $\mathbb{T}$. A token carries information on the parallelisation for which the token was generated, i.e. the origin vertex v and a number i refering to the corresponding out-arc:

$$\mathbb{T}_{(V,A)} := \{(v, i) \mid v \in V \wedge i \in \mathbb{N} \wedge i < |\mathsf{out}(v)|\} \ .$$

Each arc in the graph is assigned a subset of $\mathbb{T}$ by the *token labelling function* $t : A \rightarrow \mathcal{P}(\mathbb{T}) \cup \{\perp\}$. Token creation occurs at vertices witch $|out(v)| > 1$. Tokens propagate and unite at vertices with $|in(v)| > 1$. Figure 3.2 illustrates the flow of tokens created at vertices 1 and 2. After all arcs have been labelled, tokens originating from the same vertex *converge* and are removed from the labelling (indicated in the figure by the curly brackets).

Finally, the labelling is used to determine the components of the graph. If for two arcs $a, b \in A, a \neq b$:

$$t(a) = t(b)$$

they mark the beginning and the end of a component $\mathcal{C}$, i.e.

$$\{a, b\} = \{\mathsf{source}(\mathcal{C}), \mathsf{sink}(\mathcal{C})\}.$$

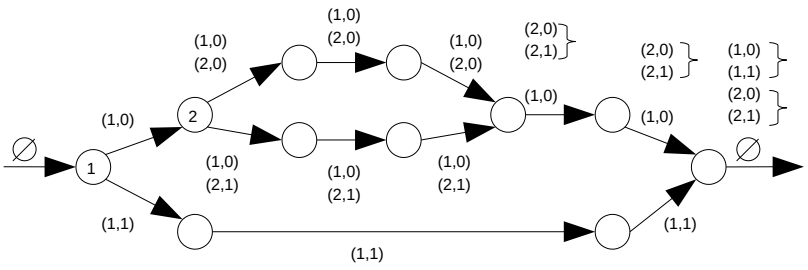


Figure 3.2: Example of converging token flow.

The resulting components can be overlapping, ambiguous and include trivial components. In Fig. 3.2 the converged labelling $\{(1, 1)\}$ appears at four arcs. Each pair of these arcs marks a valid component, but not all of them are clearly desirable. This can be avoided by a strategy presented in Sect. 3.6 which shows how to derive the correct partitioning, covering sequences and further component types.

Algorithm 1 Token-flow

Require: $t : A \to \mathcal{P}(\mathbb{T}) \cup \{\bot\}$ // initial marking
$N \subseteq V$ // to-do set of vertices
Ensure: $t : A \to \mathcal{P}(\mathbb{T})$

```
1: processVertex(v ∈ V) {
2: P(T) M := ⋃_{u∈t(in(v))} u;
3: if |out(v)| = 1 then
4:    t(elt(out(v)) := M;
5: else
6:    int i := 0;
7:    for all a ∈ out(v) do
8:       t(a) := M ∪ {(v, i++)};
9:    end for
10: end if
11: }
12:
13: tokenFlow(t, N) {
14: while N ≠ ∅ do
15:    pick(v ∈ N | t(in(v)) ⊭ ⊥);
16:    processVertex(v);
17:    N = N\v;
18:    timestamp(v);
19: end while
20: for all a ∈ A do
21:    for all v ∈ first(t(a)) do
22:       if t(a) ⊇ {(v, i) | i < |out(v)|} then
23:          t(a)\ = {(v, i) | i < |out(v)|}
24:       end if
25:    end for
26: end for
27: }
```

Algorithm 1 summarises the Token Flow procedure. For an acyclic graph with start node *start*, call the **tokenFlow** function with an initial labelling $t(start) := \emptyset$, and with $N := V \backslash \{start\}$.

The function **pick**$(e \in E \mid prop : e \to \mathbb{B})$ indeterministically selects an element e from a set E, that meets a required property, i.e. $prop(e) = true$. The function **timestamp** assigns an incrementing index to the visited elements, thus imposing an ordering on them.

The main loop of the algorithm in line 14 picks an arbitrary vertex for which all in-arcs have been labelled with tokens, and calls the **processVertex** function for it. This function computes the leaving flow from the entering flow by first merging all token sets from the entering arcs (line 2), and then propagating the resulting set to the leaving arcs. For multiple leaving arcs, each is assigned a new token in line 8. After processing, each arc is assigned a

timestamp (line 18), which is later used to identify sequence-boundaries (see Sect. 3.6). Once the main loop in **tokenFlow** terminates, converged tokens are removed (line 22).

Our reference implementation (see Sect. 4) uses depth-first-search. Vertices and arcs are visited once; thus the algorithm has a time-complexity in $O(|V| + |A|)$. Using a hash map, identifying equally labelled arcs can be achieved in constant time.

3.3 Cycles Cause Deadlocks

Algorithm 1 only works for acyclic graphs. Vertex 1 in the cycle in Fig. 3.3 cannot be processed until the back-arc $(4, 1)$ has been labelled; this however would require the flow leaving 1 be processed first, as it must pass along the vertices 2, 3 and 4. This results in a deadlock. All cycles have a back arc in the depth-first-search tree, pointing to a connecting vertex which cannot be processed because the required flow through the back-arc must pass through the vertex itself; hence, the flow cannot proceed from there, causing a deadlock. The following chapter introduces a method to avoid this problem.

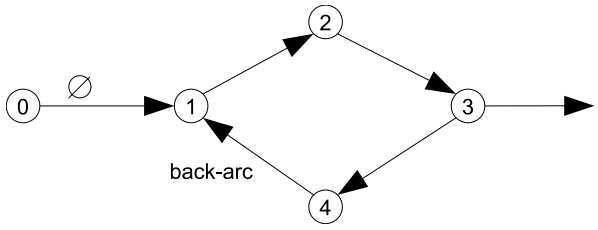


Figure 3.3: A deadlock at node 1.

3.4 Contracting Cycles

The following theorem provides an important insight into the meaning of cycles for the algorithm.

Theorem 3.4.1. *Let c be a cycle, $\mathcal{C} = (V_\mathcal{C}, A_\mathcal{C})$ a component. Then*

$$\mathsf{source}(\mathcal{C}) \in \mathsf{arcs}(c) \Leftrightarrow \mathsf{sink}(\mathcal{C}) \in \mathsf{arcs}(c) \ .$$

Proof. Let $\mathsf{source}(\mathcal{C}) \in \mathsf{arcs}(c)$ and assume $\mathsf{sink}(\mathcal{C}) \notin \mathsf{arcs}(c)$. For the path of the cycle with length $|c| = n$ define without loss of generality $(c(n), (c(1)) = \mathsf{source}(\mathcal{C})$. Thus the path starts at $c(1) = \mathsf{to}(\mathsf{source}(\mathcal{C}))$. By induction over the path length of c, we show that $\forall i : c(i) \in V_\mathcal{C}$. For $c(1) \in V_\mathcal{C}$ this is clearly true, as the source points to a vertex inside the component. Now let it be true for $c(i)$. Applying Lemma 3.1.1 yields that $c(i+1) \in V_\mathcal{C}$, because $(c(i), c(i+1)) \neq$

$\mathsf{sink}(\mathcal{C})$, and the sink is the only arc that leaves the component. Then also $c(n) \in V_{\mathcal{C}}$. But $(c(n), c(1)) = \mathsf{source}(\mathcal{C})$ violates the definition of the source $\mathcal{C}i$), for $(c(n), c(1)) \in A_{\mathcal{C}}$. It is inside the component. This contradicts the assumption. The other implication follows in analogy by the same argument in reverse direction: for the path along the cycle starting at the sink, each predecessor must not be the source because of Lemma 3.1.1. Symmetrically, the same argument contradicts the definition of the sink $\mathcal{C}ii$). □

In other words, Theorem 3.4.1 states that **source and sink of a component are never positioned across the boundaries of any cycle**. For example, in Fig. 3.4 (a) the bold arcs do not identify any component because only one belongs to a cycle the other does not. In (b) both arcs are in the same cycle and indicate a valid component. More restrictions for inter-cyclic components, as shown in (c), are discussed later.

No information on the location of components passes from the outside into any cycle interior and vice versa. Tokens convey information on component boundaries. Therefore *cycles can be isolated from the token flow* without losing information.

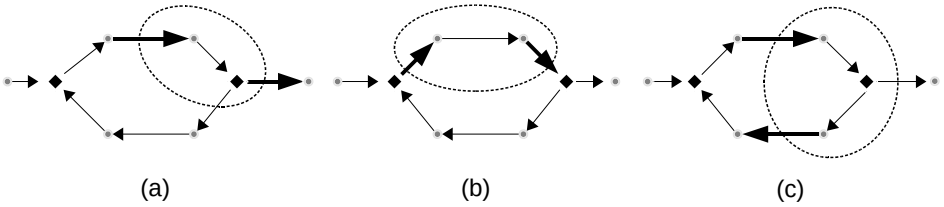


Figure 3.4: Component boundaries do not cross cycle boundaries.

To achieve this, two requirements must be met: *i)* the flow around cycles (*cycle external*) and *ii)* the flow inside cycles (*cycle internal*) must lead to a correct identification of components.

The following sections introduce the necessary preparations to enable arbitrary cyclic graphs to be processed by the token flow algorithm. In Sect. 3.4.1 we specify the external flow, in Sect. 3.4.2 the behaviour of cycle internal flow, and in Sect. 3.4.3 we give a solution to problems that arise when cycles are nested within others.

3.4.1 Cycle-External Flow.

Figure 3.5 shows two components $\mathcal{C}_1$ and $\mathcal{C}_2$ containing (simple) cycles. Each source and sink pairs must carry the same token labelling: $t(a_2) \stackrel{!}{=} t(a_1) = \emptyset \ \wedge \ t(a_4) \stackrel{!}{=} t(a_3) = \emptyset$.

To achieve this for $\mathcal{C}_1$, the tokens $(v, 0)$ and $(v, 1)$ flowing into the cycle must converge at a_2. This could be achieved by tracking the tokens separately, as they would finally arrive at a_2. For the flow inside the cycle is independent of the exterior , this is possible if all incoming

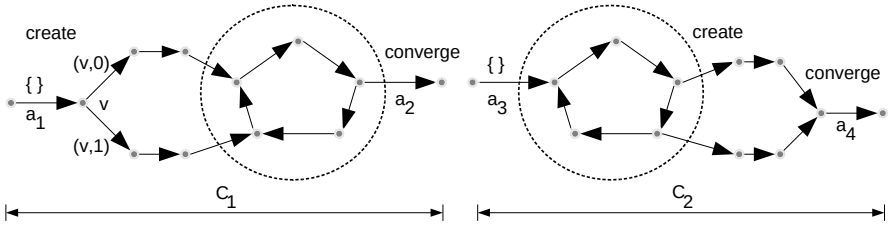


Figure 3.5: Cycle external flow.

flow merges *by the cycle*. In $\mathcal{C}_2$ the two branches flowing out of the cycle do not belong to the same component and therefore must have a different token labelling. Clearly this cannot be achieved by tracking the tokens separately, for the arcs leaving the cycle in $\mathcal{C}_2$ would receive an equal labelling. In order to keep the branches separate the tokens must, again, be created *by the cycle itself*.

Token creation and convergence are vertex properties. This gives rise to the idea of *treating cycles and vertices equally*. For this purpose we introduce *replacement vertices* for cycles, and call the process of embedding them into the graph *contraction*. Figure 3.6 illustrates the process: each arc a_i is replaced by an arc a_i' connecting with the replacement vertex.

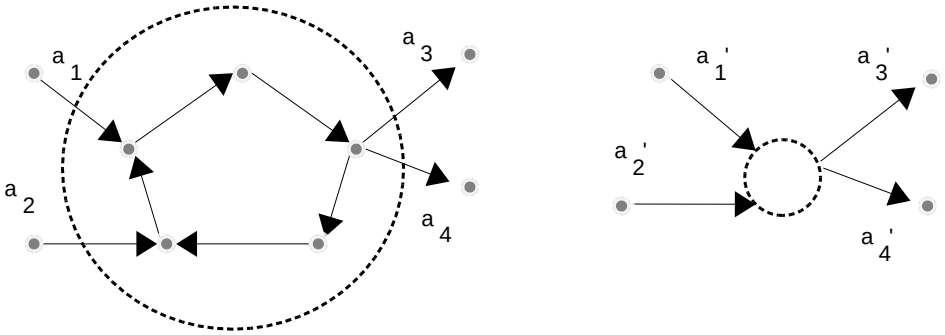


Figure 3.6: A cycle is *contracted* to a replacement vertex.

To map a cycle c onto a vertex, we first determine its incoming and leaving flow. Formally, the connectivity of a cycle c to its environment is defined by the functions in and out in analogy to those of a vertex:

$$\begin{aligned} \mathsf{in}(c) &:= \{(u,v) \in A \backslash \mathsf{arcs}(c) \mid v \in \mathsf{vertices}(c)\}, \\ \mathsf{out}(c) &:= \{(u,v) \in A \backslash \mathsf{arcs}(c) \mid u \in \mathsf{vertices}(c)\} \ . \end{aligned}$$

To contract a cycle c, we need to set the references to the replacement vertex; for all entering arcs $a \in \mathsf{in}(c)$, set $\mathsf{to}(a) := c$ and for all leaving arcs $a \in \mathsf{out}(c)$ set $\mathsf{from}(a) := c$.

Note that unlike the vertices in BPDs, contracted cycles can have multiple in- and out-

flows. This is no limitation for the token-flow algorithm and does not effect model semantics.

Contraction is only a temporary process to determine the token labellings. Storing the original references of the replaced arcs enables them to be restored after the algorithm has finished. Thus, the original input graph can be mapped to its corresponding components.

3.4.2 Cycle-Internal Flow.

After contraction, external flow does not arrive inside cycles. A means of boot-strapping internal flow is required, similar to the $\emptyset$ labelling of the start arc. We define an *initial labelling* at certain internal arcs for each cycle.

The places where to start naturally seem to be the vertices, that have a connection to the environment. We call internal vertices that are connected to a cycle c's environment *ports*:

$$\mathsf{ports}(c) := \mathsf{to}(\mathsf{in}(c)) \cup \mathsf{from}(\mathsf{out}(c)) \ .$$

Ports of a cycle cannot be contained in internal components (see Fig. 3.4(c)): if a port were included in a component then, because of its connection to the exterior, following from Lemma 3.1.1, some vertices of the environment would have to be included, too. Theorem 3.4.1 states that this is not possible.

To ensure that ports are not identified as parts of components, each flow between two ports needs different token-ids. The token flow shall begin at the ports, so we set the initial labelling at the out arcs of ports belonging to the cycle. Formally, they belong to the set

$$\mathsf{initflow}(c) := \mathsf{out}(\mathsf{ports}(c)) \cap \mathsf{arcs}(c) \ .$$

Figure 3.7(a) depicts the ports and initflow arcs of a simple cycle. The initial tokens must not converge, as can be seen in Fig. 3.7(b): if the initial labellings $t(a_1)$ and $t(a_2)$ did converge at vertex v, then $t(a_3)$ would be $\emptyset$ and thus a_3 would falsely be identified with the in and out arcs of the cycle.

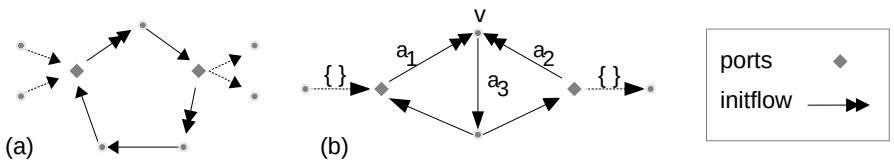


Figure 3.7: A cycle with its ports and initflow (a). Internal tokens do not converge (b).

Therefore, we expand the token set by a non-converging *internal* token type:

$$\mathbb{T}_{(V,A)} := \{(v,i) |\ v \in V \wedge i \in \mathbb{N} \wedge i < |\mathsf{out}(v)|\} \cup \{(\mathsf{int}, i) |\ i \in \mathbb{N}\}$$

The initflow arcs must be labelled with different int- tokens. In Fig. 3.7(b) for instance, the labelling is $t(a_1) = \{(\mathsf{int}, 0)\}$ and $t(a_2) = \{(\mathsf{int}, 1)\}$.

3.4.3 Substructure of Cycles.

So far, we only considered cycles as a whole, but SCCs and sub cycles can consist of several simple cycles. Contracting the SCC in Fig. 3.8 (a) leaves a simple cycle within the SCC. Without further treatment, a deadlock occurs in the internal flow of the cycle at node v. Therefore, the *sub-cycle needs to be contracted*, as shown in Fig. 3.8 (b).

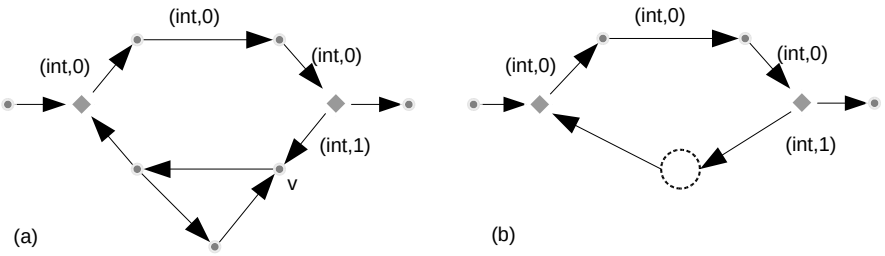


Figure 3.8: An embedded cycle (a) and its deadlock resolution (b).

Only contracting all simple cycles, however, is not always sufficient. The cycle in Fig. 3.9 consists of two simple cycles c_1 and c_2. Contracting each cycle separately leads to the situation shown in (c), *producing a deadlock*. For the SCC in the figure contracting sub-cycles *is not necessary*. The resulting labelling in this case produces correct results. Only contracting the whole SCC is required.

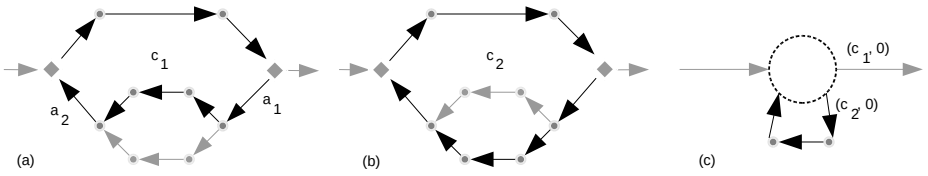


Figure 3.9: An SCC c consisting of the simple cycles c_1 and c_2.

As we have seen, contracting nested cycles is only appropriate, if they cause internal deadlocks. In Sect. 3.4.2 we argued that Internal token flow initially begins at the ports. Hence the emerging token-flow leaving ports has been labelled and they do not require further processing, therefore no internal deadlock occurs there; deadlocks can only occur in the remaining sub-graph. The sub cycles that cause deadlocks can be determined by *taking the ports from a cycle* (their leaving flow has already been labelled), and *if any cycle remains in the subgraph, it must also be contracted*.

Remaining cycles can again contain sub cycles causing deadlocks. The following recursive top-down strategy identifies all sub cycles that need contraction:

1. Find all SCCs within the (sub-) graph.
2. Contract them.
3. Repeat step 1 on the subgraph of each found SCC without its ports.

Taking the ports from the cycle in Fig. 3.9 opens up both simple cycles c_1 and c_2; no sub-cycle is left, so only the top SCC is contracted. In Fig. 3.8(a) the sub-cycle does not contain ports and thus is contracted. The remaining subgraphs are shown in Fig. 3.10.

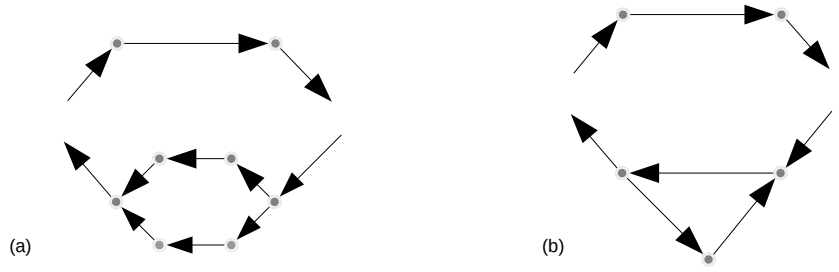


Figure 3.10: The SCCs without the ports.

Also note that taking the ports from a cycle leaves exactly those simple cycles intact, that do not contain any port vertices.

3.5 Preparations for the Token-flow Algorithm

Algorithm 2 makes the necessary preparations for the token flow algorithm on graphs containing cycles. The recursive function **contractAll** is initially called on the set containing all vertices of the graph. First all (non-trivial) SCCs are determined (line 21). Algorithms for detecting SCCs are broadly available, e.g. see Thomas H. Cormen (1994). Each SCC is contracted (line 22) which returns the replacement vertex v. The contraction replaces the in and out arcs incident on the cycle by new arcs incident on the new replacement vertex. Then in line 23, the arcs are labelled for the initial flow by assigning each arc a unique token. For each SCC a recursive call (line 27) handles the sub-cycles, as described in Sect. 3.4.1.

3.6 Exploiting Results for Component Identification

From the arc labelling, components can be derived. Only some of them are useful, however. Components should be contained within a tree-like structure, meaning for two components that either one is contained within the other (a child in the tree) or that they are disjoint (on different branches). The only case where this might occur is within sequences. In Sect. 3.6.1 their handling is described. Sect. 3.6.2 extends the token concept to enable the detection of further structures.

3.6.1 Maximum Sequences.

If a token configuration exists only once, the arc is neither source nor sink. Two arcs with equally labelled token sets are source and the sink of a valid component. If the component is trivial, it only contains a single vertex and can be neglected. Non-trivial components are *cyclic* components if they contain any contracted vertices, otherwise they are *acyclic* components.

Whether an arc is a source or a sink is determined by its *finishing timestamps* of the token flow algorithm: the arc carrying the earlier timestamp is the source. This follows from the flow properties: in depth-first search, the source must be encountered before the sink arc.

More than two arcs with equal sets of tokens mean a *sequence*. Figure 3.11(a) depicts a simple chain with the labelling $t(a_1) = t(a_2) = t(a_3) = t(a_4)$. Of all possible and valid components following from combinations of these tokens (e.g. $\mathsf{source} = a_2, \mathsf{sink} = a_4$), only the *maximum sequence* is of interest; it contains all subsequences. The maximum sequence component $\mathcal{C}_1$ can be identified by the finishing timestamps: its source carries the earliest and the sink the latest timestamp.In the example: $\mathsf{source}(\mathcal{C}_1) = a_1, \mathsf{sink}(\mathcal{C}_1) = a_4$.

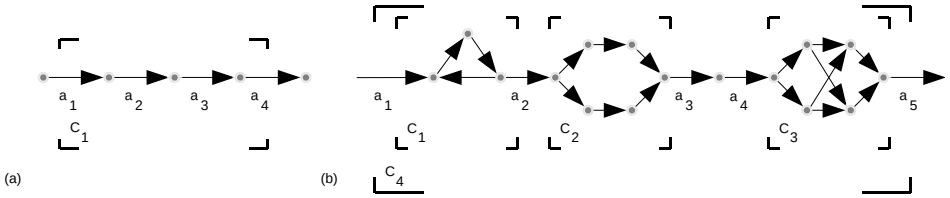


Figure 3.11: Sequence Components.

Sequences can also contain sub-components, like $\mathcal{C}_4$ containing $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ in Fig. 3.11(b). Each subcomponent is located between a pair a_i, a_{i+1}. The pair (a_1, a_2) belongs to a cycle component, whereas the pairs (a_2, a_3) and (a_4, a_5) belong to acyclic components, and (a_3, a_4) is trivial. Eventually, component $\mathcal{C}_4$ maps to a sequence including three components and one single vertex.

To summarise the component identification for arcs with equal labellings:

- For exactly two equally labelled arcs, add a component (if non-trivial).
- For more, add a sequence for the earliest and latest labelled arc.
- For each pair of arcs with succeeding timestamps, add a component (if non-trivial).

Token analysis yields a classification of components into three categories, which can be used to associate the derived components with concrete BPEL elements:

- Cyclic components containing contracted vertices either translate to <*while*> or <*repeat*> activities or can be translated using event handlers, as in Ouyang, Dumas, ter Hofstede & van der Aalst (2006*a*).

- acyclic components translate to <*switch*>, <*pick*> and <*flow*> activities (flow pattern-based translation in Ouyang et al. (2007) can be used here), and
- sequence components.

3.6.2 Partial Convergence.

Token-analysis can be extended to identify additional components, outlined by *quasi-structured* translation by Ouyang et al. (2007). The BPD in Fig. 3.12(a) depicts a pair of XOR gateways that can be matched to a component by introducing a new gateway to the graph, as shown in Fig. 3.12(b): the intermediate arc labelled $a \vee b$ is created by *splitting the gateway.*

To identify such components we introduce *partial token convergence.*

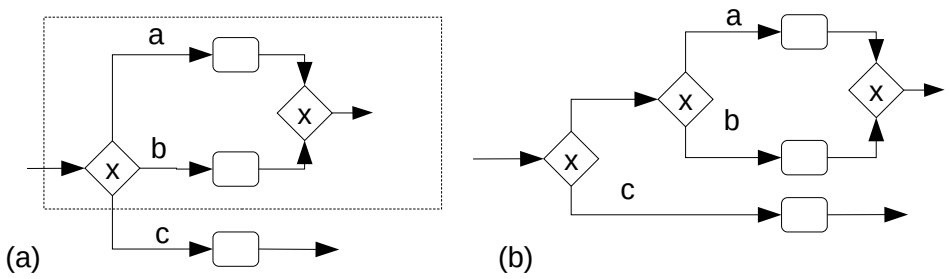


Figure 3.12: Dividing a gateway into sub-branches.

The idea is to introduce virtual arcs with a token combination belonging to the arc in a splitted gateway. For a gateway there are several different possibilities to split. Figure 3.13 shows a gateway with three branches (a) and the different possibilities to split the gateway (b,c,d). Each splitted gateway has a different partial token set. The virtual token sets are calculated for each gateway, but the gateway is only splitted, if a virtual token set matches another real labelling in the token flow. This introduces a new component and improves the structure of the graph decomposition.

Figure 3.14 shows the token-flow for splitting a gateway with multiple out-arcs (i) and multiple in-arcs (ii), as in the situation in Fig. 3.12. The gateway v in (i) has the virtual token sets $\{(v,0),(v,1)\}$, $\{(v,1),(v,2)\}$ and $\{(v,0),(v,2)\}$. At node w, the tokens combine, such that the resulting set $t(\mathsf{out}(w)) = \{(v,0),(v,1)\}$ matches a virtual configuration: gateway v must be splitted accordingly.

Likewise, merging gateways enable for splitting, like v in Fig. 3.14(ii). Here, the *virtual token sets are partial combinations of the entering arcs.* There are three entering token sets $\{(u,0),(w,0)\}$, $\{(u,0),(w,1)\}$ and $\{(u,1)\}$. The virtual token sets consist of combinations

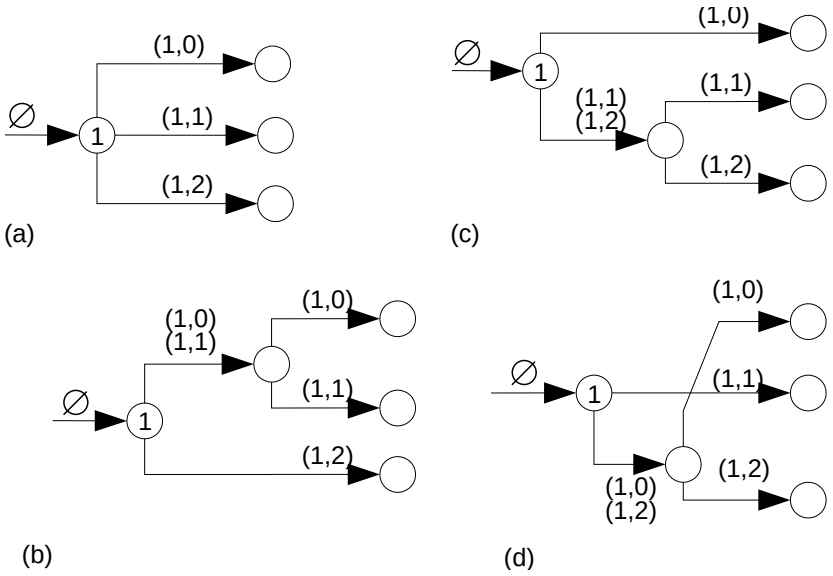


Figure 3.13: Virtual Token Sets of Splitted Gateways.

of for two entering sets each: $\{(u,0)\}$ (after $(w,0)$ and $(w,1)$ have converged), $\{(w,0)\}$ (after $(u,0)$ and $(u,1)$ have converged) and $\{(w,1)\}$ (after $(u,0)$ and $(u,1)$ have converged). $t(\mathsf{in}(w)) = \{(u,0)\}$ has a match, so the gateway splitting is identified.

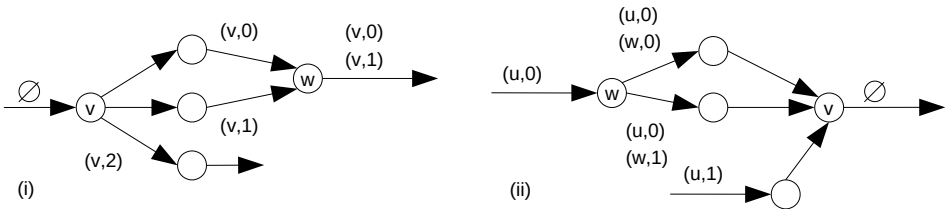


Figure 3.14: Requirements for partial convergence.

Generally at gateways with three or more entering or leaving arcs, any such intermediate token set must be considered. The following describes how to calculate the partial token sets.

Case *i)* $|\mathsf{out}(v)| > 2$. Any combination of the leaving token sets must be considered. For all subsets $M \subseteq \mathbb{T} \cap \{(v,i) | i \in \mathbb{N}\}$ of tokens created at vertex v, with $|M| \geq 2$, define virtual token sets

$$t_{part,j}(in(v)) := t(in(v)) \cup M \ .$$

j is a running index for the subsets.

Case *ii)* $|\mathsf{in}(v)| > 2$. At a merging gateway, the combinations of entering sets must be considered. This means, for any subset $e \subseteq \mathsf{in}(v)$ of entering arcs with $|e| \geq 2$, we define a

virtual token set t_{part} of the tokens:

$$t_{part,j}(\mathsf{out}(v)) := \bigcup_{u \in t(e)} u \ .$$

There are $2^{|\mathsf{out}(v)|} - |\mathsf{out}(v)| - 2$ partial combinations for case *i)* and $2^{|\mathsf{in}(v)|} - |\mathsf{in}(v)| - 2$ for case *ii)*. Token analysis must consider partial tokens for AND and XOR gateways. Tokens in t_{part} converge in the usual way. If tokens match with partial arcs, they are generated by splitting gateways (as shown in Fig. 3.12). For instance, if $t(a) = t_{part}(b)$ then the gateway node b is divided, generating a new valid component. If the gateway type is XOR, guard conditions need to be handled. The guard-condition of the intermediate arc is *the disjunction of the guards of the corresponding splitted branches.*

Algorithm 2 Mark initial flow and contract SCCs.

Require: (V, A) directed graph
Ensure: $t : A \to \mathcal{P}(\mathbb{T}) \cup \{\bot\}$ // initial marking
$N \subseteq V$ // to-do set of vertices

1: **Init:** $N := V \backslash \{start\}$;
2: $t(start) := \emptyset$;
3: **int** $i := 0$; // counter for int tokens
4: **contractAll**(V);
5:
6: **contract**$(c \in V^*)\{$
7: V v; //new replacement vertex
8: $V \cup = \{v\}$;
9: **for all** $a \in \mathsf{in}(c)$ **do**
10: $A \cup = (from(a), v)$;
11: $A \setminus = a$;
12: **end for**
13: **for all** $a \in \mathsf{out}(c)$ **do**
14: $A \cup = (v, to(a))$;
15: $A \setminus = a$;
16: **end for**
17: **return** v;
18: $\}$
19:
20: **contractAll**$(V' \subseteq V)\{$
21: **for all** $s \in$ **SCCs**(V') **do**
22: V $v :=$ **contract**(s);
23: **for all** $a \in \mathsf{initflow}(s)$ **do**
24: $t(a) := \{(\mathsf{int}, i++)\}$;
25: **end for**
26: $N \setminus = \mathsf{ports}(s)$;
27: **contractAll**$(\mathsf{vertices}(s) \backslash \mathsf{ports}(s))$;
28: **end for**
29: $\}$

4 Use Case

We now demonstrate Token Analysis and our proof-of-concept implementation. It can be downloaded from `http://sourceforge.net/projects/tokenanalysis/`. The following presents in detail all the different mapping steps from the BPD input model to the final BPEL code. We use a realistic example to give an overview of how techniques described in Sect. 3 work together.

We show the single steps of the transformation with a sample process graph for quality control in Fig. 4.1 (the graph is used for demonstration only. A real process graph would be much more complex).

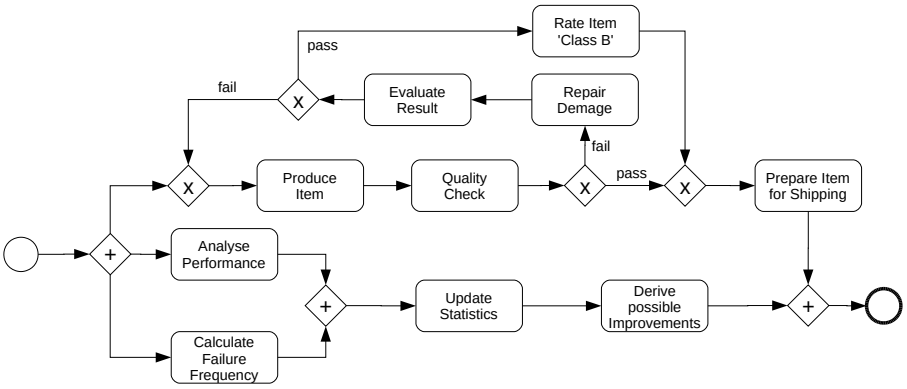


Figure 4.1: A BPD for Quality Control of Production.

The first step in analysis is, according to Sect. 3.4 the identification of the SCCs that need to be contracted. The graph contains one SCC $(2, 3, 4, 5, 6, 7, 8)$ with ports $2, 5$, and 8.(Fig. 4.2).

Taking the ports from the only SCC in Fig. 4.2 leaves no internal SCCs, as shown in Fig. 4.3. Thus, only one SCC needs to be contracted to the replacement vertex SCC 0. Next, the initflow arcs of the cycle, $(2, 3), (5, 6)$, and $(8, 2)$, are given internal token labels along the init-flow arcs for boot-strapping. The result is shown in Fig. 4.4.

The token flow algorithm is launched for the initial labelling of the SCC and an $\emptyset$ label at $(S, 1)$. First consider cycle external flow. Figure 4.5 shows the flow around the contracted vertex SCC 0 and the token configurations for the arcs, after token convergence.

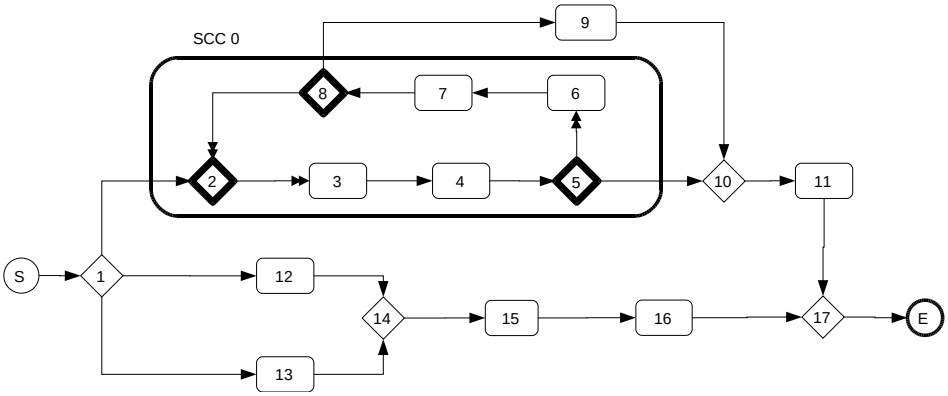


Figure 4.2: One SCC in the Quality Control Graph.

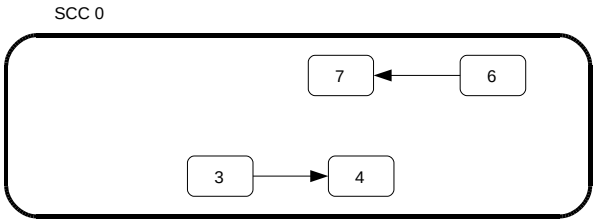


Figure 4.3: The SCC without its Ports.

There are 3 tokens created at vertex 1. Therefore, additional partial token configurations are created; they are $\{(1,0),(1,1)\}$, $\{(1,1),(1,2)\}$ and $\{(1,0),(1,2)\}$.The first token $(1,0)$ enters the cycle replacement vertex. SCC 0 has two leaving arcs $(8,9),(5,10)$. Here the replacement vertex creates 2 new tokens (SCC $0,0$) and (SCC $0,1$). The tokens propagate along the flow and at vertex 10, the ones belonging to SCC 0 converge. At an intermediate arc of vertex 1 there is a partial match with the arc $(14,15)$ carrying the tokens $\{(1,1),(1,2)\}$. Thus the gateway is splitted; the result of the splitting is depicted in Fig. 4.6, together with the remaining internal flow. Finally, all branches merge back at 17.

In the cycle internal flow no branching takes place. Therefore the tokens simply propagate up to the next ports.

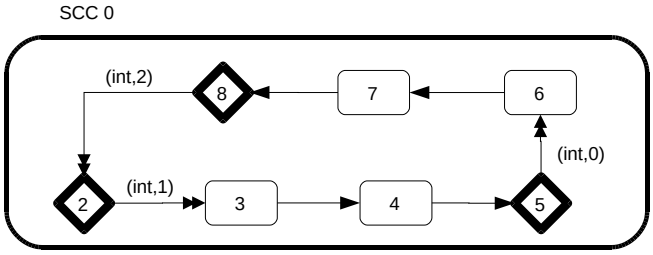


Figure 4.4: The init Flow for the SCC.

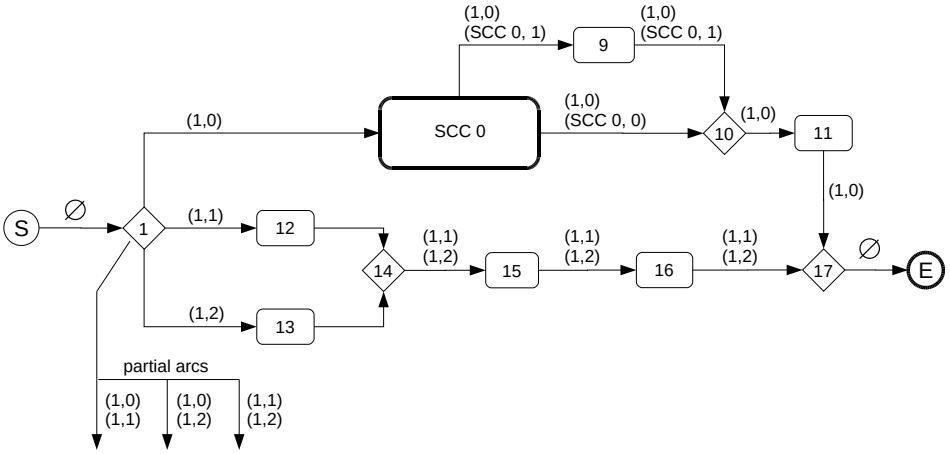


Figure 4.5: Cycle External Flow in the Control Graph.

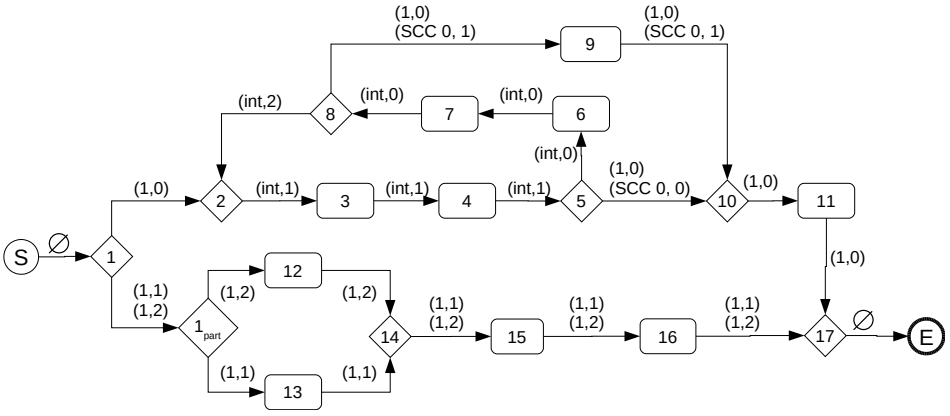


Figure 4.6: Token Flow in the Control Graph.

Figure 4.7 shows the resulting process graph with the final token sets.

In the next step, components are identified by arcs with equal token sets. The pair $(S, 1) \rightarrow (17, E)$ marks the super component $\mathcal{C}_7$ representing the entire scope. The arcs $(1, 2), (10, 11), (11, 17)$ have the same labelling. According to the rules in Sect. 3.6.1, they must form a sequence, therefore the outer pair of arcs $(1, 2) \rightarrow (11, 17)$ belongs to the sequence component $\mathcal{C}_4$. Inside there is one non-trivial sub-component $(1, 2) \rightarrow (10, 11)$ that includes the SCC (cyclic component $\mathcal{C}_3$) and one trivial component, which is omitted. Inside the cycle exist two further sequences $\mathcal{C}_1 : (5, 6) \rightarrow (7, 8)$ and $\mathcal{C}_2 : (2, 3) \rightarrow (4, 5)$ with only trivial subsequences. The partial arc $(1, 1_{part})$ matches the arcs flowing out from 14 and therefore the partial component $\mathcal{C}_5$ is created. The subsequence 15 to 17 contains only trivial components. The whole identified sequence $\mathcal{C}_6$ is $(1, 1_{part}) \rightarrow (16, 17)$.

The final step is the model translation to BPEL. As the correspondence of graph elements to BPEL blocks has now been established, the techniques from Ouyang, Dumas, ter Hofstede

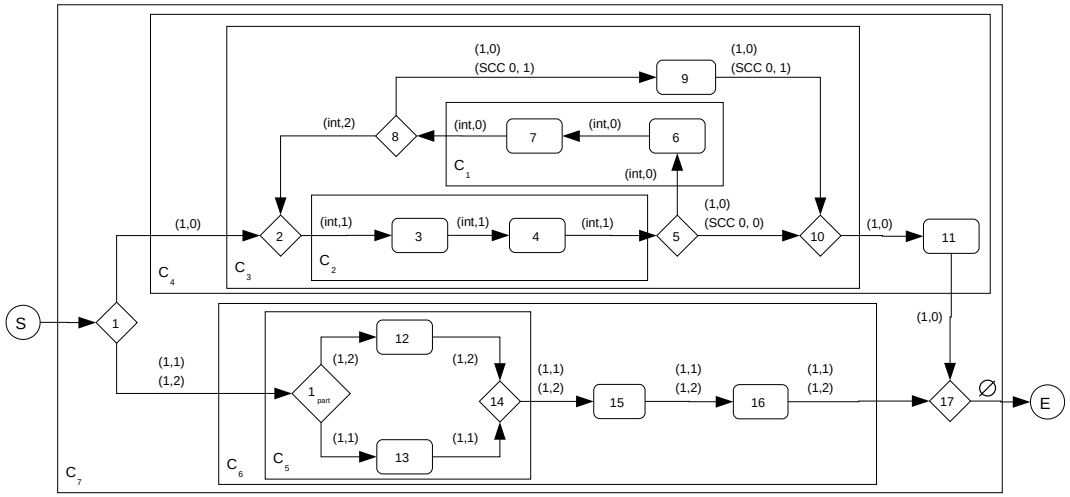


Figure 4.7: Components as Identified by the Token Labels.

& van der Aalst (2006*a*) can be directly applied. The whole process component $\mathcal{C}_7$ maps to a structured BPEL <flow>. One parallel branch consists of the structured sequence component $\mathcal{C}_6$, which contains the structured <flow> sub-component $\mathcal{C}_5$. The fact that $\mathcal{C}_6$ only consists of structured elements is due to the partial convergence. Here the strength of the approach becomes apparent. The simple integration of the gateway detection into the Token Analysis enables for structured, readable output code. If gateway 1 were not splitted, then $\mathcal{C}_6$ could not be detected and all nodes in $\mathcal{C}_6$ would have to be translated in one unstructured acyclic component $\mathcal{C}_7$.

Component $\mathcal{C}_4$ maps to a <sequence> containing $\mathcal{C}_3$ which is unstructured and cyclic. It therefore is translated following the Event-Condition-Action approach in Ouyang, Dumas, ter Hofstede & van der Aalst (2006*a*).

The sub-components $\mathcal{C}_1$ and $\mathcal{C}_2$ each map to <sequence> activities.

5 Conclusion

We have demonstrated how Token-Flow can be used to identify the components of a process graph. The identification works for cyclic graphs, too. Describing the necessary steps, we gathered theory results for cycles, that yield a cycle internal and external view of the flow. The approach also provides a classification of the components (sequence, cyclic, flow) allowing to speed up further translation steps; the actual mapping to code can be achieved with methods described in Ouyang, Dumas, ter Hofstede & van der Aalst (2006*a*), Ouyang et al. (2007). Also, the described approach enables for detection of partial components, that cannot be identified by simply checking the definition of components. Our implementation has already been successfully employed in the workflow code generation framework (Roser et al (2007)) which is e.g. used in AgilPro (`www.agilpro.eu`). The workflow code generation framework provides adapters in order to connect it to different modeling tools, transforms the graph as described above and generates code according to predefined code-templates.

We also see high potential that our work can contribute to improve the bridge between process modelling and workflow execution in practice. Process modelling approaches and tools will benefit, since restrictions on the process models towards execution with BPEL workflow engines can be reduced. The extended component identification also provides a basis to display changes in the workflow code in the process model more easily. This kind of reverse engineering will be a topic of future research. Also, a formal analysis and proof of correctness is left as future work.

Bibliography

Ammarguellat, Z. (1992), 'A control-flow normalization algorithm and its complexity', *Softw. Engin.* **13**, 237–251.

BOC (2007), 'Prozessorientierte Anwendungsentwicklung mit ADONIS : Konzepte, Services und Beratungsleistungen der BOC', Whitepaper.

Gupta, R. & Soffa, M. L. (1978), 'Region scheduling', *2nd Int. Conf. on Supercomp.* pp. 141–148.

Havlak, P. (1997), 'Nesting of reducible and irreducible loops', *ACM Trans. Program. Lang. Syst.* **19**(4), 557–567.

IBM (2006), 'Best practices for using websphere business modeler and monitor', Redpaper. `http://www.redbooks.ibm.com/redpapers/pdfs/redp4159.pdf`.

Jablonski, S. & Bussler, C. (1996), *Workflow Management - Modeling Concepts, Architecture and Implementation*, Thomson Computer Press.

Johnson, R., Pearson, D. & Pingali, K. (1994), The program structure tree: computing control regions in linear time, *in* 'PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation', ACM Press, New York, NY, USA, pp. 171–185.

Koehler, J. & Hauser, R. (2004), Untangling Unstructured Cyclic Flows - A Solution Based on Continuations, *in* 'In Proceedings of the OTM Confederated International Conferences', Vol. 3290 of *LNCS*, Springer, pp. 121–138.

Mendling, J., Lassen, K. & Zdun, U. (2006), 'Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages', *MKWI* **2**, 297–312.

Nagy, R. (n.d.), 'MDD trifft SOA', Whitepaper. `http://www.microtool.de/mt/pdf/objectif/49/mdd_soa.pdf`.

OASIS (2003), *Business Process Execution Language for Web Services (WS-BPEL) Version 1.1.*

Oliver Pera, M. (2007), 'Innovator 2007: Von geschäftsprozessen zu web services oder von business zu bpel', Whitepaper. `http://www.mid.de/fileadmin/documents/pdf/WhitePapers/Innovator_2007_WhitePaper_B2BPEL_070205.pdf`.

OMG (2006), *Business Process Modeling Notation (BPMN) Version 1.0.* `http://www.bpmn.org/Documents/OMG%20Final%20Adopted%20BPMN%201-0%20Spec%2006-02-01.pdf`.

Oracle (n.d.), 'Oracle SOA Suite - Oracle BPEL Process Manager'.

Ouyang, C., Dumas, M., Breutel, S. & ter Hofstede, A. H. (2006), Translating Standard Process Models to BPEL, *in* 'Proceedings of CAiSE 2006', Vol. 4001 of *LNCS*, Springer, pp. 417–432.

Ouyang, C., Dumas, M., ter Hofstede, A. H. & van der Aalst, W. M. (2006*a*), 'From BPMN Process Models to BPEL Web Services', *IEEE ICWS* pp. 285–292.

Ouyang, C., Dumas, M., ter Hofstede, A. H. & van der Aalst, W. M. (2006*b*), 'From Business Process Models to Process-oriented Sofware Systems: The BPMN to BPEL Way', BPM Center Report BPM-06-27.

Ouyang, C., Dumas, M., ter Hofstede, A. H. & van der Aalst, W. M. (2007), 'Pattern-based translation of bpmn process models to bpel web services', *Internat. Journal of Web Services Research JSWR* .

Ramalingam, G. (2000), On loops, dominators, and dominance frontier, *in* 'PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation', ACM Press, New York, NY, USA, pp. 233–241.

Recker, J. & Mendling, J. (2006), 'On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages', *CAiSE 2006 Workshop Proceedings* pp. 521–532.

Roser et al, S. (2007), Generation of Workflow Code from DSMs, *in* 'Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling', Montreal, Canada.

Tarjan, R. E. (1974), 'Testing flow graph reducibility', *J. Comput. Syst. Sci.* **9**, 355–365.

Thomas H. Cormen, Charles E. Leiserson, R. L. R. (1994), *Introduction to Algorithms*, The MIT electrical engineering and comp. sci. series, MIT Press, Cambridge.

van der Aalst, W. M. & Lassen, K. B. (2007), 'Translating Unstructured Workflow Processes to Readable BPEL: Theory and Implementation', *Inf. and Softw. Techn.* .

Wahli, U. et al. (2006), *Business Process Management: Modeling through Monitoring Using WebSphere V6 Products.* IBM Redbook.

White, S. (2005), Mapping BPMN to BPEL example, Technical report, IBM. `http://www.bpmn.org/Documents/Mapping%20BPMN%20to%20BPEL%20Example.pdf`.

Zhang, F. & D'Hollander, E. H. (2004), 'Using hammock graphs to structure programs', *IEEE Trans. Softw. Eng.* **30**(4), 231–245.

List of Figures