An Incremental Approach to the Analysis and Transformation of Workflows Using Region Trees

Rainer Friedrich Hauser, Michael Friess, Jochen Malte Küster, and Jussi Vanhatalo

Abstract—The 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, both analysis and transformation of workflows must be integrated to allow incremental changes of larger workflows. In this paper, we introduce the concept of a region tree (RT) for workflow models that can be used as the central data structure for both workflow analysis and workflow transformation. An RT is similar to a program structure tree and imposes a hierarchy of regions as an overlay structure onto the workflow model. It allows an incremental approach to the analysis and transformation of workflows, and thereby, significantly reduces the overhead because individual regions can be dealt with separately. The RT is built using a set of region-growing rules. The set of rules presented here is shown to be correct and complete in the sense that a workflow is region-reducible as defined through these rules if and only if it is semantically sound.

Index Terms—Business process modeling, control flow, workflow modeling, workflow transformation, workflow verification.

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 behaviors 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 in partially manual and partially automated steps from analysis models via design models to a complete and deployable information technology (IT) solution [3]. To support this development cycle including the deployment: 1) modeling tools are necessary for sketching and refining workflows; 2) support is required for validation, verification, optimization, and testing of workflows; and 3) 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.

Two main groups of graphical notations for workflows are used. Petri nets [4], specifically free-choice Petri nets [5],

have been proposed to model and analyze workflows scientifically [6]. However, the major software products and industry standards including the Unified Modeling Language 2 (UML2) Activity Diagrams [7] are based on simpler, often only informally defined process modeling languages inspired by general-purpose drawing tools and preferred by mathematically less thoroughly trained business analysts. The translation of results described in terms of Petri nets into the theory of these other process modeling languages is not always straightforward.

The challenge of validation, verification, and testing is not only 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 similar to the reduction rules for Petri nets in [4] and [5] were introduced in [8] for process modeling languages to detect structural conflicts in acyclic workflows. One of these graph-reduction rules, the so-called overlapped reduction rule defined for an infrequently occurring pattern, turned out to be insufficient as demonstrated on a sample workflow, and hence, was replaced in [9] by three other, albeit very complicated rules. In [10], another valid workflow was presented in which the original rules fail, and a case was made for using Petri nets to detect structural conflicts because they outperform the reduction algorithms operating on process models and can also handle cyclic workflows. Despite this, a rule-based reduction approach for process modeling languages is not intrinsically limited to acyclic workflows, and a carefully chosen set of rules does not only detect, but, as will be shown later, also localize errors and helps understand the structure of 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 during refinement, as discussed in [3]. One important transformation used in this process is the deployment step, which transforms the workflow into the form required by, and possibly optimized for, the runtime platform. This transformation can be quite complex, because graphical process modeling languages for workflows such as UML2 Activity Diagrams [7] and the related notation used by the IBM WebSphere Business Modeler (Modeler) [11] permit the specification of models that are less structured than are allowed by some runtime platforms such as the workflow engines for the Business Process Execution Language for Web Services (BPEL) [12]. 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 [13].

To enable workflow analysis and workflow transformation for an incremental and iterative approach, we introduce the concept of a region tree (RT). Similar to the program structure tree (PST) [14], the RT imposes a hierarchical structure on the workflow model. Individual regions can then be analyzed separately, and one region can be transformed and refined without affecting other parts. The RT can be built using rules that adapt and extend the reductions rules for detecting structural conflicts described in [8] and [9]. These rules, called region-growing rules, are used to construct a hierarchy of regions as an overlay structure 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 equivalent, more structured versions of the workflow. Unlike a PST, the RT may contain not only the special type of regions called single entry single-exit (SESE) [14], but also more general regions.

Contrary to the reduction rules in [4], [5], and [8], the transformation building the RT does not change the workflow, and thus, does not remove information. By ignoring the content of the regions, it can still be used as a reduction procedure, and the region-growing rules used in this way lead to the concept of region reducibility. It turns out that semantical soundness defined through behavioral properties of workflows is equivalent to region reducibility. In other words, a workflow is region-reducible if and only if it is semantically sound.

This paper, based on the conference paper [15] extended with the main theorems proving the equivalence of semantical soundness and region reducibility, is organized as follows. Section II introduces the basic workflow concepts. In Sections III and IV, the RT and the region-growing rules for workflows are presented. The application of the RT to combine workflow analysis and transformation is illustrated with an example in Section V. In Section VI, we prove that the set of region-growing rules is correct and complete. Finally, the paper concludes with a summary in Section VII.

II. BASIC CONCEPTS

Here, the definition of a workflow is introduced, and structural conflicts as a class of errors are discussed.

A. Workflows and Soundness

Various graphical notations for modeling workflows as process graphs 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 in [16], but not limited to only two edges. The start node, the end node, and the activity[1] are shown in Fig. 1(a), (b), and (c), respectively. The sequential control nodes, choice, and merge, in Fig. 1(d) and (e) are also called or-split and or-join.[2] The parallel control nodes, fork, and join, in Fig. 1(f) and (g) are also called and-split and and-join. These nodes can be connected
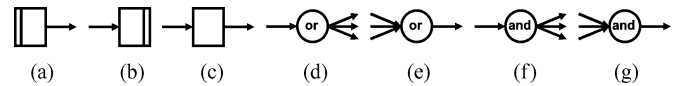

Fig. 1.   Workflow graph elements.

through edges, and a directed graph built with these elements is called a *workflow*. In the following, we assume that all workflows are *structurally sound* [17], i.e., they contain exactly one start and one end node, and there is a path from the start node to every node and a path from every node to the end node.

We introduce the semantics, i.e., the behavior, of a structurally sound workflow 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 [16]. 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.

Several concepts of semantical soundness of workflows exist. In addition to the original definition introduced for workflow nets in [6], relaxed, weak, and lazy soundness have been proposed depending on the workflow patterns in [18] to be modeled and on workflow properties such as what happens to running activities when the workflow terminates after the end node received a token [17]. We use the initial definition adapted from workflow nets and define semantical soundness as follows: executions of a workflow are described through the flow of the 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 1: A structurally sound workflow is *semantically sound* if and only if all possible executions terminate successfully.

To avoid unreachable nodes in general and infinite loops in particular, we assume further that every or-split will enable each of its outgoing edges eventually if reached often enough in the same or in different executions.

B. Structural Conflicts

Not all structurally sound workflows are semantically sound. Fig. 2 shows prototypical examples of structural conflicts. The first two were identified in [8], [16], and [19], where the situation in Fig. 2(a) is called *deadlock* and the one in Fig. 2(b) is called either *lack of synchronization* or *multiple active instances of the same activity*. The third structural conflict shown in Fig. 2(c) 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*

---

[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.
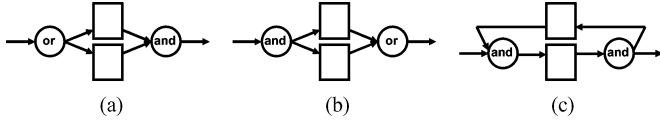
Fig. 2.   Structural conflicts.

Fig. 3.   Substitution of workflow w_2 for edge e in workflow w_1.

synchronization always receives at least two tokens, and, depending on the semantics of the or-join [16], either discards all but one token or emits one token for each token received.[3] These two structural conflicts may not necessarily be considered an error when using more relaxed definitions of soundness, but certainly need careful inspection.

In general, *deadlock* is a situation in which an and-join gets some but not all tokens, and *lack of synchronization* is a situation in which an or-join gets too many tokens on its incoming edges. The *parallel cycle* is also a kind of deadlock: it occurs 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 will never emit a token.

III. REGION ANALYSIS

Here, relevant concepts regarding workflow regions in general and the RT in particular are introduced.

A. Single-Entry Single-Exit Regions

An important type of region is the *SESE* region known from compiler theory [14]. Informally speaking, an SESE region is a set of nodes that does not contain the start and/or end node, 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 structurally sound workflow other than the start and end node build an SESE region, with the edge leaving the start node as the incoming edge and the edge going to the end node as the outgoing edge. In the following, the terms workflow and SESE region will, therefore, often be used interchangeably, and most figures will only show the SESE region instead of the workflow with start and end node. Consequently, we call an SESE region *semantically sound* if and only if the corresponding workflow with start and end node is semantically sound.

Different levels of syntactical structuredness can be defined using the concept of *substitution*, i.e., of replacing an edge e in a workflow w_1 with a structurally sound workflow w_2. As depicted in Fig. 3, edge e is removed from w_1, the start and end node 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 target of the edge going to the original end node of w_2. A part of workflow w_1 with edge e is shown in Fig. 3(a), workflow w_2 is depicted in Fig. 3(b), and the result is presented in Fig. 3(c).

---

[3]As the or-join allows different behaviors, the term *lack of synchronization* is more appropriate than *multiple instances of the same activity*.
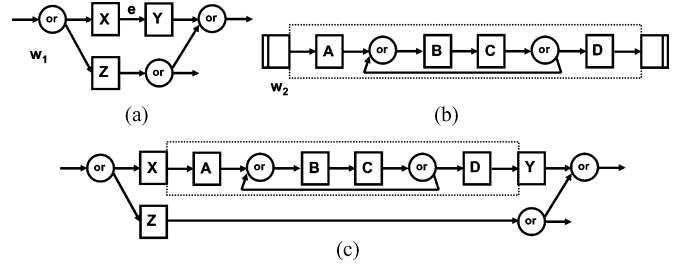
The simplest level, apart from linear workflows, i.e., workflows with a single path from start to end node, is called *structured* in [19] and consists of those workflows in which each or- or and-split has one corresponding or- or and-join, respectively.

Definition 2: A workflow is *structured* if and only if it can be constructed using the following inductive rules.
1) All structurally sound linear workflows, i.e., workflows without control nodes, are structured.
2) All structurally sound workflows with one or-split and one or-join as the only control nodes are structured.
3) All structurally sound 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.

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*.

Definition 3: A workflow is *separable* if and only if it can be constructed using the following inductive rules.
1) All structurally sound workflows with only or-splits and or-joins as control nodes are separable.
2) All structurally sound 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.

Clearly, all structured workflows are separable, but not all separable workflows are structured. Separable workflows have the following important property.

Lemma 1: All separable workflows are semantically sound.

Proof: We observe the token flow in one single SESE region that is either sequential or parallel.
1) In an 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.
2) In an acyclic SESE region with only parallel control nodes and activities (or other SESE regions), one token passes through every edge exactly once.

In both the cases, all executions terminate successfully.   ∎

However, there are semantically sound workflows that are not separable. Workflows containing so-called overlapped patterns, as shown in Fig. 4, mix and-splits with or-joins or or-splits
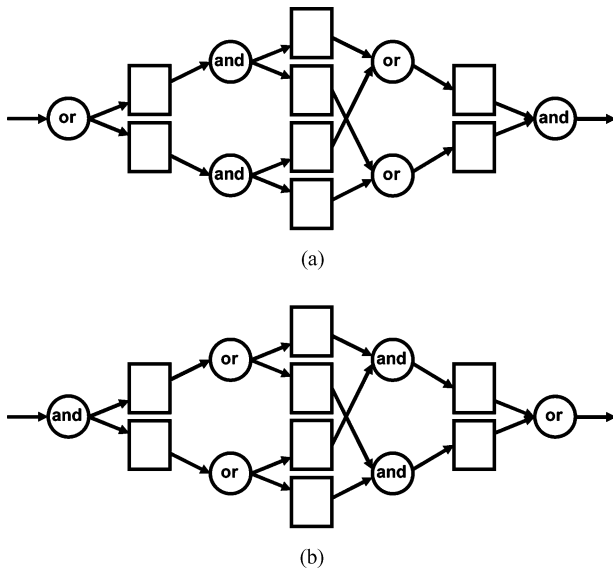
(a)

(b)

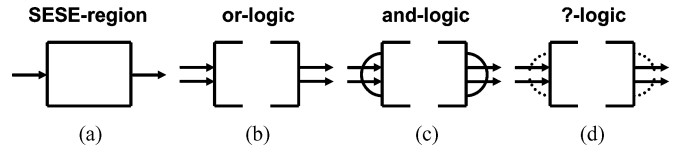Fig. 4.   Overlapped patterns.

Fig. 5.   Region graph elements with input/output logic.
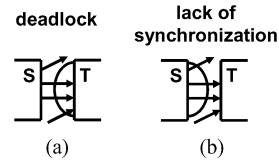

Fig. 6.   Structural conflicts between regions.

Fig. 7.   Workflow with a partition into well-defined basic regions.

with and-joins in such a way that executions of the workflow can terminate successfully. The pattern in Fig. 4(a) made it necessary to define a special rule [8]. All executions of this workflow terminate successfully, and the pattern is semantically sound. For the dual workflow, i.e., a workflow in which or-splits and -joins are replaced with and-splits and -joins, and vice versa, as in Fig. 4(b), executions can only terminate successfully if the two or-splits both enable either the upper edge or the lower edge. The pattern is, therefore, not semantically sound. If or-splits (one or more) in a workflow need additional information to make a workflow execute, their conditions are called *synchronized*. Other examples in which or-splits need synchronization of their conditions, e.g., to exit two parallel cycles at the same time, are discussed in [19].

B. Well-Defined Input and Output Logic

When seen as a black box, an SESE region behaves with respect to the flow of tokens in a similar way as does an activity. If a token enters the region through the incoming edge, a token will eventually come out on the outgoing edge. This approach of describing the behavior of regions as black boxes through their interfaces can be generalized by means of the definition of the input/output logic of a region. Informally speaking, the input logic is "or" if and only if a token on one of the incoming edges triggers the region, and it is "and" if and only if a token on all its incoming edges triggers the region. Similarly, the output logic is "or" if and only if the region eventually emits a token on one of its outgoing edges when triggered, and it is "and" if and only if the region eventually emits a token on all its outgoing edges when triggered. In this way, regions with an input/output logic can also be treated as black boxes with respect to their token-flow behavior, and the input/output logic determines the two interfaces of a region.

If the behavior of the interfaces of a region can be described in this way, we call the region and its interfaces *well defined*.
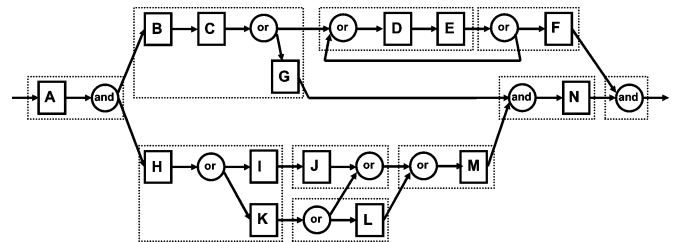
The graphical notations used in the following are presented in Fig. 5. In Fig. 5(a), an SESE region is shown. The input and output logic is "or" in Fig. 5(b), "and" in Fig. 5(c), and either unknown or irrelevant but still well defined in Fig. 5(d). If a region has one incoming edge, the input logic can be interpreted as "or" or "and," and if a region has one outgoing edge, the output logic can be interpreted as "or" or "and."

As illustrated in Fig. 6, the structural conflicts defined for acyclic workflows become incompatible input and output logic for regions. A region S with output logic "or" connected to a region T with input logic "and" through two or more edges, as in Fig. 6(a), results in a *deadlock*. A region S with output logic "and" connected to a region T with input logic "or" through two or more edges, as in Fig. 6(b), corresponds to a *lack of synchronization*.

C. Region Tree

Starting with a partition into well-defined basic regions, we can build composite regions by combining one or more 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 an *RT*, similar to the PST in [14], with the remaining single region as its root.

An initial partition in which each node of the workflow is packed into its own region is a valid starting point. Alternatively, also the initial partition in which each region contains only one single control node, but may contain in addition an arbitrary number of activities, as in Fig. 7, is valid.

For a single workflow, many different RTs can be constructed. Depending on how the RT is created, it will reveal more or less of the structure of the workflow. In the following section, we
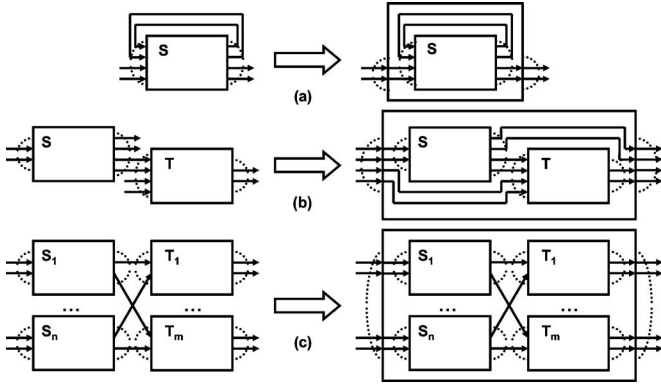
Fig. 8. Three families of region-growing rules.

Fig. 9. Rules for self-loops.

Fig. 10. Rules for two neighbors with $\{S\} = pred(T)$, $\{T\} = succ(S)$.

Fig. 11. Rules for two neighbors with $\{S\} \subset pred(T)$, $\{T\} = succ(S)$.

define region-growing rules to build the composite regions in such a way that we can detect structural errors.

IV. REGION-GROWING RULES

In this section, three families of region-growing rules will be introduced that allow new well-defined regions to be built from existing well-defined regions.

A. Families of Rules

Transformation rules have a left-hand side specifying the pattern expected by the rule and a right-hand side showing 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.

In the following, we present the three families of rules shown in Fig. 8 with their member rules, but without the error rules discussed in [15]. The simplest family of rules covers cycles, as depicted in Fig. 8(a). The family of rules for two neighbors shown in Fig. 8(b) contains different member rules for the possible input and output logics of regions S and T and depending on the successors of S or predecessors of T. These two families have a fixed number of input regions. The third family, shown in Fig. 8(c), has a variable number of input regions. It covers the overlapped patterns.

The first two families of region-growing rules are based but only to a limited extent on the reduction rules in [8] and [9]. Their more important root is compiler theory [20], explicitly the $T1 - T2$ analysis [21], the area of goto-elimination [22] and subsequent work on cycle-removal transformations for sequential (and therefore separable) workflows [23]. Although the $T1 - T2$ analysis was invented as a method for determining irreducibility, it turned out that reducibility for cycle removal is far less important than it seemed [24]. Note that these two families of rules, i.e., the rules for self-loops and for two neighbors, correspond to the $T1$ and $T2$ rule from $T1 - T2$ analysis, respectively, extended to handle irreducibility and parallelism.
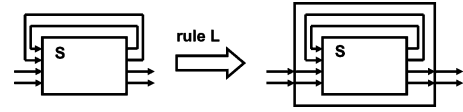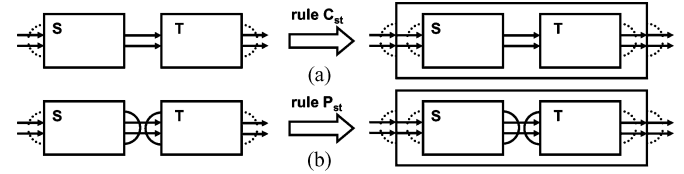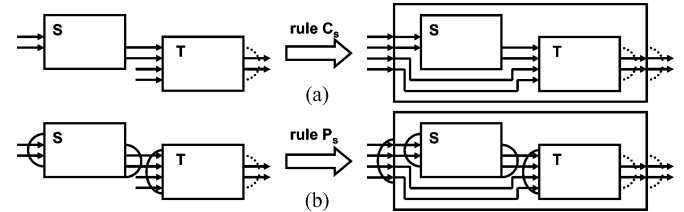
For the rules represented graphically (such as the one shown in Fig. 8), we use the following conventions: a single edge represents exactly one edge, two edges with the same source and/or target region mean one or more edges.

B. Rules for Self-Loops

The only rule for handling self-loops, i.e., edges in which the source and target are the same region, is *rule L* shown in Fig. 9. The other combinations of input/output logic for region S correspond to the structural conflicts shown in Fig. 2.

C. Rules for Two Neighbors

Two regions S and T ($S \neq T$) 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 member rules.

The first set of member rules is shown in Fig. 10. It covers the cases in which region S is the only predecessor of region T and region T is the only successor of region S: $\{S\} = pred(T)$ and $\{T\} = succ(S)$. The output logic of S and the input logic of T must be compatible. *Rule C_{st}* with "or" logic is shown in Fig. 10(c) and *rule P_{st}* with "and" logic in Fig. 10(b).

The second set of member rules is shown in Fig. 11. It covers the cases in which 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 Fig. 11(a) and *rule P_s* in Fig. 11(b) correspond to the two possible cases in which the output logic of S and the input logic of T are consistent with each other.

The third set of member rules is shown in Fig. 12. It covers the cases in which region S is the only predecessor of region T, but has successors other than T: $\{S\} = pred(T)$ and
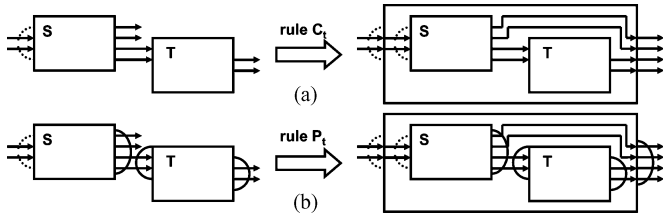
Fig. 12. Rules for two neighbors with $\{S\} = pred(T)$, $\{T\} \subset succ(S)$.
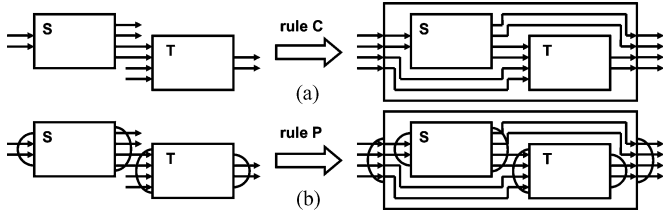

Fig. 13. Rules for two neighbors with $\{S\} \subset pred(T)$, $\{T\} \subset succ(S)$.
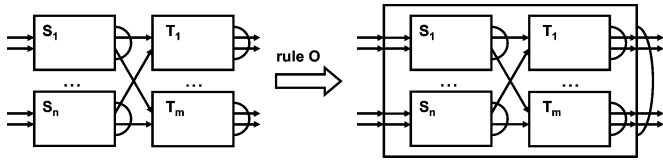

Fig. 14. Rules for overlapped patterns.

$\{T\} \subset succ(S)$. *Rule* C_t in Fig. 12(a) and *rule* P_t in Fig. 12(b) correspond to the two possible cases in which the output logic of S and the input logic of T are consistent with each other.

The fourth and final set of member rules is shown in Fig. 13. It covers the cases in which 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 Fig. 13(a) and *rule* P in Fig. 13(b) correspond to the two possible cases in which the output logic of S and the input logic of T are consistent with each other.

D. Rules for Overlapped Patterns

The overlapped pattern is a situation in which a group of n regions S_i ($n \geq 2$) is connected to a group of m regions T_j ($m \geq 2$) in such a way that from every S_i exactly one edge leads to every T_j. The only member rule allowed in this family is *rule* O shown in Fig. 14.

E. Region Reducibility

Depending on the goal, the region-growing rules can be applied differently. The effect of the application strategy as well as its properties and pitfalls such as pseudocycles have been discussed in [15]. Here, we concentrate on the set of workflows that can be resolved by the region-growing rules if these rules are used as reduction rules.

Definition 4: An SESE region is *region reducible* if it can be reduced to a single region with one incoming and one outgoing edge using the region-growing rules L, C_{st}, C_s, C_t, C, P_{st}, P_s, P_t, and O when starting from an initial partition into basic regions with at most one control node per region.
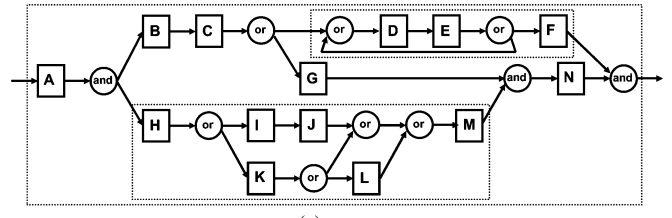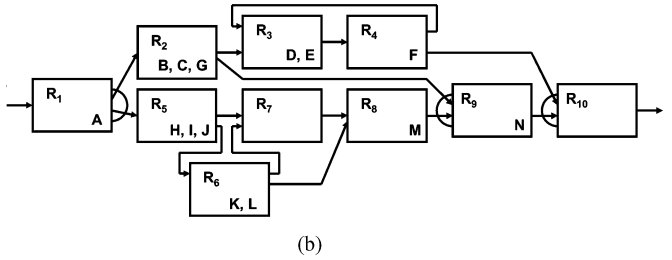
Fig. 15. Sample workflow with deadlock.

Note that rule P is not used because it is not needed and would in addition cause problems. Moreover, either rule P_s or rule P_t is redundant too. Because structurally sound workflows and SESE regions can be used interchangeably here, the concept of region reducibility can be extended to workflows in a straightforward way by removing their start and end node.

Termination and confluence are important properties of such rule-based systems. The application of the region-growing rules always terminates because each rule reduces the number of edges and either keeps the number of nodes the same or also reduces the number of nodes. Used as transformation rules, the set of region-growing rules is not confluent, because if rule L has highest priority on a cyclic sequential workflow, many cycles may be detected, but if rule L has lowest priority, all cycles are combined into one single cycle [24]. Used as reduction rules, the set of region-growing rules is, however, confluent, as will be shown later.

V. 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. 7.

A. Analysis for Structural Conflicts

The example workflow and its initial regions are shown in Fig. 15. Even with the nontrivial SESE regions marked in Fig. 15(a) by dotted rectangles, it is not obvious that it contains a deadlock. The basic regions from the initialization in Fig. 7 have been given names, and the regions are annotated with the names of the activities they contain in Fig. 15(b).

Fig. 16 shows the transformation steps until the deadlock becomes visible. Rule C_{st} is applied to regions R_3 and R_4 to obtain the state shown in Fig. 16(a) with region R_{3+4}, to which rule L can be applied, as shown in Fig. 16(b). The situation in Fig. 16(c) results 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 Fig. 16(d) and with region R_8 using rule C_{st}, as shown in Fig. 16(e). Next, rule C_t combines regions
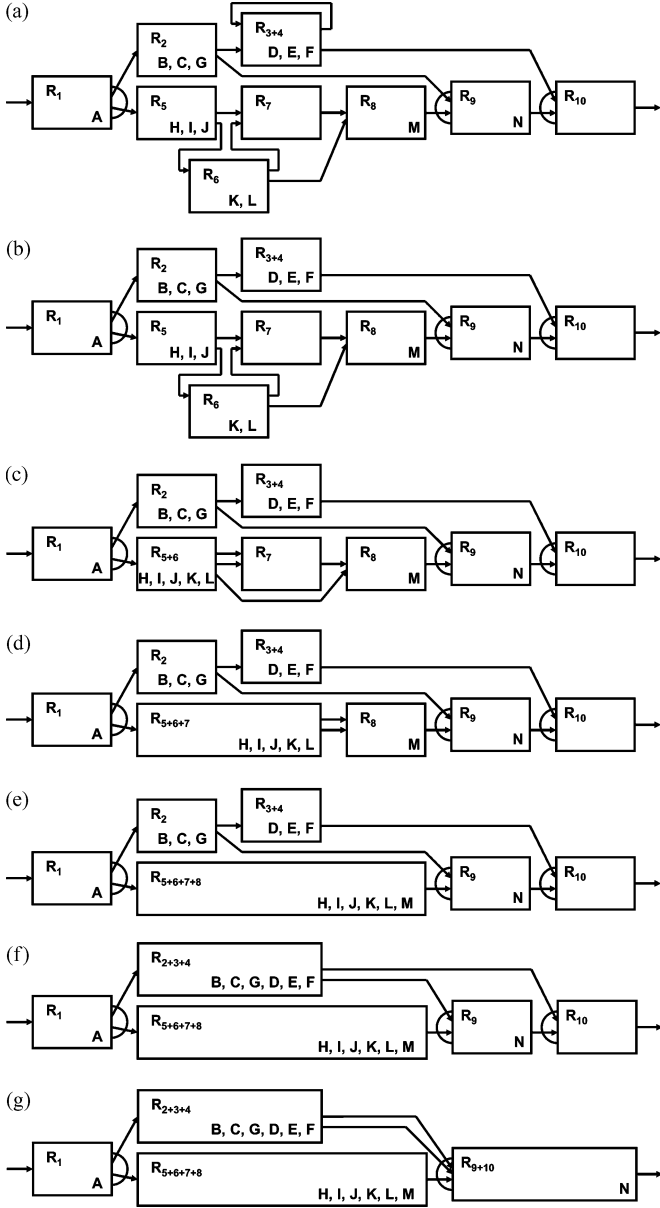
Fig. 16. Rules applied to the sample workflow with deadlock.

Fig. 17. Corrected sample workflow.

Fig. 18. Rules applied to the corrected sample workflow.

R_2 and R_{3+4}, leading to the state in Fig. 16(f). 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 Fig. 16(g), in which the deadlock between regions R_{2+3+4} and R_{9+10} becomes visible.

We consider the correction of such problems a manual step, although the algorithm that detected the conflict may come up with suggestions.[4] These suggestions can indicate which steps could lead to a structurally correct workflow, but only the de-

---

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

signer 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 to turn the or-split after activity C into an and-split.

The corrected version of the workflow is shown in Fig. 17. The workflow in Fig. 17(a) is now separable (and therefore, semantically sound according to Lemma 1), as shown by the three nontrivial SESE regions. 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 Fig. 17(b). The change is local, and only the regions affected have to be processed again, i.e., region R_2 must be regenerated and the application of rule C_t combining regions R_2 and R_{3+4} needs to be reexamined.

Resuming the transformation from here allows the remaining steps to be completed, as shown in Fig. 18. The composite

Fig. 19. Detailed content of region R_{3+4} resulting from rule L.

Fig. 20. RT for the corrected sample workflow.

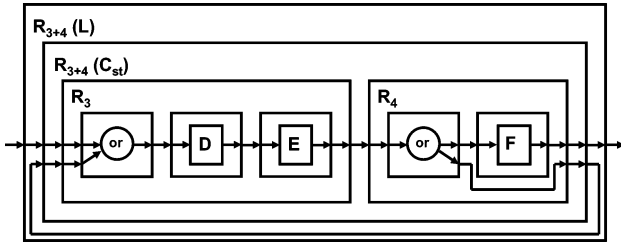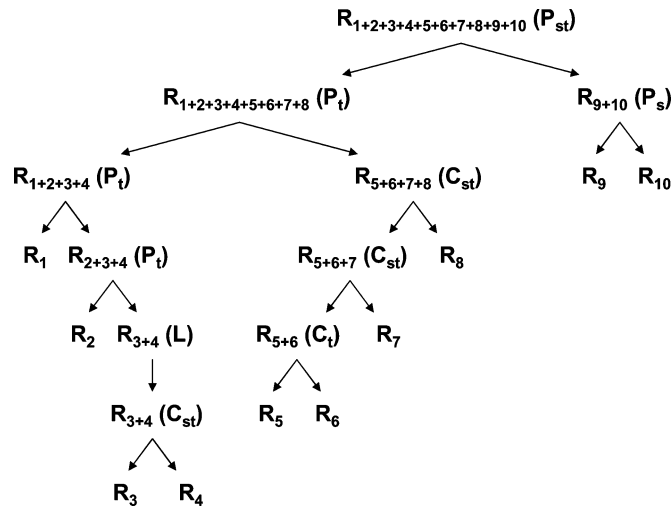region R_{3+4} in Fig. 18(a) can be merged with its predecessor R_2, leading to the four remaining regions shown in Fig. 18(b). At this point, rule P_t can merge the two composite regions in the middle into region R_1, or rule P_s can merge the same regions into region R_{9+10}. The application sequence of 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 Fig. 18(c), and, applied again to merge regions $R_{1+2+3+4}$ and $R_{5+6+7+8}$, to the situation shown in Fig. 18(d). A final application of rule P_{st} results in the single region shown in Fig. 18(e).

The content of composite regions, i.e., regions contained in other regions, is not shown in Figs. 16 and 18. The nested containment for region R_{3+4} after applying rule L, as an example, is depicted in Fig. 19. The complete RT for the corrected workflow is presented in Fig. 20 in compact form.

B. Transformation Into Structured Form

For MDE interpreted as the field of developing complete applications and other systems using visual models and modeling tools (such as the Modeler [11]), the transformation from graphical process models to the deployable code expected by the runtime platform (e.g., BPEL) is analogous to the compilation from a high-level programming language to machine code expected by the underlying hardware platform. Cycle-removal transformations and other transformations for workflows from a less to a more structured form are part of this activity.

The equivalence of workflows and the transformation of unstructured workflows into an equivalent structured form has been studied in [19]. Sequential SESE regions can always be transformed into an equivalent structured form and further to the structured BPEL activities `switch` and `while`. Although not all parallel SESE regions can be turned into an equivalent structured form, they can be directly transformed into BPEL `flow` activities plus `link` constructs. Thus, the compilation of separable workflows into BPEL is possible. The semantically sound overlapped pattern can be turned into an equivalent structured form by first duplicating the activities between the or-joins and the and-join and then switching these join nodes [16]. Thus, all region-reducible workflows can be converted into an equivalent form that can be represented in BPEL.

To demonstrate the compilation to BPEL in greater detail, we examine one region more closely and apply the transformation rules discussed in [24]. Applying these rules blindly to the part of the RT shown in Fig. 19 leads to the following BPEL skeleton code.

```
⟨while condition⟩
    ⟨invoke D /⟩
    ⟨invoke E /⟩
    ⟨switch⟩
      ⟨case condition⟩
        ⟨invoke F /⟩
    ⟨/case⟩
  ⟨/switch⟩
⟨/while⟩
```

The parameters for the `invoke` activities and the 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 [24], 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. 19), an optimization step can identify these nodes and move them out of the loop.

```
⟨while condition⟩
  ⟨invoke D /⟩
  ⟨invoke E /⟩
⟨/while⟩
⟨invoke F /⟩
```

VI. MAIN THEOREMS

The main theorems prove that a structurally sound workflow is semantically sound if and only if it is region reducible.

A. Correctness Theorem

From the definition of the region-growing rules, the following theorem is to be expected.

Theorem 1: (Correctness Theorem) If a structurally sound workflow is region reducible, it is semantically sound.

Proof: The behavior of a region with well-defined input and output logic in terms of the flow of tokens is fully described through its input and output logic, i.e., through its interfaces. Therefore, we have to show that: 1) the initial regions with at most one control node are well defined and 2) for all rules used in the definition of region reducibility, the behavior of the region on the right-hand side is the same as the behavior of the region pattern on the left-hand side. The first part of the proof is trivial because basic regions having no control node are SESE regions, and basic regions containing one control node inherit their logic from the control node.

For the second part of the proof, we show as an example that rule C_t in Fig. 12(a) is well defined and leave the proof for the other rules to the reader.[5] If the number of tokens expected by the input logic of region S is available, region S eventually submits either a token on one of its upper two edges or a token on one of the lower two edges to region T. In the latter case, region T will eventually emit a token. The new region on the right-hand side of the rule has the same input logic as region S, and, in any case, it will emit eventually one and only one token on one of its outgoing edges when the expected number of tokens is available on the input side. ∎

B. Nonseparable Patterns

Practically, all workflows used in reality are separable. Thus, workflows with an overlapped pattern come as a surprise for most people when they are first exposed to them. If such an unexpected pattern exists, the question arises whether other patterns can be found among the semantically sound workflows that are also not separable. We will show in the following lemmas that this is not the case.

Before doing this, we first define the *overlapped pattern* more precisely as n and-splits ($n \geq 2$) and m or-joins ($m \geq 2$) with one path from every and-split to every or-join such that: 1) the and-splits have no other outgoing paths; 2) the or-joins have no other incoming paths; and 3) the regions between the and-splits and or-joins are SESE regions (if the path from the and-split to the or-join is not just one single edge). The last point guarantees that the token emitted by an and-split to an or-join eventually arrives and that no other tokens are created or consumed in this part of the workflow.

Lemma 2: If a workflow w contains an SESE region that is not semantically sound, w is not semantically sound.

Proof: If the SESE region is not semantically sound, there are executions in which the region consumes a token, but does not emit one, or there are executions in which the region consumes and emits a token, but unconsumed tokens remain in the region.

1) There is no node in a workflow that can consume a token without emitting one except for the end node. Because an end node cannot be part of the SESE region, at least one
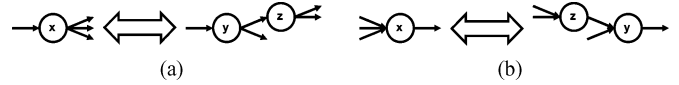

Fig. 21. Splitting and merging of control nodes.

unconsumed token must remain inside the SESE region if a token entered but no token left the region.

2) If a token remains inside the SESE region when another token leaves it, there are three possibilities: a) the workflow terminates with the unconsumed token still in the SESE region; b) the workflow never terminates; or c) the additional token leaves the SESE region as well. Only the last case needs further considerations. Because there are no synchronized conditions, the second token leaving the region may take the same path as the first token for some executions. However, there are no nodes that can emit tokens after consuming two tokens waiting on the same incoming edge, but could not emit tokens if only one token is waiting there. Therefore, w either never terminates or does so with unconsumed tokens.

In any case, w is not semantically sound. ∎

Thus, if a workflow w contains an SESE region that is not semantically sound, this cannot be fixed in another part of w.

Lemma 3: If a semantically sound workflow is not separable, it contains an overlapped pattern.

Proof: Let us assume that the structurally sound workflow w is semantically sound but not separable. We show that if all executions of w terminate successfully, w contains an overlapped pattern.

We first simplify w in such a way that the token flow, and thus, also the property of semantical soundness are not affected. (These simplifications have an obvious similarity with the reduction rules defined in [8].) Activities and, similarly, SESE regions that must be semantically sound owing to Lemma 2 can be eliminated because they only influence the timing of the token flow. Separable SESE regions can simply be removed, and nonseparable SESE regions can be checked for overlapped patterns independently. Thus, we can assume that w does not contain any SESE region except for the one resulting when the start and end node have been removed.

Furthermore, and as illustrated in Fig. 21, two adjacent control nodes of the same type can be merged into one control node of this type, or one control node (with enough edges) can be split into two adjacent control nodes of the same type if this is needed to eliminate additional separable parts of w. Assuming that x, y, and z have the same type, the split control node x with at least three outgoing edges and the two adjacent split control nodes y and z in Fig. 21(a) are equivalent, and similarly, the join control node x with at least three incoming edges and the two adjacent join control nodes y and z in Fig. 21(b) are equivalent. In this way, two direct edges from a split node to a join node of the same type can be turned into an SESE region and can be eliminated in this way.

Fig. 22 illustrates the remaining steps of the proof once all possible simplifications of this kind have been performed.

[5]Note that rule P in Fig. 13(b) is not well defined, because tokens on the upper two input edges of the new region would result in tokens on the upper two output edges even without tokens pending on the lower two input edges.
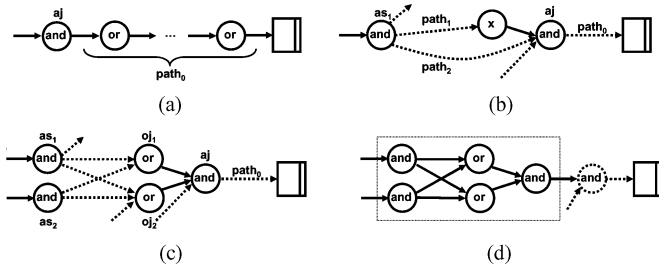
Fig. 22.   Situations discussed in the proofs of Lemmas 3 and 4.

1) Workflow w must contain at least one parallel control node because otherwise w would be separable. We select one, as shown in Fig. 22(a), having only sequential control nodes (zero or more) on a path $path_0$ to the end node. If this parallel control node emits a token, a token reaches the end node in at least one possible execution through $path_0$. The parallel control node must, therefore, be an and-join aj because an and-split would lead to unconsumed tokens, and thus, to unsuccessful executions.

2) We select an and-split as_1 and two paths $path_1$ and $path_2$ such that the two paths lead from as_1 to aj and do not contain another and-split. (If this is not possible, there must be unconsumed tokens or simplifications that have not yet been performed. Note that no other tokens must be in the workflow when aj emits a token.) At least one of the two paths, say $path_1$, must contain a node x that is not as_1 as the predecessor of aj, because the two paths would otherwise have been eliminated through the simplifications. Node x, as shown in Fig. 22(b), cannot be an and-split because we selected as_1 such that there is no other and-split on the two paths. It also cannot be an and-join because it would have been merged with aj. It cannot be an or-split because this would require synchronized conditions as a token would otherwise reach aj on $path_2$, but not necessarily on $path_1$. Thus, x must be an or-join oj_1.

3) If node oj_1 gets a token through an input edge other than the one on path $path_1$, a token must also arrive at aj via the last edge of $path_2$. As all SESE regions have been eliminated during the initial simplification step, the token arriving at oj_1 cannot come from a node on $path_1$. Thus, there must be an or-join oj_2 on $path_2$ (it is the predecessor of aj, for the same reasons as before), and there must be another and-split as_2 leading to these two or-joins, as shown in Fig. 22(c). We can repeat this argument if one of the or-joins, say oj_2, has other incoming edges. If one of the and-splits, say as_1, has other outgoing edges, the token sent on it must be consumed before aj in such a way that no deadlock occurs if as_2 emits tokens. This is only possible if there are n and-splits as_i and m or-joins oj_j arranged in such a way that whenever one of the as_i receives a token, all oj_j eventually get a token. Anything else would lead to unconsumed tokens or a deadlock at aj.

4) If tokens are emitted from one as_i to every oj_j, these tokens and only these tokens must also arrive. Because all SESE regions have been removed, this is only possible through direct edges. If aj has incoming edges not coming

from a oj_j, we split aj such that all its incoming edges come from one of the or-joins oj_j. We can, therefore, define a region as depicted in Fig. 22(d) (for the case $n = 2$ and $m = 2$), in which the only edges leading into this region are the incoming edges of the and-splits and the only edge leading out of this region is the outgoing edge of the and-join.

This region is an overlapped pattern connected to an and-join at the back.   ∎

The proof only shows that there must be an overlapped pattern in a semantically sound workflow that is not separable, but not that there are no other unexpected patterns.

Lemma 4: The overlapped pattern is the only non-separable pattern in semantically sound workflows.

Proof: The rectangle shown in Fig. 22(d) is a region with exactly n incoming edges leading to the n and-splits and one outgoing edge leading eventually to the end node. If a token comes into the region through one of the incoming edges, a token will come out on the outgoing edge. With respect to the token flow, this region is equivalent to a single or-join.

If the workflow after replacing the region with an or-join is separable, we are done. If it is not separable, we repeat the argument in the proof of Lemma 3. The original workflow has only a finite number of nodes, and each step replacing a region with an or-join reduces the number of nodes. Therefore, we reach a separable workflow in a finite number of steps.   ∎

C. Confluence Theorem

The rule application sequence is important for the creation of an RT, but is irrelevant when determining whether an SESE region or a workflow is region reducible.

Theorem 2: (Confluence Theorem) If one rule-application sequence shows that an SESE region is region reducible, also all other possible rule application sequences do so.

Proof: We assume that an SESE region W has been shown to be region reducible through one application sequence of the rules (and therefore, is semantically sound owing to Theorem 1), but another application sequence terminates before the SESE region has been reduced to a single region having one incoming and one outgoing edge. The assumption is that no rule can be applied to the remaining region graph.

The proof is similar to the one of Lemma 3. We start from the outgoing edge of the SESE region W and determine possible regions that may remain if we avoid: 1) combinations of regions that could be resolved with one of the region-growing rules and 2) configurations that lead to deadlocks and/or unconsumed tokens. The situations described during the proof are shown in Fig. 23.

The last region L, i.e., the region with the outgoing edge of W, must have "or" output logic, because no unconsumed tokens are allowed to remain in W, when a token leaves the outgoing edge of W. If an outgoing edge of L leads back to L, i.e., if there is a self-cycle, L must have "or" input logic, because a deadlock or unconsumed tokens would result otherwise, but rule L could have been applied in this situation contrary to the earlier assumption.
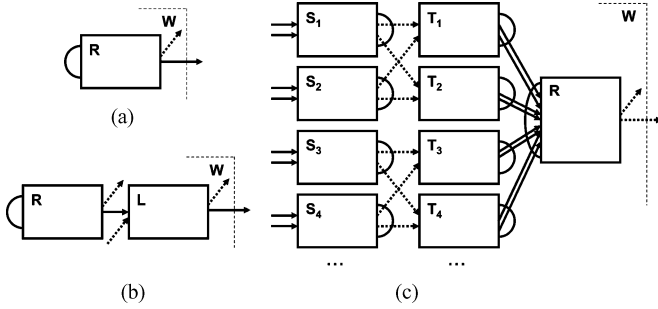
Fig. 23. Situations discussed in the proof of Theorem 2.

Thus, region L cannot be the only remaining region, and we can deduce the following structure of regions.

1) We look for a region R with "and" input and "or" output logic close to the exit of the SESE region W. If region L has "and" input logic, we select it to be region R, as shown in Fig. 23(a).

2) Otherwise, we take one of the predecessors R of L, as shown in Fig. 23(b), and note that R must have "and" input logic, because with "or" logic, one of the sequential region-growing rules for two neighbors could have been applied to R and L.

3) In both cases, R has "or" output logic and "and" input logic, and it emits the only token available in W when it emits a token because otherwise unconsumed tokens would result.

   We determine all predecessors of R and call them T_j. There must be more than one because either rule P_{st} could have been applied or there must be deadlocks and/or unconsumed tokens. If one T_j sends a token to R, all the others eventually must do so also, and we can further conclude that all T_j have "and" output logic (or only one outgoing edge), with all outgoing edges leading directly or indirectly to R, because "or" output logic would require synchronized conditions. The connections, however, cannot be indirect because either rule P_{st} would be applicable to one of the T_j and R or deadlocks and/or unconsumed tokens would result otherwise.

   Because all T_j must emit tokens to R when one T_j does, there must be regions S_i with direct or indirect paths to the T_j and with "and" output logic such that every T_j is connected to at least two S_i. (If there is only one, the input logic of the T_j would either be "and" or could be interpreted as "and," and a rule for two neighbors could have been applied to these T_j and R.) Note that not every S_i may lead to every T_j, but that the S_i and T_j can be partitioned such that every pair S_i and T_j is connected if in the same partition or are unconnected if in different partitions. This situation is shown in Fig. 23(c) for two partitions. The first partition consists of regions S_1, S_2, T_1, and T_2, and the second partition consists of regions S_3, S_4, T_3, and T_4.

   If one of the S_i in every partition gets enabled, it sends a token to every T_j in the same partition, and eventually every T_j is enabled. Thus, if one S_i per partition is enabled,

the SESE region W may emit a token. Therefore, the S_i must have "or" input logic. The connections between the S_i and the T_j must be such that whenever an S_i sends a token on the path to the T_j, a token must also arrive there. That is only possible through direct edges (or through other SESE regions). Thus, each partition contains at least two S_i and two T_j and builds an overlapped pattern. Contrary to our assumption, rule O can be applied. ∎

D. Completeness Theorem

The region-growing rules allow all semantically sound workflows to be detected.

Theorem 3: (Completeness Theorem) If a structurally sound workflow is semantically sound, it is region reducible.

Proof: We determine all SESE regions of the workflow and select an innermost region, i.e., a, SESE region that does not contain other SESE regions. Because of the definition of separable workflows and the Lemmas 3 and 4, the following three cases have to be discussed.

1) *Sequential SESE region*: As long as the SESE region contains at least two regions, one of the rules C_{st}, C_s, C_t, or C can be applied because they cover all cases of two neighbors with only "or" logic. If in the end a single region remains that has more edges than the incoming and the outgoing edge, these edges must be self-cycles and can be resolved with rule L.

2) *Parallel SESE region*: As the region is not allowed to contain cycles, there must be a path from the first region, i.e., the region with the incoming edge of the SESE region, to the last region, i.e., the region with the outgoing edge of the SESE region, having maximal path length. Rule P_{st} or rule P_t must be applicable to the first two regions on this path because the second region cannot have predecessors other than the first region in the SESE region, as otherwise a longer path than the one with maximal path length would result.

3) *Overlapped pattern in an SESE region*: The overlapped patterns can be removed with rule O. We observe that the result of rule O is the same as the result of rule C_{st} (or P_{st}) applied to a region S with "or" input logic and a region T with "and" output logic connected through a single edge.[6] Thus, there exists a different workflow that does not contain overlapped patterns, but that would have led to the same configuration. Because a semantically sound SESE region without an overlapped pattern must be separable, we can apply Theorem 2 together with the proof for sequential and parallel SESE regions.

In any case, the SESE region can be reduced to a single region with one incoming and one outgoing edge, and therefore, can be replaced by an activity without changing the behavior in terms of token flow. In this way, the entire workflow can be resolved, SESE region by SESE region, from inside out. ∎

---

[6]Compare the abstraction rule ϕ_A in [5] and the merge-fork reduction rule in [9] that are based on the same observation.

VII. Conclusion

In this paper, we introduced the RT of a workflow and the region-growing rules that allow the RT to be built in an incremental and iterative way. Three families of rules have been proposed: one for self-loops, one for processing two neighbors, and one for overlapped patterns. 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 on an example.

The first area in which we used this structural information is the detection of structural conflicts in workflows. The rules not only detect but even localize the structural conflicts called deadlock, lack of synchronization, and parallel cycles. They can handle cyclic workflows and workflows containing overlapped patterns, but 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 based on BPEL, cycles are only allowed in the form of do-while loops, and unstructured cyclic workflows, therefore, have to be transformed into this form first. This is possible because the region-growing rules, in contrast to the reduction rules in [4], [5], and [8], do not modify the original workflow, but create an overlay structure.

The region-growing rules introduced for constructing the RT can still be used as reduction rules, leading to a concept of reducibility similar to the one introduced in [8]. In the main theorems, we proved that this property, called region reducibility, is equivalent to the property of semantical soundness, and that, therefore, no additional rules are needed. As a consequence, an algorithm to detect whether a workflow is region reducible can also be used to determine whether it is semantically sound.

This paper concentrated on the demonstration of the concept of the RT, on its application to structurally and semantically sound workflows, on the definition of the region-growing rules, on their applicability as reduction rules, and on the proof of the main theorems. Future work includes extending the applicability of the RT to other areas, and the definition of further rules and concepts to handle structurally sound workflows that require synchronized conditions and other more general forms of m-out-of-n logic, e.g., interfaces expressible with pins in UML2 Activity Diagrams.

References

[1] OMG (Object Management Group), "OMG Model Driven Architecture" (MDA) [Online]. Available: http://www.omg.org/mda/

[2] S. Kent, "Model Driven Engineering," in *Proc. 3rd Int. Conf. Integr. Formal Methods (IFM'02)*, Turku, Finland, Lecture Notes in Computer Science, vol. 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," presented at the 5th Int. Workshop Graph Transformation Visual Model. Tech. (GT-VMT'06), Vienna, Austria, 2006.

[4] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[5] J. Desel and J. Esparza, *Free Choice Petri Nets*. Cambridge, U.K: Cambridge Univ. Press, 1995.

[6] W. M. P. van der Aalst, "Verification of workflow nets," in *Proc. 18th Int. Conf. Appl. Theory Petri Nets (ICATPN'97)*, Toulouse, France, Lecture Notes in Computer Science, vol. 1248, pp. 407–426.

[7] OMG (Object Management Group), "Unified Modeling Language: Superstructure" (UML2 Activity Diagrams), version 2.0. [Online]. Available: http://www.omg.org/docs/formal/05-07-04.pdf

[8] W. Sadiq and M. E. Orlowska, "Analyzing process models using graph reduction techniques," *Inf. Syst.*, vol. 25, no. 2, pp. 117–134, 2000.

[9] H. Lin, Z. Zhao, H. Li, and Z. Chen, "A novel graph reduction algorithm to identify structural conflicts," in *Proc. 35th Hawaii Int. Conf. Syst. Sci. (HICSS-35)*, 2002, pp. 289–299.

[10] W. M. P. van der Aalst, A. Hirnschall, and H. M. W. Verbeek, "An alternative way to analyze workflow graphs," in *Proc. 14th Int. Conf. Adv. Inf. Syst. Eng. (CAiSE'02)*, Toronto, ON, Canada, Lecture Notes in Computer Science, vol. 2348, pp. 535–552.

[11] IBM (International Business Machines), "WebSphere business modeler" (Modeler), Advanced Version 6.0. [Online]. Available: http://www-306.ibm.com/software/integration/wbimodeler/

[12] Microsoft *et al.*, "Business process execution language for web services" (BPEL4WS), version 1.1, 5 May 2003. [Online]. Available: ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

[13] W. Zhao, R. Hauser, K. Bhattacharya, B. R. Bryant, and F. Cao, "Compiling business processes: Untangling unstructured loops in irreducible flow graphs," *Intl. J. Web Grid Serv.*, vol. 2, no. 1, pp. 68–91, 2006.

[14] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," in *Proc. ACM Sigplan Conf. Program. Language Des. Implementation (PLDI'94)*, Orlando, FL, pp. 171–185.

[15] R. Hauser, M. Friess, J. M. Küster, and J. Vanhatalo, "Combining analysis of unstructured workflows with transformation to structured workflows," in *Proc. 10th Int. Enterprise Distrib. Object Comput. Conf. (EDOC'06)*, Hong Kong, China, pp. 129–140.

[16] R. Liu and A. Kumar, "An analysis and taxonomy of unstructured workflows," in *Proc. 3rd Int. Conf. Bus. Process Manag. (BPM'05)*, Nancy, France, Lecture Notes in Computer Science, vol. 3649, pp. 268–284.

[17] F. Puhlmann and M. Weske, "Investigations on soundness regarding lazy activities," in *Proc. 4th Int. Conf. Bus. Process Manage. (BPM'06)*, Vienna, Austria, Lecture Notes in Computer Science, vol. 4102, pp. 145–160.

[18] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow Patterns," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

[19] B. Kiepuszewski, A. H. M. ter Hofstede, and C. J. Bussler, "On structured workflow modelling," in *Proc. 12th Conf. Adv. Inf. Syst. Eng. (CAiSE'00)*, Stockholm, Sweden, Lecture Notes in Computer Science, vol. 1789, pp. 431–445.

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

[21] M. S. Hecht and J. D. Ullman, "Flow Graph Reducibility," *SIAM J. Comput.*, vol. 1, no. 2, pp. 188–202, 1972.

[22] Z. Ammarguellat, "A control-flow normalization algorithm and its complexity," *IEEE Trans. Softw. Eng.*, vol. 18, no. 3, pp. 237–251, 1992.

[23] R. Hauser and J. Koehler, "Compiling process graphs into executable code," in *Proc. 3rd Int. Conf. Generative Program. Compon. Eng. (GPCE'04)*, Vancouver, Canada, Lecture Notes in Computer Science, vol. 3286, pp. 317–336.

[24] R. Hauser, "Transforming unstructured cycles to structured cycles in sequential flow graphs," IBM Res., Rep. RZ 3624, 2005.

Rainer Friedrich Hauser was born in Zurich, Switzerland. He received the Diploma in mathematics in 1977 and the Ph.D. degree in computer science in 1984 both from the Swiss Federal Institute of Technology (ETH), Zurich.

Since 1980, he has been with IBM Zurich Research Laboratory, Zurich, where he was engaged in research on communication systems and the Open Systems Interconnection protocol. He is currently a member of a systems management research team in the Computer Science Department. He has filed a number of patents.

Dr. Hauser is a member of IBM's Quarter Century Club.

Michael Friess was born in Kehl, Germany. He received the Diplom-Ingenieur (B.A.) degree in computer sciences from the University of Cooperative Education Stuttgart, Stuttgart, Germany, in 1989.

Since 1989, he has been with the IBM Research and Development Laboratory, Böblingen, Germany, earlier as a Software Developer, and currently, as an Architect. During 1995, he was a Technical Specialist at IBM Germany, where he was engaged in research on portable software development, application development tools, and platform interoperability especially for SAP and IBM technologies. From 2002 to 2004, he was with the IBM Hursley Software Development Laboratory, U.K., as an Architect in the business integration domain and led research projects with a focus on model-driven development of business processes and services and interoperability. He has filed a number of patents. His current research interests include the choreography component of the WebSphere Process Server, focusing on client and monitoring.

Jussi Vanhatalo was born in Espoo, Finland. He received the M.Sc.(Tech.) degree from the Helsinki University of Technology, Espoo, and the second Masters' degree in research (DEA) from the University of Nice, Sophia Antipolis, France, in 2004.

During 2000–2004, he was with the Nice-business Solutions, Finland, where he worked on software projects for Nokia. He was with the Eurecom Institute, Sophia Antipolis, as an Exchange Student. Since 2004, he has been with IBM Zurich Research Laboratory, Zurich, Switzerland. His current research interests include service-oriented architectures, business process management, and business-driven development.

Jochen Malte Küster was born in Bielefeld, Germany. He received the Diplom-Informatiker and Doctoral degrees in computer science with mathematics from the University of Paderborn, Paderborn, Germany, in 2000, and the Doctoral degree in 2004.

Since 2004, he has been with IBM Zurich Research Laboratory, Zurich, Switzerland, where he is part of the Business Integration Technologies Group. His current research interests include consistency management, model transformations, business process modeling, and model-driven development of service-oriented applications.

Dr. Küster received a faculty award for his Ph.D. thesis from the University of Paderborn.