Understanding and Decomposing Control-Flow Loops in Business Process Models

Thomas M. Prinz[1], Yongsun Choi[2](✉), and N. Long Ha[3]

[1] Course Evaluation Service, Friedrich Schiller University Jena, Jena, Germany
Thomas.Prinz@uni-jena.de

[2] Department of Industrial and Management Engineering, Inje University, Gimhae, Republic of Korea
yschoi@inje.edu

[3] Faculty of Economic Information Systems, University of Economics, Hue University, Hue, Vietnam
hnlong@hueuni.edu.vn

Abstract. Business process models are usually described in a visual notation and reflect actual processes in systems. As a result, process models often are unstructured and cyclic. Unfortunately, unstructured and cyclic models are difficult to analyze and execute as research shows. Unstructuredness could be overcome using the existing studies, however, the analysis of cyclic models is still an open research problem. For this reason, this paper presents a decomposition of cyclic process models into sets of acyclic models. Together with a simple execution semantics for the acyclic models, the semantics of the decomposed model coincides with the original model if soundness is assumed. The decomposition can be achieved in a quadratic runtime complexity and gives the possibility to apply many existing analysis methodologies for acyclic process models. A short evaluation shows the feasibility of the approach.

Keywords: Process models · Loops · Decomposition · Soundness

1 Introduction

Business Process Management (BPM) examines how different operations (tasks) interact to achieve business goals [7]. This interaction is usually recorded in the form of *business process models* in special notations (like the *Business Process Model and Notation* (BPMN) [19]). Most of these process notations are visual (e.g., BPMN and Event-driven Process Chains (EPC) [14]) and the process models result from actual operational processes in systems and organizations. Therefore, such models may be unstructured and contain non-explicit *loops*, i.e., loops that result from the control-flow rather than from specific looping tasks.

The third author's work was done during his PhD program at Inje University.

C. Di Ciccio et al. (Eds.): BPM 2022, LNCS 13420, pp. 307–323, 2022.
https://doi.org/10.1007/978-3-031-16103-2_21

Unstructuredness and loops make the execution and analysis of process models difficult [2,3,8,16,17,20,22,23,30,31]. Polyvyanyy summarizes that well-structured process models are more comprehensible for humans, are more likely to contain fewer errors, and, therefore, improve their quality [20]. Arbitrary loops tend to increase the probability of errors in process models [17] and prevent the structuring of process models or at least increase the effort [20]. The *(Refined) Process Structure Tree* (RPST) describes a hierarchy of single entry and single exit (SESE) structures [2,23,30,31] that is often used to find independent structures in unstructured process models [31] and to speed up analysis [6,11], but do not help to solve the problem of *unstructured and cyclic* process components [3]. It appears that loops make the execution and analysis of process models particularly difficult. There are many efficient and simple approaches to execute and analyze processes without loops (*acyclic* processes). For example, inclusive converging gateways (OR-joins) [32] and *Dead-Path Elimination* (DPE) for WS-BPEL [18] are easy to apply and understand for acyclic process models. *Soundness* analyses [10,26] and the derivation of *behavioral relations* [11] are examples of how analyses in acyclic processes are efficient and simple. However, most of these approaches cannot be applied to cyclic processes or become more complicated to understand, implement, and prove.

Due to a large number of known state-of-the-art approaches and simpler execution semantics (e.g., for OR-joins) for acyclic process models, it would be beneficial for research and practice if there would be a transformation from a cyclic model to an acyclic model with the same execution behavior. One possibility is to transfer a process model into an RPST [2,23,30,31]. Since the RPST is a tree, it is acyclic. However, the RPST provides less information if loops are inside inherently unstructured fragments (so-called *rigids*) [20] and can, therefore, not be used in such cases. *Untanglings* [22] and *unfoldings* [8,16,20] of process models are another way of making process models acyclic. However, they use completely different representations of the original process and are not applicable to process models with OR-joins. For this reason, there is the need for a new approach to decompose an arbitrary process model with OR-joins into a set of acyclic process models while retaining its execution behavior. This paper introduces such an approach. Our approach follows ideas of Choi et al. [3], but it is applicable to process models with OR-joins and does not need to distinguish between *natural* and *irreducible* loops. A *natural* loop can only be entered at one node, while an *irreducible* loop can be entered at multiple nodes. In many situations, irreducible loops are more difficult to handle [3]. Our approach detects all loops recursively and creates acyclic versions of them. Finally, the loops in the original process are reduced to acyclic subgraphs combined with "*looping nodes*" being available in most modeling languages. If the original model is *sound*, this paper shows that the resulting acyclic processes have the same behavior as the original process. Although unsound processes exist in practice, Van Dongen et al. state that a process model should at least fulfill soundness as correctness criterion [5]. From a quality perspective, soundness is, thus, a weak constraint.

The decomposition proposed in this paper gives the following non-final improvements for research and practice regarding the state of the art: (a) It is applicable to any cyclic process model with natural and irreducible as well as nested loops. (b) The decomposition can be applied to process models with OR-splits and OR-joins. (c) Process models with complicated loop behavior should become easier to understand for humans, because they are decomposed into smaller, acyclic models. (d) A cyclic process model only needs to be transformed once. The results are again process models that can be stored and executed or analyzed as needed. (e) The execution of process models becomes trivial even with OR-joins. Actually, the kind of converging gateway becomes useless, as all converging gateways can be replaced by OR-joins. (f) Process models can be executed on any execution engine that supports acyclic processes with (sub)process calls. (g) Structuring and analyzing acyclic process models is easier and, above all, more efficient than with cyclic process models as explained before.

In summary, the contributions of this paper are: (1) Study of loops and how they can be generalized. (2) Decomposition of cyclic process models into sets of acyclic models in general process models. (2) Introduction of a semantics that executes resulting acyclic processes in the correct order to achieve the business goals of the original model (same execution semantics). (3) A short evaluation of loop decomposition regarding complexity and feasibility as well as two example applications: OR-join semantics and soundness analysis.

The rest of this paper is structured as follows: Sect. 2 introduces necessary definitions of process models, loops, and their semantics. This section is followed by Sect. 3 on related work. In Sect. 4, we describe our findings on loop structures in process models followed by a detailed description of their decomposition in Sect. 5. Section 6 describes how a decomposed process model can be executed. The complete decomposition algorithm is presented in Sect. 7. An evaluation of our loop decomposition as well as two example applications are shown in Sect. 8. Finally, Sect. 9 gives some suggestions for future research directions.

2 Preliminaries

The following section introduces notions being important to understand this work, in particular, *workflow graphs*, *loops*, *token games*, and *soundness*.

2.1 Workflow Graphs and Loops

In order to abstract process models from rich representations such as BPMN [19] and EPC [14], this paper uses *workflow graphs* [26,32]:

Definition 1 (Workflow Graph). A workflow graph $WFG = (N, E, \lambda, L)$ is a connected, directed graph (N, E). N is a set of nodes and $E \subseteq N \times N$ is a set of edges which defines the execution order between nodes. L is a set of labels $\{Start, Task, AND, OR, XOR, End\}$ and $\lambda \colon N \mapsto L$ is a total mapping that assigns to each node a label. Depending on its label, a node has different properties:

- Nodes with label *Start* (the *start nodes*) have no incoming but exactly one outgoing edge. Nodes with label *End* (the *end nodes*) have exactly one incoming but no outgoing edge. There is at least one start and one end node.

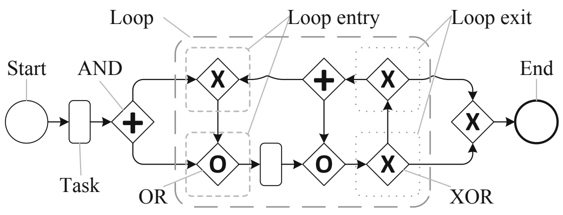

Fig. 1. A workflow graph with an irreducible loop with its loop entries and exits.

– Each node lies on a path from a start to an end node.
– Nodes with label *Task* have exactly one incoming and one outgoing edge.
– All other nodes have labels *AND*, *OR*, and *XOR*. They are divided into *split* and *join nodes*. Split nodes (i.e., AND-split, OR-split, and XOR-split) have exactly one incoming and at least two outgoing edges. Join nodes (i.e., AND-join, OR-join, and XOR-join) have at least two incoming edges and exactly one outgoing edge.

For the visualization of workflow graphs, we use notions of BPMN [19] as shown in the workflow graph in Fig. 1. Start and end nodes are represented as circles, with end nodes having thicker lines. Tasks are represented as rounded rectangles. Split and join nodes are depicted with diamonds. AND-splits and AND-joins have + signs, XOR-splits and XOR-joins have a cross, and OR-splits and OR-joins have circles in their diamonds. Later in this paper, we include special *loop nodes* (represented as rectangles with a circular arrow) in workflow graphs, informally extending Definition 1. They have at least one incoming edge and at least one outgoing edge. Details are explained during their introduction.

Loops in workflow graphs are special subgraphs:

Definition 2 (Loop/Cycle). Let $WFG = (N, E, \lambda, L)$ be a workflow graph.
A *loop* is a strongly connected component (SCC) [4] $\mathcal{L} = (N_{\mathcal{L}}, E_{\mathcal{L}})$ of *WFG* containing at least two nodes, i.e., each node of $\mathcal{L}$ has a path to each other node of $\mathcal{L}$ [1]. If *WFG* contains any loop, it is called *cyclic*. Otherwise, it is *acyclic*.

Usually, loops in workflow graphs contain at least one task node. It is important to note that a SCC is *maximal* by definition [4]. Consequently, our loops do *not* contain subloops in the classical sense. However, the approach presented in this paper later shows that *it identifies loops within loops if necessary*. A similar approach was taken by Steensgaard [28]. He also generalized the definitions of *loop entries* and *exits*:

Definition 3 (Loop Entries and Exits). Let *WFG* be a cyclic workflow graph with $\mathcal{L} = (N_{\mathcal{L}}, E_{\mathcal{L}})$ is one of its loops.

Loop Entry. All nodes of $\mathcal{L}$ that have at least one incoming edge from outside $\mathcal{L}$ are called *loop entries* and these incoming edges are *loop-entry edges*.
Loop Exit. All nodes of $\mathcal{L}$ that have at least one outgoing edge to a node outside the loop are called *loop exits* and these outgoing edges are *loop-exit edges*.

Remark 1. Each loop entry of a loop in a workflow graph is of course a join node and each loop exit of a loop is of course a split node.

Figure 1 shows a loop (gray large dashed rounded rectangle) in our example workflow graph and its loop entries (small gray rounded rectangles on the left) and loop exits (small gray rounded rectangles on the right).

Depending on the number of loop entries, two types of loops are distinguished in loop research: *natural* and *irreducible loops*. Although natural loops are more intuitive, there is no need to distinguish between the two in this paper. Finally, it is also important to note that loop-exit edges are not part of any loop:

Corollary 1. Since a loop is a maximal SCC by Definition 2, loop-exit edges cannot be part of any loop.

2.2 Semantics

The semantics of cyclic workflow graphs with OR-joins is not trivial [32]. The reason is that situations can arise where two OR-joins mutually wait for each other [15,32]. In this paper, we refer to the semantics of our previous work [24] that is complete for *sound* workflow graphs. However, *we emphasize that an additional OR-join semantics for cyclic workflow graphs is not needed after decomposition and is not required in detail in the proofs*. Therefore, it is only used here for the sake of completeness. We use a *token game* semantics in the following describing state transitions in a workflow graph $WFG = (N, E, \lambda, L)$.

A *state* S of WFG is a total mapping from the set of edges E to the set of natural numbers, $S\colon E \mapsto \mathbb{N}_0$. It describes the number of *tokens* on each edge, e.g., $S(e) = 1$ means that edge e in state S carries 1 token. An *initial state* of WFG is a state in only one outgoing edge of exactly one start node has a token. Every other edge has 0 tokens.

A node n of WFG is *waiting* in a state S if at least one incoming edge of n has a token. If n is neither an AND-join nor an OR-join, n is *enabled* if it is waiting in S. If n is an AND-join and all incoming edges of n carry a token in S, n is *enabled* in S. If n is an OR-join, then it has a waiting area $\omega(n)$ that contains all edges where n must wait for their tokens (for more details, please take a look on Prinz and Amme [24]). n is enabled in S, if it is waiting in S and no token is in n's waiting area except on n's incoming edges. If there is an enabled node n in S, then S can change into a state S' by executing n, written $S \overset{n}{\rightarrow} S'$. The resulting state S' is based on S with the following modifications: (1) Each incoming edge *in* of n with at least one token loses a token in S', except if n is an XOR-join, then only one incoming edge loses a token. (2) The number of tokens on n's outgoing edges depends on n's type. If n is an OR-split, then a non-empty set of outgoing edges of n gets an extra token in S'. If n is an XOR-split, then exactly one outgoing edge of n gets an extra token in S'. Otherwise, each outgoing edge of n gets an extra token in S'.

The execution of a workflow graph starts with an initial state, and is executed node by node, resulting in a chain of node executions and state transitions.

A state S' is *directly reachable* from a state S, depicted $S \rightarrow S'$, if there is a possible state transition $S \overset{n}{\rightarrow} S'$, i.e., node $n \in N$ is executed in state S. S' is *reachable* from a state S, depicted $S \rightarrow^* S'$, if there is a sequence/chain of directly reachable states $S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_k$, $k \geq 1$, with $S_1 = S$ and $S_k = S'$.

2.3 Soundness

Two types of structural conflicts can occur in workflow graphs, namely *deadlocks* and *lacks of synchronization* [9]. A deadlock arises from an initial state when an AND-join or OR-join is waiting in a reachable state S, but is never enabled in reachable states from S [9]. A lack of synchronization results from an initial state when an edge carries more than one token in a reachable state. A workflow graph is said to be *sound* if it has neither a deadlock nor a lack of synchronization [9]. This soundness is defined on workflow graphs with OR-joins. In our previous work, we showed that no other semantics allows running more workflow graphs sound than the one used here [24]. In the context of the present work, we assume that each XOR- and OR-split independently decides after their execution which outgoing edges receive an additional token.

3 Related Work

Finding and restructuring loops (cycles) in graphs has a long tradition in compiler theory. Tarjan defined an efficient algorithm to find cycles (SCCs) in arbitrary graphs [29]. Since one main application of finding cycles in compiler theory is optimization [12], further algorithms arose to detect in particular nested loops, i.e., loops within loops. The representation of loops within loops is called a *loop nesting forest*. Depending on the applied algorithm, different kinds of loop nesting forests can be derived from the same graph. Prominent examples are the Sreedhar-Gao-Lee [27], Steensgaard [28], and the Havlak [12] forests. As the approach in this paper decomposes loops, it leads to a loop nesting forest that may differ from those in the literature. Depending on the loop nesting forest, optimizations can be applied worse or better. Exactly what the effects are, should be investigated in the future and is beyond the scope of this paper.

The idea of finding independent structures in unstructured graphs is to improve analyses and visualization [20]. In BPM, a prominent approach is the decomposition of a graph into SESE components resulting in a tree, the RPST, that hierarchically orders those components [2,23,30,31]. Each component can be analyzed independently and, therefore, in parallel.

As explained in the introduction, loops make the analysis of process models difficult [3]. SESE decompositions do not help to improve the analysis of unstructured components with loops. Polyvyanyy et al. [21] use SESE decomposition (in the form of SPQR-trees after recognizing triconnected components) to show how sound, unstructured process models can be restructured into structured process models. Their structuring is based on *quasi block-structured* process models. A process model is quasi block-structured if its RPST does not contain a so-called *rigid* component. Loops can then occur as *Loop Case* components with single entries and single exits. Polyvyanyy et al. argue for such a loop case component that the entry and exit must be XOR gateways. This is consistent with our findings later in this paper. However, our loops can occur in any kind of component of a process model—our approach, therefore, generalizes the findings of Polyvyanyy et al. Furthermore, our process models can also contain OR-gateways. Finally, there are loops that cannot be structured and, therefore, cannot be analyzed. For this reason, Choi et al. [3] proposed a decomposition of workflow graphs into acyclic components so that loops no longer exist for soundness analysis. This decomposition separates each loop in the graph into a *forward* and *backward flow*. In natural loops, the forward flow contains the loop nodes on the paths between the loop entry and its exits; the backward flow contains all other nodes and some nodes overlapped with the forward flow. This approach is well suited for testing soundness of workflow graphs with natural loops. Irreducible loops, however, are not decomposed. They are instantiated each into multiple distinct loops depending on all combinations of how their loop entries may have tokens. When an instantiated loop has parallel loop entries, it is transformed into a natural loop. Our approach does not have to differentiate between natural and irreducible loops and decomposes both in a similar way. Moreover, it can be applied to processes with OR-splits and -joins, to which the approach of Choi et al. is not applicable.

*Untangling*s represent final extracts of executions of a process model [22]. They are similar to *instance subgraphs* [26], but work for cyclic process models free of lacks of synchronization. Although untanglings are acyclic versions of process models, they describe the processes within a different representation. One reason for this is that they have been used as index for querying process models within repositories. Our decomposition, in contrast, converts cyclic models into acyclic models within the same representation (e.g., BPMN into BPMN).

Another approach to the study of cyclic process models is *unfolding* [8,16,20]. Unfolding extends the process model (as Petri net) so that the resulting Petri net represents the same behavior as before, but without loops. Since unfoldings due to loops can be large or infinite, the resulting Petri net covers only a so-called *prefix* of the process, which represents the behavior of the entire process. Although this approach is well suited for analysis, it cannot be reused later, e.g., for execution, because it only covers an extract of the process model. In contrast, our decomposition uses (re)calls of subprocesses in loop nodes to cover the complete semantics of the original process model more naturally. In addition, since unfoldings use Petri nets, processes with OR-gateways cannot be handled.

4 Understanding Loops in Business Process Models

Loops are a classic term from programming language theory and control-flow graphs. Natural loops have a *header*, a *body*, and a *condition* [28]. The header is the entry to the loop, the body contains the code that may be executed several times, and the condition checks whether the body should be iterated or exited, i.e., a condition is a loop exit. Since business process models are very similar to control-flow graphs, it seems natural to use these terms for them as well.

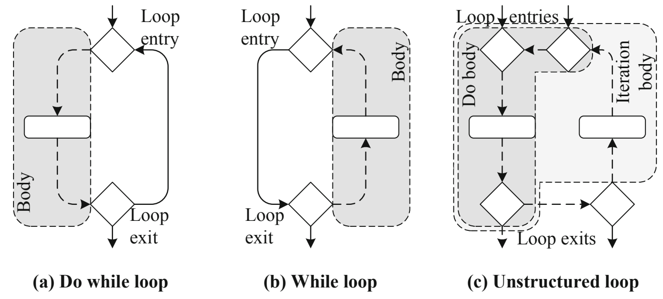

Fig. 2. Different types of loops with their loop entries, bodies, and loop exits.

4.1 Types of Loops

Classical loops can be divided into two types: *do-while* (repeat-until) and *while* loops [1]. *For* and *for-each* loops are interpreted in this context as special forms of while loops. In a do-while loop, the body of the loop is executed at least once. After the body has been executed, the loop exit (condition) is reached and it is checked whether the body is executed again. A do-while loop results in a graph as shown in Fig. 2(a). In contrast, the body of a while loop is only executed if the loop iterates at least once. Figure 2(b) shows a graph of a while loop.

Do-while and while loop structures in control-flow graphs usually result from loop structures in programming languages (e.g., `do { ... } while(cond);`). However, loops in control-flow graphs can also be unstructured, especially if *goto* statements are present [12]. On closer inspection, an unstructured loop may contain a subgraph that can be executed even though the loop does not iterate—as in a do-while loop. We call this subgraph *do-body*. And there can be a subgraph that can only be executed if the loop iterates at least once—as in a while loop. We call this subgraph *iteration-body*. Do- and iteration-bodies can be very complex, with subloops that may not pass the loop exits or entries.

A general unstructured loop can have multiple loop entries and exits. Loop entries are important for the first execution of the loop, but during iteration, they are negligible because the loop exits decide whether the loop iterates or not. For this reason, the do-body is part of the iteration-body, cf. Fig. 2*(c)*. In abstract terms, the do-body is the subgraph that starts at the loop entries and ends at its exits. It can be interpreted as the initialization of the loop or as a subgraph that must at least be executed even if the loop does not iterate. Iteration-bodies contain all nodes and edges of the loop, but can be split so that they start at the loop exits and end at them—like an unrolling of the loop.

4.2 Loops in Process Models

Process models as workflow graphs can contain all kinds of loops: do-while, while, and unstructured loops. Since workflow graphs can be designed unstructured in a graphical editor, unstructured loops with multiple entries and exits are not uncommon. To make matters worse, workflow graphs have explicit parallelism. Therefore, loop headers and nodes in loops can be executed in parallel. These circumstances make the analysis of process models particularly difficult.

From a business and intuitive point of view: When a token leaves a loop, the loop should be finished. Or in words of token game semantics: If a token is on a loop-exit edge, no other edge of the loop should carry a token anymore. Fortunately, this corresponds to the soundness property:

Theorem 1 (Single Token Loop Exits). Let *WFG* be a cyclic workflow graph with one of its loops $\mathcal{L} = (N_{\mathcal{L}}, E_{\mathcal{L}})$.

If *WFG* is sound and a loop-exit edge of $\mathcal{L}$ has a token in a state S, then the following both statements are valid:

(1) No edge in $\mathcal{L}$ has a token in S.
(2) No other loop-exit edge has a token in S.

Proof. We assume that *WFG* is sound and within a state S while a loop-exit edge ex of loop $\mathcal{L}$ carries a token.

To (1): *No edge in $\mathcal{L}$ has a token in S.* Assume there is an edge e of $\mathcal{L}$ that has a token or gets a token in a subsequent state S', $S \rightarrow^* S'$, while ex keeps its token. Since $\mathcal{L}$ is by Definition 2 a SCC, each node (and edge) within $\mathcal{L}$ has a path to each other node (and edge) in $\mathcal{L}$—and, therefore, also to ex. Let us take a path $P_{e \rightarrow ex}$ from e to ex in $\mathcal{L}$. Since each node of *WFG* has a path to at least one end node (cf. Definition 1), there is also a path $P_{ex \rightarrow end}$ from ex to a single incoming edge end of an end node. $P_{e \rightarrow ex}$ and $P_{ex \rightarrow end}$ are disjoint except of ex by Corollary 1. The combination of $P_{e \rightarrow ex}$ and $P_{ex \rightarrow end}$ results in the path $P_{e \rightarrow end}$. We now treat each XOR- and OR-split in each subsequent state of S' to always put a token on an edge on $P_{e \rightarrow end}$ (if it has any). In visual words, the tokens on ex and e follow finally the same path. Since *WFG* is sound and has no deadlock, there is a reachable state from S' where end carries at least two tokens. A lack of synchronization is reachable and, therefore, *WFG* is unsound. ↯

To (2): *No other loop-exit edge has a token in S.* Assume there would be another loop-exit edge ex' of $\mathcal{L}$ that carries a token in S. There are two possibilities without loss of generality: (a) ex' got a token some states after ex or (b) ex and ex' got the tokens within the same state transition.

To (a):*ex' got a token some states after ex* . In other words, there was a previous state, in which ex and another edge in $\mathcal{L}$ have tokens. (1) shows that this is a contradiction to the soundness of *WFG*. ↯

To (b): *ex and ex' got the tokens within the same state transition.* In each state transition, only one node is executed. Therefore, edges ex and ex' must have the same source node n. This node n has to be an AND- or OR-split since its execution results in more than one token. n—in contrast to ex and ex'—is part of $\mathcal{L}$. As a consequence, n must have at least one outgoing edge in within $\mathcal{L}$. Therefore, an execution of n can result in a state where ex and in both have tokens. This is again (1). ↯

Corollary 2. *Resulting from Theorem 1 and its proof (2) (b), each loop exit in a sound workflow graph is an XOR-split.*

Both—that every loop exit should be an XOR-split and that a loop is finished when a token leaves the loop—are intuitive but important properties of a loop in sound workflow graphs. It follows from these properties that any parallelism within a loop is synchronized before any loop exit is reached. More specifically, (1) do-bodies synchronize multiple tokens at loop entries into a single one when any loop exit is reached. (2) Each iteration of iteration-bodies starts and ends with a single token, i.e., each parallelism in iteration-bodies starts and ends in one iteration. And another important point: (3) No iteration of the iteration-body interacts with a previous iteration or with the do-body. Finally, (4) the iteration-body of a complex loop structure in a workflow graph can be replaced by a single node, making the graph acyclic. Knowing loops from control-flow graphs running not in parallel, all these facts are intuitively correct. But Theorem 1 shows that the same is true for loops in sound workflow graphs.

5 Loop Decomposition

Theorem 1 gives us the legitimacy to split a loop into two (sub) workflow graphs, one representing the do-body and the other the iteration-body. Then, the execution of a loop is done in two steps: The execution of its do-body and, subsequently, when the loop iterates, the multiple executions of the iteration-body. The separation is similar to converting a do-while loop into a while loop in programming languages where the do-body and iteration-body are the same, i.e., `do {<body>} while(cond);` is transferred to `<body> while(cond) {<body>}`. In other words, by separating the do-body from the iteration-body, an abstract do-while loop is transferred to an abstract while loop, very simplified as `<do-body> while(cond) {<iteration-body>}`.

The separation of the do- and iteration-body can generally be achieved by cutting the loop by at least one loop exit. In doing so, the do-body should contain most nodes and edges between the loop entries and the loop exits, so that execution enters the iteration-body only when required. Removing one *inner-loop* outgoing edge of a loop exit is sufficient to cut the initial loop structure. However, the more inner-loop outgoing edges of loop exits are removed, the more looping structures are destroyed. A do-body can be now defined as follows:

Definition 4 (Do-Body). Let there be a workflow graph with a loop $\mathcal{L}$.
The *do-body* of loop $\mathcal{L}$ is a maximal connected subgraph of $\mathcal{L}$ containing all loop entries and loop exits but a minimum number of *inner-loop* outgoing edges of loop exits.

Do-bodies can be identified by a variant of depth-first search. Their definition leads to the definition of iteration-bodies and cutoff edges:

Definition 5 (Iteration-Body and Cutoff Edges). Let there be a workflow graph with one of its loops $\mathcal{L}$ and $\mathcal{L}$'s do-body *Do*.

Every *inner-loop* outgoing edge of a loop exit of $\mathcal{L}$ that does not lie in *Do* is a *cutoff edge*. The *iteration-body* of $\mathcal{L}$ is the connected subgraph of $\mathcal{L}$ that contains all nodes and edges of $\mathcal{L}$ without the cutoff edges.

Because of the exclusivity of loop exits and, therefore, of cutoff edges, we can reduce each loop in a workflow graph to its do-body. To maintain loop behaviors, new *loop nodes* are inserted representing the iteration bodies of loops (for this case, Definition 1 is extended as mentioned in Sect. 2). Loop nodes may have multiple incoming and multiple outgoing edges and have an exclusive behavior according to Theorem 1. They are executed if one of their cutoff edges is taken, i.e., the cutoff edges are redirected to the loop nodes. After executing the loop nodes (their iteration-bodies), the execution can return to the previous loop-exit edges in the surrounding workflow graph. We call the modified workflow graph *reduced*. The iteration-body is extended with start and end nodes to a workflow graph:

Definition 6 (Workflow Graph Extensions). Let there be a workflow graph *WFG* with one of its loops $\mathcal{L}$.

The *$\mathcal{L}$-loop-reduced WFG* is a modified *WFG* regarding $\mathcal{L}$:
- each node and edge being part of $\mathcal{L}$ but not of $\mathcal{L}$'s do-body is removed from *WFG* (i.e., at least the cutoff edges),
- a *loop node* is inserted representing the iteration-body of loop $\mathcal{L}$,
- the cutoff edges of $\mathcal{L}$ are redirected to the new loop node,
- for each loop-exit edge, an outgoing edge from the loop node to the target of the loop-exit edge is introduced,
- each join node with one incoming edge is replaced by a single edge, and
- if non-join node has multiple incoming edges, these edges are combined with a new XOR-join in front of it.

The *extended iteration-body* $It(\mathcal{L})$ is $\mathcal{L}$'s iteration-body as a workflow graph:
- for each cutoff edge, two edges, one from a separate start node to its original target node and the other from the corresponding loop exit to a separate end node, are introduced,
- for each loop-exit edge, a corresponding edge to a separate end node is introduced, and
- each join node with only one incoming edge is replaced by a single edge.

Although iteration-bodies and extended iteration-bodies are different, we sometimes use both terms as synonyms when the context is clear.

In summary, three types of edges are redirected or added to modify and extend the original workflow graph and the iteration-bodies: loop-entry edges, loop-exit edges, and cutoff edges, each associated with a start, end, or loop node. Each cutoff edge appears twice in the extended iteration-body, once in connection with a start and once in connection with an end node.

Figure 3*(a)* shows our example workflow graph from Fig. 1 with its loop (large gray rounded rectangle). The entries of the loop are A and B, its exits F and G. The do-body contains these loop entries and exits and the subgraph between

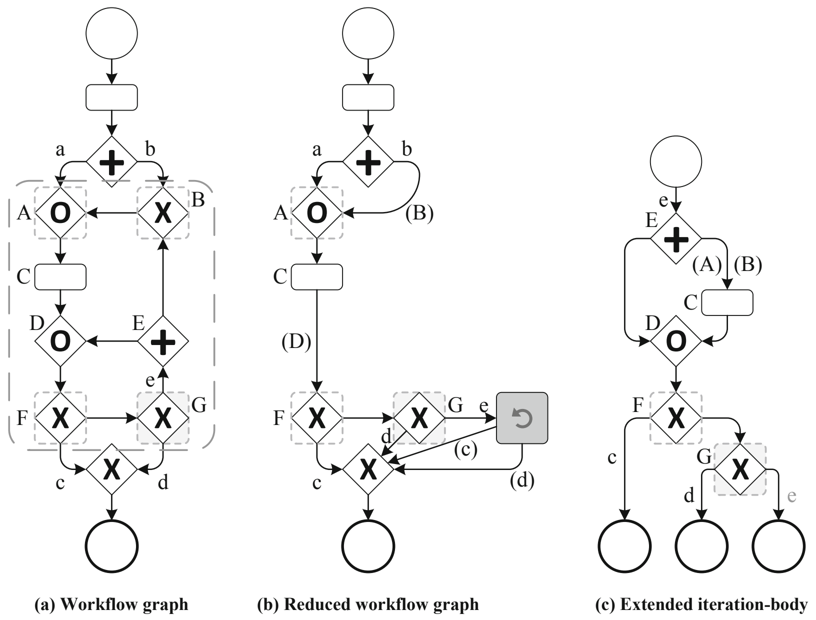

Fig. 3. Decomposition of the example workflow graph of Fig. 1 *a)* into its reduced workflow graph with a loop node *b)* and extended iteration-body *c)*.

them: A, B, C, D, F, and G. This do-body is maximal including all loop entries and exits and minimal in terms of the number of outgoing edges of loop exits. Therefore, e is a cutoff edge. The reduced version of the original workflow graph is illustrated in Fig. 3*(b)*. The reader can verify that it contains all nodes of the do-body (except B and D being replaced by edges (B) and (D)). The loop's iteration-body contains all nodes and edges of the loop, but it is truncated at the cutoff edges, e in this example, and join nodes A and B are merged into single edges (A) and (B). The extended iteration-body of Fig. 3*(c)* is the result. The graph has been arranged for clarity. We can see that the possible subloop of nodes D, F, G, and E in the original workflow graph is actually a parallel pattern and not a subloop anymore in the iteration-body. In Fig. 3*(b)* and *(c)* all edge-replaced nodes are either XOR- or OR-joins. Edge-replaced nodes can only be join nodes by Definition 6. If they could be AND-joins, the workflow graph cannot be sound following Theorem 1 and Choi et al. [3]:

Corollary 3. *Resulting from Definition 6, Theorem 1 and its proof, each join node being an edge in the reduced workflow graph or in $It(\mathcal{L})$ of a loop $\mathcal{L}$ of a sound workflow graph WFG is an XOR- or OR-join in WFG.*

Definition 6 describes a decomposition of a workflow graph into two workflow graphs regarding a single loop. In fact, both resulting workflow graphs can be cyclic again. However, since the original loop is cut on at least one outgoing edge of a loop exit, the remaining loops in both graphs are always smaller than the original loop. It is possible to identify these loops within the reduced workflow graph and iteration-body by using the same decomposition as described. In other words, we can recursively decompose each loop within a workflow graph, as Steensgaard [28] did. Eventually, this recursion terminates as the loops in each recursion become smaller and smaller.

6 Loop Execution

In the reduced workflow graph, a previously iteration-body of the loop is represented by a single node, the loop node (see Fig. 3*(b)*). The semantics of those is exclusive following Theorem 1. In other words, the special loop node is semantically a combination of an XOR-join and an XOR-split. If each loop of a workflow graph is recursively reduced to its do-body and a loop node, the workflow graph is *acyclic* by definition. Therefore, a sound, cyclic workflow graph has been transformed into an acyclic one.

The semantics of do-bodies of loops is already represented in the reduced workflow graph. To maintain the semantics of loop iterations, their extended iteration-bodies must be included. Their inclusion occurs when a corresponding loop node is enabled. When a loop node is enabled, the extended iteration-body is instantiated with a token on the corresponding start edge (e.g., the outgoing edge e of the start node in Fig. 3*(c)*). The extended iteration-body is executed until a token reaches the incoming edge of one of its end nodes. According to Theorem 1, only this incoming edge has a token in the entire iteration-body. In other words, there is always only one execution instance of the iteration-body. If the incoming edge of the end node matches the outgoing edge of one of its start nodes (e.g., the incoming edge e of the right end node in Fig. 3*(c)*), the iteration-body is terminated and instantiated again with a token on this edge (above edge e in Fig. 3*(c)*). If the incoming edge is a loop-exit edge (e.g., c and d in Fig. 3*(c)*), the loop node finishes its execution with a token on this edge.

If the original workflow graph is sound, then the execution semantics described above coincides with the behavior of the original workflow graph. But its acyclic workflow graphs are much easier to validate, verify, and execute. They should be executable on a process engine without much effort.

7 Algorithm

The combination of the recursive loop decomposition described above and the reduction of each loop and the insertion of loop nodes describes a complete decomposition algorithm from a cyclic workflow graph to a set of acyclic ones, summarized by Algorithm 1. The algorithm defines the function *decompose*, which first searches all loops within the given workflow graph, e.g., by Tarjan's algorithm for finding SCCs [29]. If there is no loop in the workflow graph, the function is finished and returns this graph. Otherwise, it initializes the set of acyclic workflow graphs. Subsequently, for each loop, the workflow graph is reduced and a loop node is inserted in the input workflow graph. The extended

Algorithm 1. Decomposition of a cyclic workflow graph WFG into a set of acyclic workflow graphs.

```
 1: function DECOMPOSE($WFG$)
 2:     Find set of loops $\mathfrak{L}$ in $WFG$.
 3:     if $\mathfrak{L} = \varnothing$ then
 4:         return $\{WFG\}$
 5:     $acyclic \leftarrow \varnothing$
 6:     for all $\mathcal{L} \in \mathfrak{L}$ do
 7:         Reduce $\mathcal{L}$ in $WFG$ to its do-body and include a loop node.
 8:         Derive the extended iteration-body workflow graph $It(\mathcal{L})$ from $\mathcal{L}$.
 9:         $acyclic \leftarrow acyclic \cup$ DECOMPOSE($It(\mathcal{L})$)
10:     $acyclic \leftarrow acyclic \cup$ DECOMPOSE($WFG$)
11:     return $acyclic$
```

iteration-body is derived and applied recursively to the *decompose* function. Since the input workflow graph has changed, *decompose* is applied again to it to find further (nested) loops. Finally, the result set of acyclic workflow graphs is returned at the end of the function.

Finding all loops in a workflow graph can be achieved in linear time, $O(N+E)$, using Tarjan's algorithm for finding SCCs [29]. Reducing the workflow graph and deriving the extended iteration-bodies can also be done in linear time mainly by using a variant of depth-first search. In the worst case, the number of (nested) loops in a workflow graph is as large as the number of edges E, i.e., the function *decompose* and the for-each-loop are both called E times at maximum. Overall, the asymptotic runtime complexity is $O(EN + E^2)$ in the worst case.

8 Evaluation and Example Applications

We have implemented our loop decomposition as a plugin for the research tool *Mojo*[1]. As input files, we used the PNML version of a library of real process models[2] from *IBM WebSphere Business Modeler* [9]. The models were originally in an XML format. The library contains 1,368 process models. 178 (approx. 13%) process models are cyclic—169 have a single and 9 have two independent SCCs (loops) by Definition 2. Therefore, there are in total 187 loops in all process models. During our decomposition, only 28 of 187 loops contain nested loops—23 contain a single nested loop and 5 contain two independent nested loops. There is no process model with a nesting depth greater than 1.

After loop decomposition, the process models are reduced to about 94% (SD 0.27) of their original size. The worst case complexity of the algorithm is quadratic, however, it appears to be linear in practice, as the number of edges visited during the decomposition does not increase quadratically. This

[1] https://github.com/guybrushPrince/mojo.plan.loop, last visited March 2022.

[2] https://web.archive.org/web/20131208132841/http://service-technology.org/publications/fahlandfjklvw_2009_bpm, last visited March 2022.

seems to depend on the nature of the process model. Without the elimination of background processes and running on a standard computer, a loop decomposition takes on average less than 1 ms (SD 0.9).

Loop decomposition has a direct effect on the execution semantics of OR-joins. The definition of the semantics for OR-joins is intuitive for acyclic but difficult for cyclic process models [24,32]. Most execution semantics in research have, finally, a problem with so-called *vicious circles*—loops in which two OR-joins are mutually waiting for each other [15]. Such vicious circles are no longer possible after decomposition into acyclic process models. Therefore, understanding the semantics of each OR-join in the decomposed models should be intuitive. Moreover, each join node can be replaced by an OR-join (or an undifferentiated join), while retaining the semantics.

Although soundness is a precondition of loop decomposition, it can be used to check soundness itself. This is the first method (to the authors' knowledge) that can be used to check soundness of cyclic process models with OR-joins. We just outline the idea without proof in the following: First, loop decomposition is applied on the process model resulting in a set of acyclic models. Starting from the acyclic version of the original process model, we can apply an arbitrary soundness verification algorithm for acyclic models with OR-splits and OR-joins (e.g., Favre and Völzer [10] or Prinz and Amme [25]). If the process is unsound, its original process is unsound too. This is valid since, otherwise, a loop could not be reduced and replaced with a loop node, being a violation against Theorem 1. Second, all of the process model's extended iteration-bodies can be simplified so that they have a single start and end node since all their start and end nodes are exclusive. Subsequently, all iteration-bodies are checked against soundness and whether Corollary 3 holds. For all sound (sub) models, simplifications and analyses can be applied iteratively. If the process model contains any unsound iteration-body, the model is naturally unsound.

9 Conclusion

In this paper, we have presented an algorithm that decomposes cyclic process models in the form of workflow graphs into sets of acyclic workflow graphs. The approach is based on a general view of loops and a simple execution semantics. The execution semantics of the acyclic workflow graphs coincides when the original workflow graph is sound. A short evaluation shows that loop decomposition is feasible in practice. Some advantages of decomposition were discussed using OR-join semantics and soundness analysis.

The BPM community should benefit from this approach. Many analyses in BPM are limited to acyclic process models and can now be applied to cyclic models. In addition, execution engines of processes can be simplified, especially regarding the semantics of OR-joins. In general and in terms of compiler theory, our presented algorithm leads to a loop nesting forest that may differ from those in the literature. The differences arise because each loop is transformed into a while-loop-like form. In the future, it will be interesting to study its impact.

For future work, we plan to incorporate algorithms and analyses from acyclic process models to our acyclic decomposition. One application could be the first detailed and complete soundness approach for cyclic workflow graphs with OR-joins. Another useful application is to combine the RPST with our loop decomposition. This should facilitate divide-and-conquer approaches for process analysis, especially for unstructured components.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Boston (1986)
2. Choi, Y., Ha, N.L., Kongsuwan, P., Han, K.H.: An alternative method for refined process structure trees (RPST). Bus. Process. Manag. J. **26**(2), 613–629 (2020). https://doi.org/10.1108/BPMJ-11-2018-0319
3. Choi, Y., Kongsuwan, P., Joo, C.M., Zhao, J.L.: Stepwise structural verification of cyclic workflow models with acyclic decomposition and reduction of loops. Data Knowl. Eng. **95**, 39–65 (2015). https://doi.org/10.1016/j.datak.2014.11.003
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
5. van Dongen, B.F., Mendling, J., van der Aalst, W.M.P.: Structural patterns for soundness of business process models. In: Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), 16–20 October 2006, Hong Kong, China, pp. 116–128. IEEE Computer Society (2006). https://doi.org/10.1109/EDOC.2006.56
6. Dumas, M., García-Bañuelosa, L., La Rosa, M., Ubaa, R.: Fast detection of exact clones in business process model repositories. Inf. Syst. **38**(4), 619–633 (2013). https://doi.org/10.1016/j.is.2012.07.002
7. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-33143-5
8. Esparza, J., Römer, S., Vogler, W.: An improvement of mcmillan's unfolding algorithm. Formal Methods Syst. Des. **20**(3), 285–310 (2002). https://doi.org/10.1023/A:1014746130920
9. Fahland, D., Favre, C., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Analysis on demand: instantaneous soundness checking of industrial business process models. Data Knowl. Eng. **70**(5), 448–466 (2011)
10. Favre, C., Völzer, H.: Symbolic execution of acyclic workflow graphs. In: Hull et al. [13], pp. 260–275. https://doi.org/10.1007/978-3-642-15618-2_19
11. Ha, N.L., Prinz, T.M.: Partitioning behavioral retrieval: an efficient computational approach with transitive rules. IEEE Access **9**, 112043–112056 (2021)
12. Havlak, P.: Nesting of reducible and irreducible loops. ACM Trans. Program. Lang. Syst. **19**(4), 557–567 (1997). https://doi.org/10.1145/262004.262005
13. Hull, R., Mendling, J., Tai, S. (eds.): Business Process Management. LNCS, vol. 6336. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15618-2
14. Keller, G., Scheer, A.W., Nüttgens, M.: Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)". Inst. für Wirtschaftsinformatik (1992)
15. Kindler, E.: On the semantics of EPCs: resolving the vicious circle. Data Knowl. Eng. **56**(1), 23–40 (2006)

16. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: von Bochmann, G., Probst, D.K. (eds.) CAV 1992. LNCS, vol. 663, pp. 164–177. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56496-9_14
17. Mendling, J., Neumann, G., van der Aalst, W.: Understanding the occurrence of errors in process models based on metrics. In: Meersman, R., Tari, Z. (eds.) OTM 2007. LNCS, vol. 4803, pp. 113–130. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76848-7_9
18. OASIS: Web Services Business Process Execution Language Version 2.0, April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf
19. Object Management Group (OMG): Business Process Model and Notation (BPMN) Version 2.0. formal/2011-01-03, January 2011. http://www.omg.org/spec/BPMN/2.0
20. Polyvyanyy, A.: Structuring process models. Ph.D. thesis, University of Potsdam (2012)
21. Polyvyanyy, A., García-Bañuelos, L., Weske, M.: Unveiling hidden unstructured regions in process models. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2009. LNCS, vol. 5870, pp. 340–356. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05148-7_23
22. Polyvyanyy, A., La Rosa, M., ter Hofstede, A.H.M.: Indexing and efficient instance-based retrieval of process models using untanglings. In: Jarke, M., et al. (eds.) CAiSE 2014. LNCS, vol. 8484, pp. 439–456. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07881-6_30
23. Polyvyanyy, A., Vanhatalo, J., Völzer, H.: Simplified computation and generalization of the refined process structure tree. In: Bravetti, M., Bultan, T. (eds.) WS-FM 2010. LNCS, vol. 6551, pp. 25–41. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19589-1_2
24. Prinz, T.M., Amme, W.: A complete and the most liberal semantics for converging OR gateways in sound processes. Complex Syst. Informatics Model. Q. **4**, 32–49 (2015). https://doi.org/10.7250/csimq.2015-4.03
25. Prinz, T.M., Amme, W.: Control-flow-based methods to support the development of sound workflows. Complex Syst. Informatics Model. Q. **27**, 1–44 (2021)
26. Sadiq, W., Orlowska, M.E.: Analyzing process models using graph reduction techniques. Inf. Syst. **25**(2), 117–134 (2000)
27. Sreedhar, V.C., Gao, G.R., Lee, Y.: Identifying loops using DJ graphs. ACM Trans. Program. Lang. Syst. **18**(6), 649–658 (1996)
28. Steensgaard, B.: Sequentializing program dependence graphs for irreducible programs. Technical report, Microsoft Research (1993)
29. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972). https://doi.org/10.1137/0201010
30. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. Data Knowl. Eng. **68**(9), 793–818 (2009). https://doi.org/10.1016/j.datak.2009.02.015
31. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models through SESE decomposition. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 43–55. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74974-5_4
32. Völzer, H.: A new semantics for the inclusive converging gateway in safe processes. In: Hull et al. [13], pp. 294–309. https://doi.org/10.1007/978-3-642-15618-2_21