The Triconnected Abstraction of Process Models

Artem Polyvyanyy, Sergey Smirnov, and Mathias Weske

Business Process Technology Group
Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany
(Artem.Polyvyanyy,Sergey.Smirnov,Mathias.Weske)@hpi.uni-potsdam.de

Abstract. Companies use business process models to represent their working procedures in order to deploy services to markets, to analyze them, and to improve upon them. Competitive markets necessitate complex procedures, which lead to large process specifications with sophisticated structures. Real world process models can often incorporate hundreds of modeling constructs. While a large degree of detail complicates the comprehension of the processes, it is essential to many analysis tasks. This paper presents a technique to abstract, i.e., to simplify process models. Given a detailed model, we introduce abstraction rules which generalize process fragments in order to bring the model to a higher abstraction level. The approach is suited for the abstraction of large process specifications in order to aid model comprehension as well as decomposing problems of process model analysis. The work is based on process structure trees that have recently been introduced to the field of business process management.

1 Introduction

Business process modeling is a well-established technique for designing and communicating how work activities are related to each other, and how these activities contribute to a business goal. To provide a common understanding of the language used, standard modeling notations are proposed, for instance, Business Process Modeling Notation (BPMN) [1], Event-driven Process Chains (EPC) [2], and Petri nets [3]. Business process models serve as a communication vehicle for different stakeholders, e.g., business analysts and software designers. Moreover, process models are used to analyze working procedures, to propose improvements, and even to provide a blueprint for a software realizing the process.

With the increasing complexity of services which companies provide to markets, business processes fulfilling these services are getting more and more complex, too. As a result, business process models often consist of dozens or even hundreds of nodes, making these models hard to understand. There is a dilemma: On the one hand, too much detail hampers the understanding of the overall process. On the other hand, this level of detail might be required for process analysis and for implementing the process in software.

There are two approaches to address the problem. Either different models serving different purposes are developed, or different models, catering to different

process modeling needs, are generated from a detailed original model. If the former approach is followed, consistency of the models is a severe problem. Changes on one level need to be reflected on other levels as well, which is often done manually. Experience shows that due to model evolution on different levels of detail, the models become inconsistent quite soon. Therefore, we opt for the latter approach: We generate different process models from a given detailed model by introducing transformation rules. These rules abstract from details of a process model and provide abstracted models that non-technical stakeholders can understand. At the same time, any evolutionary changes will be taken into account, since effectively there is only one process model, and the others are generated from it on demand. Technically, the work is based on the program parsing technique, known from the compiler theory of sequential programs [4]. The method was introduced to the business process management community in the refined process structure tree (RPST) decomposition of workflow graphs [5].

While the results in this paper are of a conceptual and rather theoretical nature, they emerged from an industry project conducted with a large health insurance company, just like a previous study focusing on pattern-based process abstraction [6]. In this initial endeavor, we developed an automated abstraction control mechanism guided by the average execution time of tasks included in a model. The proposed technique attempts to first abstract from tasks which are rarely observed. Of course, in order to allow such an abstraction control, models must be additionally annotated with the tasks' average execution times. The main limitation of the pattern-based approach is the problem of completeness, i.e., the necessity to have a full set of patterns which can support the abstraction of arbitrarily structured process models. The idea of abstraction control mechanisms is elaborated in [7], where an abstraction slider is presented.

The completeness of a set of reduction rules is a well-known problem in the analysis of Petri nets. Berthelot proposed a set of rules which can be repeatedly applied to reduce live and bounded marked graphs to a single transition [8,9]. Desel and Esparza, in [10], proposed a complete kit of reduction rules for free-choice Petri nets. In [11], Murata presents reduction rules which preserve the liveness, safeness, and boundedness properties. However, all the mentioned rules are incomplete when operating on models of an arbitrary structure. The limitations of the pattern-based abstraction and the impossibility of closing the gap by adapting the existing reduction rules have inspired this work. We define and utilize for abstraction purposes a notion of a process component which permits achieving completeness when handling a process model of an arbitrary structure.

The rest of the paper is organized as follows: The next section sketches the research field of business process model abstraction, its perspectives and its challenges. In section 3, we provide definitions and a basic corollary that form the basis for further discussion. The structural decomposition of process models into triconnected components is presented in section 4. In section 5, the components are used for process model abstraction, resulting in the triconnected abstraction technique. The paper closes with ideas on future steps and conclusions that summarize our findings.

2 Business Process Model Abstraction

This section discusses the research field of business process model abstraction (BPMA). The core aspects of BPMA are identified. Finally, we position the contribution area of BPMA to be addressed in the rest of the paper.

Business process analysts often attempt to capture every detail of handling a particular business case for inclusion in a process model, which leads to excessive numbers of modeling constructs and sophisticated model structures. In order to reduce the complexity and to allow for the faster investigation of process logic, we started to look for automated techniques to abstract, i.e., to simplify, process models. Abstraction is the result of the generalization or elimination of properties in an entity or a phenomenon in order to reduce it to a set of essential characteristics. Information loss is the fundamental property of abstraction and is its intended outcome. When modeling, business process analysts abstract from the complex reality by extracting important behavioral aspects of a process. In BPMA, we investigate problems specific to the abstraction of process model entities. The challenge lies in identifying what is a meaningful generalization of process logic aimed at removing certain characteristics while at the same time emphasizing others.

In BPMA, identified process fragments can be eliminated or replaced by concepts of a higher abstraction level which conceal, but also represent, the logic of the underlying fragments. In both cases, generalization as well as elimination, sophisticated handling mechanisms need to be proposed. We refer to such mechanisms as *abstraction steps*.

Control mechanisms combine atomic abstraction steps into *abstraction strategies*. One can envision manual strategies in which a user specifies tasks to be abstracted, semi-automated, or automated control mechanisms.

Any process abstraction methodology aims at ensuring certain properties of abstracted models. The properties should allow a semantic relation between the original and abstracted models. The key property we pursue in our approach of process abstraction is order preservation. An *order preserving abstraction* is an abstraction that ensures that neither new task execution order constraints can appear after abstraction, nor existing ones (except for generalized ones) go away. For instance, assume that task A should be abstracted. Let f_A be a process fragment affected in the abstraction step (f_A contains A). As a result of abstraction, fragment f_A gets replaced by task F. If task B belongs to f_A, information about execution order constraints between task A and task B is lost. However, an order preserving abstraction ensures that between any pair of tasks not in f_A, e.g., task C and task D, execution order constraints are preserved. Furthermore, an order preserving abstraction guarantees that execution order constraints between any task not in f_A, e.g., task E, and any task in f_A, task A or task B in our example, are the same as between task E and task F. In the end, an order preserving abstraction secures the overall process logic to be reflected in the abstracted model.

A business process model abstraction methodology is a compromised combination of requirements and techniques picked out from all of the discussed

abstraction aspects. Usually, such a combination is guided by project specific use cases. This paper primarily contributes to the BPMA aspects of discovering fragments which are structurally suitable for abstraction and further performing abstractions. Effectively, we define a structural fragment type which is accepted as a unit of process logic abstraction, provide mechanisms for the discovery of a complete set of process fragments suitable for abstraction, and specify the algorithm which aims at abstracting from a given task in a process model by utilizing the discovered fragments.

3 Preliminaries

In this section, we introduce basic definitions. We start with a process model formalism adapted from [12] which is based on generic modeling concepts. A process model consists of a set of tasks and their structuring using directed control flow edges and gateway nodes that implement process routing decisions.

Definition 1. $P = (N, E, type)$ is a *process model* if $N = N_T \cup N_G$ is a set of nodes, where N_T is a nonempty set of tasks and N_G is a set of gateways; the sets are disjoint. $E \subseteq N \times N$ is a set of directed edges between nodes defining control flow. $type : N_G \rightarrow \{and, xor, or\}$ is a function that assigns a control flow construct to each gateway. (N, E) is a connected graph—a *process graph*. Each task $t \in N_T$ can have at most one incoming and at most one outgoing edge $(|\bullet t| \leq 1 \wedge |t\bullet| \leq 1)$, where $\bullet t$ stands for a set of immediate predecessor nodes $(\bullet t = \{n \in N | (n, t) \in E\})$ and $t\bullet$ stands for a set of immediate successor nodes $(t\bullet = \{n \in N | (t, n) \in E\})$ of task t. A task $t \in N_T$ is a *process entry* if $|\bullet t| = 0$. A task $t \in N_T$ is a *process exit* if $|t\bullet| = 0$. There is at least one process entry task and at least one process exit task. Each gateway is either a split or a join. A gateway $g \in N_G$ is a *split* if $(|\bullet g| = 1 \wedge |g\bullet| > 1)$. A gateway $g \in N_G$ is a *join* if $(|\bullet g| > 1 \wedge |g\bullet| = 1)$.

To be able to refer to parts of a process model, we define a process fragment. A process fragment is a connected part of a process model.

Definition 2. A *process fragment* $F = (N_F, E_F, type_F)$ of a process model $P = (N, E, type)$, where $N_G \subset N$ is a set of gateways of P, consists of a connected subgraph (N_F, E_F) of the process graph (N, E) of P and function $type_F$, which is a restriction of the function $type$ of P to a set $N_F \cap N_G$.

Within a process fragment, nodes can be classified in regard to their structural relation to the whole process model.

Definition 3. A node $n \in N_F$ is a *boundary* node of a process fragment $F = (N_F, E_F, type_F)$ in a process model $P = (N, E, type)$ if n is a process entry of P, a process exit of P, or there exist edges $e_i \in E_F$ and $e_j \in E \backslash E_F$ adjacent through n. A non-boundary node $n \in N_F$ of F is an *internal* node of F.

Boundary nodes of a process fragment can be distinguished as fragment entries and fragment exits based on the directions of incident control flow edges.

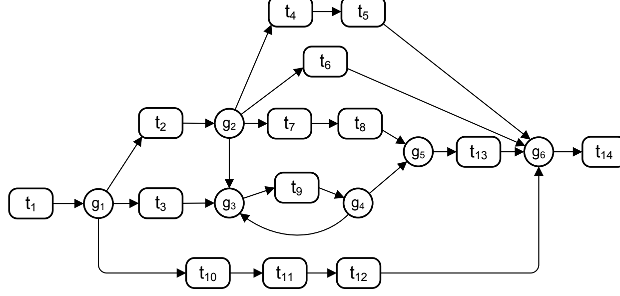


Fig. 1. A process model

Definition 4. Let $n \in N_F$ be a boundary node of a process fragment $F = (N_F, E_F, type_F)$ in a process model $P = (N, E, type)$, then:

- A node n is a *fragment entry* of F if all the incoming edges of n are outside of F ($\bullet n \subseteq N \backslash N_F$) or all the outgoing edges of n are inside of F ($n \bullet \subseteq N_F$).
- A node n is a *fragment exit* of F if all the outgoing edges of n are outside of F ($n \bullet \subseteq N \backslash N_F$) or all the incoming edges of n are inside of F ($\bullet n \subseteq N_F$).

Finally, we recognize a special class of process fragments—process components.

Definition 5. A *process component* $C = (N_C, E_C, type_C)$ is a process fragment with two boundary nodes: one fragment entry and one fragment exit.

This notion of a component was first introduced in [4] as a concept of a *proper subprogram*. A process component is a process fragment in which it is assured that if control flows through a fragment's edge, it has first entered the process fragment through the fragment entry and will subsequently leave the process fragment through the fragment exit.

Structurally, a process component is a self-contained block of process logic with strictly defined boundaries. Semantically, a process component can be addressed as a detailed specification of task execution scenarios. Hence, any process component can be formalized as a WF-net [13] of, potentially, an arbitrary structure. Therefore, in the triconnected abstraction approach, a process component is accepted as a unit of meaningful aggregation of process logic, i.e., detailed specifications get represented by a corresponding task concept. In the following sections, we discuss issues relevant to the identification and abstraction of process components in process models.

We require process models to be *structurally sound* [12], i.e., a process model should have exactly one process entry, exactly one process exit, and each process model node should be on a path from the process entry to the process exit. The prerequisite introduces a minimal correctness notion for process models—subjects for abstraction. Moreover, the stated structural requirement is crucial when it comes to the discovery of process components in process models. Figure 1 provides an example of a process model suitable for abstraction.

4 The Triconnected Decomposition

This section explains how to discover process fragments that relate to the notion of a process component as defined in section 3. First, we give the basic intuition inherent in the algorithm. Afterwards, we show the relation of the discovery process to the approach of SPQR-tree [14,15] decomposition. Finally, we discuss SPQR-tree fragments in the context of process models.

4.1 Basic Approach for Process Component Discovery

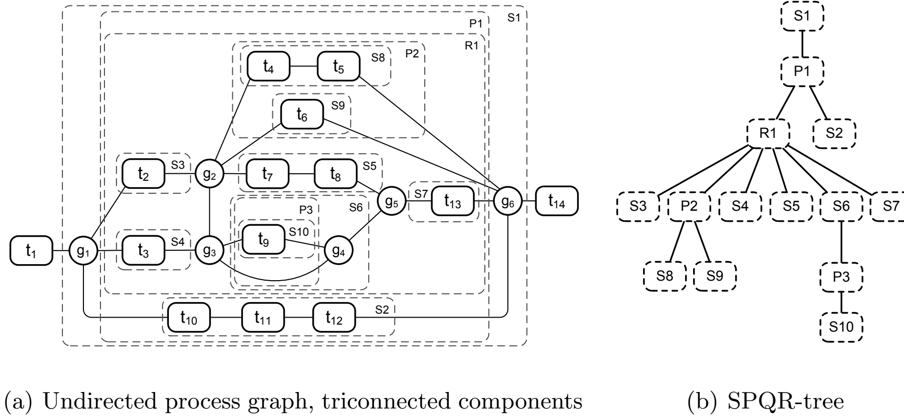
A search for a process component in a process model is guided by its definition (see Definition 5), which states that a process component is a process fragment with two boundary nodes. Boundary nodes are the nodes that connect the fragment to the model, i.e., if removed the fragment becomes disconnected from the model. Thus, in order to discover a process component, one must first look for a *separation pair*—a pair of nodes that disconnect a process fragment from the rest of the process model. For instance, gateways g_3 and g_4 disconnect task t_9 in the process model from Figure 1. Afterwards, the boundary nodes of the fragment need to be tested to give one fragment entry and one fragment exit.

A separation pair divides process model into two fragments. In order to find all fragments with two boundary nodes, the rationale of the described discovery step must be applied to each of the two fragments, resulting in a divide and conquer algorithm design. Each recursive thread terminates once the problem cannot be further subdivided, i.e., there is no separation pair in a process fragment.

The described algorithm is in fact the algorithm for the discovery of triconnected components in a graph. Connectivity is a property of a graph. It is known that a graph is k-connected if there exists no set of $k-1$ elements, each a vertex or an edge, whose removal makes the graph disconnected (there is no path between some node pair in a graph). Such a set is called a separating $(k-1)$-set. Separating 1- and 2-sets of graph vertices are called cutvertices and separation pairs. 1-, 2-, and 3-connected graphs are referred to as connected, biconnected, and triconnected, respectively. Each recursive thread of the algorithm terminates once it encounters a triconnected component.

4.2 SPQR-Tree Decomposition

In order to discover process components, one can use SPQR-tree decomposition. SPQR-tree decomposition is a decomposition of an undirected biconnected multigraph induced by its split pairs aimed at identifying its triconnected components. A *split pair* is either a separation pair or a pair of adjacent nodes. Process models are connected, but not necessarily biconnected. For example, the process model from Figure 1 has cutvertex g_1. However, it is always possible to make a process model biconnected by adding a back edge connecting a process exit with a process entry. The requirement of structural soundness ensures that every process model has exactly one process entry and exactly one process exit.



(a) Undirected process graph, triconnected components (b) SPQR-tree

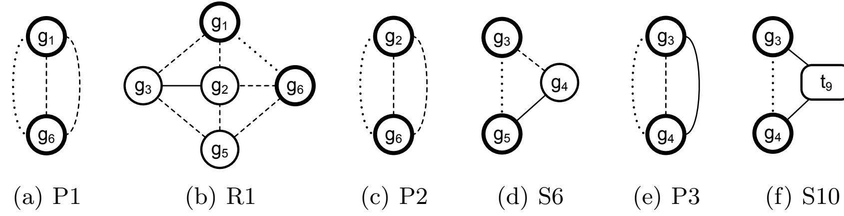
Fig. 2. SPQR-tree process model decomposition

The algorithm for the discovery of triconnected components of a graph was first proposed by Hopcroft and Tarjan in [16]. Later, Tarjan and Valdes in [4] applied the algorithm for sequential program parsing to obtain the *parse tree* (or *the tree of the triconnected components*). The tree was studied as SPQR-tree in [14,15]. [16,17,18] show the path towards a linear time complexity algorithm implementation of SPQR-tree decomposition. The decomposition results in triconnected components of four structural types, in the following using the SPQR-tree terminology, S, P, Q, and R types.

- *Trivial case.* A split pair is a pair of adjacent graph vertices—a fragment consists of one edge—the Q-type fragment.
- *Parallel case.* A split pair is a pair of adjacent graph vertices in k distinct edges ($k \geq 2$)—the P-type fragment.
- *Series case.* A split pair is a pair of graph vertices giving a maximal sequence of vertices and consists of k nodes and k edges ($k \geq 3$)—the S-type fragment.
- *Rigid case.* If none of the above cases applies, a fragment is a triconnected fragment—the R-type fragment.

SPQR-tree decomposition of the process model from Figure 1 is exemplified in Figure 2. Each process fragment corresponds to a triconnected component of the model and is defined by edges that are inside or intersect with a corresponding region visualized with a dashed line in Figure 2(a). Fragment names hint at structural fragment types, e.g., $P1$, $P2$, and $P3$ are all parallel case fragments. Boundary nodes of a fragment are the nodes incident with edges crossing the region borderline and are outside of the region.

Figure 2(b) shows an SPQR-tree that visualizes hierarchical fragment relations. Fragment $P1$ contains fragments $R1$ and $S2$ and is fully contained within fragment $S1$. Each SPQR-tree node represents a *fragment skeleton*, i.e., basic structure of a fragment and its relations with a parent and child fragments. Figure 3 shows



(a) P1 (b) R1 (c) P2 (d) S6 (e) P3 (f) S10

Fig. 3. SPQR-tree fragment skeletons

fragment skeletons of SPQR-tree nodes from Figure 2(b). Boundary nodes are highlighted with a thick borderline, e.g., nodes g_1 and g_6 in fragment $R1$ (see Figure 3(b)). Each fragment skeleton can consist of edges of three types. Original graph edges are drawn with solid lines, whereas dotted and dashed lines represent *virtual edges*. Each virtual edge is shared between two fragment skeletons and hints at a parent-child relation. An edge visualized by a dotted line shows a child relation of the fragment skeleton with another skeleton which contains the same virtual edge; a dashed line signals a parent relation. For instance, the fragment skeleton from Figure 3(f) contains one virtual edge (g_3, g_4), which hints at a child relation with another fragment skeleton that contains the same virtual edge—fragment skeleton $P3$ (see Figure 3(e)). In order to obtain the graph fragment given by fragment skeleton $P3$, one must "glue" it together with fragment skeleton $S10$ along virtual edge (g_3, g_4). Once the fragments are combined, the virtual edge is removed. In general, a graph fragment represented by an SPQR-tree node can be obtained by combining all its descendants.

SPQR-tree provides process model decomposition that ignores control flow edge directions. At this point, there has still been no distinction made between entry and exit boundary nodes; obtained fragments still cannot be classified as process components.

4.3 SPQR-Tree Fragments in the Context of Process Models

In this section, we examine fragments obtained after the SPQR-tree decomposition of a process model, i.e., edges of a process graph are directed and nodes distinct as tasks and gateways.

In general, an SPQR-tree can be rooted to any node. However, in the context of a process model it makes sense to root the tree to a node representing the fragment containing the deliberately introduced back edge (node $S1$ in Figure 2(b)). As a result, one obtains the structural hierarchical refinement of a process model.

Further observations are: Task nodes can only be present, but are not always necessarily present (see Figure 3(d)), inside of S-type fragments, while boundary fragment nodes are always gateways. The former property comes from the definition of the S-type fragment. Any sequence of nodes in a process graph can only be formed by task nodes embraced by gateways. Thus, any maximal

sequence, also composed of one task (see Figure 3(f)), is recognized as the S-type fragment with two boundary gateways: one at sequence entry and another at sequence exit. This also means that other fragment skeletons are composed of gateways only, which testifies the latter property.

Until now, we have recognized sequences as S-type fragments. Q-type fragments stand for original process graph edges, e.g., the edge (g_4, g_5) of fragment skeleton from Figure 3(d). P-type fragments (see Figures 3(a), 3(c), and 3(e)) allow identification of block and loop structures within process models. The control flow of the process model from Figure 1 specifies fragments $P1$ and $P2$ as blocks and fragment $P3$ as a loop (there exists a back edge between boundary nodes g_3 and g_4). The fragment from Figure 3(b) is the triconnected fragment that explicitly defines what makes the process model graph-structured. There are no R-type fragments in a block-structured process model. A block-structured process model can be inductively composed based on sequence, block, and loop patterns (S-type and P-type fragments) [19].

Finally, we are ready to make the concluding proposition of section 4:

Theorem 1. *Any process fragment obtained after SPQR-tree decomposition of a structurally sound process model is a process component.*

Proof. Any process fragment obtained after SPQR-tree decomposition of a process model has two boundary nodes. A pair of boundary nodes of a process fragment is a split pair of the process model. Thus, it is necessary to show that one of the boundary nodes is a fragment entry and the other is a fragment exit.

First, we show that any boundary node of a process fragment induced by SPQR-tree decomposition is either a fragment entry or a fragment exit. All the edges incident with a boundary node are divided into two disjoint sets of those inside and those outside the fragment. Definition 4 states that a boundary node of a process fragment is a fragment entry or a fragment exit if either all the incoming or all the outgoing edges incident with the node are either the edges of the fragment or are outside the fragment. As explained above, any boundary node is a gateway. For any gateway, either a set of all incoming edges or a set of all outgoing edges consists of one element (see Definition 1). The relation of this one edge, either belonging to the process fragment or not, defines the relation of the whole set. Therefore, any boundary node can only expose the logic of a fragment entry or a fragment exit.

The rationale towards a formal proof of the "pure" logic of a boundary node of a process fragment can be approached as follows. Let $P = (N, E, type)$ be a process model, $F = (N_F, E_F, type_F)$ be a process fragment of P. Let us define auxiliary predicates:

- $i : E \times N \to \{true, false\}$ is $true$ if $e \in E$ is the incoming edge of node $n \in N$, $false$ otherwise, and
- $o : E \times N \to \{true, false\}$ is $true$ if $e \in E$ is the outgoing edge of node $n \in N$, $false$ otherwise.

One can now define predicates which check if a node $n \in N$ can be an entry of F—$canEnter$, or an exit of F—$canExit$:

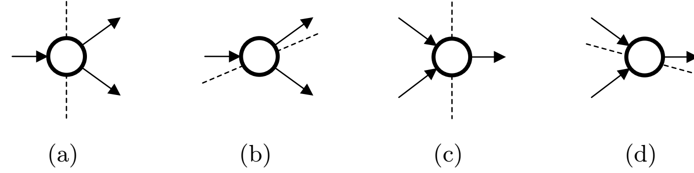


Fig. 4. All possible combinations for edge separation on internal and external fragment edges for a boundary gateway connecting three edges

- $canEnter(n, F) = \exists e_1 \in E \backslash E_F \exists e_2 \in E_F : i(e_1, n) \wedge o(e_2, n)$,
- $canExit(n, F) = \exists e_1 \in E_F \exists e_2 \in E \backslash E_F : i(e_1, n) \wedge o(e_2, n)$.

In order to show that any boundary fragment node cannot at the same time expose entry and exit logic, one must show that the logical statements $canEnter(n, F) \models \neg canExit(n, F)$ and $canExit(n, F) \models \neg canEnter(n, F)$ hold. Hence, one must show that $canEnter(n, F) \wedge canExit(n, F)$ is a false statement on all interpretations which in a prenex normal form says:

$$\exists e_1 \in E \backslash E_F \exists e_2 \in E_F \exists e_3 \in E_F \exists e_4 \in E \backslash E_F : i(e_1, n) \wedge o(e_2, n) \wedge i(e_3, n) \wedge o(e_4, n)$$

If n is a split gateway, the statement might evaluate to $true$ only if e_1 and e_3 are bound to the same edge. This, however, is impossible, as e_1 and e_3 belong to different sets which are disjoint: E and $E \backslash E_F$. The same rationale applies for a join gateway and edges e_2 and e_4. Therefore, a logical expression $canEnter(n, F) \wedge canExit(n, F)$ always evaluates to $false$, which proves the pure logic of any boundary node of F.

Figure 4 shows all possible combinations of internal and external fragment edges incident with a boundary gateway which connects three edges. The dashed line separates edges on fragment's internal and external edges. Regardless of a separation and a gateway type, control flow is only allowed to "penetrate" a fragment's boundary in one direction, either to enter or to leave a process fragment.

Finally, it is necessary to show that only one arrangement of boundary nodes is possible, i.e., one of the nodes is a fragment entry and the other is a fragment exit. We show this by contradiction; the settings of two fragment entries or two fragment exits are not possible under the correctness criteria imposed on a process model—a process model is structurally sound. Two cases can be reduced to one. For instance, in case of a fragment with two exits, one can discuss a two entry fragment formed by the edges outside the two exit fragment. A process fragment with two entries violates the requirement of a structurally sound process model which states that each node in a process model is on a path from a process entry to a process exit. Once we enter a two entry fragment, we never leave it. Any node of a two entry fragment cannot be on a path from the process entry to the process exit. Therefore, one of the boundary nodes must be a fragment exit.

If the process entry and the process exit are the boundary nodes of a process fragment, the process entry is a fragment entry and the process exit is a fragment exit. □

5 The Triconnected Abstraction

This section presents the triconnected abstraction. The approach is based on the decomposition technique described in section 4. First, we define abstraction rules. Afterwards, we combine the rules into the process model abstraction algorithm.

5.1 Abstraction Rules

The triconnected process model abstraction technique is founded on the idea of interchanging process fragments with process tasks of higher abstraction levels. In this section, we present abstraction rules that utilize process components obtained after SPQR-tree decomposition for this purpose. The approach assumes abstraction control mechanism that delivers collection of tasks to be abstracted in the process model.

Once a task to abstract is selected, it uniquely identifies the S-type fragment that contains the task and its structural relation within SPQR-tree. There can be seven types of SPQR-tree edges based on the types of adjacent nodes of S-, P-, and R-type; Q-type fragments are not considered. Edges of (S, S)-type and (P, P)-type are recognized as single fragments of S- or P-type, respectively. Edges are proposed as $(parent, child)$ pairs. Out of seven edge types, four connect S-type nodes: (S, P), (S, R), (P, S), and (R, S). The abstraction rules we propose operate within a single series case process fragment, or assume one of the four stated structural relations of an S-type process fragment.

Sequential (Q-Type) Abstraction A task in a process model can be structured in a sequence with other tasks. We implement abstraction of this task by aggregation with one of its neighbors. Any maximal sequence of tasks is recognized within an S-type process component. Thus, the abstraction is performed locally, i.e., within one process component.

Figure 5 shows an example of a *sequential abstraction* performed inside of the S-type process component. The structure of the original process component is given on the left of the figure. The component is a maximal sequence of three tasks. The example ignores boundary gateway logic, which can be either split or join. In the case task A or C should be abstracted, selection of the neighbor task to aggregate with is obvious—it is task B. However, if task B triggers abstraction, the selection is delegated to the abstraction control mechanism. If structural

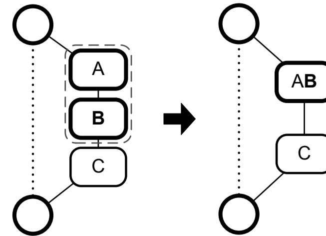


Fig. 5. Sequential abstraction

process model generalization is of interest, abstraction control mechanism can allow nondeterministic task choice. In the example, task A is selected to be aggregated with task B, the corresponding process fragment is enclosed in the region with a dashed borderline and constitutes a single Q-type component.

The process component structure on the right of Figure 5 is the output of the sequential abstraction step. As a result, tasks A and B are aggregated into one task AB that semantically corresponds to the activity of first accomplishing task A and then task B. The process component keeps its structural type—the S-type. Sequential abstraction preserves SPQR-tree structure.

S-Type Abstraction A maximal sequence of tasks in a process model can consist of one task. The situation might occur in the original model or be a result of the prior application of sequential abstractions. This task can be structured in a sequence with process components of P-type or R-type. Within SPQR-tree, such structural relations are captured by (S, P)- or (S, R)-type edges. If it is necessary to abstract the task, aggregation with a neighbor component is performed to result in *S-type abstraction.*

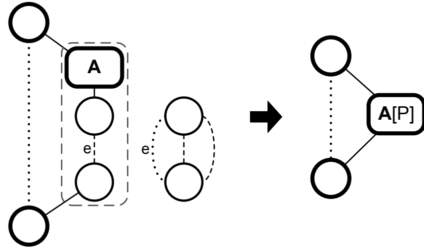


Fig. 6. S-type abstraction

Figure 6 shows an example of S-type abstraction. Task A is designed for abstraction (highlighted with a thick borderline on the left in Figure 6). Task A has no neighbor task—sequential abstraction is not possible. However, the task is in a sequence with the P-type component to form the abstraction fragment in the region enclosed by the dashed borderline. The result of S-type abstraction is given on the right of the figure. Abstraction results in task $A[P]$, which semantically corresponds to the activity of first accomplishing task A and then performing a process fragment captured by the P-type component. S-type abstraction results in SPQR-tree transformation. The branch representing the abstracted component gets removed. Abstraction leads to a restructuring of the S-type component that contained the task which triggered abstraction. However, the component retains its type—the S-type.

S-type abstraction is presented by means of a structural relation of an (S, P)-type edge in SPQR-tree. The procedure for an (S, R)-type edge is analogous. In the example, the boundary gateways of the abstracted component are reduced. In general, if a boundary gateway of an abstracted component is shared with some other process component, it must be preserved in the abstracted model.

P-Type Abstraction Sequential and S-type abstractions tend to generalize S-type components into simple components. A *simple component* is a S-type component composed of a single task (see Figure 3(f)). Simple components are structured by (P, S)- or (R, S)-type edges in SPQR-tree. If a task from

a simple component is selected for abstraction and its parent component is a P-type component, *P-type abstraction* is performed. The task is aggregated with some other child component of the parent component. The selection of the child component to aggregate with is carried out by the abstraction control mechanism.

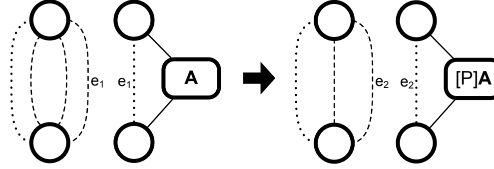


Fig. 7. P-type abstraction

Figure 7 shows an example of P-type abstraction. Task A is selected for abstraction. The task is highlighted with a thick borderline and is the only task of the simple component (shown on the left of Figure 7). The simple component is the child component of the P-type component. It shares virtual edge e_1 with its parent. The result of the P-type abstraction step is given on the right of the figure. Two child components of the P-type component are aggregated into one simple component that contains task $[P]A$. This task semantically corresponds to the execution of two abstracted branches following the type of the boundary gateways. The obtained simple component shares virtual edge e_2 with the parent P-type component.

P-type abstraction results in SPQR-tree transformation. The branch that represents the abstracted component is completely removed. The number of child components of the parent parallel component is reduced by one. If the P-type component initially contains two branches, abstraction results in a single branch. Afterwards, the boundary gateways must be reduced if they do not specify any routing logic, i.e., have single incoming edge and single outgoing edge. In such a case, the P-type component node is further reduced in the SPQR-tree to represent a single task within the next level parent component.

R-Type Abstraction A task intended for abstraction can be contained in a simple component within a process model that is a child of a R-type component. Such a structural relation is specified by a (R, S)-type edge within SPQR-tree. *R-type abstraction* is proposed to handle this situation. As a result of the R-type abstraction the task is aggregated with the whole parent component.

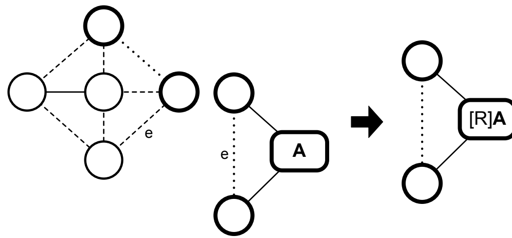


Fig. 8. R-type abstraction

Figure 8 shows an example of R-type abstraction. Task A is selected to be abstracted. The task is highlighted with a thick borderline and is the only task of the simple component on the left of the figure. The simple component is the child of the R-type component (the same component as in Figure 3(b)). The simple component shares the virtual edge e with its parent and corresponds to fragment $S7$ from Figure 2. The result of R-type abstraction step is given on the right of Figure 8. The abstraction results in the aggregation of the whole parent

R-type component into a simple component that has task $[R]A$ and boundary gateways of the R-type component. The task semantically corresponds to the execution of the whole rigid component.

R-type abstraction results in SPQR-tree transformation. The abstracted R-type component gets replaced by a simple component. The branch of the R-type fragment is completely removed. Similar to P-type abstraction, the boundary gateways can be skipped to further reduce the resulting simple component.

5.2 Abstraction Algorithm

Section 5.1 presented four abstraction rules. The rules cover all possible structural relations of a task in a process model. In this section, we organize them into a procedure that handles a single abstraction step of a task. As input, the algorithm obtains a process model, its SPQR-tree decomposition, and a task to abstract. As output, the algorithm delivers a process model with the specified task abstracted. Algorithm 1 formalizes the procedure in pseudo code.

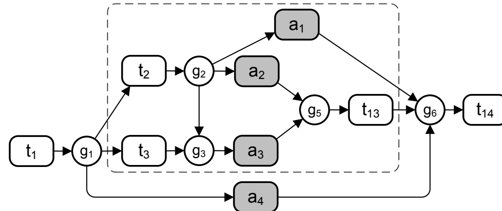


Fig. 9. An abstracted process model

The algorithm orchestrates abstraction rules and attempts to aggregate a minimal number of tasks at each abstraction step; empirical insights for the proposed solution were obtained in [6]. In line 1, the component c which contains task a is identified—it is a S-type component. If c is not a simple component (line 2), then either it has a neighbor task (line 3) or a neighbor component (line 4) that can be aggregated with task a. Otherwise (line 5), abstraction of task a depends on the parent component of c. If c is the root component of SPQR-tree, then p consists of a single task a and there is nothing else to abstract (line 6). Otherwise, get the parent component of c—component cp (line 7). If cp is a P-type component (line 8) or a R-type component (line 9), then P-type abstraction or, respectively, R-type abstraction is performed.

Algorithm 1 The Triconnected Abstraction

TriAbstraction(ProcessModel p, SPQRtree t, Task a)

1. c := component of process model p from SPQR-tree t containing task a
2. **if** c is not a simple component **then**
3. **if** a has neighbor task in c **then** perform *sequential abstraction* of a
4. **else** perform *S-type abstraction* of a
5. **else** // c `is a simple component`
6. **if** c is the root component in t **then** p is already abstracted to one task **return**
7. cp := get a parent component of c in SPQR-tree t
8. **if** cp is P-type component **then** perform *P-type abstraction* of a
9. **if** cp is R-type component **then** perform *R-type abstraction* of a

Algorithm 1 provides a formal relation between an original model and an abstracted one. The triconnected abstraction is the order preserving abstraction. Figure 9 shows the abstraction example of the process model from Figure 1. In the example, a collection of tasks selected for abstraction caused process components $S2$, $S5$, $S6$, $S8$, $S9$, $S10$, $P2$, and $P3$ to get abstracted. These tasks can be t_6, t_8, t_9, t_{10}, and t_{12} (see Figure 1). After abstraction, aggregating tasks a_1, a_2, a_3, and a_4, highlighted with grey background in the figure, conceal the process logic of abstracted components. For instance, task $a1$ is the abstraction of two branches: one composed of tasks t_4 and t_5, and the other of a single task t_6. The type of gateway g_2 specifies the behavioral relation of both branches inside the abstracted task. Task a_1 can be derived using a single P-type abstraction step triggered by task t_6 or by a series of sequential then P-type abstractions if first triggered by either t_4 or t_5. The only R-type component of the process model, shown in the region enclosed by the dashed borderline in Figure 9, is not abstracted. An algorithmic step aimed at abstracting any of the tasks contained within the region will cause the whole component to aggregate into one task.

6 Conclusions

In this paper, we investigated how the SPQR-tree decomposition of process models can help the task of process model abstraction, in particular the discovery of structurally meaningful process model fragments and their aggregation. We defined abstraction rules based on the notion of a process component and proposed their arrangement in the algorithm.

The triconnected abstraction technique defines structural model transformations and can be generalized to any process modeling notation which uses directed graphs as the underlying formalism. Limitations of the triconnected abstraction technique come from restrictions on process model structure. Process models must be free of self-loop structural patterns (should have no cutvertices), and must contain no "mixed" gateways with multiple incoming and multiple outgoing edges (should decompose onto process components). The limitations described above can be overcome by a preprocessing step which transforms mixed gateways into a sequence of first a join, then a split. Alternatively, one can generalize abstraction mechanisms to operate with the RPST decomposition [5].

While the results have proven very useful to our project partner, the abstraction mechanisms only take into account the structure of a business process. In particular, the user of the abstraction might decide that certain activities need to be present in several or even all abstractions. In this case, the application of the mechanisms introduced in this paper needs to be restricted. Therefore, studies regarding the methodology of abstractions need to complement the more technical studies reported in this paper. In future works, we also plan to investigate multiple entry multiple exit components; this should allow further decomposition of rigid case fragments in process models. Theorem 1 gives promising insights into the problem of RPST computation, which we plan to develop in the following work. A promising research direction is to look into how the triconnected ab-

straction technique can be employed for decomposing problems of process model verification—process model behavior analysis, and which model properties are preserved by the abstraction rules.

References

1. OMG: Business Process Modeling Notation, Version 1.2. (January 2009)
2. Keller, G., Nüttgens, M., Scheer, A.: Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)". Technical Report 89, University of Saarland (1992)
3. Petri, C.: Kommunikation mit Automaten. PhD thesis, Institut für instrumentelle Mathematik, Bonn, Germany (1962)
4. Tarjan, R.E., Valdes, J.: Prime Subprogram Parsing of a Program. In: Proceedings of the 7th Symposium on Principles of Programming Languages (POPL), New York, NY, USA, ACM (1980) 95–105
5. Vanhatalo, J., Völzer, H., Koehler, J.: The Refined Process Structure Tree. In: Proceedings of the 6th International Conference on Business Process Management (BPM), Milan, Italy (September 2008) 100–115
6. Polyvyanyy, A., Smirnov, S., Weske, M.: Reducing Complexity of Large EPCs. In: Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (MobIS: EPK), Saarbruecken, Germany (November 2008)
7. Polyvyanyy, A., Smirnov, S., Weske, M.: Process Model Abstraction: A Slider Approach. In: Proceedings of the 12th IEEE International Enterprise Distributed Object Computing Conference (EDOC), Munich, Germany (September 2008)
8. Berthelot, G.: Checking Properties of Nets using Transformation. In: Advances in Petri Nets 1985, London, UK, Springer-Verlag (1986) 19–40
9. Berthelot, G.: Transformations and Decompositions of Nets. In: Advances in Petri nets 1986, London, UK, Springer-Verlag (1987) 359–376
10. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge University Press, New York, NY, USA (1995)
11. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE **77**(4) (1989) 541–580
12. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer Verlag (2007)
13. Aalst, W.: Verification of Workflow Nets. In Azéma, P., Balbo, G., eds.: Application and Theory of Petri Nets, Berlin, Germany, Springer Verlag (1997) 407–426
14. Battista, G.D., Tamassia, R.: Incremental Planarity Testing. In: Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS). (1989)
15. Battista, G.D., Tamassia, R.: On-Line Maintenance of Triconnected Components with SPQR-Trees. Algorithmica **15**(4) (1996) 302–318
16. Hopcroft, J.E., Tarjan, R.E.: Dividing a Graph into Triconnected Components. SIAM Journal on Computing **2**(3) (1973) 135–158
17. Fussell, D., Ramachandran, V., Thurimella, R.: Finding Triconnected Components by Local Replacement. SIAM Journal on Computing **22**(3) (1993) 587–616
18. Gutwenger, C., Mutzel, P.: A Linear Time Implementation of SPQR-Trees. In: Proceedings of the 8th International Symposium on Graph Drawing (GD), London, UK, Springer-Verlag (2001) 77–90
19. Liu, R., Kumar, A.: An Analysis and Taxonomy of Unstructured Workflows. In: Proceedings of the 3rd International Conference on Business Process Management (BPM), Nancy, France (September 2005) 268–284