Token Analysis of Graph-Oriented Process Models

Mathias Götz, Stephan Roser, Florian Lautenbacher and Bernhard Bauer
Programming Distributed Systems Lab
University of Augsburg
Augsburg, Germany
{*goetz, roser, lautenbacher, bauer*}*@ds-lab.org*

***Abstract*—In Business Process Management, graph-based models are used to represent coordination protocols between collaborating partners as well as for service orchestration and composition. At runtime however, current process engines are commonly based on mainly block-structured languages, such as 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 paper presents a novel approach based on *Token Analysis* to automatically identify components. The technique also allows for optimizations by integrating further steps in the translation of process graphs to executable workflows.**

***Index Terms*—graph-transformation; process; modeling; bpmn; bpel**

I. Introduction

Graph-based models – commonly expressed in the *Business Process Modelling Notation* (BPMN, [1]) – enable intuitive modeling of processes: they are understandable and usable also by non IT-staff. 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. The expressive power of graph-based languages, however, also has some disadvantages. The integrity of the model cannot be ensured on a syntactic level like e.g. by compilers in imperative programming, but requires further analysis.

However at runtime, current workflow engines are often based on imperative block-structured languages. The *Business Process Execution Language for Web Services* (shortly: BPEL, [2]) is a block-oriented language including graph-based control links. Language elements 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 all business analysts.

[3] 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. BPMN is relevant at an early stage of BPM design and BPEL in the late stage of execution. To close the gap between modeling and execution, recent work present transformation strategies.

[4] and [5] provide mappings from sub-graphs of BPMN models – henceforth called *components*– to equivalent BPEL code. Automatic translation requires to determine all components and choose the best mapping strategies for them. For large scaled input, a structured and efficient approach is needed. [1] 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 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 flow analysis for compilers and multi-threaded processor design, are complex and not well adjusted to generating BPEL code.

The objective of this work starts with the basic Token Analysis idea and transforms it into a convenient segmentation algorithm, customized to business process diagrams (BPDs, see [1]) not requiring human input. We aim at building a bridge between the ideas proposed in the papers on token-flow by [1] and the mapping routines by [5]. From the given informal description of the Token concept, we derive a machine-executable algorithm. This involves investigating cyclic graph structures and their handling during the analysis.

The overall result is an automatic and efficient algorithm for the segmentation of (process) graphs into components. The results improve the translation procedure: we introduce a new method called *partial token convergence* to detect additional components that only become available through gateway 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. Our extended Token Analysis approach works for arbitrary graphs, includes optimizations in the translations procedure as well as an enhanced readability of generated code. The implementation of the Token Analysis algorithm, including automatic mapping to BPEL code, has proven that the algorithm successfully integrates into the overall transformation. The developed Java framework can be used to enhance existing modeling tools with BPEL export.

This paper is structured as follows: Section II gives an overview of transformation strategies from graph to block structured models. Section III presents the extended Token Analysis algorithm, which is applied to an example graph in Section IV. Finally, Section V concludes and discusses future work.

II. 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.

[6] uses an *annotated graph*, i.e. annotations at special vertices that refer to portions of successively translated BPEL code. In succeeding steps, subgraphs translate to BPEL. The paper describes several different mapping strategies such as *Element-Preservation*, *Element-Minimization*, *Structure-Identification*, *Structure-Maximization* and *Event-Condition-Action-Rules* for the transformation from a process graph to BPEL.

[7] describes a semi-automatic pattern-based approach which is based on Petri nets. [8] instead shows a fully automatic approach based on event-condition-action rules which is applied to BPMN as input producing BPEL code. The paper describes *event-action* rules, using BPEL event handlers that represent unbounded parallelism by an unbounded number of threads within a <scope> running simultaneously. The strategy is further improved in [5] to a compact representation that can directly be implemented. Although the mappings provide translations for arbitrary graphs, the produced code severely lacks readability. As a remedy, [5] suggests the combination with other translation types for well-structured sub-graphs on the same basis as Structure-Identification in [4]. 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 formalized in detail. Based on the decomposition of a process graph into components, the paper presents an incremental bottom-up overall translation. [9] further extends the translation by mappings for components that are not well-structured, but can be translated without the use of event-handlers. However, the resulting BPEL code is often difficult to maintain due to the use of <links>.

In [10] a control flow normalisation algorithm 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.

[11] identifies all single-entry single-exit (SESE) regions in a graph and structures them in a tree, with larger SESE regions at the top and smaller regions at the bottom. They claim that their algorithm "*runs faster than Lengauer and Tarjan's algorithm*" using cycle equivalence. Their approach does also identify quasi-structured patterns whereas our work seeks only for perfect SESE regions in the first place.

The problem of finding *SESE* graphs in workflows is a concern in compiler construction and parallel programming. Control regions facilitate instruction scheduling for pipelined machines [12]. In the field of interval analysis, irreducible loops are characterized by single entry regions. [13] 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 is 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.

[14] present an algorithm to build a loop nesting tree for arbitrary control flow from which reducible and irreducible regions can be identified. For building the loop tree, the paper uses an extended version of Tarjan's algorithm [15], 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 paper 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.

[16] also computes SESE regions in process models. Therefore, they require that first a process structure tree of the target model is build which can afterwards be traversed in order to generate BPEL-code. Similarly [17] describes a refined version of the process structure tree and how it can be computed for the transformation between BPMN and BPEL. In our approach the SESE regions are computed directly without building the process structure tree first.

As already mentioned, [1] describes informally a mapping based on Token Analysis. The token concept is related to Petri Nets but 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."

[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.

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" [1, pp. 202ff]. Simple loops with an out-degree of two map to <while> activities. 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 paper and the additional example [18] make it clear, that the approach relies on human interaction.

Section III extends this concept to work automatically. Unlike in [1], the method does not stop whenever a component can be identified, but calculates the labeling in one run (token-flow) from which component boundaries are deduced (Token Analysis). Quasi-structured patterns as in [9] can be identified by introducing *Partial Token Convergence*. All concepts in this paper are oriented towards a seamless integration of the mappings described in [5], [9].

III. Algorithms

The following presents a formal description of the token-flow algorithm that grounds on the concepts outlined in [1]. First, we summarize basic mathematical foundations about set theory. The problem that cycles cause deadlocks in this algorithm is investigated in Sect. III-B, and *contraction* is proposed as a solution. The developed techniques to handle arbitrary cyclic graphs are formalized in Sect. III-C. Finally, Sect. III-D shows how the components are derived covering maximum sequences and partial convergence.

Much of the used notation is adapted from [5] except that the discussion is not restricted to BPDs, but rather based upon directed graphs. This develops more general results that can be applied to different problems.

Let S be some set. $\mathcal{P}$ is the power 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:

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

A *directed graph* is a tuple (V, A). V contains the *vertices*, $A \subseteq V \times V$ the directed *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 we 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 w.l.o.g. 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 III.1 (Cycle)**.** A path $c = v_1, \ldots, v_n$ forms a *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 paper we do not consider graphs that contain unreachable cycles. We assume that all vertices are reachable from the start-vertex. Components are connected subsets with the following properties.

Definition III.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 III.1. *For a component $\mathcal{C}$ and for all $v \in V_\mathcal{C}$:*

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

Proof: 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, e.g. the graph in Fig. 1 would contain a component with $\mathsf{source} = a_1$ and $\mathsf{sink} = a_2$).

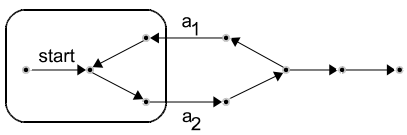

Fig. 1. Components do not contain start events.

A. Token-flow Algorithm

We now give a precise and formal description of the token *propagation* mechanism described in [1]. The token-flow is calculated in two steps: first, 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. To each arc a subset of tokens called *token labeling* is assigned. 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 labeled with the same token: At vertices with out degree > 1, new tokens are created. The out-arcs are labeled with the union of the arriving token sets and the newly generated tokens. At merging gateways, the out-arc is labeled 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 labeled. This strategy performs a blocking wait, and it does not work for arbitrary graphs. Deadlocks can only occur at cycles. They can be separated from the graph by applying the techniques described in Section III-B.

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 labeling (re-combination). Components can be derived by matching pairs of arcs with equal token sets. Different to [1], the process does not stop whenever a component is encountered but continues until all the arcs have been labeled. The procedure does not have to start again, and enables the recognition of more advanced, interleaved structures.

We now define the set of *tokens* $\mathbb{T}$. A token carries information on the parallelization for which the token was generated, i.e. the origin vertex v and a number i referring 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 labeling function* $t : A \to \mathcal{P}(\mathbb{T}) \cup \{\bot\}$. Token creation occurs at vertices with $|out(v)| > 1$. Tokens propagate and unite at vertices with $|in(v)| > 1$. Figure 2 illustrates the flow of tokens created at vertices 1 and 2. After all arcs have been labeled, tokens originating from the same vertex *converge* and are removed from the labeling (indicated in the figure by the curly brackets).

Finally, the labeling is used to determine the components of the graph. If for two arcs $a, b \in A, a \neq b : t(a) = t(b)$ holds, they mark the beginning and the end of a component $\mathcal{C}$, i.e. $\{a, b\} = \{\mathsf{source}(\mathcal{C}), \mathsf{sink}(\mathcal{C})\}$.

The resulting components can be overlapping, ambiguous, and include trivial components. In Fig. 2 the converged labeling $\{(1,1)\}$ appears at four arcs. Each pair of these arcs marks

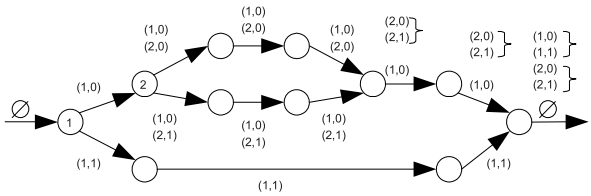

Fig. 2. Example of converging token-flow

Algorithm 1 Token-flow

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

1: **processVertex**$(v \in V)$ {
2: $\mathcal{P}(\mathbb{T})\ M := \bigcup_{u \in t(\mathsf{in}(v))} u$;
3: **if** $|\mathsf{out}(v)| = 1$ **then**
4: $t(\mathsf{elt}(\mathsf{out}(v)) := M$;
5: **else**
6: **int** $i := 0$;
7: **for each** $a \in \mathsf{out}(v)$ **do**
8: $t(a) := M \cup \{(v, i++)\}$;
9: **end for**
10: **end if**
11: }
12:
13: **tokenFlow**(t, N) {
14: **while** $N \neq \emptyset$ **do**
15: **pick**$(v \in N \mid t(\mathsf{in}(v)) \neq \bot)$;
16: **processVertex**(v);
17: $N = N \backslash v$;
18: **timestamp**(v);
19: **end while**
20: **for each** $a \in A$ **do**
21: **for each** $v \in \mathsf{first}(t(a))$ **do**
22: **if** $t(a) \supseteq \{(v,i) \mid i < |\mathsf{out}(v)|\}$ **then**
23: $t(a) = t(a) \backslash \{(v,i) \mid i < |\mathsf{out}(v)|\}$
24: **end if**
25: **end for**
26: **end for**
27: }

a valid component, but not all of them are desirable. This can be avoided by a strategy presented in Sect. III-D which shows how to derive the correct partitioning, covering sequences, and further component types.

Algorithm 1 summarizes the token-flow procedure. For an acyclic graph with start node $start$, call the **tokenFlow** function with an initial labeling $t(start) := \emptyset$, and with $N := V \backslash \{start\}$. The function **pick**$(e \in E \mid prop : e \to \mathbb{B})$ non-deterministically 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 labeled 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, a new token is assigned to each arc in line 8. After processing, to each arc a timestamp (line 18) is assigned, which is later used to identify sequence-boundaries (see Sect. III-D). Once the main loop in **tokenFlow** terminates, converged tokens are removed (line 22).

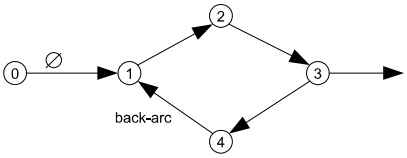

Fig. 3. A deadlock at node 1

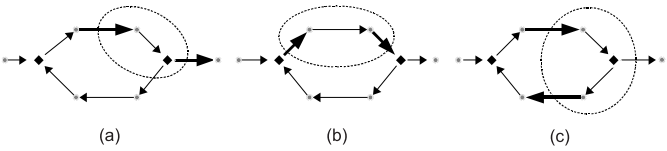

Fig. 4. Component boundaries do not cross cycle boundaries

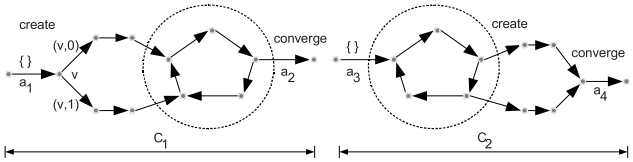

Fig. 5. Cycle external flow

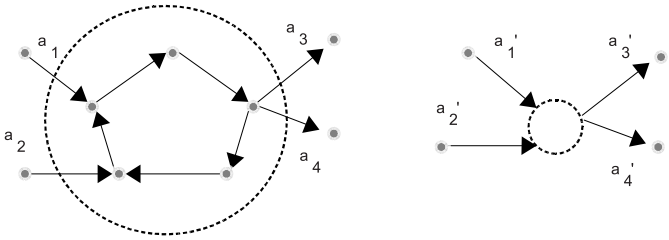

Fig. 6. A cycle is *contracted* to a replacement vertex

B. Contracting Cycles

Algorithm 1 only works for acyclic graphs. Vertex 1 in the cycle in Fig. 3 cannot be processed until the back-arc $(4, 1)$ has been labeled; this would require the flow leaving vertex 1 to be processed first, as it must pass along the vertices 2, 3, and 4. The result is 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. This section introduces a method to avoid this deadlock problem.

Theorem III.1. *Let c be a cycle, $\mathcal{C} = (V_\mathcal{C}, A_\mathcal{C})$ a component. Then*

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

Theorem III.1 provides an important insight into the meaning of cycles for the algorithm (the proof can be found in [19]). It states that source and sink of a component are never positioned across the boundaries of any cycle. For example, in Fig. 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. To achieve this, two requirements must be met: *i)* the flow around cycles (*external*) and *ii)* the flow inside cycles (*internal*) must lead to a correct identification of components.

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

1) Cycle-External Flow: Figure 5 shows two components $\mathcal{C}_1$ and $\mathcal{C}_2$ containing (simple) cycles. Each source and sink pair must carry the same token labeling: $t(a_2) \stackrel{!}{=} t(a_1) = \emptyset \ \wedge \ t(a_4) \stackrel{!}{=} t(a_3) = \emptyset$.

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. Because the flow inside the cycle is independent of the exterior, this is possible if all incoming flows merge when entering 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 labeling. In order to keep the branches separate the tokens must 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 6 illustrates the process: each arc a_i is replaced by an arc a_i' connected with the 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 $\mathsf{from}(a) := c$. Contraction is only a temporary process to determine the token labelings. Storing the original references of the replaced arcs allows to restore them after the algorithm has finished. Thus, the original input graph can be mapped to its corresponding components.

2) Cycle-Internal Flow: After contraction, external flow does not arrive inside cycles. A kind of bootstrapping internal flow is required. We define an *initial labeling* 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

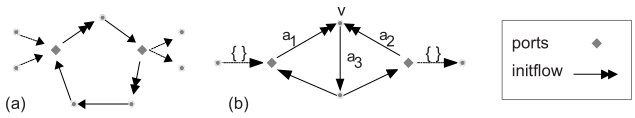

Fig. 7. A cycle with its ports and initflow (a). Internal tokens do not converge (b)

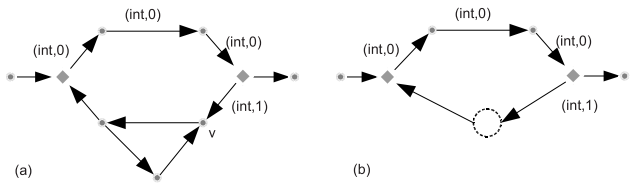

Fig. 8. An embedded cycle (a) and its deadlock resolution (b)

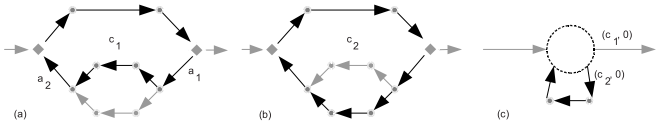

Fig. 9. An SCC c consisting of the simple cycles c_1 and c_2

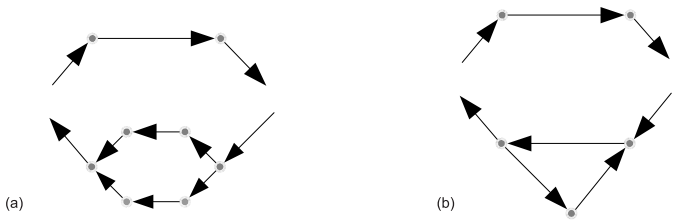

Fig. 10. The SCCs without the ports

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. 4(c)): if a port would be included in a component, then (because of its connection to the exterior) some vertices of the environment would need to be included, too. Theorem III.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, thus we set the initial labeling 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 7(a) depicts the ports and initflow arcs of a simple cycle. The initial tokens must not converge, as can be seen in Figure 7(b): if the initial labelings $t(a_1)$ 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.

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 labeled with different int-tokens. In Figure 7(b) for instance, the labeling is $t(a_1) = \{(\mathsf{int}, 0)\}$ and $t(a_2) = \{(\mathsf{int}, 1)\}$.

3) Substructure of Cycles: A cycle s is called *strongly connected component* (SCC), if it is maximal, i.e. if for all cycles $c : \mathsf{vertices}(c) \cap \mathsf{vertices}(s) \neq \emptyset \Rightarrow \mathsf{vertices}(c) \subseteq \mathsf{vertices}(s)$. SCCs and sub cycles can consist of several simple cycles. Contracting the SCC in Fig. 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. 8 (b).

Only contracting all simple cycles, however, is not always sufficient. The cycle in Fig. 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 labeling in this case produces correct results. Only contracting the whole SCC is required.

As we have seen, contracting nested cycles is only appropriate, if they cause internal deadlocks. The sub cycles that cause deadlocks can be determined by *taking the ports from a cycle* (their leaving flow has already been labeled) 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. 9 opens up both simple cycles c_1 and c_2; no sub-cycle is left, thus only the top SCC is contracted. In Fig. 8(a) the sub-cycle does not contain ports and thus is contracted. The remaining subgraphs are shown in Fig. 10.

C. 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 [20]. 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 labeled 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. III-B.

D. Exploiting Results for Component Identification

From the arc labeling, components can be derived. 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

Algorithm 2 Mark initial flow and contract SCCs.

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

```
Init: N := V\{start};
t(start) := ∅;
int i := 0; // counter for int tokens
contractAll(V);

contract(c ∈ V*){
V v; //new replacement vertex
V = V ∪ {v};
for each a ∈ in(c) do
    A = A ∪ (from(a), v);
    A = A \a;
end for
for each a ∈ out(c) do
    A = A ∪ (v, to(a));
    A = A \a;
end for
return v;
}

contractAll(V' ⊆ V){
for each s ∈ SCCs(V') do
    V v := contract(s);
    for each a ∈ initflow(s) do
        t(a) := {(int, i++)};
    end for
    N = N \ports(s);
    contractAll(vertices(s)\ports(s));
end for
}
```

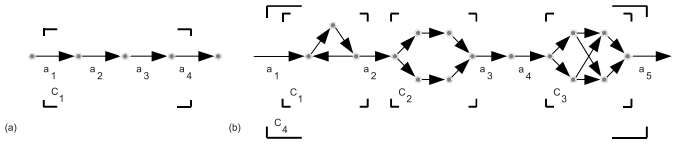

Fig. 11. Sequence Components

branches). The only case where this might occur is within sequences. In this section we describe their handling and extend the token concept to enable the detection of further structures.

1) Maximum Sequences: Two arcs with equally labeled token sets are the 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 represent a *sequence*. Figure 11(a) depicts a simple chain with the labeling $t(a_1) = t(a_2) = t(a_3) = t(a_4)$. 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: $\mathsf{source}(\mathcal{C}_1) = a_1, \mathsf{sink}(\mathcal{C}_1) = a_4$.

Sequences can also contain subcomponents, like $\mathcal{C}_4$ containing $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ in Fig. 11(b). Each subcomponent is located between a pair a_i, a_{i+1} of arcs. 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 summarize the component identification for arcs with equal labelings:

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

Token Analysis yields a classification of components into 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 [5].
- Acyclic components translate to <*switch*>, <*pick*> and <*flow*> activities (flow pattern-based translation in [9] can be used here), and
- Sequence components.

2) Partial Convergence: Token-analysis can be extended to identify additional components, outlined by *quasi-structured translation* by [9]. The BPD in Fig. 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. 12(b): the intermediate arc labeled $a \vee b$ is created by *splitting the gateway*. To identify such components we introduce *partial token convergence*.

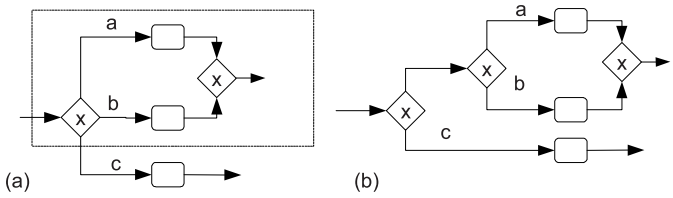

Fig. 12. Dividing a gateway into sub-branches

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 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 labeling in the token-flow. This introduces a new component and improves the structure of the graph decomposition.

Figure 14 shows the token-flow for splitting a gateway with multiple out-arcs (i) and multiple in-arcs (ii), as in the

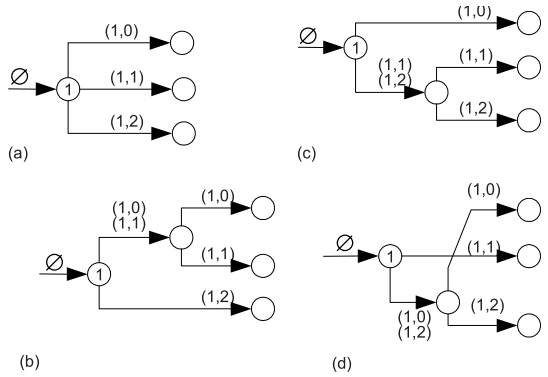

Fig. 13. Virtual Token Sets of Splitted Gateways

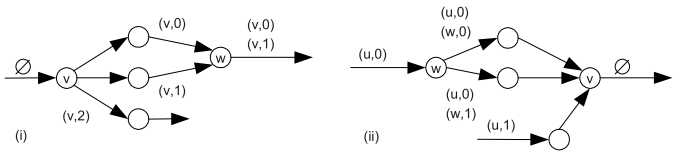

Fig. 14. Requirements for partial convergence

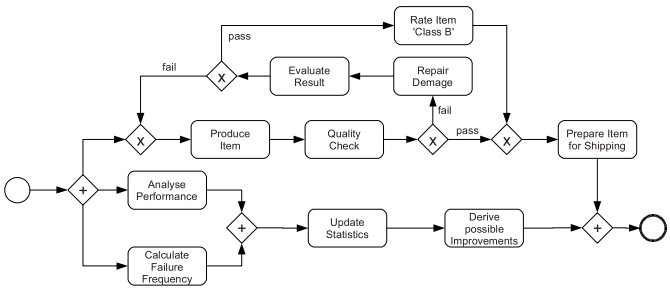

Fig. 15. A BPD for Quality Control of Production

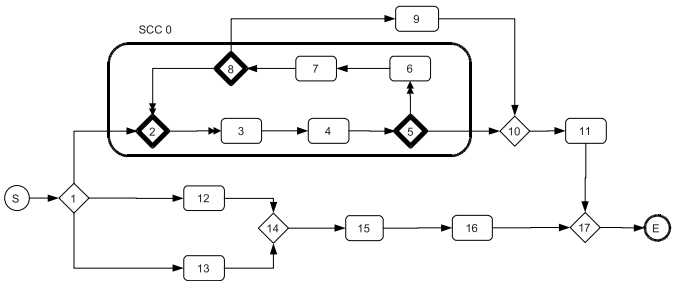

Fig. 16. One SCC in the Quality Control Graph

situation in Fig. 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. 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 of 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.

Generally, at gateways with three or more entering or leaving arcs, each such intermediate token set must be considered. In the following we describe 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 (where j is a running index for the subsets)

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

Case *ii)* $|\mathsf{in}(v)| > 2$. At a merging gateway, the combinations of entering sets must be considered, i.e. 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. 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*.

IV. Use Case

In the following we describe our approach utilizing our proof-of-concept implementation. It can be downloaded from http://sourceforge.net/projects/tokenanalysis/. We present the mapping steps from the BPD input model to the graph-based model for BPEL and use a realistic example to give an overview of how techniques described in Sect. III work together. We show the single steps of the transformation with a sample process graph for quality control in Fig. 15.

The first step in analysis is according to Sect. III-B 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. 16). Taking the ports from the only SCC in Fig. 16 leaves no internal SCCs, as shown in Fig. 17a. 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. 17b.

The token-flow algorithm is launched for the initial labeling of the SCC and an $\emptyset$ label at $(S,1)$. First consider cycle external flow. Figure 18 shows the flow around the contracted vertex $SCC\ 0$ and the token configurations for the arcs, after token convergence. Three tokens are created at vertex 1. Therefore,

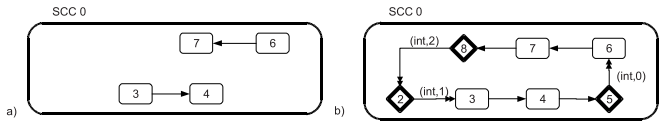

Fig. 17. The SCC without its Ports and the init flow for the SCC

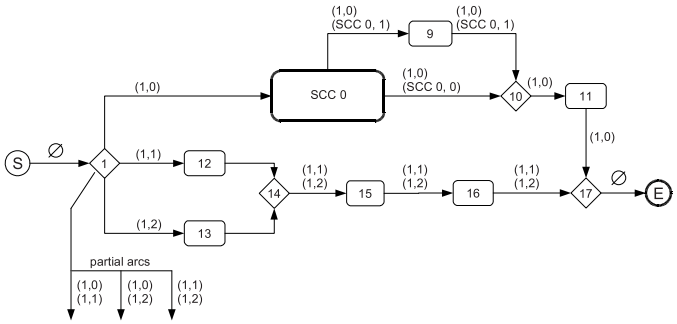

Fig. 18. Cycle External Flow in the Control Graph

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 two 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. 19, together with the remaining internal flow. Finally, all branches merge back at node 17. In the cycle internal flow no branching takes place. Therefore, the tokens simply propagate up to the next ports. Fig. 20 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 labeling. According to the rules in Sect. III-B, 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

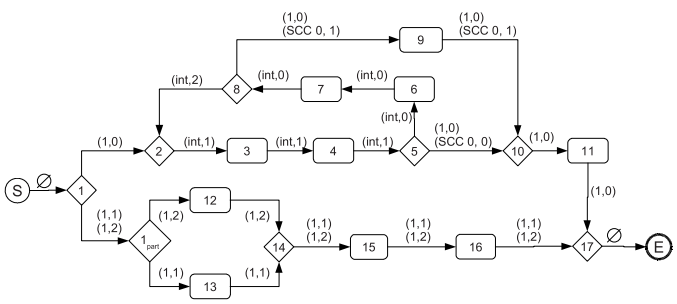

Fig. 19. Token-flow in the Control Graph

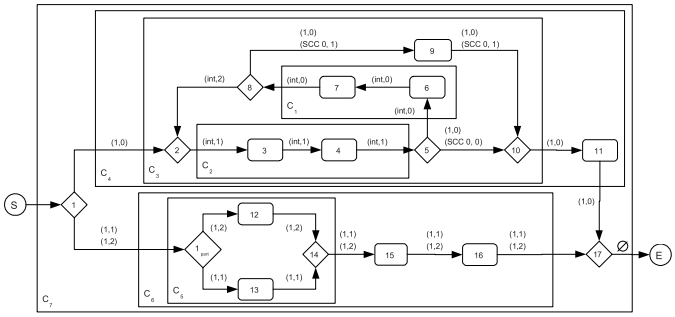

Fig. 20. Components as Identified by the Token Labels

$(1,2) \rightarrow (10,11)$ 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 sub-sequences. The partial arc $(1,1_{part})$ matches the arcs flowing out from 14 and 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 generation of BPEL code. As the correspondence of graph elements to BPEL blocks has now been established, this can be done using templates. 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 in the Token Analysis algorithm enables structured, readable output code. If gateway 1 had not been splitted, then $\mathcal{C}_6$ could not be detected and all nodes in $\mathcal{C}_6$ would have to be translated in an unstructured acyclic component $\mathcal{C}_7$.

V. Conclusion

We have demonstrated how Token Analysis can be used to identify the components of a process graph. The identification works for cyclic graphs, too. Describing the necessary steps, we gathered theoretical 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 [5], [9]. Also, the described approach enables the 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 [21] which is e.g. used in the AgilPro process suite (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. This allows (different to [16]) to switch to other modeling tools (such as a BPMN modeler)

easily and to change the transformation templates from BPEL 1.1 to BPEL 2.0 very quick.

We see high potential that our work can contribute also to other research areas besides the generation of block-structured code (such as BPEL). We already applied it for the validation of process models (similar to [22]) as well as for an adaptation mechanism of existing process models.

The extended component identification provides also a basis to display changes in the workflow code in the process model more easily. Such reverse engineering as well as formal analysis and proof of correctness is topic of future research.

REFERENCES

[1] OMG, "Business Process Modeling Notation Specification, Version 1.2," formal/09-01-03, January 2009, http://www.omg.org/spec/BPMN/1.2.

[2] OASIS, *Business Process Execution Language for Web Services 1.1*, 2003.

[3] J. Recker and J. Mendling, "On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages," *CAiSE, Luxembourg*, vol. 4001, June, 5-9 2006.

[4] J. Mendling et al, "Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages," in *MKWI*, vol. 2, 2006, pp. 297–312.

[5] C. Ouyang et al, "From BPMN Process Models to BPEL Web Services," *IEEE ICWS*, pp. 285–292, 2006.

[6] J. Mendling et al, "On the Transformation of Control Flow between Block-oriented and Graph-oriented Process Modeling Languages," *Int. J. Business Process Integration and Management*, vol. 3, no. 2, September 2008.

[7] W. van der Aalst and K. B. Lassen, "Translating Unstructured Workflow Processes to Readable BPEL," *Information and Software Technology*, vol. 50, no. 3, pp. 131–159, 2008.

[8] C. Ouyang, M. Dumas, S. Breutel, and A. H. ter Hofstede, "Translating Standard Process Models to BPEL," in *Proceedings of CAiSE 2006*, ser. LNCS, vol. 4001. Springer, 2006, pp. 417–432.

[9] C. Ouyang, M. Dumas, A. H. ter Hofstede, and W. M. van der Aalst, "Pattern-based Translation of BPMN Process Models to BPEL Web Services," *Internat. Journal of Web Services Research (JSWR)*, 2007.

[10] J. Koehler et al, "Declarative techniques for model-driven business process integration," *IBM Systems Journal*, vol. 44, no. 1, 2005.

[11] R. Johnson, D. Pearson, and K. Pingali, "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*. New York, NY, USA: ACM Press, 1994, pp. 171–185.

[12] R. Gupta and M. L. Soffa, "Region scheduling," *2nd Int. Conf. on Supercomp.*, pp. 141–148, 1978.

[13] F. Zhang and E. H. D'Hollander, "Using Hammock Graphs to Structure Programs," *IEEE Trans. Softw. Eng.*, vol. 30, no. 4, pp. 231–245, 2004.

[14] P. Havlak, "Nesting of reducible and irreducible loops," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 4, pp. 557–567, 1997.

[15] R. E. Tarjan, "Testing flow graph reducibility," *J. Comput. Syst. Sci.*, vol. 9, pp. 355–365, 1974.

[16] L. Garcia-Banuelos, "Pattern identification and classification in the translation from BPMN to BPEL," in *OTM 2008, Part I*, ser. LNCS, R. Meersman and Z. Tari, Eds., no. 5531. Springer-Verlag Berlin Heidelberg, 2008, pp. 436–444.

[17] J. Vanhatalo, H. Völzer, and J. Koehler, "The refined process structure tree," in *Proceedings of BPM 2008*, ser. LNCS, M. Dumas, M. Reichert, and M.-C. Shan, Eds., no. 5240. Springer-Verlag, 2008, pp. 100–115.

[18] S. A. White, "Using BPMN to Model a BPEL Process," *BPTrends*, March 2005.

[19] M. Götz, S. Roser, F. Lautenbacher, and B. Bauer, "Using Token Analysis to Transform Graph-Oriented Process Models to BPEL," University of Augsburg, Tech. Rep., 2008, TR 2008-08.

[20] T. H. Cormen, C. E. Leiserson, R. L. Reivest, and C. Stein, *Introduction to Algorithms*, ser. The MIT electrical engineering and Comp. Sci. Series. Cambridge: MIT Press, 2001.

[21] S. Roser, F. Lautenbacher, and B. Bauer, "Generation of Workflow Code from DSMs," in *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, Montreal, Canada, 2007.

[22] J. Vanhatalo, H. Völzer, and F. Leymann, "Faster and more focused control-flow analysis for business process models through SESE decomposition," in *ICSOC*, 2007.