Unveiling Hidden Unstructured Regions in Process Models

Artem Polyvyanyy[1], Luciano García-Bañuelos[2]*, and Mathias Weske[1]

[1] Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2–3, D-14482 Potsdam, Germany
(Artem.Polyvyanyy,Mathias.Weske)@hpi.uni-potsdam.de
[2] Institute of Computer Science, University of Tartu
J. Liivi 2, Tartu 50409, Estonia
luciano.garcia@ut.ee

Abstract. Process models define allowed process execution scenarios. The models are usually depicted as directed graphs, with gateway nodes regulating the control flow routing logic and with edges specifying the execution order constraints between tasks. While arbitrarily structured control flow patterns in process models complicate model analysis, they also permit creativity and full expressiveness when capturing non-trivial process scenarios. This paper gives a classification of arbitrarily structured process models based on the hierarchical process model decomposition technique. We identify a structural class of models consisting of block structured patterns which, when combined, define complex execution scenarios spanning across the individual patterns. We show that complex behavior can be localized by examining structural relations of loops in hidden unstructured regions of control flow. The correctness of the behavior of process models within these regions can be validated in linear time. These observations allow us to suggest techniques for transforming hidden unstructured regions into block-structured ones.

Keywords: Process structure tree, process model analysis, process model correctness, process model transformation

1 Introduction

Software engineers employ principles of conceptual modeling to encapsulate all the information about real world entities in formal models, e.g., specification of behavior. The research field of business process management [1] investigates the problem of capturing behavioral aspects of real world entities in process models. Process models are widely used to design, analyze, and improve how companies organize operational processes. Furthermore, process models are used in the design of distributed software systems and to provide a blueprint for systems that realize the processes.

* Supported by the Estonian Science Foundation and the European Regional Development Fund through the Estonian Centre of Excellence in Computer Science.

Process modeling languages, e.g., BPMN [2], formalize process models as directed graphs, where edges specify execution order constraints, task nodes represent business activities, and gateway nodes define the control flow routing logic of a model. Graph-based process modeling languages allow a great level of expressiveness for business analysts capturing process scenarios in models. In order to fulfill business goals, analysts can come up with arbitrarily structured process models. However, the degree of freedom which analysts gain is the primary source of errors [3,4,5]. One can easily end up with a model which encodes undesired scenarios, e.g., ones that never reach the goal state of a process or ones that result in the uncontrolled concurrent execution of business activities. Additionally, process models with complex structures aggravate the computational complexity of model analysis tasks, e.g., the task of checking the correctness of the process model's behavior—the (behavioral) soundness [6]. One widely accepted solution to address identified problems is to restrict the freedom, i.e., to restrain the structural principles of the composing models. This, however, limits the creativity during the business process model design phase. Furthermore, the restrictions on allowed structural patterns often result in models with replicated task nodes as well as replicated structural patterns, which are heavily introduced in order to capture envisioned scenarios.

In this paper, we use the SPQR-tree process model decomposition technique, known from compiler theory [7] and introduced to the business process management field in [8], to derive a structural classification of process models. We identify a structural class of process models which are composed solely of block-structured fragments. However, when these fragments are combined in a model they form regions of unstructured process behavior. In these regions control flow can enter and afterwards leave an individual fragment through the same node in a looping pattern. We refer to such regions as "hidden" unstructured regions.

Hidden unstructured regions have been identified before [7,8]. They are referred to as *non-prime subprograms* in [7] and as *directed bond fragments* in [8]. However, we take a step forward as we provide a characterization of their structural properties and describe a linear time method for verifying the soundness of the underlying process model. Additionally, we show that existing flow graph restructuring techniques can be adapted to transform hidden unstructured regions. Restructured process models become suitable for translation to block-structured languages such as BPEL [9,10], e.g., for execution.

The rest of the paper is organized as follows: The next section discusses the related work. In section 3, the SPQR-tree decomposition technique is presented. First, we exemplify the technique with the help of undirected graphs. Afterwards, we discuss the implications of the decomposition if performed on process models. In section 4, we present the notion of a behavioral correctness for process models. Section 5 presents structural process model classes and discusses the identification and behavioral analysis of hidden unstructured regions in process models. Section 6 shows that existing techniques can be adapted to transform hidden unstructured regions into block-structured process models. The paper closes with ideas on future steps and conclusions that summarize our findings.

2 Related Work

Kiepuszewski, Hofstede, and Bussler [4] identify the classes of unstructured process graphs that can be transformed to equivalent structured versions. Their classification relies on two parameters: the presence of parallel control flow (i.e., *and* gateways) and the presence of loops. Three categories are identified. The first category is restricted to *xor* logic only and allows loops. As the underlying logic is simple, one can use techniques developed for flowchart restructuring to transform a model in this category to an equivalent structured form (e.g., [11]). They also showed that restructuring can be achieved either by node duplication or by the use of auxiliary variables, but some cases can only be restructured with the use of the aforementioned variables. The second category comprises acyclic models which allow internal *and* logic. As the authors point out, the presence of *and* gateways may induce structural problems such as deadlocks or lack of synchronization. They further analyze this category and conclude that most unstructured models carrying internal *and* logic cannot be restructured except for a subset of the models with an overlapping structure [12]. Finally, the authors describe a category consisting of models with loops and with internal *and* logic. They concluded that, in general, well-behaved cyclic process can be restructured. But, the authors also presented a well-behaved cyclic process that cannot be restructured that uses variables to synchronize some of the parallel paths. An open question remains, whether there exist well-behaved arbitrary processes without the variables that cannot be restructured.

Liu and Kumar [5] extended the work by Kiepuszewski et al. Their aim was to identify the source of unstructuredness in process models and to investigate the scenarios and configurations giving rise to structural conflicts. Their taxonomy is built on top of three dimensions. First, the notion of *corresponding control elements* or *corresponding pair*, i.e., a split and a join gateways which have at least two distinct control flow paths going from the split gateway and reaching the join gateway. Corresponding pairs are further classified in *proper* ones if both gateways use the same logic and, otherwise, in *mismatched* ones. Second, the notion of *nesting of corresponding pairs*. Nesting is further divided in proper and improper, and quantified according to the number of intermediate improperly nested gateways: first-order improper nested, if only one intermediate gateway exists, and so on. The third parameter being considered is the presence of loops in the process model. The authors proceed by describing families of process models which are variations on the three dimensions. The first family corresponds to first-order improper nested graphs. To illustrate the characteristics of this family, they present a process model topology with four gateways and enumerate all variants (changing the gateway logic) and identify the potential structural conflicts. The second family corresponds to second-order improper nested graphs, for which a similar analysis is performed. In this context the authors identify the overlapping structure as the only sound configuration allowing mismatched corresponding pairs. The last family analyzed is that of first-order improper nested graphs with loops. The analysis of higher level improper nesting has been left open.

In this paper, we do some first steps in investigating how a process structure tree, in particular the SPQR-tree, can be employed for the efficient analysis of process model behavior. Similar to [4], we identify and investigate different structural process model classes. However, their focus is on restructuring suitability. In contrast, we want to investigate the structural fragment types discovered by the SPQR-tree decomposition. The fragments are finer-grained, and allow us to analyze both their structural and their behavioral properties. In this way, we identify classes of fragments for which correctness of process model behavior can be validated in linear time. This also differs from the approach described in [5]. Their study is based on the analysis of a single case for each family of process models with an arbitrary number of gateways, for which the set of possible variants on gateway logic has been enumerated. However, we consider that such approach lacks of generality. In our approach, the analysis is simplified by the fact that the SPQR-tree decomposition by itself gives the hints about the structural properties. We base our reasoning on this properties and derive the restrictions to be observed by sound process models.

3 The SPQR-Tree Decomposition

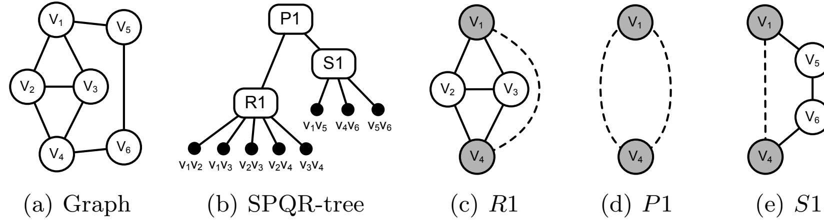
This section explains the technique of structural process model decomposition. First, the technique is exemplified by performing the decomposition of undirected graphs. Afterwards, we discuss the implications of the decomposition technique if performed on process models.

3.1 Graph Decomposition

The SPQR-tree decomposition is a decomposition of an undirected biconnected multigraph aimed at identifying its triconnected fragments. 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. 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.

The algorithm for the discovery of triconnected fragments of a graph was first proposed in [13]. Afterwards, in [7], the algorithm was applied to sequential program parsing. It was proposed to decompose the directed program graph into the *parse tree* (or *the tree of the triconnected components*). The parse 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. The parse tree was studied as SPQR-tree in [14,15]. [13,16,17] show the path towards a linear time complexity algorithm implementation of the SPQR-tree decomposition. The decomposition results in process fragments of four structural types: S-, P-, Q-, and R-type fragments.

- *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.



(a) Graph (b) SPQR-tree (c) $R1$ (d) $P1$ (e) $S1$

Fig. 1. SPQR-tree decomposition of a graph

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

Figure 1 exemplifies the SPQR-tree decomposition. Figure 1(a) shows an undirected graph, whereas Figure 1(b) gives its SPQR-tree decomposition. The names of SPQR-tree nodes hint at the structural types of their underlying fragments, e.g., $S1$ is the series case fragment, $P1$ is the parallel case fragment, and $R1$ is the rigid case fragment. The $v_i v_j$ nodes represent Q-type structural fragments—graph edges.

Each SPQR-tree node represents a *fragment skeleton*, i.e., the basic structure of a fragment and its relations with other fragments. Figures 1(c), 1(d), and 1(e) show the fragment skeletons of the SPQR-tree from Figure 1(b). Each fragment skeleton consists of the original graph edges (drawn with solid lines) and *virtual edges* (drawn with dashed lines). Each virtual edge is shared between two fragment skeletons and hints at a structural relation between skeletons in the SPQR-tree, whereas each original edge is contained in one skeleton. The nodes that form the separation pairs of the graph are highlighted with a grey background, e.g., nodes v_1 and v_4 in fragment skeleton $R1$. These nodes, when removed, disconnect fragment $R1$ from the rest of the graph. Observe that separation pairs are only discovered in a set of graph nodes with the number of coincident edges higher than two. This aligns with the definition of the series case fragment. In order to obtain a maximal sequence, any adjacent S-type fragments within the SPQR-tree must be combined. Otherwise, one can discover a combinatorial set of separation pairs within the series case fragments, e.g., $v_1 v_4$, $v_1 v_6$, and $v_4 v_5$ within the $S1$ fragment skeleton (see Figure 1(e)). Similarly, the structural relation of a parallel case fragment within a parallel case fragment should be recognized as a single P-type fragment.

In order to obtain the original graph, one must "glue" all the fragment skeletons pair-wise in any order, along the virtual edges, i.e., merge adjacent vertices of the shared virtual edge. Once fragment skeletons are combined, the shared virtual edge is removed.

3.2 Process Model Decomposition

In this section, we examine fragments obtained after the SPQR-tree decomposition of a process model. We start with the definition of a process model adopted from [1], which is also the generalization of the definition proposed in [18]—gateways that are both split and join are allowed in a process graph.

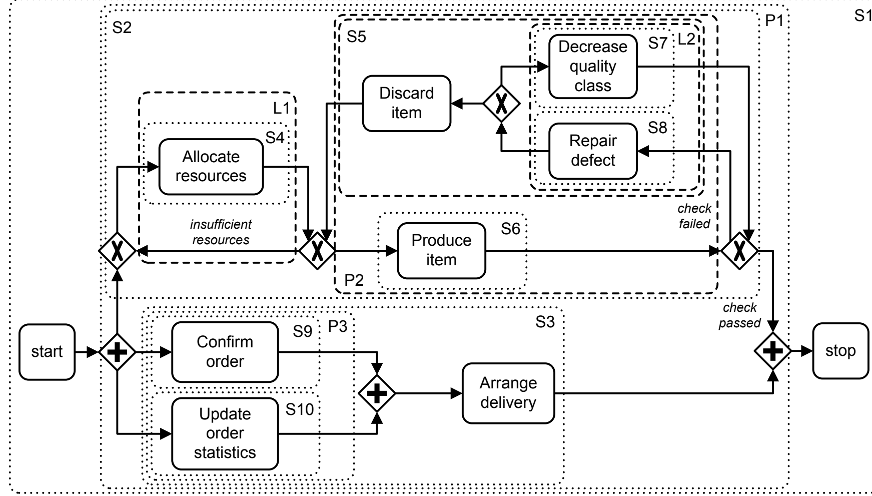


Fig. 2. A process model and its SPQR-tree node fragments

Definition 1 (Process Model). A *process model* is a tuple $P = (N, E, type)$, where:

- $N = N_T \cup N_G$ 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,
- $type : N_G \rightarrow \{and, xor\}$ is a function that assigns a control flow construct to each gateway.

Moreover, (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 $g \in N_G$ has either more than one incoming edge, or more than one outgoing edge. 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 ($|\bullet g| > 1 \wedge |g\bullet| = 1$). A gateway $g \in N_G$ is a *mixed* gateway if ($|\bullet g| > 1 \wedge |g\bullet| > 1$).

We require process models to be structurally correct, i.e., structurally sound. Figure 2 gives an example of a structurally sound process model.

Definition 2 (Structural Soundness). A process model is *structurally sound* if there is exactly one process entry, exactly one process exit, and each process model node is on a path from the process entry to the process exit.

Similar to the case with graphs, one can construct an SPQR-tree of a process graph and obtain the decomposition of a process model's control flow edges. In general, an SPQR-tree can be rooted to any node. However, in the context of process models it makes sense to root the tree to the node which represents an S-type fragment that contains a process entry and a process exit. In this case, one obtains a structural process model hierarchy [8,18,7], i.e., a containment hierarchy of sets of process model edges. The hierarchy shows a refinement of structural patterns that collectively build up a process model, starting with a top level series case fragment.

In order to properly address parts of a process model, we define a process fragment as a connected subgraph of a process model.

Definition 3 (Process Fragment). A *process fragment* of a process model $P = (N, E, type)$ is a tuple $F = (N_F, E_F, type_F)$, where $N_G \subset N$ is a set of gateways of P, which 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 function *type* of P to the set $N_F \cap N_G$.

The fact that the SPQR-tree decomposition of a process model delivers a concrete hierarchical containment of edges (the root node is always fixed) allows to uniquely identify process fragments that are represented by fragment skeletons. A process fragment that corresponds to a certain node, or a fragment skeleton, of the SPQR-tree is obtained by gluing its fragment skeleton with all its descendent skeletons in the SPQR-tree hierarchy. We refer to such a process fragment as an *SPQR-tree node fragment*. For instance, the $S1$ fragment in Figure 3 is a series case fragment that corresponds to the whole process model shown in Figure 2. In general, the SPQR-tree root fragment corresponds to the whole process model. Figure 3 shows the SPQR-tree decomposition of the process model from Figure 2. Observe that the Q-type fragments, i.e., the control flow edges, are not visualized for simplicity reasons. In Figure 2, each SPQR-tree node fragment is enclosed in the region with a dashed or dotted borderline, i.e., a fragment is formed by control flow edges enclosed in or intersecting the region and nodes adjacent in a set of edges that form the fragment.

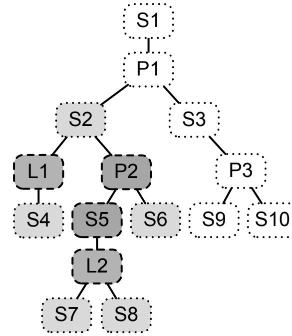


Fig. 3. SPQR-tree of the process model from Figure 2

In the following, we give a classification of nodes contained in a process fragment.

Definition 4 (Boundary, Internal, Entry, Exit nodes). A node is either a boundary or an internal node of a process 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 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:
  - 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 fragment.

We also employ the directed property of the process graph edges and the definition of entry and exit nodes to recognize a special type of process fragments—process components.

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

This notion of a component was first introduced in [7] as a concept of a *proper subprogram*. Process fragments that are not components are *non-components*. The importance of process fragments with a single entry and a single exit logic was identified in [19,8]. In [18], we showed how process components can be used for the task of process model abstraction, i.e., the discovery of reducible process fragments. The implication of the technique is fragmentation of structural as well as behavioral process model analysis tasks. In [18], we proved that for a process model which forbids mixed gateways, all SPQR-tree node fragments are process components. The observation is captured in Theorem 1.

Theorem 1. *Any SPQR-tree node fragment of a structurally sound process model, which forbids mixed gateways, is a process component.*

However, for the generalized definition of a process model (see Definition 1), it does not necessarily hold that all SPQR-tree node fragments are components. In the process model from Figure 2, fragments $S5$, $L1$, $L2$, and $P2$ (the corresponding SPQR-tree nodes are highlighted with dark grey background in Figure 3) are non-components. For each of the fragments, one of the boundary nodes is neither an entry, nor an exit. In Figure 2, non-components are enclosed in the regions with a dashed borderline, whereas components are enclosed in the regions with a dotted borderline.

In the following part of the paper, we will investigate the structural particularities of the parallel case process fragments, which are non-components, and their influence on the behavioral analysis of process models.

4 Correctness of Process Models

Section 3 stated a structural requirement of a process model correctness—structural soundness. In this section, we discuss the correctness property relevant to the dynamics of processes—(behavioral) soundness.

Before one can judge the correctness of process behavior, the execution semantics of process models has to be specified. We specify execution semantics of process models by proposing a mapping to Petri nets [20]. The mapping procedure is adopted from [21] and is visualized in Figure 4. Figure 4(a) shows all possible patterns of a process model edge that connects tasks or gateways of either *xor* type or *and* type. During the mapping procedure, each process model edge is mapped onto the corresponding Petri net pattern proposed in Figure 4(b). In addition to the mapping rules proposed in Figure 4, one has to add a source place i which enables a transition corresponding to the process entry, and a sink place o which is the only output place of the transition corresponding to the process exit. Similar to [21], we name the described mapping—the *petrify* mapping. The result of the *petrify* mapping is a workflow net [6]. Also, similar to [21], one can show that the mapping of a process model onto a Petri net results in a free choice Petri net [22]. Places with multiple outgoing arcs can only result when mapping process model edges that have a common *xor* type gateway as a source node. However, each of these outgoing arcs always has only one transition as a target; this transition models the choice decision.

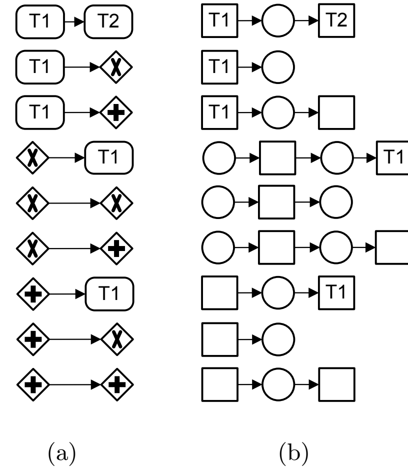


Fig. 4. Mapping process models to Petri nets

The mapping specifies execution semantics of process models as follows: A *xor* split forwards control flow along one of the outgoing edges. A *xor* join merges multiple alternative threads of control flow without synchronization. An *and* split concurrently forwards control flow along all the outgoing edges. An *and* join synchronizes multiple alternative threads of control flow. A mixed gateway first behaves as a join and then as a split of the corresponding gateway type.

At this point, we are ready to define the (behavioral) soundness property of a process model.

Definition 6 (Soundness). A process model is *sound* if the *petrify* mapping of the process model results in a sound workflow net.

Wil van der Aalst showed in [6] how the soundness property of workflow nets relates to the properties of liveness and boundedness, i.e., a workflow net is sound

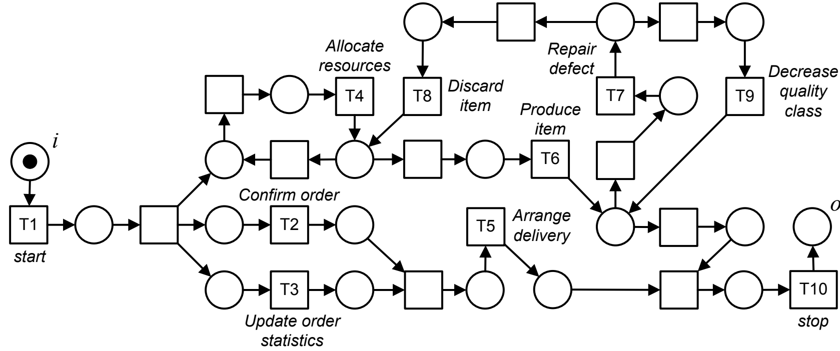


Fig. 5. A workflow net, the *petrify* mapping of the process model from Figure 2

if and only if the extended net is live and bounded, where the extended net is obtained by adding an extra transition $t*$ to the workflow net which connects the sink place o with the source place i. Liveness and boundedness of free choice Petri nets can be checked in polynomial time [22]. Therefore, the soundness of free choice Petri nets as well as the soundness of process models can be validated in polynomial time.

Alternatively, the soundness property of a process model can be deduced from the absence of structural conflicts in the process model.

Theorem 2. *If a workflow net is a result of the petrify mapping of a process model which is free of dead-locks and free of lack of synchronization, it is sound.*

Taking into consideration that any workflow graph obtained as a mapping of a process model is a free choice Petri net, the proof of Theorem 2 is analogous to the proof of Theorem 2 in [21].

Figure 5 shows a workflow net, which is the result of the *petrify* mapping of the process model from Figure 2. The workflow net is sound; the soundness can be validated in polynomial time. Therefore, according to Definition 6, the process model from Figure 2 is sound.

5 Hidden Unstructured Regions in Process Models

In this section, we identify regions in process models that hide unstructured process logic and discuss structural constraints of these regions implied by model correctness properties. We start the discourse by identifying structural classes of process models based on the SPQR-tree decomposition for which soundness can be decided in linear time. One can define structural process model classes based on the presence or absence of certain structural case fragments in process models and the notion of a process component. For instance:

Definition 7 (Block-structured Process Model). A process model is *block-structured* if the SPQR-tree decomposition of the process graph contains no R-type fragments and all SPQR-tree node fragments are process components.

In the general case, SPQR-tree node fragments can also be non-components:

Definition 8 (Quasi Block-structured Process Model). A process model is *quasi block-structured* if the SPQR-tree decomposition of the process graph contains no R-type fragments.

To conclude, a process model is *graph-structured* if the SPQR-tree decomposition of the process graph contains at least one R-type fragment. In the following, we examine in detail the block-structured classes of process models. Before we continue with the discussion, we identify loop case fragments which are the special types of parallel case fragments.

Definition 9 (Directed Fragment). An SPQR-tree node fragment is *directed* if one of its boundary nodes has only outgoing incident edges among fragment edges, and the other has only incoming incident edges among fragment edges.

The directed property assures that once control flow enters the fragment, it does not reach the fragment entry before the fragment exit. Also, if control flow reaches the fragment exit it passes control outside the fragment. Process fragments that are not directed are *non-directed* fragments. Fragment $S3$ from Figure 2 is directed, whereas fragment $S5$ is not. A directed process fragment is clearly a process component.

Process components are useful for behavioral analysis. One can expect that once control flow enters a component through the fragment entry, it also leaves the component through the fragment exit exactly once. This observation was developed in [18] to propose the process model abstraction technique which aggregates process fragments, per process component base, into tasks of a higher abstraction level. Therefore, if one is assured of the correctness of a process component, the component can be seen as a Q-type fragment which passes control flow from its entry to its exit.

Definition 10 (Loop Case Fragment). An *L-type* (or *loop case*) fragment is a parallel case fragment for which: (i) the entry of the fragment is also the exit for at least one of its child fragments, and (ii) each child skeleton specifies a directed SPQR-tree node fragment.

Fragment $L2$ in Figure 2 is an L-type fragment. $L2$ contains two directed fragments $S7$ and $S8$ and the entry of $L2$ is the exit of $S8$; there exists a cyclic path that goes through the boundary nodes of $L2$. Contrary, fragment $P2$ in Figure 2 is a P-type fragment, but not an L-type fragment; $P2$ contains a non-directed fragment $S5$.

It is a straightforward task to check whether a block-structured process model is sound. One must check if the gateway types of the boundary nodes for each P-type fragment which is also a component match, meaning both gateways are either of *xor* type or of *and* type. To ensure the soundness of a block-structured process model, any L-type component of the process model must be structured by boundary gateways of *xor* type. If the fragment entry of an L-type component is an *and* type gateway, there is a dead-lock situation, while if the fragment exit

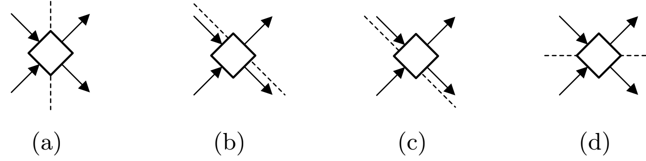


Fig. 6. All unique combinations for edge separation on internal and external fragment edges for a boundary node of a fragment which is a mixed gateway

of an L-type component is an *and* type gateway, there is a live-lock situation. This also implies a lack of synchronization at the fragment exit. We summarize the conditions in the following theorem:

Theorem 3. *A block-structured process model is sound if and only if:*

- *for each P-type component the boundary gateways match in type,*
- *for each L-type component the boundary gateways have xor type.*

In the case of a quasi block-structured process model, it is not obvious what checks should be applied to non-components in order to validate the soundness of the process model. To address the challenge, we investigate structural particularities of non-components. The first characteristic of non-components follows from Theorem 1:

Corollary 1. If an SPQR-tree node fragment of a process model is a non-component, then at least one of its boundary nodes is a mixed gateway.

By examining the structure of a boundary fragment node which is a mixed gateway, one can make a stronger statement, which is captured in Lemma 1.

Lemma 1. *An SPQR-tree node fragment is a non-component if and only if it has a boundary node that is a mixed gateway that has at least one incoming and at least one outgoing edge both among internal and external fragment edges.*

Proof. Figure 6 shows combinations of incident edge settings with a boundary node which is a mixed gateway. The dashed lines divide the edges onto internal and external fragment edges. The combinations are obtained by allowing only incoming, only outgoing, or both incoming and outgoing edges at each side of a dashed line. Each combination represents a collection of edge settings, where every edge stands for an arbitrary number of edges (but at least one) that have the same structural relation (incoming or outgoing, internal or external) with the boundary node. Out of the total of nine possible combinations, two interfere with the requirement of structural soundness: all incident edges are incoming or all incident edges are outgoing. Moreover, such nodes are not even gateways. Three combinations are the mirror copies of patterns shown in Figures 6(a), 6(b),

and 6(c). Hence, the four unique combinations are visualized in Figure 6. Observe that out of all possible edge settings, only the one given in Figure 6(d) allows paths in both directions across the fragment's boundary and, therefore, is neither an entry nor an exit of the fragment (see Definition 4). Finally, any other edge setting allows paths only in one direction and, hence, a node is either an entry or an exit of the fragment (see Definition 4). □

A fragment with at least one boundary node like the node from Figure 6(d) is a non-directed fragment. For quasi block-structured process models there is a strong relation between directed fragments and L-type fragments.

Lemma 2. *An SPQR-tree node fragment of a quasi block-structured process model is a non-directed fragment if and only if it is an L-type fragment or it contains an L-type fragment that shares a boundary node with it.*

Proof. A non-directed fragment contains incoming and outgoing incident edges with one of its boundary nodes. In a quasi block-structured process model this node is the boundary node of some P-type fragment. The P-type fragment must have Q-type and/or S-type fragments as child fragments (see [15,18]). Every Q-type fragment is directed. If all the S-type fragments which are the children of the P-type fragment are directed, then the fragment is an L-type fragment. If there exists an S-type fragment which is a non-directed fragment, then it must share a boundary node with a P-type fragment. Hence, the above described logic can be recursively applied to this P-type fragment. Eventually, we will reach an L-type fragment which also shares a boundary node with the initially investigated non-directed fragment. Therefore, if a fragment is a non-directed fragment, then it is either an L-type fragment or it contains an L-type fragment that shares a boundary node with it.

The reverse direction of the proposition is trivial to show. At least one of the boundary nodes of the SPQR-tree node fragment is incident with incoming and outgoing edges contained in the fragment. This node is also the boundary node of the L-type fragment. Hence, the fragment is a non-directed fragment. □

Finally, we are ready to make the concluding statement which characterizes the nature of non-components in quasi block-structured process models.

Lemma 3. *If an SPQR-tree node fragment of a quasi block-structured process model is a non-component, then either it is an L-type fragment, or it shares a boundary node with an L-type fragment.*

Proof. If an SPQR-tree node fragment of a quasi block-structured process model is a non-component, then it has a boundary node which is a mixed gateway that has incoming and outgoing edges among internal and external fragment edges (Lemma 1). Every fragment with a boundary node as in Figure 6(d) is clearly a non-directed fragment. In a quasi block-structured process model, a non-directed fragment is an L-type fragment or it contains an L-type fragment that shares a boundary node with it (Lemma 2). □

The fact that each non-component of a quasi block-structured process model shares a node with an L-type fragment allows us to define criteria for checking the soundness of a quasi block-structured process model.

Theorem 4. *A quasi block-structured process model is sound if and only if:*

- *for each P-type component the boundary gateways match in type,*
- *for each L-type component the boundary gateways have xor type,*
- *for each non-component the boundary gateways have xor type.*

Proof. In addition to the criteria proposed in Theorem 3, one needs to show that non-components cause structural conflicts, either a dead-lock, or a lack of synchronization, when they have a boundary node of the *and* type. Every non-component is either an L-type fragment, or it shares a boundary node with an L-type fragment (Lemma 3). Moreover, a boundary node of the non-component which is neither its entry nor its exit is a boundary node of an L-type fragment contained in the non-component and has incoming and outgoing incident edges outside the non-component. If this node is an *and* type gateway, it introduces the uncontrolled concurrency (a lack of synchronization) conflict to the process model. If the process model is mapped to the workflow net using the function *petrify*, one can always observe an unbounded place, which is an output place of the transition that corresponds to the *and* type gateway of the L-type fragment and is on a path that leads outside of the fragment. □

Theorems 3 and 4 conclude that the soundness of quasi block-structured process models can be checked in linear time. The SPQR-tree of a process graph can be constructed in linear time [17]. The directed property of a fragment as well as an L-type fragment structure can be checked locally. Finally, model soundness can be checked by performing a post-order traversal to the SPQR-tree of the process graph while verifying if the structural constraints defined in Theorem 4 hold for every fragment.

6 Transformation of Hidden Unstructured Regions

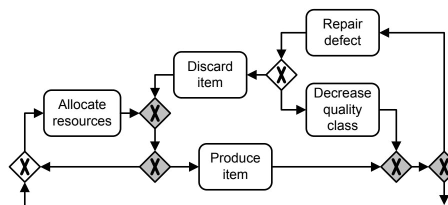


Fig. 7. Splitting conflicting gateways on the running example process

The problem of program control flow graph restructuring has largely attracted the attention of the compiler construction research community. In contrast to flow-graphs, process models can be structured with advanced constructs such as *and* type gateways that may preclude the use of those general approaches. However, our focus is on non-component fragments which, according to Theorem 4, have *xor* type gateways as boundary nodes. This

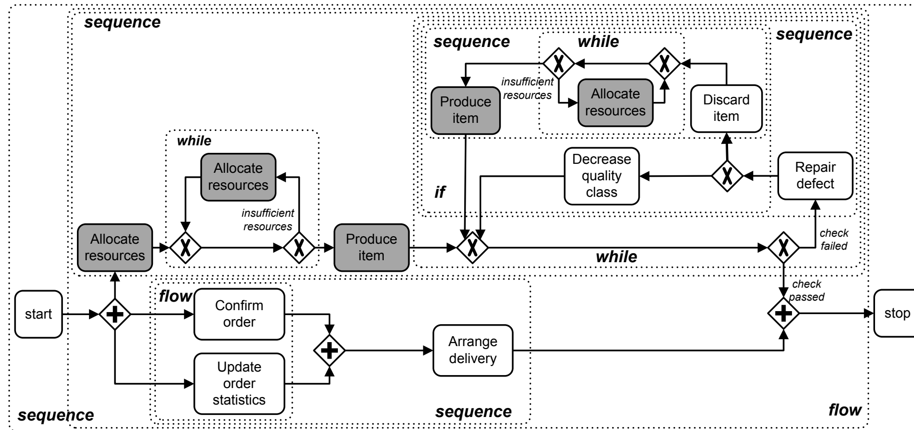


Fig. 8. The restructured process model for the process model from Figure 2 and its mapping to BPEL

fact allows us to conclude that there exist techniques that can be applied for transforming quasi block-structured models into block-structured ones.

Please note that restructuring techniques do not consider mixed gateways. Hence, prior to applying a restructuring, conflicting mixed gateways must be split as follows: A first gateway is used for collecting all incoming edges, whereas a second one is used for collecting outgoing edges. Finally, an edge is added to connect the first gateway with the other one. To illustrate this procedure, consider Figure 7, which presents fragment $S2$ in the running example after splitting the two conflicting gateways (i.e., shaded gateways). $S2$ corresponds to the non-component fragment highlighted in the SPQR-tree in Figure 3.

Splitting conflicting gateways unveils the unstructured logic which is hidden by non-component fragments. For instance, a closer look at Figure 7 allows us to identify two unstructured loops, one with two entry points (left-hand side) and the other with two exit points. It is worth noting that the resulting region would be enclosed in a R-type fragment if the corresponding SPQR-tree was updated.

Figure 8 presents the restructured process model of the running example (the transformation was done using the technique proposed in [11]). Please note that some nodes have been replicated, i.e., tasks with dark grey background. In general, restructuring methods rely on node replication and/or on encoding the control flow via additional variables and gateways. The translation of the resulting process model to a block-structured language is straightforward (see [8]). Although the process model still contains mixed gateways, the SPQR-tree decomposition provides enough structural information to support the translation. To illustrate this, Figure 8 presents the mapping of the control flow of the running example to BPEL constructs.

7 Conclusions

In this paper, we describe a structural classification for process models based on the SPQR-tree decomposition. Three classes are identified: block-structured, quasi block-structured, and graph-structured process models. We focus on quasi block-structured models which contain regions with hidden unstructured behavior. Although this kind of regions was described before [7,8], we take a step forward as we characterize their structural properties and show that their soundness can be verified in linear time. We also show that existing techniques for flow-graph restructuring can be applied to transform hidden unstructured regions into block-structured equivalent process fragments. It is worth noting that in the cases where a quasi block-structured process graph contains *and* type gateways, all of them are part of block-structured fragments. As a consequence, hidden unstructured fragments rely only on *xor* logic, which enables the use of restructuring techniques. The evoked transformation may be required if a quasi block-structured process model must be translated into a block-structured language such as BPEL, e.g., for execution.

In the future work, we want to extend our approach to address the analysis and transformation of unstructured process models, i.e., process models whose SPQR-tree decomposition contains R-type fragments. We believe that the structural constraints imposed by L-type fragments to their enclosing regions can be exploited to speed up the verification of unstructured process models, for instance, if R-type fragments contain internal loops. We are also interested in further investigating the analysis and the transformation of process models which use *or* type gateways.

References

1. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer Verlag (2007)
2. OMG: Business Process Modeling Notation, Version 1.2. (January 2009)
3. Laue, R., Mendling, J.: The Impact of Structuredness on Error Probability of Process Models. In Kaschek, R., Kop, C., Steinberger, C., Fliedl, G., eds.: UNISCON. Volume 5 of Lecture Notes in Business Information Processing., Springer Verlag (2008)
4. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.: On Structured Workflow Modelling. In: Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE), London, UK, Springer Verlag (2000)
5. 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
6. 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
7. 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

8. 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
9. Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, C.K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A.: Web Services Business Process Execution Language Version 2.0. OASIS Standard. (April 2007)
10. Margolis, B.: SOA for the Business Developer: Concepts, BPEL, and SCA (Business Developers series). Mc Press (2007)
11. Oulsnam, G.: Unravelling Unstructured Programs. The Computer Journal **25**(3) (1982) 379–387
12. Lin, H., Zhao, Z., Li, H., Chen, Z.: A Novel Graph Reduction Algorithm to Identify Structural Conflicts. In: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS). Volume 9., Washington, DC, USA, IEEE Computer Society (2002) 289
13. Hopcroft, J.E., Tarjan, R.E.: Dividing a Graph into Triconnected Components. SIAM Journal on Computing **2**(3) (1973) 135–158
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. Fussell, D., Ramachandran, V., Thurimella, R.: Finding Triconnected Components by Local Replacement. SIAM Journal on Computing **22**(3) (1993) 587–616
17. 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
18. Polyvyanyy, A., Smirnov, S., Weske, M.: The Triconnected Abstraction of Process Models. In: Proceedings of the 7th International Conference on Business Process Management (BPM), Ulm, Germany (September 2009)
19. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In: Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC). Volume 4749., Springer Verlag (September 2007) 43–55
20. Petri, C.: Kommunikation mit Automaten. PhD thesis, Institut für instrumentelle Mathematik, Bonn, Germany (1962)
21. Aalst, W., Hirnschall, A., Verbeek, H.: An Alternative Way to Analyze Workflow Graphs. In Pidduck, A.B., Mylopoulos, J., Woo, C.C., Özsu, M.T., eds.: Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE). Volume 2348 of Lecture Notes in Computer Science., Springer Verlag (2002) 535–552
22. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge University Press, New York, NY, USA (1995)