Combining Analysis of Unstructured Workflows with Transformation to Structured Workflows

Rainer Hauser*, Michael Friess†, Jochen M. Küster*, and Jussi Vanhatalo*
*IBM Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland, {rfh,jku,juv}@zurich.ibm.com
http://www.zurich.ibm.com/csc/bit/bpia.html
†IBM Deutschland Entwicklung GmbH, D-71032 Böblingen, Germany, mfriess@de.ibm.com

Abstract— **Analysis of workflows in terms of structural correctness is important for ensuring the quality of workflow models. Typically, this analysis is only one step in a larger development process, followed by further transformation steps that lead from high-level models to more refined models until the workflow can finally be deployed on the underlying workflow engine of the production system. For practical and scalable applications, analysis and transformations of workflows must both be integrated to allow incremental changes of larger workflows.**

In this paper, we introduce the concept of a region tree for workflow models that can be used as the central data structure for both workflow analysis and workflow transformation. A region tree is similar to a program structure tree and imposes a hierarchy of regions onto the workflow model. It allows an incremental approach to analysis and transformation of workflows and thereby significantly reduces the overhead because individual regions can be dealt with separately.

I. Introduction

Graphical notations for workflow models or business process models (in the following called workflows) have been used for a long time to describe behavior in terms of a control-flow between activities and their temporal relations. Workflows are therefore a relatively advanced area, in which model-driven architecture (MDA) [1] or, in a broader sense, model-driven engineering (MDE) [2] concepts and methods have been applied. The development of an application based on graphical models is a complicated process, leading from analysis models via design models to a complete and deployable IT solution through partially manual and partially automated steps [3]. To support this development cycle including the deployment, (1) tools are required for validation, verification, optimization and testing of workflows, and (2) algorithms are needed for transforming a workflow into the elements and structure required by the underlying workflow or execution engine (in the following called runtime platform).

The challenge of validation, verification and testing is to discover errors and unexpected behaviors as early as possible, but also not to restrict the designer by imposing unnecessary overhead. Structural conflicts (most importantly deadlocks) as one source of errors in a workflow can be detected by various methods. Graph reduction rules were introduced in [4] to detect structural conflicts in acyclic workflows. One of the rules in [4], the so-called overlapped reduction rule defined for an infrequently occurring pattern, turned out to be insufficient as demonstrated on a sample workflow and was replaced in [5] by three others, though very complicated rules. In [6], another valid workflow was presented in which the original rules fail, and a case was made for using Petri-net theory and tools to detect structural conflicts because they outperform the reduction algorithms operating on workflow graphs and can also handle cyclic workflows. Despite this, rule-based approaches are not intrinsically limited to acyclic workflows and are advantageous because they can localize errors as well as help understand the structure of general workflows.

In a model-driven approach to workflow modeling, workflow analysis for structural conflicts is not a one-time activity but is performed repeatedly on different (or even the same) parts of a larger workflow model. The main reason for this is that a workflow model is iteratively refined from a more abstract to a more concrete form [3]. One important transformation used in this refinement process is the deployment step, which transforms the workflow into the form required by, and possibly optimized for, the runtime platform. This transformation is not always straightforward, because graphical modeling languages for workflows, such as Unified Modeling Language 2 (UML2) Activity Diagrams [7] and the notation used by the IBM WebSphere Business Modeler (Modeler) [8], allow specification of models that are less structured than allowed by some runtime platforms such as the workflow engines for the Business Process Execution Language for Web Services (BPEL) [9]. In BPEL, unstructured cycles, for example, must be converted to structured do-while loops. Thus, if the target runtime platform is based on BPEL the unstructured parts of the workflow that cannot be represented in BPEL must be resolved into structured constructs [10].

To enable workflow analysis and workflow transformation for an incremental and iterative approach, we introduce the concept of a region tree. This region tree imposes a hierarchical structure on the workflow model similar to the program structure tree [11]. Individual regions can then be analyzed separately, and one region may be transformed and refined without affecting other parts of the workflow model. The region tree can be built by rules that adapt and extend the reductions rules for detecting structural conflicts in [4][5]. These rules, called region-growing rules, are used to construct a hierarchy of regions in which structural conflicts become visible at the interfaces between interacting regions. The resulting tree of regions can be used to optimize workflows and to transform unstructured or partially structured workflows into more structured, equivalent versions of the workflow. Unlike

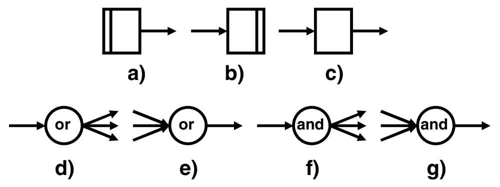

Fig. 1. Workflow graph elements

the program structure tree, the region graph may contain not only the special type of regions called single-entry-single-exit (SESE) regions [11], but also more general regions.

The paper is organized as follows. Section II introduces the basic workflow concepts. In Sections III and IV, the region tree and the region-growing rules for workflows are presented, and in Section V, their application is discussed. The combination of analysis and transformation using the region tree is shown on an example in Section VI. Finally, the paper concludes with a summary in Section VII.

II. Basic Concepts

General workflows and useful subclasses of workflows with their properties are introduced and discussed.

A. Well-formed and Well-behaved Workflows

Various graphical notations for workflows exist, but they are all based on the concept of directed graphs. We use the definition of workflows and the graphical representation for their elements, as shown in Fig. 1, similar to the notation used in [12] (but not limited to only two edges). The start node, the end node, and the activity[1] are shown in 1a, 1b, and 1c, respectively. The sequential control nodes choice and merge in 1d and 1e are also called or-split and or-join[2]. The parallel control nodes fork and join in 1f and 1g are also called and-split and and-join. These nodes can be connected through edges, and a directed graph built with these elements is called a *workflow*. In the following we assume that the workflows are well-formed.

Definition 1: A workflow is *well-formed* if and only if it has the following properties:

1) There is exactly one start node with no incoming edges and one outgoing edge.
2) There is exactly one end node with one incoming edge and no outgoing edges.
3) Every activity has exactly one incoming and one outgoing edge.
4) Every or- and and-split has exactly one incoming edge and at least two outgoing edges.
5) Every or- and and-join has at least two incoming edges and exactly one outgoing edge.
6) There is a path from the start node to every node, and a path from every node to the end node.

[1]Note that the start and end nodes are sometimes considered activities and sometimes no-op elements, although the distinction is not relevant here.

[2]The term "or" is slightly misleading, and the term "xor" is sometimes used instead, because one and only one edge is assumed to be enabled.

We introduce the semantics (i.e., the behavior) of a well-formed workflow rather informally in terms of Petri-net-like tokens. The start node emits a token on its outgoing edge when the workflow is started. An activity starts when a token arrives on its incoming edge (i.e., when the incoming edge is enabled), and eventually ends by sending a token to its outgoing edge. The end node consumes a token on its incoming edge and terminates the workflow. The or-split emits a token on one of its outgoing edges after consuming a token on its incoming edge. The or-join emits a token on its outgoing edge after consuming a token on one of its incoming edges (and thus behaves according to the "multiple executions" semantics defined in [12]). The and-split consumes a token on its incoming edge and emits a token on all outgoing edges. The and-join emits a token on its outgoing edge after consuming a token on all incoming edges. As we allow cyclic workflows, we assume that every or-split will enable each of its outgoing edges eventually if reached often enough.

Executions of a workflow are defined through the flow of these tokens. An execution *terminates* as soon as the end node consumes a token. It terminates *successfully* if at this point no other tokens are present in the workflow.

Definition 2: A well-formed workflow is *well-behaved* if and only if all possible executions terminate successfully.

B. Structured and Separable Workflows

There are several levels of how structured a workflow is. The simplest level, apart from linear workflows (i.e., workflows with a single path from start to end node), are called *structured* in [13] and consist of those workflows in which each or- or and-split has one corresponding or- or and-join, respectively. If the sequential parts of a workflow (i.e., those parts only using or-splits and -joins) can be separated from the parallel parts (i.e., those parts only using and-splits and -joins), we call the workflow *separable*.

To define these levels, we introduce the concept of a *single-entry-single-exit* (SESE) region from compiler theory [11]. Informally speaking, a SESE region is a set of nodes such that there is exactly one edge (called the incoming edge) leading from nodes not in the region to nodes in the region and exactly one edge (called the outgoing edge) leading from nodes in the region to nodes not in the region. The set of all nodes of a well-formed workflow other than the start and end node build a SESE region[3], with the edge leaving the start node as the incoming edge and the edge going to the end node as the outgoing edge.

We further introduce the concept of *substitution*, i.e., of replacing an edge e in a workflow w_1 with a well-formed workflow w_2. Edge e is removed from w_1, the start node and the end node of w_2 are removed from w_2, and the remaining parts of w_2 are plugged into w_1 such that the original source of e becomes the new source of the edge leaving the original start node of w_2 and the original target of e becomes the new

[3]In the following figures, we will often only show the SESE regions instead of the workflows with start and end nodes, because many properties of workflows can be generalized as properties of SESE regions.

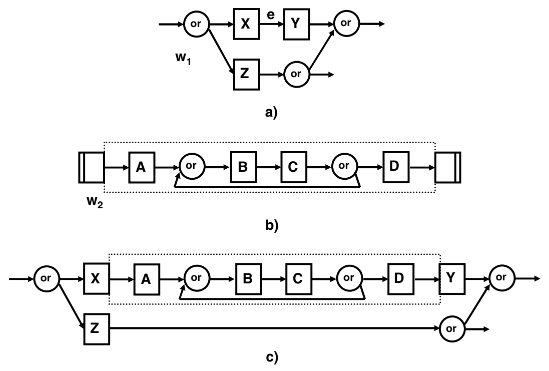

Fig. 2. Substitution of workflow w_2 for edge e in workflow w_1

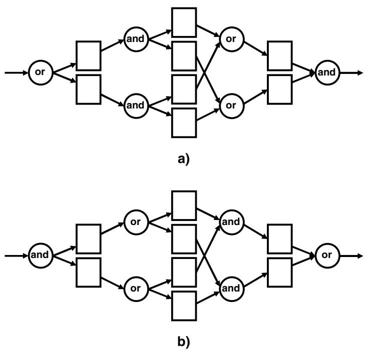

Fig. 3. Overlapped patterns

target of the edge going to the original end node of w_2. This substitution is depicted in Fig. 2. A part of workflow w_1 with edge e is shown in 2a, workflow w_2 to be plugged into w_1 in 2b, and the result of the substitution in 2c.

Definition 3: A workflow is *structured* if and only if it can be constructed using the following inductive rules:

1) All well-formed linear workflows (i.e., workflows without control nodes) are structured.
2) All well-formed workflows with one or-split and one or-join as the only control nodes are structured.
3) All well-formed acyclic workflows with one and-split and one and-join as the only control nodes are structured.
4) If w_1 and w_2 are structured workflows and w_1 contains an edge e, the result of replacing e with w_2 in w_1 is structured.

In structured workflows, there is a corresponding join control node for each split control node. Thus, structured workflows can be represented in XML such that split and join control nodes correspond to start and end tags as is done in BPEL. All well-formed sequential workflows (i.e., workflows with only sequential control nodes) have an equivalent structured form as shown in [13]. For workflows with parallelism this is, however, not true, and BPEL needs, as a consequence, the link element in addition to the flow element.

Definition 4: A workflow is *separable* if and only if it can be constructed using the following inductive rules:

1) All well-formed workflows with only or-splits and or-joins as control nodes are separable.
2) All well-formed acyclic workflows with only and-splits and and-joins as control nodes are separable.
3) If w_1 and w_2 are separable workflows and w_1 contains an edge e, the result of replacing e with w_2 in w_1 is separable.

The inductive construction rules guarantee that separable workflows can be partitioned into SESE regions such that each region contains either only sequential or only parallel control nodes. Obviously, all structured workflows are separable, but there are separable workflows that are not structured.

Lemma 1: All separable workflows are well-behaved.

Proof: In a SESE region with only sequential control nodes, all nodes (or other SESE regions) consume one token and eventually emit one token. Thus, there is exactly one token in the region between the time the token enters the region and the time it leaves it[4]. In an acyclic SESE region with only parallel control nodes, one token passes through every edge exactly once. ■

As just shown, all structured and separable workflows are well-behaved, but there are well-behaved workflows that are not separable. The so-called overlapped patterns shown in Fig. 3 mix and-splits with or-joins or or-splits with and-joins in such a way that executions of the workflow can terminate successfully. The pattern in 3a made it necessary to define a special rule in [4], and all its executions terminate successfully. For the dual pattern (i.e., the pattern where or-splits and -joins are replaced with and-splits and -joins and vice versa) in 3b, executions can only terminate successfully if the two or-splits both enable either the upper edge or the lower edge, and the pattern is therefore not considered well-behaved. If two or more or-splits in a workflow need additional information to make the workflow execute, their conditions are called *synchronized*. Other examples in which two or-splits need synchronization of their conditions (e.g., to exit two parallel cycles at the same time) are discussed in [13].

C. Structural Conflicts

Not all well-formed workflows are well-behaved. Fig. 4 shows simple cases of structural conflicts. The first two were identified in [4][12][13], where 4a is called *deadlock* and 4b is either called *lack of synchronization* or *multiple active instances of the same activity*. The third structural conflict shown in 4c can only occur in cyclic workflows. We call it *parallel cycle*.

The and-join of the *deadlock* never emits a token if the or-split only gets a single token. The or-join of the *lack of synchronization* always receives at least two tokens and, depending on the semantics of the or-join [12], either discards all but one token or emits one token for each of them[5]. Both structural conflicts may not always be seen as an error, but are certainly dangerous and need careful inspection. In general,

[4]As mentioned above, we assume that all outgoing edges of an or-split are eventually taken to avoid infinite loops in cyclic workflows.

[5]As the or-join allows different behaviors, the name *lack of synchronization* is more appropriate than the name *multiple instances of the same activity*.

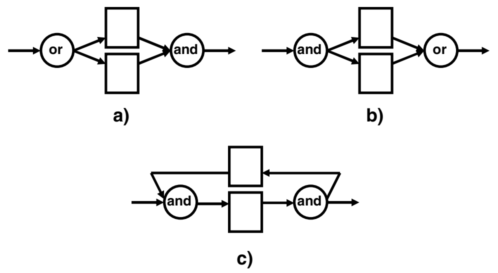

Fig. 4. Structural conflicts

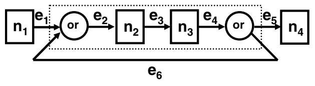

Fig. 5. Nodes and edges inside and outside of a region

deadlock is a situation in which an and-join gets some but not all tokens on its incoming edges, and *lack of synchronization* is a situation in which an or-join gets more than one token on its incoming edges. The *parallel cycle* is also a kind of deadlock: if the nodes on the path from the and-split back to the and-join can only get a token through the path from the and-join to the and-split, the and-join never emits a token.

III. Region Analysis

Regions with their input and output logic are introduced, and region trees for workflows are defined.

A. Regions

A SESE region is a part of a workflow that can be hidden behind an interface to the remaining parts of workflow. The interface in this case is very simple as it consists of an input and an output edge. This concept can be generalized to more complex interfaces defined through multiple incoming edges with an input logic and multiple outgoing edges with an output logic. These concepts are defined in the following.

A region is a set of nodes and a set of edges as depicted in Fig. 5. Nodes n_2 and n_3 belong to the region, n_1 and n_4 do not. Edges e_2, e_3 and e_4 are inside the region, e_1, e_5 and e_6 are outside. If an edge is inside the region, also its source and target node must be inside the region (e.g., edge e_2), but an edge may be outside the region even if its source and target are inside (e.g., edge e_6).

Further, as regions can be nested, there are basic regions that link a region to the elements of the workflow, and composite regions that contain only other regions. We illustrate the nesting of regions with the sample workflow shown in Fig. 6, where some regions are depicted as rectangles. The innermost rectangles are basic regions, all other rectangles are composite regions.

Definition 5: A *basic region* is a tuple $R = (N_R, E_R)$, where N_R is a subset of N (the set of nodes of the workflow) and E_R is a subset of E (the set of edges of the workflow), with the restriction that the source and the target node of an edge in E_R must be in N_R.

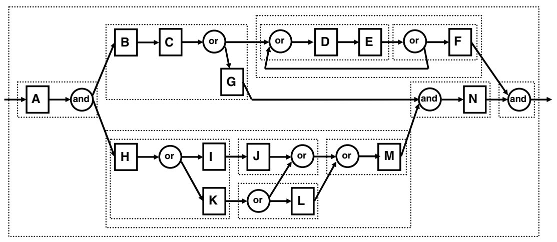

Fig. 6. A tree of well-defined regions

Definition 6: A *composite region* is a tuple $R = (N_R, E_R)$, where N_R is a set of regions and E_R is a set of edges between regions, with the restrictions that (1) either R_1 is contained in R_2 or R_2 is contained in R_1 or $R_1 = R_2$ if R_1 and R_2 both contain the same workflow node n, and (2) the source and the target region of an edge in E_R must be in N_R.

In the following we assume that the basic regions form a partition of the workflow nodes, i.e., every workflow node is contained in exactly one basic region. One possible partition packs every workflow node n into a region $R_n = (N_{R_n}, E_{R_n})$ with $N_{R_n} = \{n\}$ and $E_{R_n} = \emptyset$. This special partition is called the *single-node basic partition*. Another partition is shown in Fig. 6, where each basic region contains exactly one control node and each activity is either put into the region to its left or to its right.

When building new composite regions first from basic regions and later from other composite regions, there is always a set of top-level regions not contained in another region. These top-level regions also form a partition of the workflow nodes because of restriction (1) in Definition 6.

Any partition into regions is a directed graph in which the nodes are the regions and the edges are the edges between these regions. Note, however, that this graph may not be an ordinary directed graph because multiple edges may lead from a region R_1 to a region R_2. As part of a directed graph, regions have successors and predecessors, and we use the predicate $succ(R)$ for the set of all successor regions of region R and the predicate $pred(R)$ for the set of all predecessor regions of region R.

The basic regions of the single-node basic partition inherit their behavior from the node they contain. The region containing the start or end node have no incoming or no outgoing edge, respectively. The regions containing activities are SESE regions. The nodes containing control nodes have either multiple outgoing or multiple incoming edges, and one can assign an input and output logic to them. The output logic of a single-node basic region containing an or- or and-split is “or” or “and”, respectively. Similarly, the input logic of a single-node basic region containing an or- or and-join is “or” or “and”, respectively.

The concept of input and output logic can be extended to other regions. If the workflow is not partitioned using the single-node basic partition, the basic regions may not have a well-defined input and/or output logic. A region, for example, containing an or-split with one outgoing edge leading to an

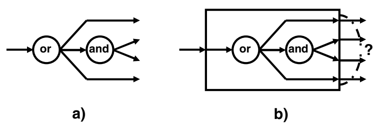

Fig. 7. Region with no well-defined output logic

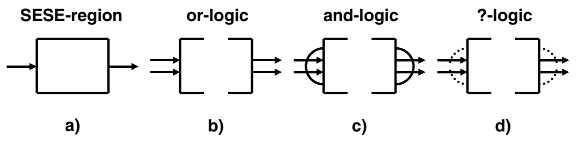

Fig. 8. Region graph elements with input/output logic

and-split as depicted in Fig. 7 has no unique output logic. The two control nodes in 7a can be put into a region as in 7b, but the four outgoing edges of this region do not have a behavior that can be described with "or" or "and".

Definition 7: The *input logic* and *output logic* of a region is called well-defined in the following cases:

1) The input logic of a region is "or" if and only if the region eventually enables one or all of its outgoing edges (depending on the output logic) when any one of its incoming edges is enabled.
2) The input logic of a region is "and" if and only if the region eventually enables one or all of its outgoing edges (depending on the output logic) when all of its incoming edges are enabled.
3) The output logic of a region is "or" if and only if the region eventually enables one and only one of its outgoing edges when one or all of its incoming edges (as required by the input logic) are enabled.
4) The output logic of a region is "and" if and only if the region eventually enables all of its outgoing edges when one or all of its incoming edges (as required by the input logic) are enabled.

The graphical notations for the input/output logic of a region are shown in Fig. 8. In 8a, a SESE region is shown. The input and output logic is "or" in 8b, "and" in 8c, and either unknown or irrelevant but still well-defined in 8d. If a region has zero or one incoming edge, the input logic can be interpreted as "or" or "and". Similarly, if a region has zero or one outgoing edge, the output logic can be interpreted as "or" or "and".

Definition 8: A region is called *well-defined* if it has a well-defined input logic and a well-defined output logic.

The structural conflicts defined for acyclic workflows become incompatible input/output logic for regions as shown in Fig. 9. A region S with output logic "or" connected to a region T with input logic "and" through two or more edges as in 9a results in a *deadlock*, and a region S with output logic "and" connected to a region T with input logic "or" through two or more edges as in 9b corresponds to a *lack of synchronization*.

B. Region Tree

Starting with a partition into well-defined basic regions, composite regions can be built by combining one or more

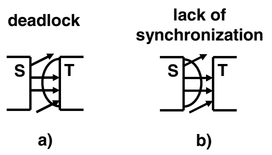

Fig. 9. Structural conflicts between regions

regions into a new region that is also well-defined. If we continue in this way, we may finally reach the point where only a single region is left. The resulting structure is called a *region tree* (RT), similar to the program structure tree (PST) in [11], with the remaining single region as its root.

Definition 9: A *region tree* (RT) of a workflow is a tree of regions where (1) all nodes are well-defined regions, (2) the leaf nodes are basic regions forming a partition of the workflow nodes, (3) all other nodes are composite regions, and (4) the root is a region.

For a single workflow, many different RTs can be constructed. Depending on how the RT is created, it reveals more or less of the structure of the workflow. Fig. 6 shows one possible RT for a sample workflow. The basic regions are well-defined because they contain activities and only a single control node. The composite regions are also well-defined because they are SESE regions, although composite regions may have a more complex input and/or output logic. In Section IV, we define rules that create well-defined composite regions from a set of well-defined input regions, and in Section V, we discuss different strategies for applying these rules.

IV. Region-growing Rules

Three generic region-growing rules are introduced that allow new well-defined regions to be built from existing well-defined regions.

A. Well-defined Rules

Transformation rules have a left-hand side specifying the pattern expected by the rule and a right-hand side that shows the result of the application of the rule if the pattern matches. The left-hand side of such a region-growing rule is a set of regions assumed to be well-defined, and the right-hand side is a single new region. We call a region-growing rule *well-defined* if the resulting new region is well-defined whenever the input regions on the left-hand side are well-defined. Fig. 10 shows an example of a rule that is not well-defined. The new region cannot have "and" input logic because it would emit a token on its upper output edge after consuming two tokens on its upper two input edges instead of waiting for tokens on all four input edges. It also cannot have "or" input logic because it requires at least two tokens in order to emit a token instead of one.

Any well-defined region-growing rule can be applied in different ways, depending on the purpose intended. The three applications in Fig. 11 will be used in the following as appropriate. A rule can ignore the content of the regions as

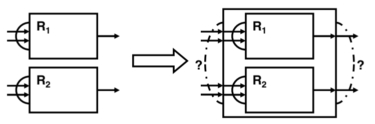

Fig. 10. Not well-defined region-growing rule

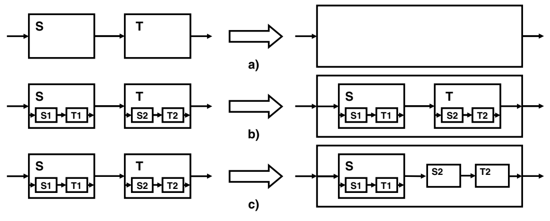

Fig. 11. Possible applications of well-defined region-growing rules

in 11a because all information required for the application of further rules is stored in the interfaces (i.e., the input/output logic). Used this way, a rule becomes a reduction rule. If the content of the regions is needed and the rules acts as a transformation rule, all intermediate regions created by previous applications of rules may be kept, as in 11b, or some of the regions introduced by other rules may be dissolved, as region T is in 11c.

In the following, we present the three generic rules shown in Fig. 12 and their well-defined subrules. The simplest generic rule is depicted in 12a and covers the cycles in a workflow. The generic rule with the most subrules is the one for two neighbors, shown in 12b, because different subrules are needed for the possible input and output logics of regions S and T and depending on the successors of S or predecessors of T, respectively. The overlapped patterns are handled by the generic rule shown in 12c.

These generic region-growing rules are only based on the reduction rules in [4][5] to a limited extent. Their more important root is compiler theory [14], with the T1-T2 analysis [15] to be mentioned explicitly, the area of goto-elimination [16] and subsequent work on cycle-removal transformations for sequential workflows [17]. Although the T1-T2 analysis was invented as a method to determine irreducibility [14], it turned out that reducibility for cycle-removal is far less important than it seemed [18]. The generic rules for self-cycles and

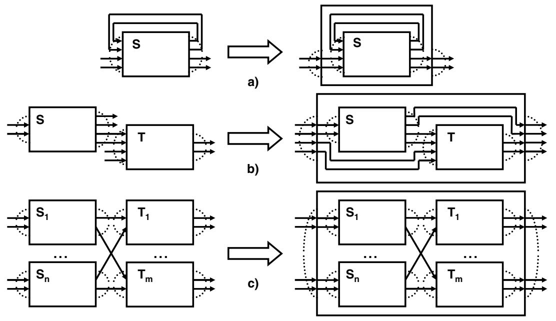

Fig. 12. The three generic region-growing rules

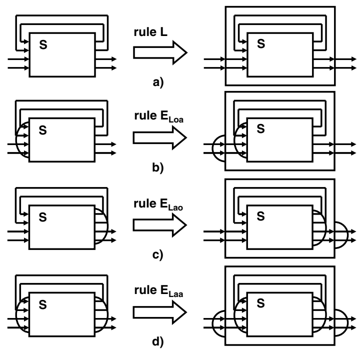

Fig. 13. Rules for self-loops

two neighbors correspond to the T1 and T2 rule from T1-T2 analysis, respectively, extended to handle irreducibility and parallelism [18].

For the graphically represented rules (such as the one shown in Fig. 12), we use the following conventions: a single edge represents exactly one edge, whereas two edges mean one or more edges. In order to handle the region containing the start or the end node, a variant of one rule in which regions are allowed to have zero incoming or outgoing edges is needed.

B. Rules for self-loops

The set of rules for handling self-loops (i.e., edges where the source and the target are the same region) is shown in Fig. 13. *Rule* L in 13a is the only rule in this set that is not an error. *Rule* E_{Loa} in 13b, E_{Lao} in 13c and E_{Laa} in 13d correspond to the three structural conflicts in Fig. 4. In the case of a self-loop, compatibility means that region S has an input logic that is the same as its output logic. Therefore, the situation 13b corresponds to deadlock as in Fig. 4a and the situation 13c to lack of synchronization as in Fig. 4b. Although the input/output logic of region S is compatible, the situation in 13d is also a deadlock, however it is only possible in cyclic workflows. It is called parallel cycle and is shown in Fig. 4c.

C. Rules for two neighbors

Two regions S and T with one or more edges leading from S to T build the pattern for the rules for two neighbors. Depending on whether region S has successors other than T and region T has predecessors other than S, this group is split into four sets of subrules.

The first set of subrules in this group is shown in Fig. 14. It covers the cases where region S is the only predecessor of region T and region T is the only successor of region S: $\{S\} = pred(T)$ and $\{T\} = succ(S)$. *Rule* S in 14a takes two SESE regions and creates one SESE region containing them. Although it is actually a special case of *rule* C_{st} in 14b and *rule* P_{st} in 14c, we consider it important enough to get its own rule-name such that it can be assigned an independent priority in the transformation algorithm. To resolve the regions containing the start or the end node, region S may have no incoming and/or region T no outgoing edge in rule S, although we assume that this special rule for resolving start

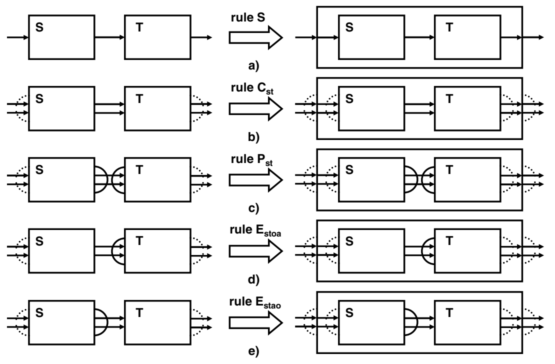

Fig. 14. Rules for two neighbors with $\{S\} = pred(T)$, $\{T\} = succ(S)$

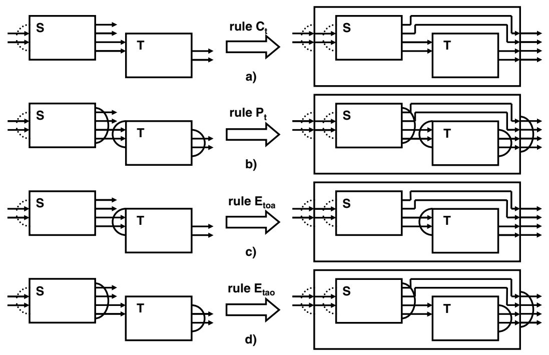

Fig. 16. Rules for two neighbors with $\{S\} = pred(T)$, $\{T\} \subset succ(S)$

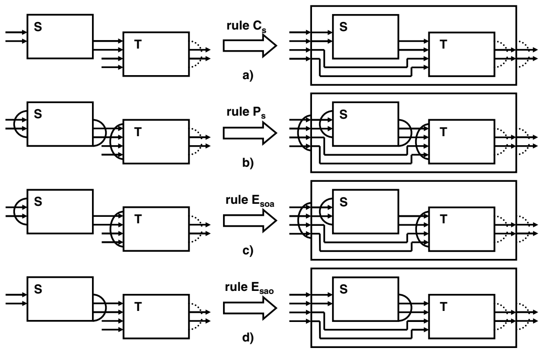

Fig. 15. Rules for two neighbors with $\{S\} \subset pred(T)$, $\{T\} = succ(S)$

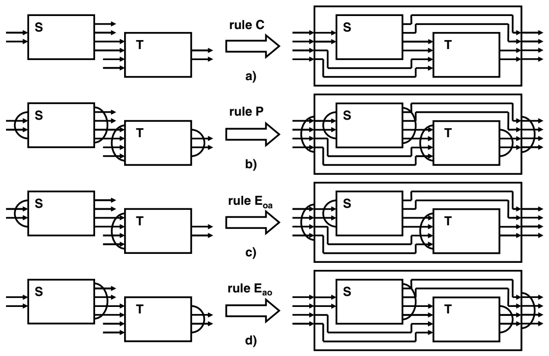

Fig. 17. Rules for two neighbors with $\{S\} \subset pred(T)$, $\{T\} \subset succ(S)$

and end nodes is only applied as one of the very last steps. To enumerate the possible structural conflicts, we created special error rules as we did for the rules for self-cycles. *Rule* E_{stoa} in 14d corresponds to a deadlock and *rule* E_{stao} in 14e to a lack of synchronization.

The second set of subrules in this group is shown in Fig. 15. It covers the cases where region T is the only successor of region S but has predecessors other than S: $\{S\} \subset pred(T)$ and $\{T\} = succ(S)$. *Rule* C_s in 15a and *rule* P_s in 15b correspond to the two possible cases where the output logic of S and the input logic of T are consistent with each other. Also for this situation, we defined special error rules. *Rule* E_{soa}, shown in 15c, corresponds to a deadlock and *rule* E_{sao}, shown in 15d, to a lack of synchronization.

The third set of subrules in this group is shown in Fig. 16. It covers the cases where region S is the only predecessor of region T but has successors other than T: $\{S\} = pred(T)$ and $\{T\} \subset succ(S)$. *Rule* C_t in 16a and *rule* P_t in 16b correspond to the two possible cases where the output logic of S and the input logic of T are consistent with each other. Again, we defined special error rules. *Rule* E_{toa}, shown in 16c, corresponds to a deadlock and *rule* E_{tao}, shown in 16d, to a lack of synchronization.

The fourth and final set of subrules in this group is shown in Fig. 17. It covers the cases where region S has successors other than T and region T has predecessors other than S: $\{S\} \subset pred(T)$ and $\{T\} \subset succ(S)$. *Rule* C in 17a and *rule* P in 17b correspond to the two possible cases where the output logic of S and the input logic of T are consistent with each other. The error rules for this situation are defined as well. *Rule* E_{oa} in 17c corresponds to a deadlock and *rule* E_{ao} in 17d to a lack of synchronization. Note that these two patterns are only errors if there are at least two edges leading from S to T, because otherwise the output logic of S and the input logic of T are compatible.

D. Rules for the overlapped patterns

The overlapped pattern is a situation where a group of n regions S_i is connected to a group of m regions T_j in such a way that from each S_i exactly one edge leads to each T_j. The two corresponding region-growing rules are shown in Fig. 18. *Rule O* in 18a is well-defined, and *rule* E_O in 18b is an error rule because it would require synchronized decision

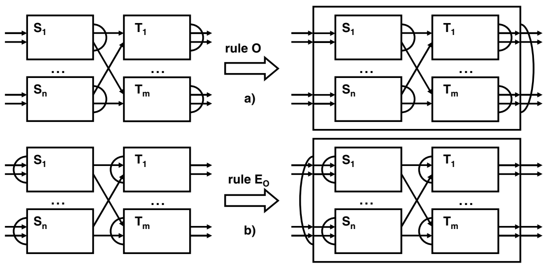

Fig. 18. Rules for overlapped patterns

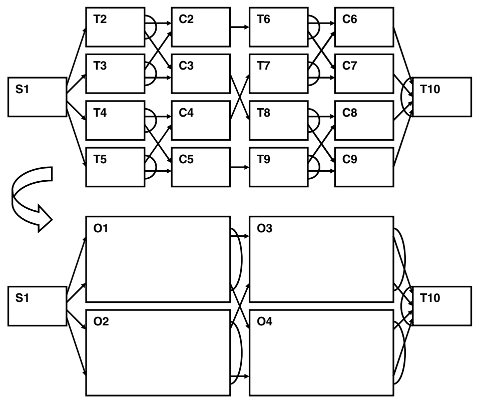

Fig. 19. Overlapped rule applied to counterexample in [6]

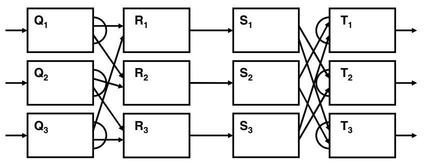

Fig. 20. Overlapped patterns with synchronized conditions

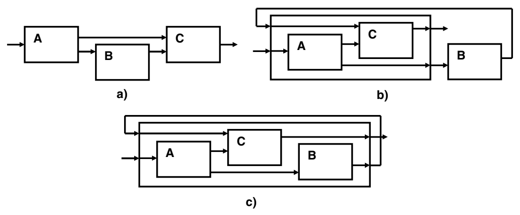

Fig. 21. Pseudo-cycle introduced by rule C

conditions in the regions S_i to ensure that all executions of a workflow containing this pattern (that is otherwise well-defined) will terminate successfully. Overlapped patterns with other input/output logic combinations for regions S_i and T_j are not possible or can be resolved by the rules for two neighbors. (Note also that the two regions S_1 and T_1 in Fig. 18a match the pattern for the rule E_{ao} in Fig. 17d without the restriction that there must be at least two edges leading from region S_1 to T_1.)

Fig. 19 demonstrates that the counterexample presented in [6] can be resolved using rule O. This workflow is basically a pattern of four overlapped patterns arranged in a square in such a way that they in turn also form an overlapped pattern when resolved with rule O.

The problem of the original reduction rule for the overlapped pattern in [4] is (1) that the pattern to the left side of the rule to be matched also includes the sources of all the edges leading to the regions S_i as well as the targets of all the edges leaving the regions T_j, and (2) that all these sources and targets were assumed to be a single region. In the counterexample, regions $T6$ and $T7$, for example, do not have the same predecessor, and regions $C2$ and $C3$, for example, do not have the same successors. This assumption is not necessary as the counterexample shows.

Because synchronized conditions are not allowed, only the overlapped pattern where all regions have "and" output logic is possible. If we allowed synchronized conditions (by redefining what "well-behaved" means) and thus turned rule E_O from an error into an ordinary rule, this additional rule would not be sufficient to resolve all possible workflows that would become well-behaved under the new definition, as can be demonstrated with the example pattern in Fig. 20.

A well-defined region as defined in this paper can only have m-out-of-n logic with either $m = 1$ ("or" logic) or $m = n$ ("and" logic), but the example in Fig. 20 would require a 2-out-of-3 logic, because whenever one of the regions Q_1, Q_2, Q_3 receives a token, two of the three regions S_1, S_2, S_3 eventually get a token. The selection of the outgoing edges of these regions must be synchronized to avoid a deadlock. If, for example, Q_1 gets the initial token, it sends a token to R_1 and R_2, and these two regions pass the token on to regions S_1 and S_2. In order to enable one of the three regions T_1, T_2 and T_3 and to make sure that the workflow does not get stuck in a deadlock at this point, the two regions S_1 and S_2 must "know" which other region got the second token so that both can decide to send their token to T_3, in this case.

V. Application of the Rules

Some issues related to the application of the rules are discussed, and the rules are applied to find region trees for a workflow.

A. Pseudo-cycles

As none of the rules is supposed to introduce a cycle, any acyclic workflow should remain acyclic. As demonstrated in Fig. 21, it is, however, possible to turn an acyclic workflow into one that seems to be cyclic if the rules for two neighbors are applied without care. Rule C with region A as source and region C as target in 21a leads to the situation in 21b, and rule C_t applied to the resulting new region as source and region B as target creates a new region with a self-cycle. We call such artificially introduced cycles *pseudo-cycles*. They are not as dramatic as they may seem to be because (1) the problematic rules are not needed to resolve acyclic or reducible workflows, and (2) cycles and pseudo-cycles can no longer be distinguished in highly cyclic (i.e., irreducible) workflows [18].

Lemma 2: Well-formed acyclic sequential workflows can be completely reduced using rules C_{st} and C_t, and well-formed acyclic parallel workflows can be completely reduced using rules P_{st} and P_t.

Proof: The leftmost split-node on a path from the start to the end node with maximal length can always be resolved together with a neighbor to the right. ■

Using a similar argument, it can be shown that every well-formed acyclic sequential or parallel workflow can be reduced with rules C_{st} and C_s or P_{st} and P_s, respectively, without the need for further rules.

Lemma 3: Rules C_s, P_s, C_t, P_t and the corresponding error rules cannot introduce pseudo-cycles into an acyclic workflow.

Proof: The edges of a directed acyclic graph form a partial order relation on the nodes $m \prec n$ (there exists a path from m to n). These rules handle regions S and T with $S \prec T$, but only if no region R exists with $S \prec R \prec T$. ■

B. Application Strategy

So far, we have only presented the region-growing rules but not specified how they are supposed to play together to build an RT for a workflow. In the following, we assume that the workflow is partitioned through an initialization step into basic regions. This partition can be the single-node basic partition, a partition into basic regions containing exactly one control node as shown in Fig. 6, or any other partition into well-defined regions.

Starting from an initial partition any algorithm based on these rules will eventually terminate independently of the application order of the rules, because the number of regions and/or edges seen at the top-level is reduced by every application of a rule. When such an algorithm terminates, more than one single region may be left. A workflow containing the pattern in Fig. 20, for example, will not lead to a single region. If it is crucial that the algorithm always returns an RT, i.e., one single region at the top-level, an artificial region (marked erroneous) can be introduced that contains all remaining regions as children.

The rules are not confluent because already the rules for sequential workflows are not confluent, and different priorities for the rules have different side-effects, but lead all to correct and "equivalent" results [18]. As a reasonable strategy, we will (1) give rules S, L and O higher priority than the rules for two neighbors, and (2) always make sure that the simpler rules for two neighbors have higher priority than the more complex ones[6]. Because of pseudo-cycles, rule P will not be used.

Structural conflicts can be detected in two ways. Either the explicit error rules (such as E_{Loa}) are used and the resulting new regions are marked erroneous, or these rules are not used and the algorithm terminates whenever it cannot find a further rule that is applicable. In the second case, only a single structural error is detected.

The algorithm outlined as pseudo-code in Fig. 22 shows an implementation that gives rules C_s and P_s lower priority than rules C_t and P_t and uses explicit error rules. The *input* is assumed to contain the set of basic regions created by the initialization step, and the *output* of the algorithm is the set of remaining regions. Inside the do-until loop (i.e., repeat-loop), the variable *regions* contains the top-level regions at the current point of the transformation. The algorithm tries to apply one of the rules in *ruleSet[1]*. If no matching set of regions is found (indicated by an empty *match*), the algorithm tries to apply one of the rules in *ruleSet[2]* and so on. If a matching set of regions is found (indicated by a non-empty *match*), the respective rule

[6]Rules C_s, C_t, P_s and P_t are of equal complexity and can be given the same priority. The only reason for assigning a lower priority to C_s than to C_t is the setting of conditions in or-splits [18].

```
$regions \leftarrow input$
$ruleSet[1] \leftarrow \{S, L, O\}$
$ruleSet[2] \leftarrow \{C_{st}, P_{st}\}$
$ruleSet[3] \leftarrow \{E_{Loa}, E_{Lao}, E_{Laa}, E_{stoa}, E_{stao}, E_O\}$
$ruleSet[4] \leftarrow \{C_t, P_t\}$
$ruleSet[5] \leftarrow \{E_{toa}, E_{tao}\}$
$ruleSet[6] \leftarrow \{C_s, P_s\}$
$ruleSet[7] \leftarrow \{E_{soa}, E_{sao}\}$
$ruleSet[8] \leftarrow \{C\}$
$ruleSet[9] \leftarrow \{E_{oa}, E_{ao}\}$
repeat
  $applied \leftarrow false$
  for $level \leftarrow 1$ to 9 while $\neg applied$ do
    for all $rule \in ruleSet[level]$ while $\neg applied$ do
      $match \leftarrow findMatch(rule, regions)$
      if $| match | \geq 1$ then
        $newRegion \leftarrow rule.apply(match)$
        $regions \leftarrow regions \setminus match$
        $regions \leftarrow regions \cup \{newRegion\}$
        $applied \leftarrow true$
      end if
    end for
  end for
until $\neg applied$
$output \leftarrow regions$
```

Fig. 22. Algorithm in pseudo-code

is applied, creates and returns a new region *newRegion*, the matching regions are removed from *regions*, the new region is added to *regions*, and the boolean variable *applied* is set to true to ensure that the next round starts again with *ruleSet[1]*. If the new region has been created by one of the error rules, the region is marked as erroneous. In this algorithm, the handling of the regions containing start and end nodes has been left out, and it can therefore be used to transform not only complete workflows but also SESE regions[7].

VI. Combined Analysis and Transformation

The combination of the analysis of workflows for structural conflicts and their transformation into a more structured form is demonstrated on the example workflow from Fig. 6.

A. Analysis for Structural Conflicts

The example workflow and its initial regions are shown in Fig. 23. Even with the non-trivial SESE regions highlighted in 23a as dotted rectangles, it is not obvious that it contains a deadlock. The basic regions from the initialization in 23b have been given names, and the regions are annotated with the names of the activities they contain.

Fig. 24 shows the transformation steps until the deadlock becomes visible. Rule C_{st} is applied to regions R_3 and R_4 to get to the state in 24a with region R_{3+4}, to which rule L can be applied as shown in 24b. The situation in 24c results

[7]Note that all SESE regions can be determined (e.g., using the algorithm in [11]), and the region-growing rules can be applied to these regions.

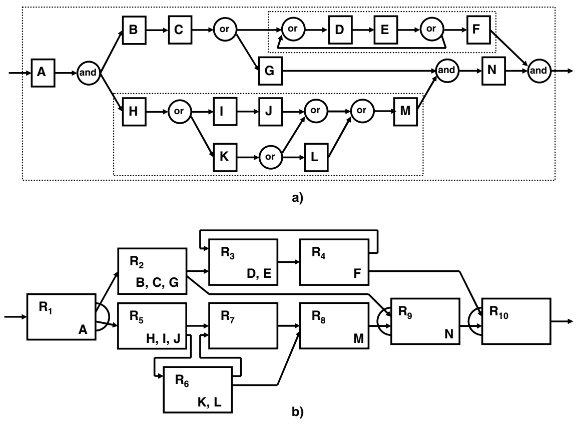
Fig. 23. Sample workflow with deadlock

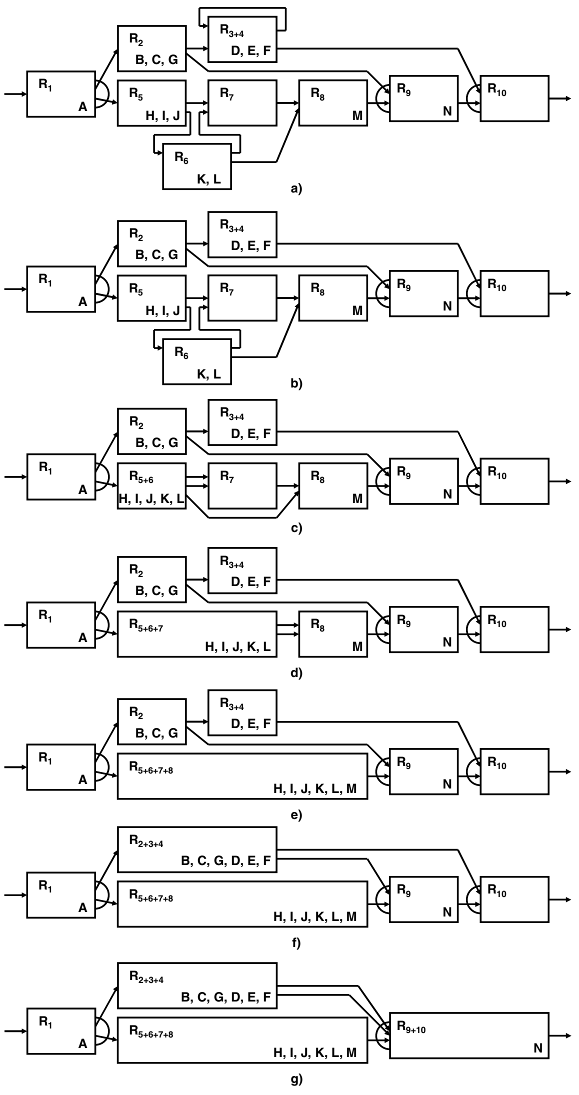
Fig. 24. Rules applied to the sample workflow with deadlock

from the application of rule C_t to regions R_5 and R_6. The new region can be combined with region R_7 using rule C_t again as shown in 24d and with region R_8 using rule C_{st} as shown in 24e. Next, rule C_t combines regions R_2 and R_{3+4}, leading to the state in 24f.

At this point, either rule P_t can be applied to region R_1 and the composite region $R_{5+6+7+8}$, or rule P_s can be applied to regions R_9 and R_{10}. (Note, however, that the two regions R_1 and R_{2+3+4} have incompatible output logic such that rule P_t cannot be applied to these two regions.) As the sequence in which the rules are applied in this situation has no significant influence on the result, we apply rule P_s first and get the state shown in 24g, in which the deadlock between regions R_{2+3+4} and R_{9+10} becomes visible.

We consider the correction of such problems a step that has to be performed manually by the designer, although the algorithm that detected the conflict may come up with suggestions[8]. These hints can indicate which steps could lead to a structurally correct workflow, but only the designer can determine which solution is the right one given what the workflow is supposed to do. In this example, the problem can be resolved by changing either the or-split after activity C into an and-split or all three and-splits in the workflow into or-splits. We assume that here the correct choice is turning the or-split after activity C into an and-split.

The corrected version of the workflow is shown in Fig. 25. The workflow in 25a is now separable (and therefore well-behaved) as the three non-trivial SESE regions show. It consists of two sequential SESE region (one cyclic, one acyclic), both contained in a parallel SESE region. Because of the correction, region R_2 gets an output logic "and" in 25b. The change is local, and only the affected regions have to be transformed again, i.e., region R_2 must be regenerated and the application of rule C_t combining the basic region R_2 and the composite region R_{3+4} needs to be re-examined.

Resuming the transformation from here allows the few remaining steps to be completed as shown in Fig. 26. The composite region R_{3+4} in 26a can be merged with its predecessor R_2, leading to the four remaining regions shown in 26b. At this point, rule P_t can merge the two composite regions in the middle into region R_1, or rule P_s can merge the same regions into region R_{9+10}. The application sequence for the rules has no significant impact in this case, and we just select one possible sequence. Rule P_t, for example, merging regions R_1 and R_{2+3+4}, leads to the situation shown in 26c, and, applied again to merge regions $R_{1+2+3+4}$ and $R_{5+6+7+8}$, to the situation shown in 26d. A final application of rule P_{st} results in the single region shown in 26e.

For space reasons, the content of the composite regions (i.e., the regions contained in other regions) is not shown in Figs. 24 and 26. As an example of how the internals of such a region would look like, the nested containment is depicted in Fig. 27 for region R_{3+4} after the application of rule L. It visualizes a part of the RT. The complete RT for the corrected workflow is presented in Fig. 28 in compact form.

[8]The number of changes needed to fix the problem in the workflow belongs to the criteria on which such suggestions could be based.

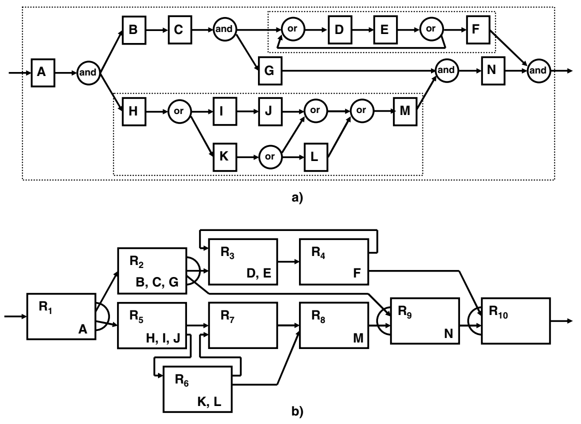

Fig. 25. Corrected sample workflow

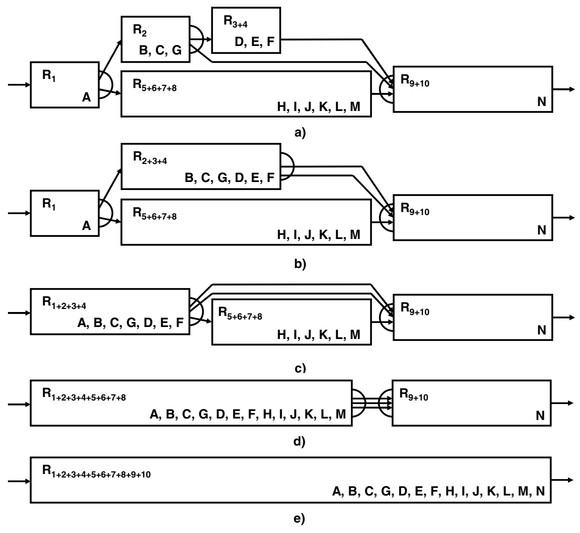

Fig. 26. Rules applied to the corrected sample workflow

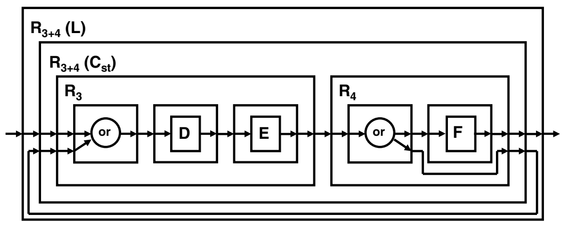

Fig. 27. Detailed content of region R_{3+4} resulting from rule L

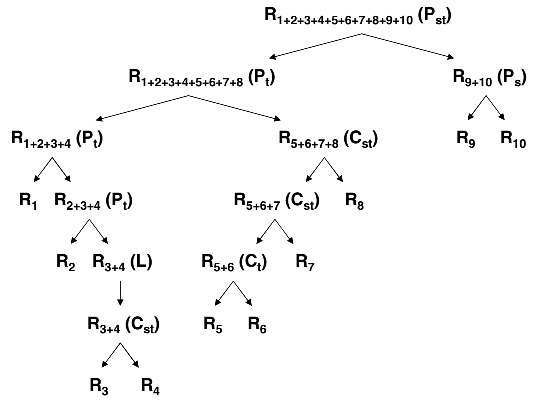

Fig. 28. Region tree for the corrected sample workflow

B. Transformation into Structured Form

A separable workflow consists of nested SESE regions that are either sequential or parallel. The sequential regions can always be transformed into an equivalent[9] structured form [13] and further to the structured BPEL activities `switch` and `while`. Although not all parallel regions can be turned into an equivalent structured form [13], they can be directly transformed into BPEL `flow` activities plus `link` constructs. Thus, the compilation of separable workflows into BPEL is possible.

Workflows that are not separable must contain SESE regions with sequential as well as parallel control nodes. With the region-growing rules, such regions can only occur through rule O or one of the error rules. The input pattern of rule O, i.e., the overlap pattern, can be turned into an equivalent structured form by duplicating the activities between the or-joins and the and-join and then switching these join nodes [12]. The error rules correspond to structural errors that cannot occur in well-behaved workflows. Thus, all well-behaved workflows that can be handled by the region-growing rules introduced in this paper can be converted into an equivalent form that can be represented in BPEL.

[9]For a definition of equivalence, see [13].

To demonstrate the compilation to BPEL in more detail, we examine one region more closely and apply the transformation rules discussed in [18]. Applying these rules blindly to the part of the RT containing region R_{3+4} created by rule L, i.e., the part of the RT shown in Fig. 27, leads to the following BPEL skeleton code:

```
<while condition>
  <invoke D />
  <invoke E />
  <switch>
    <case condition>
      <invoke F />
    </case>
  </switch>
</while>
```

Parameters for the `invoke` activities and conditions for the `while` and `switch` activities have not been set, but it is assumed here that they could be derived from the original workflow. Because the cycle would be better represented by a do-until than by a do-while loop, the condition of the loop must also guarantee that activities D and E are invoked at least once.

As described in [18], rules C_t and C_s tend to move nodes (such as activity F in this example) from the right and from the left, respectively, into the cycles, although these nodes would better stay outside. Because the area of the workflow contributing to a cycle is well-known (see Fig. 27), an optimization step can identify these nodes and move them out of the loop:

```
<while condition>
  <invoke D />
  <invoke E />
</while>
<invoke F />
```

VII. Conclusion

In this paper, we introduced the region tree of a workflow and some transformation rules that allow the region tree to be built in an incremental and iterative way. Three generic rules can be distinguished: one for self-loops, one for processing two neighbors, and one for the overlapped pattern. The regions detected by the rules and the interfaces between them (as defined through the input/output logic of a region) reveal structural information about the workflow that is useful for further applications. We combined two such applications to demonstrate the power of the region tree.

The first area in which we used this structural information is the detection of structural conflicts in workflows. These rules not only detect but also localize the structural conflicts called deadlock, lack of synchronization, and parallel cycles. They can handle cyclic workflows and workflows containing the overlapped pattern, but so far they cannot handle workflows that would require synchronized conditions.

The second possible application area explored in this paper is the transformation (or compilation) of unstructured or insufficiently structured workflows into a more structured form as expected by some runtime platforms. If, for example, the workflow is supposed to be deployed on a workflow engine that is based on BPEL, cycles are only allowed in the form of a do-while loop, and unstructured cyclic workflows therefore have to be transformed into this form first.

This paper concentrated on the demonstration of the concept of the region tree. Future work includes the application of the region tree in other areas, and the definition of additional rules and concepts needed to handle all well-formed workflows even if they require synchronized conditions and general forms of m-out-of-n logic.

References

[1] OMG (MDA), *Model Driven Architecture*, http://www.omg.org/mda/, March 2002.

[2] S. Kent, *Model Driven Engineering*, Proc. 3rd International Conference on Integrated Formal Methods (IFM'02), Turku, Finland, May 2002, LNCS 2335, pp. 286-298.

[3] J. Koehler, R. Hauser, J. Küster, K. Ryndina, J. Vanhatalo, and M. Wahler, *The Role of Visual Modeling and Model Transformation in Business-driven Development*, Proc. 5th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'06), Vienna, Austria, April 2006.

[4] W. Sadiq and M.E. Orlowska, *Analyzing Process Models Using Graph Reduction Techniques*, Information Systems, 25(2), pp. 117-134, 2000.

[5] H. Lin, Z. Zhao, H. Li, and Z. Chen, *A Novel Graph Reduction Algorithm to Identify Structural Conflicts*, Proc. 35th Hawaii International Conference on System Sciences (HICSS-35), 2002.

[6] W.M.P. van der Aalst, A. Hirnschall, and H.M.W. Verbeek, *An Alternative Way to Analyze Workflow Graphs*, Proc. 14th International Conference on Advanced Information Systems Engineering (CAiSE'02), Toronto, Canada, May 2002, LNCS 2348, pp. 535-552.

[7] OMG (UML2), *Unified Modeling Language: Superstructure*, version 2.0, http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf, August 2005.

[8] IBM (Modeler), *WebSphere Business Modeler*, Advanced Version 6.0, http://www-306.ibm.com/software/integration/wbimodeler/.

[9] BPEL4WS, *Business Process Execution Language for Web Services*, version 1.1, ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf, May 2003.

[10] W. Zhao, R. Hauser, K. Bhattacharya, B.R. Bryant, and F. Cao, *Compiling Business Processes: Untangling Unstructured Loops in Irreducible Flow Graphs*, Int. J. Web and Grid Services, 2(1), pp. 68-91, 2006.

[11] R. Johnson, D. Pearson, and K. Pingali, *The Program Structure Tree: Computing Control Regions in Linear Time*, Proc. ACM Sigplan Conference on Programming Language Design and Implementation (PLDI'94), Orlando, Florida, June 1994, pp. 171-185.

[12] R. Liu and A. Kumar, *An Analysis and Taxonomy of Unstructured Workflows*, Proc. 3rd International Conference on Business Process Management (BPM'05), Nancy, France, September 2005, LNCS 3649, pp. 268-284.

[13] B. Kiepuszewski, A.H.M. ter Hofstede, and C.J. Bussler, *On Structured Workflow Modelling*, Proc. 12th Conference on Advanced Information Systems Engineering (CAiSE'00), Stockholm, Sweden, June 2000, LNCS 1789, pp. 431-445.

[14] A. Aho, R. Sethi, and J. Ullman, *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[15] M.S. Hecht and J.D. Ullman, *Flow Graph Reducibility*, SIAM J. Comput., 1(2), pp. 188-202, 1972.

[16] Z. Ammarguellat, *A Control-Flow Normalization Algorithm and Its Complexity*, Software Engineering, 18(3), pp. 237-251, 1992.

[17] R. Hauser and J. Koehler, *Compiling Process Graphs into Executable Code*, Proc. 3rd International Conference on Generative Programming and Component Engineering (GPCE'04), Vancouver, Canada, October 2004, LNCS 3286, pp. 317-336.

[18] R. Hauser, *Transforming Unstructured Cycles to Structured Cycles in Sequential Flow Graphs*, IBM Research Report RZ 3624, August 2005.