Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik
an der Universität Potsdam

Nr. 26

The Triconnected Abstraction of Process Models

Artem Polyvyanyy, Sergey Smirnov and Mathias Weske

Potsdam 2008

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de/ abrufbar.

Universitätsverlag Potsdam 2008
http://info.ub.uni-potsdam.de/verlag.htm

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 4623 / Fax: -4625
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik and der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

Erscheinungsweise: aperiodisch.
Das Manuskript ist urheberrechtlich geschützt.
Druck: allprintmedia GmbH, Berlin

ISSN 1613-5652
ISBN 978-3-940793-65-2

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
D-14482 Potsdam, Germany
{Artem.Polyvyanyy,Sergey.Smirnov,Mathias.Weske}@hpi.uni-potsdam.de

Abstract. Business process modeling is a creative task carried out by humans. Business analysts capture process knowledge in models. Process models are decompositions of processes into well recognized business tasks and their structuring by means of control flow. As outcome of a creative practice, models can be composed from tasks of different abstraction levels, i.e., low level tasks with a short and centralized lifecycles and general activities spanning over company departments. In this paper we propose to utilize process model control flow structure for the purpose of generalization of low level tasks to tasks of higher abstraction level. We use SPQR-tree hierarchical process model decomposition for identification of process model components—control flow structures with a self-contained logic suitable for abstraction. The approach allows the highest granularity as compared to existing techniques.

Key words: business process model abstraction, SPQR-tree process model decomposition, triconnected abstraction

1 Introduction

Many engineering disciplines benefit from knowledge reuse. In order to communicate the knowledge it should be formalized. The engineering discipline of business process management proposes to formalize the process knowledge in the form of process models [1]. Research community and industry propose modeling notations to aid in process formalization, e.g., Business Process Modeling Notation (BPMN) [2], Event-driven Process Chains (EPC) [3], Petri nets [4], etc. In general, the formalization initiatives treat processes as collections of individual tasks with isolated logic and execution order constraints defined by control flow. Each process task, as well as each process model fragment is an abstraction of concrete work practices to be accomplished during process execution.

By developing process models, business analysts apply abstractions of real world procedures and capture information sufficient to fulfill the purpose envisioned for a model, e.g., the precise instructions for automating business procedures or the core process logic for the fast process investigation by management. In business process model abstraction initiative [5,6] we aim at study of common process model generalization principles, e.g., in order to allow automated derivation of coarse models—those designed for company management, from detailed ones—those designed to support low level process execution.

In the core of the process model abstraction methodology lies the idea of exchange between different process abstraction entities, e.g., between a process model fragment and a task. Such an exchange step should result in generalization of process model details. Thus, the resulting process model reflects the process on a higher abstraction level. In this paper we propose which process model fragments can be generalized to one task of a higher abstraction level. The methodology for a discovery of the fragments within a process model is suggested. Afterwards, the abstraction rules that allow generalization of identified fragments are defined and organized in the algorithm. The approach allows handling of arbitrary graph structured process models with the highest level of granularity as compared to existing techniques.

The rest of the paper is organized as follows. In the next section we sketch the research field of business process model abstraction, its perspectives and challenges. In section 3 we provide definitions and a basic corollary that give the basis for further discussion. Afterwards, in section 4 the approach of the hierarchical process model decomposition into triconnected graph fragments is presented. Following, in section 5 the fragments are employed for the task of process model abstraction to result in the triconnected abstraction. The paper closes with pointers to related work, ideas on future steps aimed at refinement of the proposed approach, and conclusions that summarize our findings.

2 Business Process Model Abstraction

This section explains business process model abstraction (BPMA). In [6] we summarized the results of the research project together with the industry partner; it resulted in the pattern-based abstraction approach. In [5] we discussed the visionary idea of the slider mechanism for control of process model abstraction. Now, we shape the obtained experience to obtain the common understanding of BPMA.

Abstraction is the process or result of generalization or removing properties from an entity or a phenomenon in order to reduce it to a set of essential characteristics. Therefore, information loss is the fundamental property of abstraction and is its intended outcome. In software engineering, abstraction is the fundamental concept of object-oriented programming paradigm [7]. Engineers abstract from complex reality of objects by extracting only important properties and behavioral aspects. In BPMA, we investigate the problems relevant and specific to abstraction of process model entities. It is a challenge to identify what are the meaningful aggregations of process knowledge aimed to generalize or remove certain process characteristics and to emphasis the other.

BPMA is about finding meaningful aggregations. In the context of process knowledge, the search for meaningful aggregations is about finding process model fragments with well-defined and self-contained process logic, i.e., fragments constituting complete and independent components with precisely stated behavior.

BPMA is about performing abstractions. Identified process model fragments might be removed or replaced by concepts of a higher abstraction level that con-

ceal, but also represent, the logic of the underlying fragments. In both cases of elimination or aggregation of process knowledge, sophisticated handling mechanisms need to be proposed. We refer to such mechanisms as *abstraction steps*.

BPMA is about abstraction control, i.e., combining individual abstraction steps into *abstraction strategies* [6]. Even after that we know how to derive process fragments, and after we have developed formal methods for performing abstractions, still the approach for combining abstractions has to be specified. One can envision manual strategies, where a user specifies what should be abstracted, semi-automated, or fully automated control mechanisms. For instance, an automated mechanism can be guided by the average execution time of tasks included in a model and try to first abstract from tasks which are rarely observed. Of course, in order to allow such abstraction control, models must be additionally annotated with average task execution times.

BPMA is about abstraction properties. Each abstraction step can be characterized by its properties designed to allow their qualitative comparison, and thus encourages introduction of new and improved abstraction step handling mechanisms. For instance, *abstraction smoothness* is a property of abstraction steps that shows how many process tasks are generalized in one step. The less the abstraction smoothness, the more flexibility is allowed for tuning abstraction levels.

BPMA is about preserving process model properties. Abstractions result in model transformations. However, an arbitrary process model transformation cannot be accepted as a process abstraction. Any process abstraction has to ensure certain properties of abstracted process models. The properties are designed to allow relation between original and abstracted models. The key property of process model abstraction is *order preservation*.

Definition 1. An *order preserving abstraction* is an abstraction that assures that neither new task execution constraints can appear after abstraction, nor existing (except for generalized ones) go away.

For instance, assume that task A should be abstracted in the current abstraction step. Let f_A be a process fragment affected by this abstraction step (f_A contains A). As a result of abstraction, fragment f_A gets replaced by task F. If task B also belongs to f_A, information about the ordering constraints between tasks A and B is lost. However, the order preserving abstraction should assure that for any pair of tasks not in f_A, e.g., tasks C and D, the ordering constraints between them are preserved. Furthermore, the order preserving abstraction must guarantee that the execution order constraints between any task not in f_A, e.g., task E, and any task in f_A, task A or B in our example, are the same as between tasks E and F. In the end, the 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 mentioned abstraction aspects. Usually, such a combination is guided by project specific settings. This paper primarily contributes to the BPMA aspects of finding meaningful aggregation fragments and performing abstractions.

3 Preliminaries

This section introduces basic definitions. We start with a process model formalism adopted from [1] that is based on generic modeling concepts. A process model consists of a set of tasks and their structuring by directed control flow edges and gateway nodes that implement process routing decisions.

Definition 2. $P = (N, E, type)$ 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 mutually 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 to each gateway a control flow construct. (N, E) is a connected graph. Each task $t \in N_T$ can have no more than one incoming and 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. There is at least one *process entry* task and one *process exit* task. 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 $(|t\bullet| = 0)$. Each gateway can be either *split* or *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 refer to a portion of a process model we define a process model fragment.

Definition 3. A *process model fragment* $F = (N_F, E_F, type)$ of a process model $P = (N, E, type)$ is a connected subset of process model nodes $N_F \subseteq N$ incident in a subset of process model edges $E_F \subseteq E$.

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

Definition 4. A node $n \in N_F$ can either be a *boundary* node or an *internal* node of a process model fragment F in a process model P:

- A node $n \in N_F$ is a boundary node of F if n is a process entry or a process exit of P, or if there exist edges $e_i \in E_F$ and $e_j \in E \backslash E_F$ adjacent through n. A boundary node can be a *fragment entry* or a *fragment exit* node:
  - A node $n \in N_F$ is a fragment entry 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 $n \in N_F$ is a fragment exit 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 $(\bullet n \subseteq N_F)$
- A non boundary node is an internal node of a process model fragment.

Finally, we identify a special type of a process model fragment—a *process component*.

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

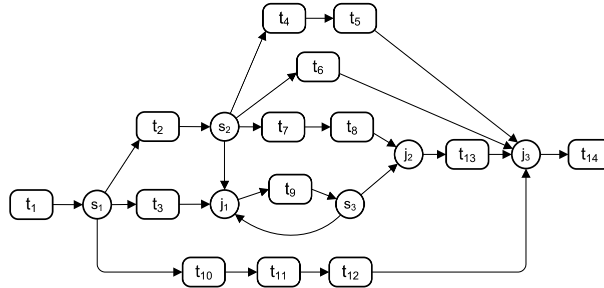


Fig. 1. Example of a process model

This notion of a component was first introduced in [8] as a concept of a *proper subprogram*. A process component is a fragment of a process model such that if control flows through the fragment edge it is assured that it has first entered the fragment through the entry node and is guaranteed to afterwards leave the fragment through the exit node.

A process component can be treated as a self contained block of the process logic with strictly defined boundaries. As such, a process component can be accepted as a unit of meaningful aggregation of the process knowledge. In fact, it is accepted as such in our approach. Therefore, in the following sections we will discuss issues relevant to identification and abstraction of process components in process models.

Further, we require all handled process models to be sound [9]. The requirement is primarily driven by an attempt to avoid hiding of process model errors, e.g., livelocks or deadlocks, during abstraction. Figure 1 gives an example of a process model suitable for abstraction assuming that split and join semantics of the gateway nodes makes it sound, e.g., if all the gateways are of *xor* type. The soundness requirement implicitly states that a process model should have exactly one entry task and exactly one exit task.

4 Triconnected Decomposition

This section explains how to discover process model fragments that relate to the notion of a process component 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 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 which states that a process component is a process model fragment with two boundary nodes (see Definition 5). The boundary nodes of a process fragment

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 process model nodes that disconnect a process fragment from the rest of the process model (e.g., gateways j_1 and s_3 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 of process model nodes divides a model into two fragments: a component candidate and the rest of the process model. In order to find all the fragments with two boundary nodes, the rationale of the described discovery step must be recursively applied to each of the fragments to obtain divide and conquer algorithm design. Each recursive thread terminates once the problem cannot be further subdivided, i.e., it is not possible to find a separation pair in the process fragment.

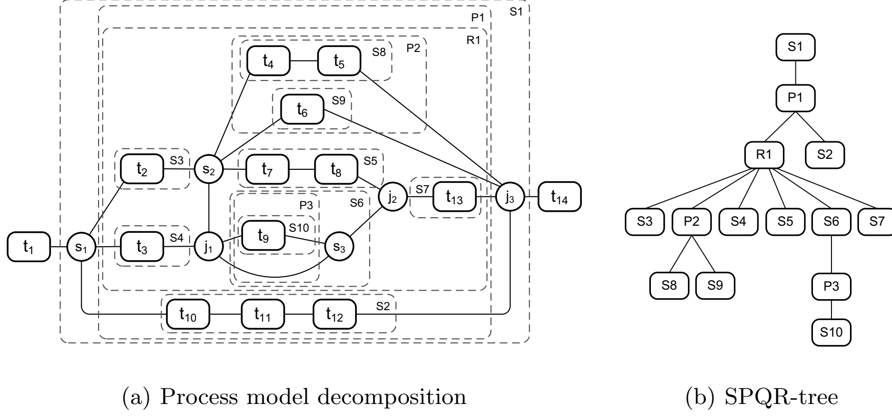
The described algorithm is in fact the algorithm for discovery of triconnected fragments in a graph. Connectivity is a property of a graph. It is known that a graph is k-connected if there exists no set of $k-1$ elements, each a vertex or an edge, whose removal makes the graph disconnected (there is no path, ignoring edge directions, between some pair of nodes 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 fragment.

4.2 SPQR-Tree Decomposition

In order to discover process component candidates we use SPQR-tree decomposition. SPQR-tree decomposition is a hierarchical decomposition of an undirected (edge directions are ignored) biconnected multi graph aimed to identify its triconnected fragments. Process models are connected, but not necessarily biconnected. For example, the process model from Figure 1 can be disconnected by removing the cutvertice gateway s_1. However, it is always possible to make any process model biconnected by adding a back edge that connects a process exit with a process entry (by chance, the soundness requirement guarantees the existence of exactly one process entry and one process exit).

SPQR-tree was first introduced by Di Battista and Tamassia [10] and was further explained in detail in [11]. [12,13,14] show the path towards a linear time complexity algorithm of SPQR-tree decomposition. In its core, SPQR-tree is a hierarchical representation of graph fragments induced by its *split pairs*, where a split pair is either a separation pair, or a pair of adjacent vertices. It was shown in [10,11], that split pairs result in process fragments of four structural types: S, P, Q, and R.

- *Trivial Case.* A split pair is a pair of adjacent graph vertices—the fragment consists of one edge—the Q-type fragment.



(a) Process model decomposition (b) SPQR-tree

Fig. 2. Example of SPQR-tree process model decomposition

- *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 to give a maximal sequence of vertices—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 visualized in Figure 2. Figure 2(a) shows process model fragments. Each fragment is defined by edges that are inside or intersect with a corresponding region (visualized with a dashed line). Fragment names also hint on structural fragment types, e.g., $P1$, $P2$, and $P3$ are all the parallel case fragments of type P. Boundary nodes of a fragment are the nodes incident with the edges crossing the region borderline and are outside of the region. The fragments are either disjoint or fully contained within parent fragments; this is also an implication of the recursion principle for fragment discovery described in section 4.1.

Figure 2(b) shows SPQR-tree that visualizes hierarchical fragment relations. Fragment $P1$ contains fragments $R1$ and $S2$ and is fully contained inside fragment $S1$. Each SPQR-tree node represents *fragment skeleton*, i.e., its basic structure and relations with parent and child fragments. Figure 3 shows fragment skeletons of SPQR-tree nodes from Figure 2(b). The boundary nodes in fragments are highlighted with a thick borderline, e.g., nodes s_1 and j_3 in fragment $R1$ (see Figure 3(b)). Each fragment skeleton might 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 on a parent-child relation. An edge visualized with a dotted line informs on a child relation of the fragment skeleton with another containing the same virtual edge, whereas a dashed line signals on a parent relation. For instance, the fragment skeleton from Figure 3(f) contains one virtual

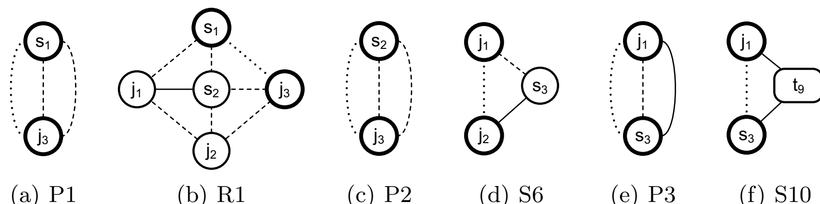


Fig. 3. SPQR-tree fragment skeletons

edge (j_1, s_3), which also hints on 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 the fragment skeleton $P3$ one must "glue" it together with the fragment skeleton $S10$ along the virtual edge (j_1, s_3). Once the fragments are combined, the virtual edge is removed. In general, a graph fragment represented by an SPQR-tree node is obtained by combining all the child fragment skeletons down the SPQR-tree hierarchy.

SPQR-tree provides process model graph decomposition ignoring control flow edge directions. Up till now there is no distinction 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 SPQR-tree fragments in the case when the properties of a process model are preserved while decomposition procedure, i.e., edges are assumed to be directed and nodes distinct as tasks and gateways.

In general, SPQR-tree can be rooted to any node, however in a process model context it makes sense to root the tree to a node representing the fragment containing deliberately introduced back edge (node $S1$ in Figure 2(b)). By doing so, we obtain a structural hierarchical refinement of the process model.

Further observations are: task nodes can only be present (but not always necessarily, 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 model 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 and testifies the latter property.

Until now we have recognized sequences as S-type SPQR-tree fragments. The Q-type fragments stand for original process model graph edges, e.g., edge (s_3, j_2) of fragment skeleton from Figure 3(d). The P-type fragments (see Figures 3(a), 3(c), and 3(e)) allow identification of block and loop structures within

process model graphs. 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 j_1 and s_3). 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) [15].

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

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

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

First, let us show that any boundary node of a process model fragment induced by SPQR-tree decomposition can either be the fragment entry or the fragment exit, but not both. 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 fragment is the fragment entry or the 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 incoming edges or a set of outgoing edges consists of one element (Definition 2). Thus, the relation of this one edge (either it belongs to the fragment or not) defines the relation of the whole set. Therefore, the boundary gateway can only expose the logic of the fragment entry or the fragment exit.

The rationale towards the formal proof of the "pure" logic nature of a boundary fragment node can be approached as follows. Let $p = (N, E, type)$ be a process model, $f = (N_f, E_f, type)$ be a process model fragment of p with a boundary node $n \in N_f$. Let us define auxiliary predicates:

- $i : E \times N \rightarrow \{true, false\}$ is $true$ iff $e \in E$ is the input edge of node $n \in N$,
- $o : E \times N \rightarrow \{true, false\}$ is $true$ iff $e \in E$ is the output edge of node $n \in N$.

Let us define the predicate that checks if a boundary node $n \in N_f$ can expose the entry logic of a process model fragment f:

$$canEnter(n, f) = \exists e_1 \in E \backslash E_f \exists e_2 \in E_f : i(e_1, n) \wedge o(e_2, n).$$

Analogously, one can check whether a node can expose the exit logic of a process model fragment:

$$canExit(n, f) = \exists e_1 \in E_f \exists e_2 \in E \backslash E_f : i(e_1, n) \wedge o(e_2, n).$$

In order to show that any boundary fragment node cannot be the entry and the exit at the same time, one must show that the logical statements

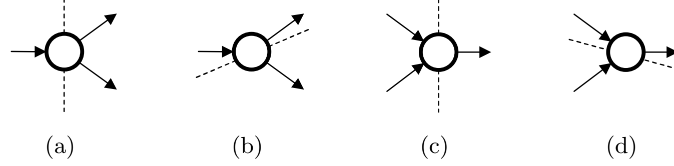


Fig. 4. All possible combinations for edge separation on internal and external fragment edges for a boundary gateway that connects three edges

$canEnter(n, f) \models \neg canExit(n, f)$ 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:

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

In the case when 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 is however not possible as e_1 and e_3 belong to different sets which are disjoint: E and $E \backslash E_f$. 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 testifies the pure logic of any boundary fragment gateway.

Figure 4 shows all possible combinations of internal and external fragment edges incident with a boundary gateway that connects three edges. The dashed line defines separation of edges on the fragment internal and the fragment external edges. Regardless of a separation and a gateway type, it is only allowed for control flow to penetrate the fragment boundary in one direction (either to enter or to exit the fragment).

Finally, it is required to show that only one installation of boundary nodes is possible, i.e., one of the nodes is the fragment entry and another is the 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). Both cases can be reduced to one. For instance, in case of a fragment with two exits, we can discuss a two entries fragment formed by the edges outside the two exits fragment. A two entries process model fragment violates the requirement of a structurally sound process model which states that each node in the process model is on the path from the process entry to the process exit. The process entry and the process exit are always in one process model fragment which is not a two entries fragment. Otherwise, the process entry has a node that leads to it. Once we proceed with the execution of a task inside of a two entries fragment we never leave it. Therefore, any node inside of the two entries fragment can not be on the path from the process entry to the process exit. Thus, one of the boundary nodes must be the fragment exit. □

5 Triconnected Abstraction

In this section we present the process model abstraction principles. The approach is based on the process model decomposition described in section 4. First, we define abstraction rules. Afterwards, we provide the triconnected abstraction algorithm.

5.1 Abstraction Rules

The developed process model abstraction mechanism proposes to interchange process abstraction concepts of process model fragments with process tasks of higher abstraction level. In this section we present abstraction rules that for this purpose utilize process components obtained after SPQR-tree decomposition. The approach assumes manual abstraction control, i.e., a user specifies which process task, or a collection of tasks, of the original process model needs to be generalized. Afterwards, the process components containing these tasks get abstracted.

Once a task to be abstracted is selected, it uniquely identifies a S-type component and its structural relation within SPQR-tree. In total, there can be seven types of SPQR-tree edges based on the types of connected nodes of S-, P-, and R-type (Q-type fragments are not considered). The edges of (S, S)-type and (P, P)-type are not possible in SPQR-tree, such structural hierarchies are recognized as single fragments of S- or P-type respectively. Out of seven possible edge types, four connect a node of S-type: (S, P), (S, R), (P, S), and (R, S) (edges are proposed as $(parent, child)$ pairs). Following, the proposed abstraction rules operate inside a single process component of S-type, or assume one of the four stated structural S-type process component relations.

Sequential (Q-Type) Abstraction A task in a process model can be structured in a sequence with other tasks. This task can be abstracted in the process model by aggregating it with one of its neighbors. Any maximal sequence of process tasks is recognized within a single 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 the S-type process component. The original process component structure is given at the left of the figure. The component corresponds to 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 is the one to be abstracted, the selection of the neighbor to aggregate is obvious—it is task B. However, in the case when task B triggers abstraction, the selection should be carried out by the abstraction control mechanism. In the case

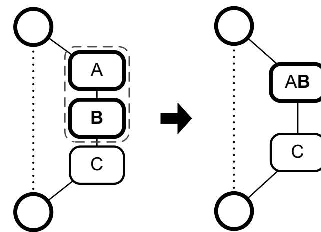


Fig. 5. Sequential abstraction example

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

The process component structure at 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 B.

As a result of abstraction, the sequence of three tasks turned into the sequence of length two. The process component preserved its structural type—the S-type. SPQR-tree stayed unchanged. Sequential abstraction is characterized by abstraction smoothness of 2, i.e., the fragment composed of two tasks is replaced by a single task.

S-Type Abstraction A maximal sequence of tasks can consist of one task in a process model, i.e., there is no direct task neighbor. Such a situation might also result after applying sequential abstractions. However, a task can be in a sequence with process components of P- or R- types. Such structural relations are captured in (S, P) and (S, R) edges within SPQR-tree. If it is necessary to abstract the task, aggregation with a neighbor component is performed to result in *S-type abstraction*.

Figure 6 shows the example of S-type abstraction. Task A is designed to be abstracted (highlighted with a thick borderline at the left in Figure 6). Task A has no task neighbor—sequential abstraction is not possible. However, the task is in a sequence with the P-type component to give the abstraction fragment enclosed in the region with dashed borderline. The result of the S-type abstraction step is given at the right of the figure. The abstraction results in one task $A[P]$ that semantically corresponds to the activity of first accomplishing task A and then performing a process fragment captured in the P-type component.

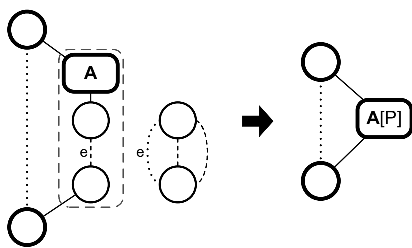


Fig. 6. S-type abstraction example

S-type abstraction results in SPQR-tree transformation. The tree branch that represents abstracted component is completely removed. The abstraction leads to restructuring of the S-type component that contained the task which triggered abstraction. However, the component remains of the same type—the S-type.

S-type abstraction was presented by means of aggregation of (S, P) SPQR-tree edge. In the case of (S, R) edge, the procedure is analogous. In the example, the boundary gateways of aggregated component were removed. However, if one of the boundary gateways of the aggregated component is shared with boundary gateway of the parent S-type component, it must be preserved. S-type abstraction is characterized by smoothness

of equal or higher than 2. The smoothness of S-type abstraction step is 2 if the aggregated component contains only one task, e.g., a P-type component with two paths—one empty path and another containing a single task. We assume each process component to include at least one task.

P-Type Abstraction Sequential and S-type abstractions tend to generalization of S-type components to *simple components*. Simple components are the S-type components with two boundary gateways and a single task (see Figure 3(f)). Simple components are the leaves of SPQR-tree and are structured by (P, S) or (R, S) edges. In case a task from a simple component is selected to be abstracted and it has a P-type parent component, the *P-type abstraction* is performed. The task is aggregated with some branch (child component) of the parent component. The selection of the child component to aggregate with is performed by the abstraction control mechanism.

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

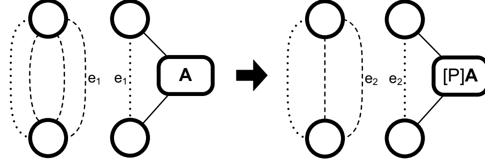


Fig. 7. P-type abstraction example

P-type abstraction results in SPQR-tree transformation. The tree branch that represents abstracted component is completely removed. Thus, the number of child components of the parent parallel component is reduced by one. In the case when a P-type component initially contains two branches, the abstraction results in a single branch. Afterwards, the boundary gateways must be reduced in the case they do not specify any routing logic, e.g., have a single entry and a single exit 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.

P-type abstraction is characterized by abstraction smoothness of equal or higher than 1. The smoothness of P-type abstraction step is 1 if a simple component is aggregated with an empty path of the parent parallel component.

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

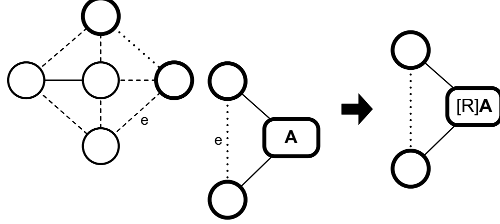


Fig. 8. R-type abstraction example

Figure 8 shows the example of the R-type abstraction. Task A is designed to be abstracted. The task is highlighted with a thick borderline and is the only task of the simple component at the left of the figure. The simple component is the child of the R-type component (same component as in the example from Figure 3(b)). The simple component shares virtual edge e with its parent and corresponds to component $S7$ from Figure 2. The result of the R-type abstraction step is given at the right of Figure 8. The abstraction results in the whole parent R-type component aggregation into a simple component with task $[R]A$ and boundary gateways of the R-type component. The obtained task semantically corresponds to the execution of the whole rigid component.

R-type abstraction results in SPQR-tree transformation. The abstracted R-type component node gets replaced by a simple component node. Similar to P-type abstraction, the boundary gateways can get eliminated to further result in reduction of resulted simple component. R-type abstraction is characterized by smoothness of equal or higher than 1. The abstraction smoothness of R-type abstraction step is 1 if the simple component is the only S-type child component of the rigid component, i.e., all the other paths are empty. However, in most cases one should expect higher abstraction smoothness values.

5.2 Abstraction Algorithm

In this section we propose the triconnected process model abstraction algorithm. In section 5.1 we presented four abstraction rules. Now, we organize them in a procedure that handles a single task abstraction step. As input, the algorithm obtains a process model, its SPQR-tree decomposition structure, and a task to abstract. As output, the algorithm provides the process model with the specified task abstracted. Alg. 1 formalizes the algorithm in a pseudo code.

Algorithm 1 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

Alg. 1 orchestrates individual abstraction rules to pursue best abstraction smoothness; empirical insights for the solution were obtained in [6]. In line 1, component c containing task a is identified—it is a S-type component. If c is not simple (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 for process p, then p consists of a single task a and there is nothing else to abstract (line 6). If process model has other tasks than a, 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 or R-type abstraction has to be performed respectively.

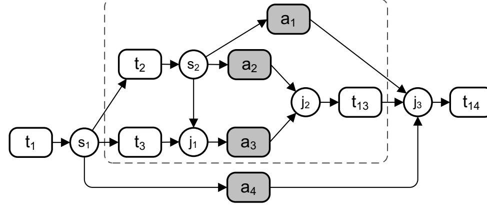


Fig. 9. Abstracted process model

Theorem 2 *A triconnected abstraction is an order preserving abstraction.*

Proof. The property of order preservation for triconnected abstraction can be deduced from the four types of abstraction rules it is composed of: Q-, S-, P-, and R-type, and the notion of a process model component (Definition 5). Each abstraction operates with a fragment obtained after SPQR-tree decomposition of a process model. For any pair of tasks outside the fragment the ordering constraints are not affected. After abstraction the fragment is replaced by a single path between boundary fragment nodes with a single aggregating task on that path. This task represents the logic of the aggregated fragment and is in the same order relation to other tasks of the model as any of the aggregated tasks; the aggregating task and all the aggregated tasks communicate with the process model through the same logic of the same boundary fragment nodes. □

Figure 9 shows the abstraction example of the process model from Figure 1. In the example, the "to be abstracted" tasks selection 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. The only R-type component of the process model (enclosed into the region with dashed borderline in Figure 9) is not abstracted. The wish to abstract any of the contained tasks will cause the complete component to aggregate into one task; the smoothness of such abstraction step gets as high as 6. Also, an order in which tasks are selected for abstraction can influence abstraction smoothness. In the future work, we would like to concentrate on lowering abstraction smoothness with a help of more fine-granular process model decompositions and sophisticated abstraction control mechanisms.

6 Related Work and Conclusions

In this paper we investigated SPQR-tree process model decomposition for the task of process model abstraction—discovery of meaningful process model fragments and their aggregation. We defined abstraction rules based on the notion of the process component and proposed their arrangement in the algorithm.

The topic of business process model abstraction evolved from the project with the industry partner that had a demand for generalization of its detailed process models. First, we proposed the pattern-based abstraction [6]. Its limitations originate from the impossibility to foresee all the patterns that need to be handled. Afterwards, we looked into the applicability of the single-entry-single-exit (SESE) process model decomposition [16] for abstraction of process models and evaluated the abstraction smoothness of the approach on a set of real-world process models. Comparing to the pattern-based approach, SESE-based abstraction allows systematic treatment of all recognized control flow structures that subsume the patterns from [6]. The proposed technique of triconnected abstraction outperforms SESE-based abstraction in smoothness, i.e., allows more fine-granular process component discovery. For example, a SESE decomposition cannot discover process model fragments $S6$, $P1$, $P2$, $P3$, and $R1$ shown in Figure 2. Triconnected abstraction is by now the best known to us technique that allows discovery of fine-granular process components and their further order preserving abstraction in graph structured process models. The proposed abstraction methodology is validated in the implemented tool prototype.

The refined process structure tree (RPST) decomposition technique [17] introduced the tree of triconnected components [8], known from the compiler theory of sequential programs, to the business process management community. The tree of triconnected components is in the core of SPQR-tree decomposition. RPST further extends the decomposition methodology to allow identification of process components when a process model allows existence of "mixed" gateways, i.e., gateways with multiple entries and multiple exits. The triconnected abstraction can be extended to handle additional component structures introduced by RPST. Simple transformation of all mixed gateways of a process model to a sequence of "pure" join and then "pure" split allows as-is application of triconnected abstraction algorithm (see Theorem 1). Also, there is an ongoing work for transformation of the problematic mixed gateways in a model to allow derivation of RPST process components with SPQR-tree decomposition. Finally, in order to further increase the granularity of process components, we plan to investigate multiple-entry-multiple-exit (MEME) components.

References

1. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer Verlag (2007)
2. OMG: Business Process Modeling Notation Specification. 1.1 edn. (January 2008)
3. Keller, G., Nüttgens, M., Scheer, A.: Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)". Technical Report Heft 89,

Veröffentlichungen des Instituts für Wirtschaftsinformatik University of Saarland (1992)
4. Petri, C.: Kommunikation mit Automaten. PhD thesis, Bonn, Germany (1962)
5. Polyvyanyy, A., Smirnov, S., Weske, M.: Process Model Abstraction: A Slider Approach. In: EDOC'08: Proceedings of the 12th IEEE International Enterprise Distributed Object Computing Conference, Munich, Germany, IEEE Computer Society (September 2008)
6. Polyvyanyy, A., Smirnov, S., Weske, M.: Reducing Complexity of Large EPCs. In: Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (MobIS: EPK), Saarbruecken, Germany (November 2008)
7. Booch, G., Maksimchuk, R.A., Engel, M.W., Young, B.J., Conallen, J., Houston, K.A.: Object-Oriented Analysis and Design with Applications (3rd Edition). Addison-Wesley Professional (April 2007)
8. Tarjan, R.E., Valdes, J.: Prime Subprogram Parsing of a Program. In: POPL'80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (1980) 95–105
9. Aalst, W.: Verification of Workflow Nets. In Azéma, P., Balbo, G., eds.: Application and Theory of Petri Nets, volume 1248 of LNCS, Berlin, Germany, Springer Verlag (1997) 407–426
10. Battista, G.D., Tamassia, R.: Incremental Planarity Testing. In: FOCS. (1989) 436–441
11. Battista, G.D., Tamassia, R.: On-Line Maintenance of Triconnected Components with SPQR-Trees. Algorithmica **15**(4) (1996) 302–318
12. Fussell, D., Ramachandran, V., Thurimella, R.: Finding Triconnected Components by Local Replacement. SIAM J. Comput. **22**(3) (1993) 587–616
13. Gutwenger, C., Mutzel, P.: A Linear Time Implementation of SPQR-Trees. In: GD'00: Proceedings of the 8th International Symposium on Graph Drawing, London, UK, Springer-Verlag (2001) 77–90
14. Hopcroft, J.E., Tarjan, R.E.: Dividing a Graph into Triconnected Components. SIAM Journal on Computing **2**(3) (1973) 135–158
15. Liu, R., Kumar, A.: An Analysis and Taxonomy of Unstructured Workflows. In: Business Process Management. (2005) 268–284
16. Johnson, R., Pearson, D., Pingali, K.: The Program Structure Tree: Computing Control Regions in Linear Time, ACM Press (1994) 171–185
17. Vanhatalo, J., Völzer, H., Koehler, J.: The Refined Process Structure Tree. In: BPM. (2008) 100–115

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
25	978-3-940793-46-1	**Space and Time Scalability of Duplicate Detection in Graph Data**	Melanie Herschel, Felix Naumann
24	978-3-940793-45-4	**Erster Deutscher IPv6 Gipfel**	Christoph Meinel, Harald Sack, Justus Bross
23	978-3-940793-42-3	**Proceedings of the 2nd. Ph.D. retreat of the HPI Research School on Service-oriented Systems Engineering**	Alle Professoren des HPI
22	978-3-940793-29-4	**Reducing the Complexity of Large EPCs**	Artem Polyvyanyy, Sergy Smirnov, Mathias Weske
21	978-3-940793-17-1	**"Proceedings of the 2nd International Workshop on e-learning and Virtual and Remote Laboratories"**	Bernhard Rabe, Andreas Rasche
20	978-3-940793-02-7	**STG Decomposition: Avoiding Irreducible CSC Conflicts by Internal Communication**	Dominic Wist, Ralf Wollowski
19	978-3-939469-95-7	**A quantitative evaluation of the enhanced Topic-based Vector Space Model**	Artem Polyvyanyy, Dominik Kuropka
18	978-939-469-58-2	**Proceedings of the Fall 2006 Workshop of the HPI Research School on Service-Oriented Systems Engineering**	Benjamin Hagedorn, Michael Schöbel, Matthias Uflacker, Flavius Copaciu, Nikola Milanovic
17	3-939469-52-1 / 978-939469-52-0	**Visualizing Movement Dynamics in Virtual Urban Environments**	Marc Nienhaus, Bruce Gooch, Jürgen Döllner
16	3-939469-35-1 / 978-3-939469-35-3	**Fundamentals of Service-Oriented Engineering**	Andreas Polze, Stefan Hüttenrauch, Uwe Kylau, Martin Grund, Tobias Queck, Anna Ploskonos, Torben Schreiter, Martin Breest, Sören Haubrock, Paul Bouché
15	3-939469-34-3 / 978-3-939469-34-6	**Concepts and Technology of SAP Web Application Server and Service Oriented Architecture Products**	Bernhard Gröne, Peter Tabeling, Konrad Hübner
14	3-939469-23-8 / 978-3-939469-23-0	**Aspektorientierte Programmierung – Überblick über Techniken und Werkzeuge**	Janin Jeske, Bastian Brehmer, Falko Menge, Stefan Hüttenrauch, Christian Adam, Benjamin Schüler, Wolfgang Schult, Andreas Rasche, Andreas Polze
13	3-939469-13-0 / 978-3-939469-13-1	**A Virtual Machine Architecture for Creating IT-Security Labs**	Ji Hu, Dirk Cordel, Christoph Meinel