Automatic transformation from graphical process models to executable code

Automatic Transformation from Graphical Process Models to Executable Code

Rainer Hauser, Thalwil, Switzerland
(rainer.hauser@gmail.com)

May 2010

Summary

Model-driven engineering envisions a paradigm shift from block-oriented, textual descriptions of behavior to graph-oriented, visual descriptions similar in significance to the paradigm shift from low-level assembler languages to high-level programming languages. One challenging question in this area is how to transform a behavioral model of a system from a graph-oriented language (e.g., UML Activity Diagrams or BPMN) into a block-oriented language (e.g., BPEL) or vice versa. Part of this problem is the transformation from unstructured (usually visual) process models to structured (executable) program code to which this paper proposes a solution that can handle all sound sequential, parallel and mixed regions. It has been implemented as an extension of the IBM WebSphere Business Modeler and applied to many business processes including dozens from the IBM reference model for insurance companies called Insurance Application Architecture.

Key words: Process modeling; control flow; model transformation; UML; BPMN; BPEL

1 Introduction

Complexity of software systems is already now a challenging problem and will become even more challenging in the future. Maintaining the application infrastructure of an enterprise and keeping track of its changes is a rather difficult task because of its sheer size. Larger scale systems require higher-level abstractions [1], and research in the area of model-driven engineering thus searches for solutions to control complexity through abstraction [2].

With placing the focus differently, the basic idea of model-driven engineering can be interpreted in various ways including the following:

1. Model-driven architecture (MDA) as defined by OMG [3] sees the behavior of a system modeled on two levels in such a way that the platform-independent model (PIM) abstracts some details away from the platform-specific model (PSM). The step from PIM to PSM is supposed to be performed, at least partially, through automatic model transformations.

2. A second, related view is based on the so-called business/IT gap [4], i.e., the fact that business and technical analysts speak a different language, use different models and are concerned with different levels of detail. The assumption is that the business analyst draws boxes with arrows outlining the behavior while the IT professional completes the design and implements it in a conventional programming language.

3. A third view is based on the analogy that sees the paradigm shift – using the terminology suggested in [5] – from block-oriented, textual languages to graph-oriented, visual languages as a similarly dramatic step as the one from low-level assembler languages to high-level programming languages. The graphical models are assumed to completely describe the behavior, and therefore can be either directly executed or automatically transformed (compiled) into executable code.

All three views see a graphical model GM and another, mostly textual model TM that are supposed to describe the same behavior, possibly on different levels of abstraction, and are confronted with the challenging question how to guarantee that both models are consistent with each other. In view (1), the transformation from PIM to PSM generates a skeleton solution that has to be completed manually. This leads to a reconciliation problem, because the PSM, when the PIM changes, has to be either recreated from scratch (and thus all manual modifications get lost) or synchronized with its previous version [6]. In view (2), the GM is sometimes only drawn manually on paper and used informally, and the TM is maintained independently. However, a GM is often required for documentation purposes and should therefore be accurate and reconciled with the TM. Finally, the TM is automatically created from the GM in view (3). Therefore, transformations to deduce the TM from the GM are either needed or would be useful in all three views.

The graph-oriented model GM used for describing the behavior of a system on one side is usually a graph with nodes and arrows where the nodes represent actions and the arrows continuations (i.e., temporal dependencies between these actions). Actions are therefore either control actions (i.e., decision and merge nodes for sequential or fork and join for parallel parts) or activities indicating single, atomic (i.e., not further structured) tasks. Arrows are a kind of goto-instructions and specify, together with the control actions, the control flow of the modeled system. The block-oriented model TM used for describing the executable side of a system on the other hand is a specification in a structured and usually imperative programming language (i.e., with conditional and repetitive constructs but no goto-statements). Thus, one of the main differences between the two approaches is that graph-oriented models are unstructured and block-oriented models are structured.

The transformation from a graph-oriented model GM into a block-oriented model TM is therefore not mainly a problem of transforming a graphical to a textual model, but of transforming from an unstructured form with goto-instructions to a form that can be represented by a structured, imperative programming language. It is closely related to the problem of goto-elimination. A repetitive group of activities in the graph-oriented model, for example, is simply a cycle in terms of graph theory, but has to be transformed into a structured while-loop (or a similar loop-construct) in the block-oriented model.

As will be discussed in the next section, transformations between the GM and the TM have been studied either for abstract and artificial classes of modeling languages or for concrete, well-known and widely used modeling languages such as Activity Diagrams in UML [7] and the Business Process Modeling Notation (BPMN) [8] on the graph-oriented side and the Business Process Execution Language (BPEL) [9] on the block-oriented side. The solution presented in the following is based on abstract languages defined such that the results can be applied to UML Activity Diagrams, BPMN and BPEL but does not get lost in the idiosyncrasies of these languages. It concentrates on the transformation from an unstructured, graph-oriented to a structured, block-oriented model.

This paper is structured as follows: The background and related work is presented in Section 2. The block- and graph-oriented languages used to specify behavior are introduced in Section 3, and the transformation between them is discussed, together with its implementation and application, in Section 4. Finally, Section 5 concludes and outlines future work.

2 Background and Related Work

General issues and known approaches to transform unstructured models into structured form are discussed.

2.1 Background

Imperative, block-oriented, structured languages such as COBOL, PL/1 or C have been used for several decades to describe the behavior of systems through programs. The structural elements needed to specify a program written in such a language are very simple. There are sequences of actions, there are conditional blocks written in form of if-then-else-statements or generalizations to more than two branches, and there are various forms of loops all representable as while-loops. The goto-statement however has been banned. Important as programming concepts, but of little significance for the discussion here is that code can further be partitioned into subroutines and that recursion can be used instead of loops for repetitive behavior.

Already in the early days of programming, flow-charts have been introduced as a graphical form for representing behavior. Algorithms have often been designed this way and were later transformed manually into program code. The flow-chart can therefore be seen as the starting point for today's graph-oriented languages intended to represent workflows and business processes, but has been extended – as some of the imperative, block-oriented languages – to allow expressing concurrency. While the basic expressiveness of the imperative, block-oriented languages has stayed simple and stable for decades, and not many experts in the field argue anymore in favor of goto-statements, the expressiveness of the graph-oriented languages for workflows and business processes is still debated. The well-known workflow patterns [10] give a good overview of what is currently considered necessary and/or useful. BPMN, UML Activity Diagrams and BPEL have been examined in their light [11].

A program written in a high-level programming language can either be interpreted directly or compiled into executable code in order to run it. Similarly, a process model specified in a graph-oriented language can either be interpreted (e.g., as executable UML [12]) or transformed into another form for its execution. We look here only at the transformation approach and concentrate on the transformation into a more structured form.

Depending on the context, however, the term *structured* can mean different things. For block-oriented programming languages, it stands for the absence of goto-statements. A short note by Dijkstra [13] started a long and lively discussion about structured programming and triggered research leading to several structured programming methodologies including [14]. For graph-oriented behavioral models, the situation is less clear. Certain features of flow-charts and the lack of others (including the unstructured cycles and the missing constructs for structured loops) led to a search for other graphical representations of behavior better suited for structured programs [15]. As flow-charts not only remained in use but also new and related graphical modeling languages for workflows and business processes came into existence, refined definitions of what properties make a model structured were needed. A rigorous notion has been defined in [16] where a workflow is considered structured when there is a one-to-one relationship between decision nodes and corresponding merge nodes on one side and between fork nodes and corresponding join nodes on the other. Unlike the same term in the field of block-oriented languages, it is not intended to specify what good programming practices are, but is useful as an abstract concept when discussing business process models and their properties. The really important question for a graph-oriented process model, however, is not whether it is structured, but whether it is *sound*, and there are various, although closely related definitions of soundness such as the one in [17].

2.2 Related Work

There is a lot of work going on in the area of transformation between different behavioral modeling languages ranging from simple mappings between two models with very similar structure to complex transformations between models with fundamentally different structures. Simple mappings such as [18, 19] are not within the scope of this paper and will therefore not be discussed further. This includes also partial mappings such as [20], where only those elements in the two modeling languages are mapped that are very similar while all the other elements are ignored. Although they present quite a promising approach, we leave ontology-based transformation methods such as [21] on the side as a direction more or less orthogonal to our intent.

Petri-nets have been used to model asynchronous concurrent systems for quite some time. With finite instead of instantaneous transition time they can be used to modularly build complex systems from simpler ones by inserting well-formed blocks [22]. This method of stepwise refinement guarantees certain soundness properties of the generated Petri-nets. Until today, the fundamental work around workflows has been dominated by Petri-nets as the basic modeling approach [23]. As valuable as they are for mathematically trained researchers, the business analysts never accepted them for modeling their business processes. Thus, even the teams well-known for their research related to Petri-nets moved in the past few years partially to other modeling approaches. UML Activity Diagrams [7] have gained some interest, but more so did BPMN [8]. The differences between the two modeling languages is, however, not very significant with respect to the main transformation problems. Thus, research concentrated on the basic modeling capabilities of both languages under the name *standard workflow models* [24] or *standard process models* [25]. The main characteristic of these modeling languages is their unstructured continuations in the control flow, i.e., the fact that they are based on the concept of directed graphs.

Transformation from structured languages such as BPEL to unstructured languages such as BPMN is possible [5]. Therefore, business processes could be implemented in BPEL and automatically transformed to BPMN or UML Activity Diagrams for documentation purposes. Although the transformation in this direction is undoubtedly valuable in certain circumstances, we will only consider the opposite direction, because one of the main reasons for model-driven engineering is reduction of complexity for which – according to the hope behind the model-driven dream – graphical models (with tools to zoom in and out) are better suited.

A simple, incomplete and partially manual transformation from BPMN to BPEL is sketched on an example in [26]. Based on the three requirements completeness, automation and readability, a more serious and more sophisticated transformation algorithm between the same two languages is described in [27]. It splits a BPMN process into well-structured and non-well-structured components. The well-structured components, corresponding to parts of the model that are structured as defined in [16] rather restrictively, are transformed into the corresponding structured elements in BPEL while the non-well-structured, i.e., the remaining and therefore more complex components, are covered by the event handler construct in BPEL in a similar way as in [25]. This pattern-based approach, still closely tailored to BPEL, has been further refined with respect to how the well-structured patterns are identified and transformed [28]. This direction of research tries to use – or sometimes even misuse – specific features of BPEL, experiments with different transformation strategies [5], and although the algorithm described in the very recent publication [29] intends to produce readable BPEL automatically and indeed overcomes limitations of earlier attempts, it cannot translate every workflow automatically, but needs sometimes manual transformations that can be stored into a component library for later reuse by the algorithm. In [30], the latest paper known to us related to BPMN-to-BPEL transformation, the approach of [27] is taken again and extended to using three different approaches, such that well-structured components are mapped directly, non-well-structured but acyclic components are handled using BPEL links, and non-well-structured and cyclic components are again resolved using the BPEL event handler construct.

The main difficulty of the transformation is the fact that BPEL does not support unstructured cycles while graph-oriented process model languages are based on them and usually model repetitive behavior this way. This problem and further challenges of the transformation are discussed in [31, 32, 33, 34]. Because the fact that the arbitrary cycles in graphical models present the main difficulty when transforming such models, they were the starting point of our research. The primary goal was not to transform graph-oriented modeling languages to BPEL specifically, but to structured block-oriented languages in general. Thus, using constructs specific to BPEL such as the event handler was not our intent. We used – similar to the standard workflow models or process models – a kind of standard block-oriented, imperative programming model that supports sequences, conditionals and loops. The problems to be solved turned out to be very similar to those of goto-elimination in compiler theory several decades ago [35].

Our aim was to resolve arbitrary cycles (including overlapping cycles) as allowed in languages such as BPMN into structured constructs required by languages such as BPEL. Compiler theory helped to solve the case for sequential, reducible program structures [36] with reducibility as defined in [37]. The solution has been extended in [38] to allow the transformation of any sound process model with soundness defined as in Petri-net theory. We also examined a rather different approach based on the correspondence between a finite automaton and a regular expression [39]. The original business process is transformed into a finite automaton F, F is transformed into a regular expression R in the language REL, and R is finally transformed into BPEL code. Because business process languages can express behavior that is not representable with regular expressions, REL has been extended [40].

If we assume there is a program that detects errors in process models and transforms those that are sound fully automatically into correct and directly deployable BPEL processes, readability of the result is no longer an issue and efficiency becomes the more prominent feature. However, as long as some manual adjustments are still needed, readability seems to be an important property of such a transformation program although it has to be clarified what readability exactly means. For well-structured components, it is obvious what we might call good readability. For arbitrarily nested cycles, however, it is no longer as clear because there is no “natural” loop-structure as we will see below.

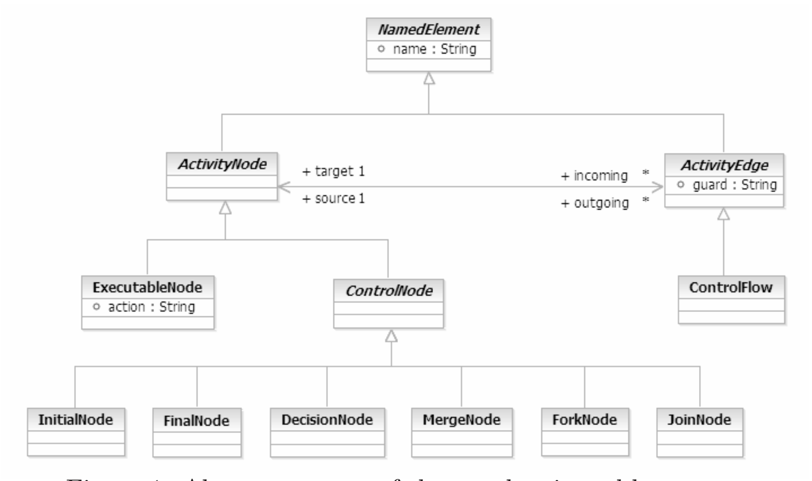


Figure 1: Abstract syntax of the graph-oriented language.

2.3 Positioning of the Paper

This paper is based on the conference paper [36] where we discussed a solution for transforming sequential, reducible graph-oriented models into block-oriented models. A first implementation of this approach transformed models specified with the IBM WebSphere Business Modeler [41] (based on a variant of the UML Activity Diagrams) and used the activity type `StructuredActivityNode` to encapsulate the parts of the model already transformed. Using an invisible overlay structure of regions instead led to a second implementation that was much leaner and could also handle irreducible models and concurrency. In the resulting conference paper [38], we explored the theoretical aspects of these invisible regions and combined analysis for structural errors and transformation of sound models into a more structured form. The transformation itself, however, had only been sketched. Therefore, the main purposes of the paper presented here are firstly to give a more detailed description for the complete transformation of sound business processes including irreducibility and concurrency, but secondly to describe it also in easily understandable terms and thus leaving away issues related to regional and structural analysis.

3 Two Modeling Languages

The two languages used in the following to specify behavior in the graph-oriented and the block-oriented paradigm, respectively, are introduced.

3.1 Graph-Oriented Language

Figure 1 shows the abstract syntax used in [36] as the subset of UML Activity Diagrams needed for the graph-oriented modeling language. The model of a business process consists of nodes and edges. An `ActivityNode`, i.e., a node, is either an `ExecutableNode` or a `ControlNode`, and an `ActivityEdge`, i.e., an edge, is always a `ControlFlow`.

The `ExecutableNode` represents the atomic tasks to be accomplished by a business process, and the six subclasses of the `ControlNode` together with the single subclass of `ActivityEdge` represent the control flow and thus the temporal (and conditional) relations between the tasks. The nodes `DecisionNode` and `MergeNode` allow describing conditional execution of sequential parts, and the nodes `ForkNode` and `JoinNode` allow describing concurrent execution of parallel parts of the business process. (The node `StructuredActivityNode` for structuring processes has been omitted for simplicity.)

For representing concrete business processes, we use the syntax from [38] shown in Figure 2 where the elements in Figures 2a and 2b are called start and end node and correspond to `InitialNode` and `FinalNode` of the abstract syntax, respectively. The element in Figure 2c is called activity (or basic activity) and corresponds to the `ExecutableNode` of the abstract syntax. The remaining elements in

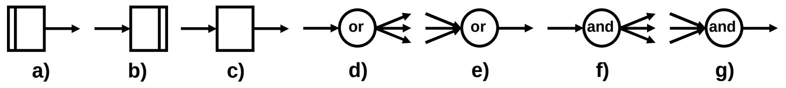


Figure 2: Elements of the concrete graph-oriented language.

Figures 2d, 2e, 2f and 2g are called decision, merge, fork and join and correspond to `DecisionNode`, `MergeNode`, `ForkNode` and `JoinNode` of the abstract syntax, respectively. The elements decision and merge are also called or-split and -join, and the elements fork and join are also called and-split and -join. An arrow corresponds to `ActivityEdge` in the abstract model, and its *guard* specifies the condition within a `DecisionNode`.

Using these elements, concrete business processes can be built. We assume further well-formedness (also called structural soundness) constraints:

1. There is exactly one start node, i.e., a node with no incoming and one outgoing edge.
2. There is exactly one end node, i.e., a node with one incoming and no outgoing edge.
3. There is a path from the start node to every node and a path from every node to the end node.
4. The guards for all edges are *true* except for the outgoing edges of a decision. If n edges leave a decision with $expr_1, \ldots, expr_n$ as guards, then the following two conditions must hold:

$expr_1 \vee expr_2 \vee \ldots \vee expr_n = true$	complete	(PM1)
$expr_i \wedge expr_j = false$ (for $i \neq j$)	deterministic	(PM2)

Figure 3 is an example of a business process. It contains cycles, even overlapping cycles, and shows sequential, conditional logic as well as concurrency. The conditions, i.e., guards, of the or-splits are not shown. The lower two threads are not independent because activity L does not only have to wait for activity K, but also for activity I. Activity I can only start when activity F has completed and activity L can therefore only start when activity F has completed. This makes the execution of the two threads dependent.

3.2 Block-Oriented Language

Figure 4 shows the abstract syntax for BPEL used in [36] extended with the class `Link`. The model of execution is based on the `Process` pointing to the initial `ExecutionElement`. The `Sequence` contains code to be executed sequentially, and the `Flow` contains code to be executed in parallel. The `Invoke` executes a task, i.e., a not further specified activity. The element `Switch` with the `Case` and the element `While` allow conditional and repetitive execution, respectively. The element `Assign` makes it possible to modify a `Variable` used, for example, in the *condition* of `Case` and `While` elements. The `Link` allows specifying more complex synchronization patterns than independent treads in a `Flow`. If, for example, a `Sequence` S is supposed to complete before the task `Invoke` T can start, a `Link` with *source* $= S$ and *target* $= T$ is added to the *links* of the `Flow`.

We use a BPEL-like, but less verbose language to represent a concrete model in the block-oriented language. Figure 5 shows an example that is the possible result of a transformation from the example

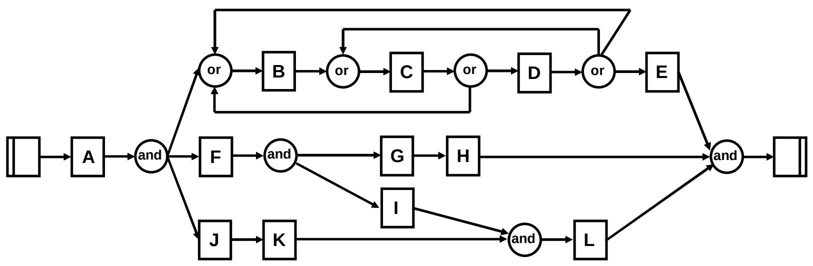


Figure 3: Example of a process in the concrete graph-oriented language.

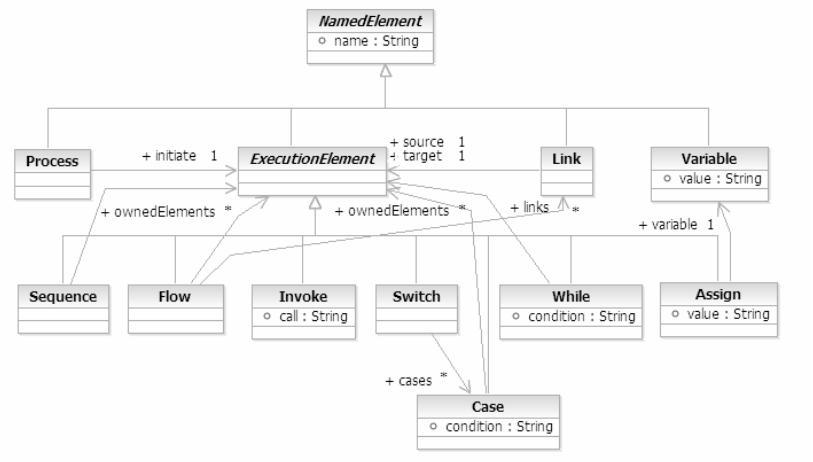


Figure 4: Abstract syntax of the block-oriented language.

in Figure 3 into the block-oriented modeling language. All elements could have an attribute *name*, but we only assigned them partially, for example, where needed to refer to an element in an `assign` or a `link` element. As a convention, we use the name of the task for the corresponding `invoke` element and therefore assume that they are unique. The order of the owned elements of a `flow` is irrelevant, and temporal constraints are modeled using `link` and `sequence` elements. In Figure 5, the fact that task L can only start when task I has completed is represented through the `link` with I as source and L as target. The fact that task K can only start when task J has completed is however represented as a `sequence`. Instead of a `switch` element with a set of `case` elements as in BPEL 1.1, we use an `if` element as in BPEL 2.0, but without `elseif` and `else` that are not needed because of (PM1) and (PM2). Instead of a `while` we use a `repeat` element that tests the condition at the end and thus executes its content at least once. The `variable name=next[S1]` (with scope $S1$) is used for the conditions in the `repeat` and `if` elements and is set based on boolean values such as CB, i.e., the condition that the execution proceeds from task C to B. To improve readability, the conditions for the `repeat` and `if` elements as well as the values for the `assign` elements are written in a way that is not correct XML.

```
<process>
 <sequence>
  <invoke name=A />
  <flow>
   <link source=F target=S2 />
   <link source=F target=I />
   <link source=S3 target=L />
   <link source=I target=L />
   <sequence>
    <sequence name=S1>
     <variable name=next[S1] />
     <assign next[S1]:=B />
     <repeat>
      <if next[S1]=B>
       <invoke name=B />
       <assign next[S1]:=C />
      </if>
      <invoke name=C />
      <if CB>
       <assign next[S1]:=B />
      </if>
      <if CD>
       <assign next[S1]:=D />
      </if>
      <if next[S1]=D>
       <invoke name=D />
       <if DB>
        <assign next[S1]:=B />
       </if>
       <if DC>
        <assign next[S1]:=C />
       </if>
       <if DE>
        <assign next[S1]:=E />
       </if>
      </if>
     </repeat next[S1]!=E>
     <invoke name=E />
    </sequence>
   </sequence>
   <sequence name=S2>
    <invoke name=G />
    <invoke name=H />
   </sequence>
   <sequence name=S3>
    <invoke name=J />
    <invoke name=K />
   </sequence>
   <invoke name=F />
   <invoke name=I />
   <invoke name=L />
  </flow>
 </sequence>
</process>
```

Figure 5: Example of a process in the concrete block-oriented language.

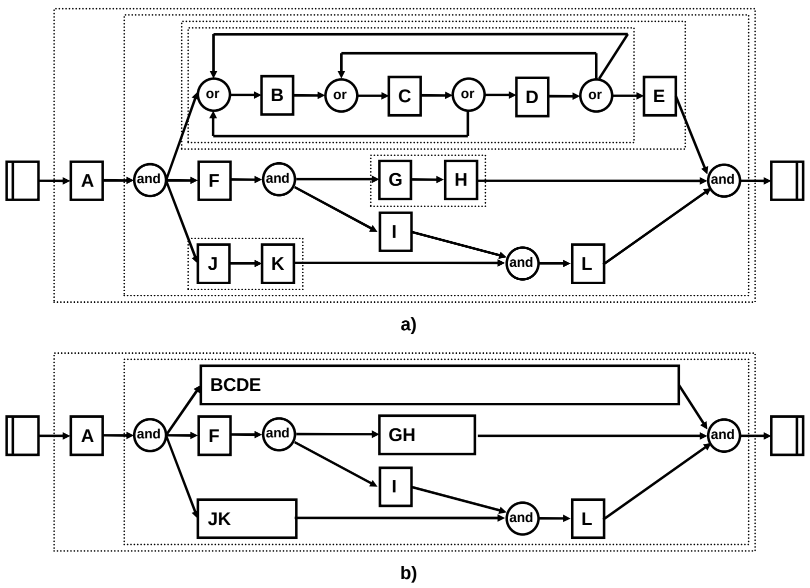


Figure 6: Example with its non-trivial structured activities.

4 Transformation

The transformation from behavior models in the graph-oriented language into models in the block-oriented language is presented.

4.1 Structured Activities

Regions in the graph-oriented model with a single entry and a single exit play an important role. They are called *structured activities* in the following. If the details of a structured activity are hidden within a subprocess, this structured activity can, in all respects relevant for the transformation to the block-oriented language, be treated the same way as a basic activity. Especially, an activity (basic or structured) can be transformed independently of the rest of the process resulting in one `ExecutionElement` (possibly with children) of the abstract block-oriented language.

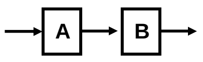
In Figure 6, the structured activities in the model of Figure 3 are high-lighted. Figure 6a shows all structured activities that are not trivial, i.e., consist of at least two `ActivityNode` elements. The structured activity containing activities G and H and the one containing the activities J and K are simple sequences without conditional or repetitive elements and without concurrency. The structured activity containing the or-join before activity B and all the nodes between it and the or-split before activity E can be called sequential because it only contains activities and sequential control nodes, i.e., or-splits and -joins. These three structured activities do not contain other structured activities. The sequential structured activity together with activity E builds also a simple sequence. The remaining two non-trivial structured activities are depicted in Figure 6b with the inner, already described structured activities shown as ordinary activity boxes to emphasize the similarity between basic and structured activities. The inner of these two structured activities can be called parallel because it only contains activities and parallel control nodes, i.e., and-splits and -joins. It builds together with the basic activity A a simple sequence.

The transformation from the graph-oriented model to the block-oriented model starts with the determination of the structured activities. As single-entry-single-exit regions, they can be determined in linear time [42]. The result is a set of nested structured activities that can be categorized as follows:

1. *simple sequences* contain only activities but neither sequential nor parallel control nodes,

2. *sequential regions* contain only sequential control nodes and, optionally, some activities,

3. *parallel regions* contain only parallel control nodes and, optionally, some activities,



```
<sequence name=S>
 <invoke name=A />
 <invoke name=B />
</sequence>
```

Figure 7: Sample transformation of a simple sequence.

4. *mixed regions* contain sequential as well as parallel control nodes and, optionally, some activities.

Although a region with only control nodes makes little sense, it is allowed and the activities in the last three categories are therefore optional. In the following, we further assume that regions of the same category have been merged where it makes sense. Two adjacent simple sequences, for example, can always be combined into one. This is not crucial for the transformation but reduces the level of nesting.

The nested activities are transformed independently from inside out, i.e., such that the innermost activities are transformed first. Basic activities are transformed into `invoke` elements and structured activities are transformed depending on their category. A simple sequence becomes a `sequence`, a sequential region becomes `repeat` and `if` elements packed into a `sequence` for the definition and assignment of the `variable` used in their conditions, a parallel region becomes a `flow`, and a mixed region is converted first into equivalent sequential and parallel regions as will be discussed in the following.

4.2 Transformation of Simple Sequences

A simple sequence consists of two or more activities executed unconditionally one after the other in the given order. The transformation of a simple sequence is rather obvious as shown in Figure 7 for an example of two tasks A and B.

The elements of a simple sequence in the graph-oriented model are assumed to be transformed already into `invoke` elements if corresponding to basic activities or into `sequence` or `flow` elements if corresponding to structured activities. These elements are added in the order within the simple sequence to a new `sequence` element with a name that can be automatically generated. This new `sequence` can be used for the rest of the transformation instead of the activities in the simple sequence.

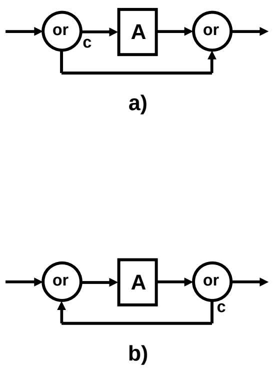
There is no technical reason for treating simple sequences as a special category because they could easily be handled as part of one of the other categories. They would, however, be transformed differently depending on whether they are part of a sequential or part of a parallel region.

4.3 Transformation of Sequential Regions

The transformation of sequential regions is very similar to goto-elimination studied in the context of structured programming [35, 43]. An arrow in a graph-oriented model is a goto in visual form. It can lead either downstream or upstream. Figure 8 shows a possible transformation of an example for both these situations. Figure 8a presents the result for the downstream (conditional) and 8b for the upstream (repetitive) case. (For compactness and better readability, the condition of the `repeat` element is, as mentioned above and as would not be legal in correct XML, written in form of a simple string instead of an attribute and, even more illegal in XML, placed in the end tag.) Real graph-oriented models can contain nested and overlapping cycles and are usually much more complex than these two simple examples and the other equally simple patterns covered as well-structured by [27].

Because all graph-oriented models without parallelism have an equivalent structured form [16], they can be transformed into a block-oriented model. In [36], we have proposed two different solutions for handling *sequential graph-oriented* models:

1. The *finite state machine transformation* compiles every model into a single loop that does not reflect the inherent program structure of the original model. Each basic activity is transformed into an `if` element containing the `invoke` element and some conditional `assign` elements that



```
<if c>
 <assign next:=A />
</if>
<if !c>
 <assign next:=null />
</if>
<if next=A>
 <invoke name=A />
</if>

<repeat>
 <invoke name=A />
 <if c>
  <assign next:=A />
 </if>
 <if !c>
  <assign next:=null />
 </if>
</repeat next=A>
```

Figure 8: Sample transformation of sequential regions.

ensure the correct execution of the `invoke` elements. This is possible due to a theorem often associated with the names Böhm and Jacopini [44] although this is not quite correct [45].

2. The *goto-elimination method* is a rule-based approach that handles models only if they are reducible, because it is based on [35] and thus on the two rules of the T1-T2 analysis [46] as a method to determine reducibility. To resolve irreducibility, node splitting introduced in [47] and optimized in [48] was assumed to be needed.

As shown in [49], reducibility turned out to be less crucial for the second method than has been assumed. Further rules can be introduced to cover irreducibility, and the complete set of rules allows transforming any sequential model. The conditions needed in the `if` and `repeat` elements have to be set accordingly.

The resulting rule-based approach needs an initialization as shown in Figure 9. The activities, e.g., basic activity A in Figure 9a, and the surrounding control nodes are first merged into single nodes with multiple incoming and multiple outgoing edges. (Note that in some cases, empty activities have to be introduced.) The conditions of the or-splits, i.e., the $expr_i$ satisfying (PM1) and (PM2), are the guards of the edges. They are expressions over variables used and modified inside the activities. To keep the logic of the control flow, we introduce a new variable `next[E]` with scope E such as `next[S1]` in Figure 5 (often simply written as `next` if the scope is clear) for the sequential region that is currently transformed. As depicted in Figure 9b, values are assigned to this variable in a second step. The resulting block of code with the code for the activity and the conditional assignments to this variable is referred to as `<element name=A />` in the following.

After the initialization, the edges $M \rightarrow N$ between these elements are eliminated step by step through a set of two rules until only one element with one incoming and one outgoing edge remains. One rule

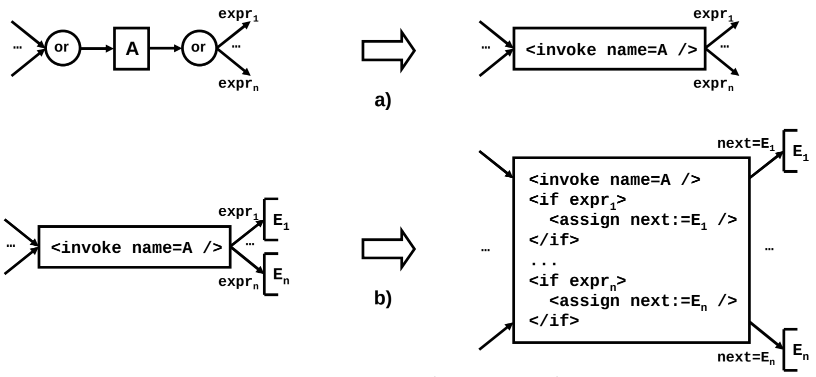


Figure 9: Initialization of the transformation.

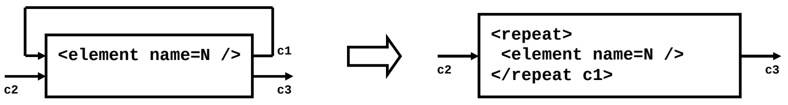


Figure 10: Transformation rule L.

covers self-cycles, i.e., edges with $M = N$. It corresponds to the T1 rule of the T1-T2 analysis. The second rule, actually a family of rules, handles two neighbors, i.e., edges with $M \neq N$, and merges the two elements M and N. It corresponds to the T2 rule and extends it to handle also irreducibility. The rules are pattern-based, i.e., if a match is found for the pattern on the left side of the rule the modifications on the right side are applied.

The first rule is called rule L and is shown in Figure 10. The element on the left side with the self-cycle and with code `<element name=N />` assigned is replaced by the element on the right side without self-cycle and with the `<repeat>` block assigned as the new `<element name=N' />`. The conditions `c1`, `c2` and `c3` are expressions over atomic terms of the form `next=Ei`. Note that rule L is responsible for eliminating cycles in a sequential region and is not needed if a sequential region is acyclic.

The second rule is shown in Figure 11 as a set of four subrules that are called C_{st}, C_s, C_t and C and cover the possible cases that the source M of the edge has or has not successors other than N and the target N of the edge has or has not predecessors other than M. Figure 11a shows the case where M has only N as successor and N has only M as predecessor. This subrule is called C_{st}. Figure 11b shows the case where N has predecessors other than M. This subrule is called C_s. Figure 11c shows the case where M has successors other than N. This subrule is called C_t and corresponds together with C_{st} to the T2 rule of the T1-T2 analysis. Figure 11d shows the most general case where M has successors other than N, and N has predecessors other than M. Subrule C alone would be sufficient to cover all cases because the other subrules are just special cases where certain conditions are false. The two elements on the left side with code `<element name=M />` and `<element name=N />` assigned are replaced by the element on the right side with the `sequence` block assigned as the new `<element name=N' />`. Also here the conditions `c1` to `c9` are expressions over atomic terms of the form `next=Ei`. Note that rule C_{st}, equivalent to the transformation of simple sequences, is needed because other rules may introduce new simple sequences.

If more than one rule is applicable, a strategy is needed that determines the next rule to be applied. The subrules of the second rule suggest a natural strategy. Rules C is only used when neither rule C_s nor

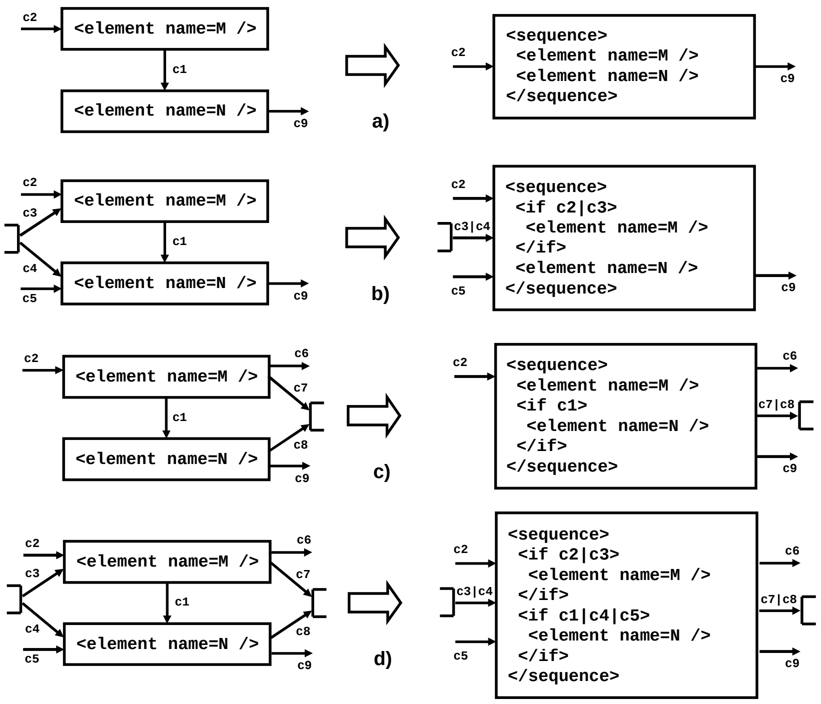
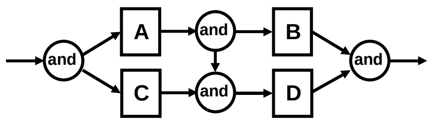


Figure 11: Transformation rules C.



```
<flow name=F>
 <link source=A target=B />
 <link source=A target=D />
 <link source=C target=D />
 <invoke name=A />
 <invoke name=B />
 <invoke name=C />
 <invoke name=D />
</flow>
```

Figure 12: Sample transformation of a parallel region.

rule C_t can be applied. Rules C_s and C_t are only used if rule C_{st} cannot be applied. Whether rule L has higher priority than the subrules of rule C is, however, a matter of taste. If rule L has high priority, many `repeat` elements are introduced. If it has low priority, multiple loops may be combined into a single `repeat` element. If it has lowest priority, cyclic models result in a single `repeat` element [49]. In this sense, workflows, and therefore also programs, do not have a natural loop-structure. Note also that rule C can introduce pseudo-cycles [38].

4.4 Transformation of Parallel Regions

The handling for parallel regions is very different from the one for sequential regions, because parallel regions cannot always be transformed into equivalent structured form [16]. This is the reason why the `link` element is needed in addition to the `flow` element in BPEL. These links must be acyclic, but there is no need for cyclic links because parallel regions are not allowed to contain cycles [38]. Figure 12 shows a possible transformation result of a parallel model. The ordering of the `invoke` statements within the `flow` is completely irrelevant, because all execution dependencies are expressed using `link` elements.

A parallel region in a graph-oriented model is acyclic, contains only control nodes that are parallel, and the edges represent timing constraints. A path from activity M to activity N means that N can only start after M has terminated, and if there is no path in the graph model from M to N, the two activities can run independently. This property can be expressed as a partial ordering relation $\prec$ with $A \prec B$ if and only if there is a path from A to B with non-zero length. This relation is a partial ordering: (1) $A \not\prec A$ (irreflexivity), (2) $A \prec B \rightarrow B \not\prec A$ (asymmetry), and (3) $A \prec B \wedge B \prec C \rightarrow A \prec C$ (transitivity). The lack of cycles guarantees irreflexivity and asymmetry.

The transformation of a parallel region is therefore straightforward. We create a new `flow` element, add the necessary `link` elements to it, and add the corresponding entries `<element name=N />` for all activities N in the region. Because of the transitivity, we only need a `<link source=M target=N />` for $M \prec N$ if there is no activity X such that $M \prec X$ and $X \prec N$.

4.5 Transformation of Mixed Regions

In most graphical models, the sequential and parallel regions are separated and can be transformed independently as discussed in the previous subsections. As shown in [50], there is, however, a rarely occurring pattern, called the *overlapped pattern*, where this is not the case. Figure 13 shows it in its simplest form. In general, m and-splits ($m \geq 2$) and n or-joins ($n \geq 2$) with exactly one path from every

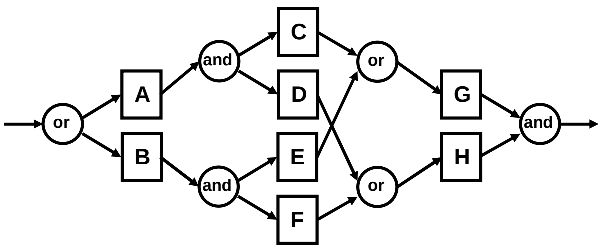


Figure 13: An overlapped pattern.

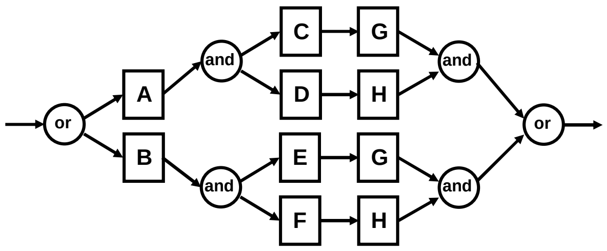


Figure 14: Structured equivalent of overlapped pattern.

and-split to every or-join is possible.

Overlapped patterns can be transformed into an equivalent form where sequential and parallel regions can be separated [51]. By duplicating the activities G and H and by switching the or- and and-joins, the overlapped pattern in Figure 13 is turned into the form shown in Figure 14 where sequential and parallel regions are no longer mixed. The new model with the duplicated activities contains two parallel structured activities inside a sequential structured activity. Nested overlapped patterns are a bit trickier, because the edges leading to A and B may come from different or-splits, and/or the edges from G and H may lead to different and-joins [38].

The overlapped pattern with m and-splits and n or-joins is one possible situation where or- and and-logic are mixed in such a way that a model, without restructuring, cannot be decomposed into structured activities with either only sequential or only parallel control nodes. If there is one such situation, the question arises whether there are other, similar situations where or- and and-logic are mixed. This is, however, not the case and the only mixed patterns are the overlapped patterns [52]. For the details, on how they can be resolved even if they are combined in such a way that overlapped patterns build the nested parts of other overlapped patterns, see [53].

4.6 The Algorithm

The algorithm outlined in pseudo-code presented in Figure 15 shows the structure and main steps of the transformation. It gets a *graphicalModel* as input, finds and processes all basic and structured activities, i.e., all single-entry-single-exit regions, transforms them from inside out and returns the resulting *blockModel* as output.

Because the algorithm is only guaranteed to produce correct results if the graphical model is sound, we assume that the input has been validated before the transformation starts. Validation can even be combined with the analysis of the regions and the application of the rules as discussed in [52].

The subroutine *findSingleEntrySingleExitRegion* determines first all basic and structured activities, eliminates secondly mixed regions by duplicating activities, maximizes next all consecutive regions of the same type where desired, and finally builds the containment relation of the remaining regions. Its result is a hierarchy of regions corresponding to basic activities, simple sequences, sequential regions and parallel regions. The method *getNext* applied to *regions* returns the next region in the hierarchy for processing such that all children have been processed before the parent is processed (i.e., depth first).

In the while-loop, a new element with a unique name is allocated for the region. Depending on whether it corresponds to a basic activity, a simple sequence, a sequential region or a parallel region, it is processed differently, but as either shown in the algorithm or in Figures 10 and 11, a piece of code is always determined and assigned to the element based on the code of its children (if there are children). The `code` is an `invoke` for a basic activity, a `sequence` for a simple sequence or a sequential region, and a `flow` for a parallel region. The last element determined for the outermost region is associated with the complete graph, and its `code` is therefore the model in the block-oriented language, i.e., the result of the transformation.

The generation of the `code` for an element corresponding to a structured activity from the `code` of its children, i.e., the subelements, is explicitly shown for simple sequences and parallel regions. For sequential regions, most of it is, however, hidden in the method *rule.update.apply* and is therefore demonstrated on an example and discussed in detail in the following.

```
procedure $Transform(graphModel)$ returns $blockModel$:

  $regions \leftarrow findSingleEntrySingleExitRegions(graphicalModel)$
  $element \leftarrow null$
  while $region \leftarrow regions.getNext()$ do
    $element \leftarrow allocate(region)$
    $name \leftarrow element.getName()$
    if $element \in BasicActivities$ then
      $element.code.add($<invoke name=$name$ />$)$
    end if
    if $element \in SimpleSequences$ then
      $element.code.add($<sequence name=$name$>$)$
      for all $subelement \leftarrow element.getNextSubelement()$ do
        $element.code.addAll(subelement.code)$
      end for
      $element.code.add($</sequence>$)$
    end if
    if $element \in SequentialRegions$ then
      $element.code.add($<sequence name=$name$>$)$
      $element.code.add($<variable name=next[$name$] />$)$
      $applied \leftarrow false$
      for all $rule \in strategy$ while $\neg applied$ do
        for all $edge \in element.edges$ while $\neg applied$ do
          if $rule.pattern.match(edge)$ then
            $rule.update.apply(edge, element)$
            $applied \leftarrow true$
          end if
        end for
      end for
      $element.code.add($</sequence>$)$
    end if
    if $element \in ParallelRegions$ then
      $element.code.add($<flow name=$name$>$)$
      for all $(source, target) \leftarrow element.getNextDependency()$ do
        $(name_1, name_2) \leftarrow (source, target).getName()$
        $element.code.add($<link source=$name_1$ target=$name_2$ />$)$
      end for
      for all $subelement \leftarrow element.getNextSubelement()$ do
        $element.code.addAll(subelement.code)$
      end for
      $element.code.add($</flow>$)$
    end if
  end while
  $blockModel \leftarrow element.code$
```

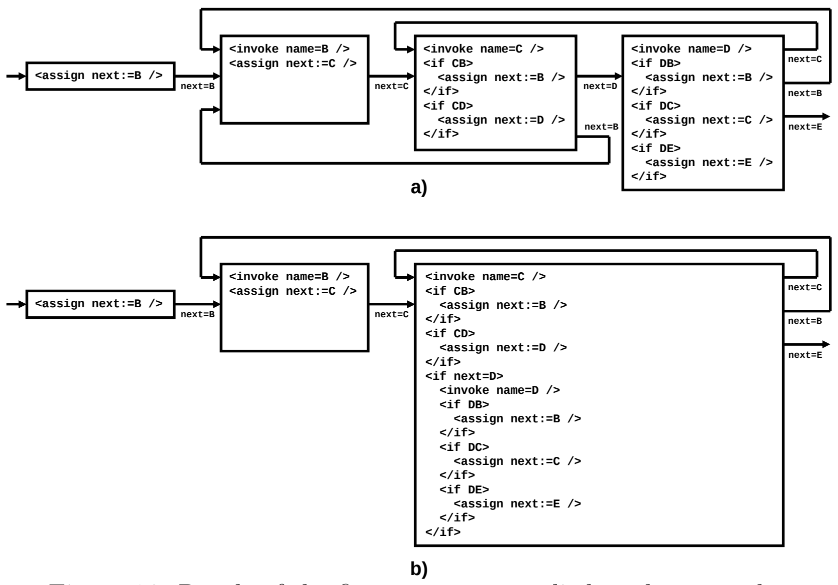
Figure 15: The algorithm.

4.7 Example

The result of the first two steps of the transformation algorithm applied to the sequential structured activity in the sample process of Figure 3 is presented in Figure 16. After initialization and introduction of an empty task for setting the variable `next`, the region contains the four elements shown in Figure 16a. The last two elements corresponding to the original tasks C and D can be combined into one element using rule C_t leading to the situation in Figure 16b. Note that the original edges from C to B and from D to B are merged and become one single edge (as in the T1-T2 analysis). Its guard is `next=B | next=B` or simply `next=B`.

In the next step, the new element corresponding to C merged with D can either be derecursivated, i.e., its self-cycle can be removed using rule L, or it can be merged with the element corresponding to B using rule C_s. We will follow both possibilities because they reveal different interesting properties of the transformation and of the effect of the rule-application strategy.

For the first strategy, we observe that the sample model in Figure 3 is reducible, and that the rules C_s and



a)

```
<assign next:=B />
```

```
<invoke name=B />
<assign next:=C />
```

```
<invoke name=C />
<if CB>
  <assign next:=B />
</if>
<if CD>
  <assign next:=D />
</if>
```

```
<invoke name=D />
<if DB>
  <assign next:=B />
</if>
<if DC>
  <assign next:=C />
</if>
<if DE>
  <assign next:=E />
</if>
```

b)

```
<assign next:=B />
```

```
<invoke name=B />
<assign next:=C />
```

```
<invoke name=C />
<if CB>
  <assign next:=B />
</if>
<if CD>
  <assign next:=D />
</if>
<if next=D>
  <invoke name=D />
  <if DB>
    <assign next:=B />
  </if>
  <if DC>
    <assign next:=C />
  </if>
  <if DE>
    <assign next:=E />
  </if>
</if>
```

Figure 16: Result of the first two steps applied to the example.

C are therefore not needed. The intermediate results of a sequence of rule applications starting from the state in Figure 16b is shown in Figure 17. Rule L can be applied leading to the situation in Figure 17a. In a next step, rule C_{st} applied to the element corresponding to B and the merged C and D results in the situation depicted in Figure 17b. The self-cycle can again be resolved using rule L, and we get the two elements in Figure 17c that could trivially be merged using rule C_{st} a second time.

Without the rules C_s and C introduced to resolve irreducible models, the result of the transformation is deterministic in this example because at any point during its execution exactly one rule is applicable, and the three edges leading backwards are always transformed into two structured loops independent of the strategy. This is not always the case. A strategy giving rule L higher priority than rule C_t may turn multiple edges all leading backwards to the same node into multiple `repeat` elements while a strategy where L has lower priority than rule C_t merges these cycles into a single `repeat` element [49].

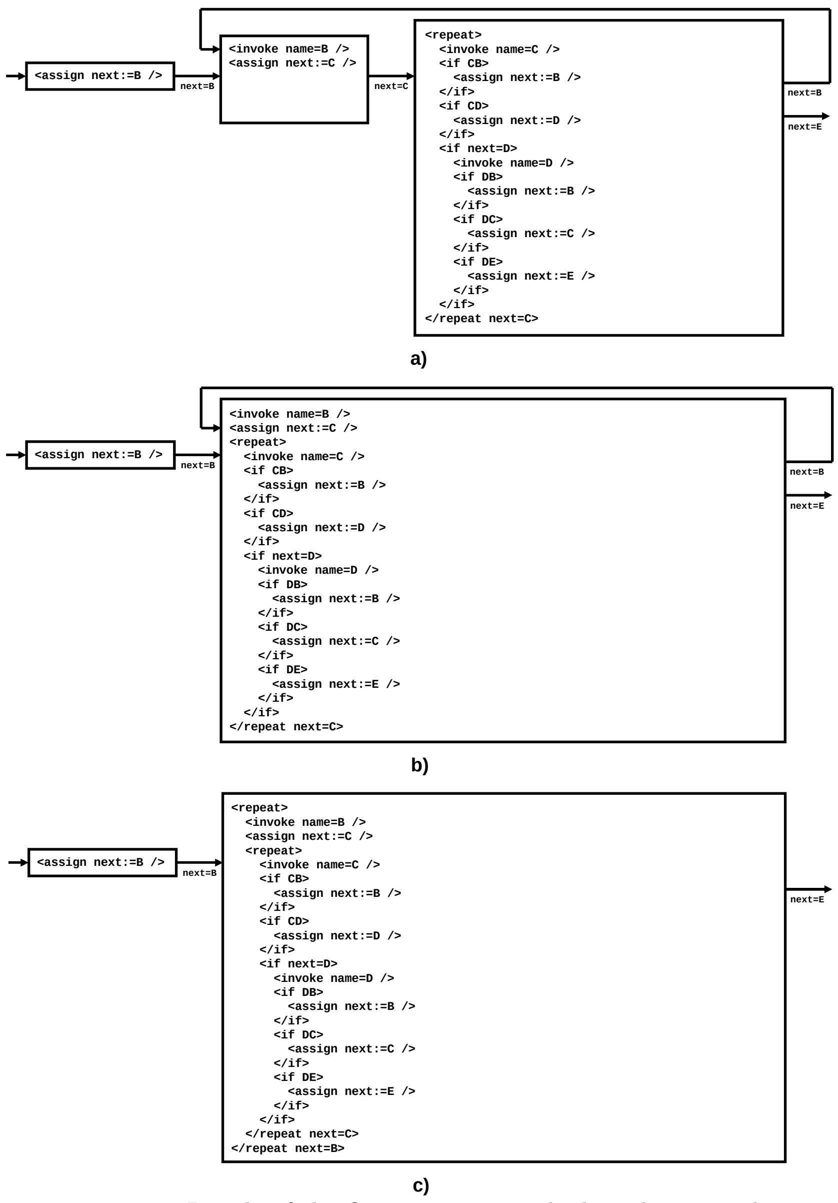
In the second strategy, the rule C_s is allowed and has higher priority than rule L. The elements corresponding to B and the merged C and D in Figure 16b can be merged with rule C_s into a single element, before the existing self-cycle is removed. This strategy is shown in Figure 18. All three edges leading backwards are merged into one single self-cycle as illustrated in Figure 18a. It gets the guard `next=B | next=C`. Because the resulting large element with the self-cycle can now be entered either with `next=B` or with `next=C`, the code corresponding to B becomes conditional. A derecursivation step using rule L removes the self-cycle and leads to the situation shown in Figure 18b. As with the other strategy, applying rule C_{st} would merge the remaining two elements into one and would thus complete the transformation.

Note that without the rules C_s and C, all edges leading to the same element X have the same guard at any point in time during the transformation. It is of the form `next=Y` where Y is the first subelement in X. The code of an element therefore never needs an if-statement at the beginning to conditionally execute an activity. (Reducible processes can be characterized by having single entries into cycles.) Because the second element N has also other predecessors in addition to the first element M in rules C_s and C, this is no longer the case when we allow these two rules.

The code in Figure 5 has been generated using rule C_s. Note also the condition `next[S1]!=E` in the `repeat` element. It is in this example equivalent to `next[S1]=B | next[S1]=C`. In general, if all but one edge leaving an element lead backwards to the same target, i.e., form a single cycle, such an optimization step is allowed.

4.8 General Remarks

The transformation algorithm handles sequential regions iteratively using a rule-based approach, while concurrent regions are transformed in a single step. Both parts of the algorithm assume that the input



a)

```
<assign next:=B />
```

```
<invoke name=B />
<assign next:=C />
```

```
<repeat>
  <invoke name=C />
  <if CB>
    <assign next:=B />
  </if>
  <if CD>
    <assign next:=D />
  </if>
  <if next=D>
    <invoke name=D />
    <if DB>
      <assign next:=B />
    </if>
    <if DC>
      <assign next:=C />
    </if>
    <if DE>
      <assign next:=E />
    </if>
  </if>
</repeat next=C>
```

b)

```
<assign next:=B />
```

```
<invoke name=B />
<assign next:=C />
<repeat>
  <invoke name=C />
  <if CB>
    <assign next:=B />
  </if>
  <if CD>
    <assign next:=D />
  </if>
  <if next=D>
    <invoke name=D />
    <if DB>
      <assign next:=B />
    </if>
    <if DC>
      <assign next:=C />
    </if>
    <if DE>
      <assign next:=E />
    </if>
  </if>
</repeat next=C>
```

c)

```
<assign next:=B />
```

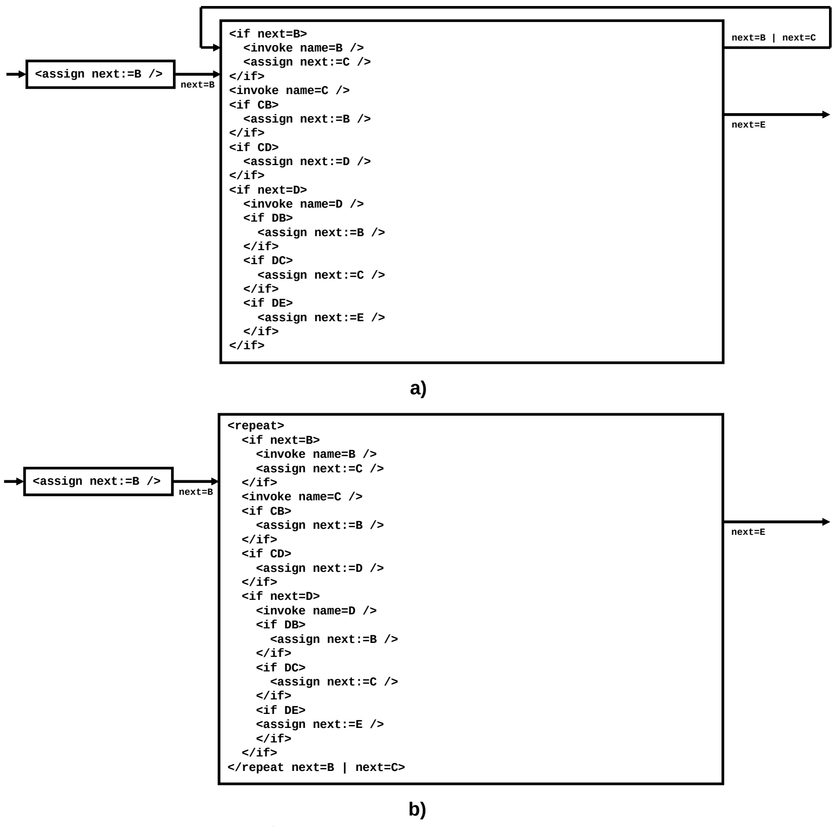
```
<repeat>
  <invoke name=B />
  <assign next:=C />
  <repeat>
    <invoke name=C />
    <if CB>
      <assign next:=B />
    </if>
    <if CD>
      <assign next:=D />
    </if>
    <if next=D>
      <invoke name=D />
      <if DB>
        <assign next:=B />
      </if>
      <if DC>
        <assign next:=C />
      </if>
      <if DE>
        <assign next:=E />
      </if>
    </if>
  </repeat next=C>
</repeat next=B>
```

Figure 17: Result of the first strategy applied to the example.

process model is sound as defined in [38]. An extended version of the algorithm where analysis and transformation are combined into an incremental approach is shown in [52]. The rule-based transformation proposed for sequential regions can be seen as a solution somewhere between the *finite state machine transformation* and the *goto-elimination method* approach discussed, as mentioned before, in [36]. The variable `next` (or `next[E]`) serves as a substitute for the goto-statement similarly to the variable `nextNode` in the finite state machine transformation. However, the set of rules proposed here extracts the program structure inherent in the original graph-oriented model similarly to the goto-elimination method, but can also handle irreducible processes.

In the terminology of [35], the initialization step is called *precalculation*, the effect of rule L is called *derecursivation*, and the effect of the rules C_{st} and C_t is called *substitution and elimination*. Because of the special variable `next` and the way it is used, *if-distribution* is not needed, and *factorization* only occurs when the two rules C_s and C, i.e., the rules used to resolve irreducible processes, are applied.

The transformation can handle all sound process models specified in a graph-oriented modeling language with the syntactic features similar to the ones shown in Figure 2. Completeness for the parallel parts is obvious. Because every edge can be removed from the model by a rule, the transformation for the sequential parts is also complete. See [52] for the semantics of the formalism and for further issues related



```
<assign next:=B />
```

```
<if next=B>
  <invoke name=B />
  <assign next:=C />
</if>
<invoke name=C />
<if CB>
  <assign next:=B />
</if>
<if CD>
  <assign next:=D />
</if>
<if next=D>
  <invoke name=D />
  <if DB>
    <assign next:=B />
  </if>
  <if DC>
    <assign next:=C />
  </if>
  <if DE>
    <assign next:=E />
  </if>
</if>
```

```
<assign next:=B />
```

```
<repeat>
  <if next=B>
    <invoke name=B />
    <assign next:=C />
  </if>
  <invoke name=C />
  <if CB>
    <assign next:=B />
  </if>
  <if CD>
    <assign next:=D />
  </if>
  <if next=D>
    <invoke name=D />
    <if DB>
      <assign next:=B />
    </if>
    <if DC>
      <assign next:=C />
    </if>
    <if DE>
    <assign next:=E />
    </if>
  </if>
</repeat next=B | next=C>
```

Figure 18: Result of the second strategy applied to the example.

to correctness and confluence in general, and [49] for more details on the influence of the priorities of the rules in particular. For the completeness of the algorithm and one single rule-based approach for all models even in the presence of overlapped patterns, consult [53].

4.9 Implementation and Application

The transformation has been implemented as an extension of the IBM WebSphere Business Modeler [41]. The implementation, however, did not transform to BPEL or another block-oriented language, but to the same graph-oriented language as the input model (a variant of the UML Activity Diagrams). A first version that handled only sequential and reducible models used the activity type `StructuredActivityNode` to simulate the BPEL `Sequence` and the `LoopNode` to simulate the BPEL `While`. The deep nesting that resulted did not lead to easily comprehensible output models. A second version that could handle also irreducible models and models containing concurrency used an overlay of invisible regions instead of the `StructuredActivityNode` elements leading to a much leaner nesting structure of the output models.

There were several reasons why the algorithm has been implemented as a transformation from graph-oriented to graph-oriented models and not from graph-oriented to block-oriented models:

1. The result of the transformation could more easily be analyzed in visual than in textual form.

2. The Modeler allowed not only unstructured cycles but contained also the element `LoopNode` necessary to represent structured cycles.

3. An elaborate transformation from the Modeler to BPEL already existed, that worked for `LoopNodes` but could not handle cyclic models.

4. The *cycle-removal* part of the transformation could be used as one step in a sophisticated refinement methodology leading from *analysis models* to *design models* [54].

For the sequential parts, the algorithmic problems to be solved were more or less the same as for a transformation leading to block-oriented models. For the concurrent parts, the algorithm only needed to identify the regions.

The WebSphere Business Modeler allows the specification of the same or very similar behavior in redundant ways. We therefore defined and implemented various transformations, e.g., to refine the models, to tailor the message flow, and to bring a process model into a specific normal form [55]. These transformations could be invoked individually but could also be chained together to form more complex transformations. The cycle-removal part of the transformation presented in this paper and the available transformation of acyclic models to BPEL were used as steps in a broader model refinement methodology that is described in [54], where also an example of an input model together with the corresponding output model after cycle-removal is shown.

The methodology has been applied to many processes in the IBM Insurance Application Architecture (IAA) [56], a reference model for insurance companies. The cycle-removal part of the transformation from unstructured to structured models helped to identify several logical errors where the cycle-structure turned out to be different from what was intended.

As we were involved – see [57] – in discussions concerning the OMG initiative searching for a language to specify query, views and transformations (QVT) [58], the question arises whether we used any of the proposed QVT languages. The answer is no, because we had to integrate the transformation into the code base of the Modeler written in Java. We built, however, a framework for transformations in general and for rule-based transformations in particular. As we used Java directly, we specified the transformation – although personally more in favor of declarative methods – in a purely imperative way. The framework turned out to be very useful as it was extremely easy to create simple new transformations that could be chained together with existing transformations. The performance of the transformations, including the framework, could be neglected even for large, highly cyclic models. In all these respects, using our own infrastructure was not an issue. For maintainability and in order to reuse the transformation code in other environments, it would have been preferable, however, to use a standardized model transformation language.

5 Conclusions

In this paper, we have introduced an algorithm that transforms unstructured process models in a graph-oriented language such as BPMN into the structured form of a block-oriented language such as BPEL. The transformation can handle all sound process models fully automatically. A graphical model is first split into sequential and parallel regions. This is possible, because mixed regions can be converted into equivalent regions that can be split into sequential and parallel regions, although this conversion step has the disadvantage that it duplicates some of the activities.

The resulting regions are processed from inside out. Sequential regions are transformed using two rules that have been inspired by the T1-T2 analysis known from compiler theory, but extended to cover also irreducibility. Parallel regions are processed by analyzing the paths in the graph and thus by determining the dependencies as a partial ordering relation. (Alternatively, rules similar to the rules for the sequential parts can be introduced also for the parallel parts. This way the whole process model can be handled using one single transformation method.)

Although the graph-oriented language used in this paper is complete in the sense that it can model any behavior, it cannot express all workflow patterns discussed in the literature directly as built-in constructs. Our transformation algorithm can therefore not handle all possible process models in all possible graph-oriented languages without adaptation. Some proposed extensions and more elaborate workflow patterns clearly make sense, and future work is required to extend our approach in order to cover them as well. For other extensions, it is however not beyond all doubt that they are desirable. Remember that PL/1 does not only allow goto-statements jumping to constant labels but even allows variables for labels to make a program completely incomprehensible, and this was considered a valuable feature at some time.

References

[1] Shaw M. Larger Scale Systems Require Higher-Level Abstractions. 5th International Workshop on Software Specification and Design, 1989: 143-146.

[2] Kent S. Model Driven Engineering. 3rd International Conference on Integrated Formal Methods (IFM 2002), May 2002. LNCS 2335: 286-298. DOI: 10.1007/3-540-47884-1.

[3] OMG Model Driven Architecture. http://www.omg.org/mda/ [1 May 2010].

[4] Orriens B, Yang J. Bridging the Gap between Business and IT in Service Oriented Collaboration. IEEE International Conference on Services Computing (SCC 2005), July 2005; **1**: 315-318. DOI: 10.1109/SCC.2005.115.

[5] Mendling J, Lassen KB, Zdun U. Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages. In Lehner F, Nösekabel H, Kleinschmidt P, eds.: *Multikonferenz Wirtschaftsinformatik 2006*. GITO-Verlag: Berlin, 2006; **2**: 297-312.

[6] Sendall S, Küster JM. Taming Model Round-Trip Engineering. OOPSLA and GPCE Workshop on Best Practices for Model Driven Software Development (OOPSLA/GPCE 2004), October 2004. http://www.softmetaware.com/oopsla2004/sendall.pdf [1 May 2010].

[7] OMG Unified Modeling Language (UML). http://www.uml.org/ [1 May 2010].

[8] OMG Business Process Modeling Notation (BPMN). http://www.bpmn.org/ [1 May 2010].

[9] OASIS Business Process Execution Language for Web Service Version 1.1 (5 Mai 2003). http://www.oasis-open.org/committees/download.php/2046/BPEL V1-1 May 5 2003 Final.pdf [1 May 2010].

[10] van der Aalst WMP, ter Hofstede AHM, Kiepuszewski B, Barros AP. Workflow Patterns. *Distributed and Parallel Databases* 2003; **14**(1): 5-51. DOI: 10.1023/A:1022883727209.

[11] Wohed P, van der Aalst WMP, Dumas M, ter Hofstede AHM, Russell N. On the Suitability of BPMN for Business Process Modelling. 4th International Conference on Business Process Management (BPM 2006), September 2006. LNCS 4102: 161-176. DOI: 10.1007/11841760_12.

[12] Rumpe E. Executable Modeling with UML. In *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*. Idea Group Publishing: Hershey, 2002; 697-701.

[13] Dijkstra EW. Go To Statement Considered Harmful. *Communications of the ACM* 1968; **11**(3): 147-148. DOI: 10.1145/362929.362947.

[14] Jackson MA. *Principles of Program Design*. Academic Press: London, 1975.

[15] Nassi I, Shneiderman B. Flowchart Techniques for Structured Programming. *ACM SIGPLAN Notices* 1973; **8**(8): 12-26. DOI: 10.1145/953349.953350.

[16] Kiepuszewski B, ter Hofstede AHM, Bussler CJ. On Structured Workflow Modeling. 12th Conference on Advanced Information Systems Engineering (CAiSE 2000), June 2000. LNCS 1789: 431-445. DOI: 10.1007/3-540-45140-4.

[17] Puhlmann F, Weske M. Investigations on Soundness Regarding Lazy Activities. 4th International Conference on Business Process Management (BPM 2006), September 2006. LNCS 4102: 145-160. DOI: 10.1007/11841760.

[18] Gardner T. UML Modeling of Automated Business Processes with a Mapping to BPEL4WS. 1st European Workshop on Object Orientation and Web Services (ECOOP 2003), July 2003. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.2955 [1 May 2010].

[19] Korherr B, List B. Extending the UML 2 Activity Diagram with Business Process Goals and Performance Measures and the Mapping to BPEL. 2nd International Workshop on Best Practices of UML (BP-UML 2006), November 2006. LNCS 4231: 7-18. DOI: 10.1007/11908883_4.

[20] Bordbar B, Staikopoulos A. On Behavioural Model Transformation in Web Services. ER 2004 Workshops CoMoGIS, CoMWIM, ECDM, CoMoA, DGOV, and eCOMO, November 2004. LNCS 3289: 667-678. DOI: 10.1007/b101694.

[21] Norton B, Cabral L, Nitzsche J. Ontology-Based Translation of Business Process Models. 4th International Conference on Internet and Web Applications and Services (ICIW 2009), May 2009. DOI: 10.1109/ICIW.2009.77.

[22] Valette R. Analysis of Petri Nets by Stepwise Refinements. *Journal of Computer and System Sciences* 1979; **18**(1): 35-46. DOI: 10.1016/0022-0000(79)90050-3.

[23] van der Aalst WMP. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems and Computers* 1998; **8**(1): 21-66. DOI: 10.1142/S0218126698000043.

[24] Kiepuszewski B, Hofstede AHM, van der Aalst WMP. Fundamentals of Control Flow in Workflows. *Acta Informatica* 2003; **39**(3): 143-209. DOI: 10.1007/s00236-002-0105-4.

[25] Ouyang C, Dumas M, Breutel S, ter Hofstede AHM. Translating Standard Process Models to BPEL. 18th International Conference on Advanced Information System Engineering (CAiSE 2006), June 2006. LNCS 4001: 417-432. DOI: 10.1007/11767138.

[26] White SA. Using BPMN to Model a BPEL Process. *BPTrends*, March 2005. http://www.bptrends.com/publicationfiles/03-05 WP Mapping BPMN to BPEL- White.pdf [1 May 2010].

[27] Ouyang C, Dumas M, ter Hofstede AHM, van der Aalst WMP. From BPMN Process Models to BPEL Web Services. IEEE International Conference on Web Services (ICWS 2006), 2006: 285-292. DOI: 10.1109/ICWS.2006.67.

[28] Ouyang C, Dumas M, ter Hofstede AHM, van der Aalst WMP. Pattern-based Translation of BPMN Process Models to BPEL Web Services. *International Journal of Web Services Research* 2008; **5**(1): 42-62.

[29] van der Aalst WMP, Lassen KB. Translating Unstructured Workflow Processes to Readable BPEL: Theory and Implementation. *Information and Software Technology* 2008, **50**(3), 131-159. DOI: 10.1016/j.infsof.2006.11.004.

[30] Ouyang C, Dumas M, van der Aalst WMP, ter Hofstede AHM, Mendling J. From Business Process Models to Process-Oriented Software Systems. *ACM Transactions on Software Engineering and Methodology* 2009, **19**(1): 2:1-2:37. DOI: 10.1145/1555392.1555395.

[31] Roser S, Lautenbacher F, Bauer B. Generation of Workflow Code from DSMs. OOPSLA Workshop on Domain-Specific Modeling, October 2007. http://www.dsmforum.org/events/DSM07/papers/roser.pdf [1 May 2010].

[32] Recker J, Mendling J. On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. 11th International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD 2006), June 2006. http://eprints.qut.edu.au/4637/ [1 May 2010].

[33] Indulska M, Recker J, Green P, Rosemann M. Are We There Yet - Seamless Mapping of BPMN to BPEL4WS. Americas Conference on Information Systems (AMCIS 2007), August 2007. http://aisel.aisnet.org/amcis2007/439 [1 May 2010].

[34] Weidlich M, Decker G, Grosskopf A, Weske M. BPEL to BPMN: The Myth of a Straight-Forward Mapping. Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE (OTM 2008), November 2008. LNCS 5331: 265-282. DOI: 10.1007/978-3-540-88871-0_19.

[35] Ammarguellat Z. A Control-Flow Normalization Algorithm and Its Complexity. *IEEE Transactions on Software Engineering* 1992, **18**(3): 237-251. DOI: 10.1109/32.126773.

[36] Hauser R, Koehler J. Compiling Process Graphs into Executable Code. 3rd International Conference on Generative Programming and Component Engineering (GPCE 2004), October 2004. LNCS 3286: 317-336. DOI: 10.1007/b101929.

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

[38] Hauser R, Friess M, Küster JM, J. Vanhatalo. Combining Analysis of Unstructured Workflows with Transformation to Structured Workflows. 10th International Enterprise Distributed Object Computing Conference (EDOC 2006): 129-140, October 2006. DOI: 10.1109/EDOC.2006.21.

[39] Zhao W, Hauser R, Bhattacharya K, Bryant BR, Cao F. Compiling Business Processes: Untangling Unstructured Loops in Irreducible Flow Graphs. *International Journal on Web Grid Services* 2006; **2**(1): 68-91. DOI: 10.1504/IJWGS.2006.008880.

[40] Zhao W, Bryant BR, Cao F, Hauser R, Bhattacharya K, Tao T. Transforming Business Process Models in the Presence of Irreducibility and Concurrency. *International Journal on Business Process Integration and Management* 2007; **2**(1): 37-48. DOI: 10.1504/IJBPIM.2007.014103.

[41] IBM WebSphere Business Modeler. http://www.ibm.com/software/integration/wbimodeler/ [1 May 2010].

[42] Johnson R, Pearson D, Pingali K. The Program Structure Tree: Computing Control Regions in Linear Time. ACM Sigplan Conference on Programming Language Design and Implementation (PLDI 1994): 171-185. DOI: 10.1145/178243.178258.

[43] Erosa AM, Hendren LJ. Taming Control Flows: A Structured Approach to Eliminating Goto Statements. International Conference on Computer Languages (ICCL 1994), May 1994: 229-240. DOI: 10.1109/ICCL.1994.288377.

[44] Böhm C, Jacopini G. Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules", *Communications of the ACM* 1966; **9**(5): 366-371. DOI: 10.1145/355592.365646.

[45] Harel D. On Folk Theorems. *Communications of the ACM* 1980; **23**(7): 379-389. DOI: 10.1145/358886.358892.

[46] Hecht MS, Ullman JD. Flow Graph Reducibility. *SIAM Journal on Computing* 1972; **1**(2): 188-202. DOI: 10.1145/800152.804919.

[47] Cocke J, Miller RE. Some Analysis Techniques for Optimizing Computer Programs. 2nd Hawaii International Conference on System Science (HICSS 1969), January 1969: 143-146.

[48] Unger S, Mueller F. Handling Irreducible Loops: Optimized Node Splitting versus DJ-Graphs. *ACM Transactions on Programming Languages and Systems* 2002; **24**(4): 299-333. DOI: 10.1145/567097.567098.

[49] Hauser R. Transforming Unstructured Cycles to Structured Cycles in Sequential Flow Graphs. IBM Research Report RZ 3624, 2005. http://www.rainerhauser.ch/public/reports/RZ3624.pdf [1 May 2010].

[50] Sadiq W, Orlowska ME. Analyzing Process Models Using Graph Reduction Techniques. *Information Systems* 2000; **25**(2), 117-134. DOI: 10.1016/S0306-4379(00)00012-0.

[51] Liu R, Kumar A. An Analysis and Taxonomy of Unstructured Workflows. 3rd Conference on Business Process Management (BPM 2005), September 2005. LNCS 3649: 268-284. DOI: 10.1007/11538394.

[52] Hauser R, Friess M, Küster JM, Vanhatalo J. An Incremental Approach to the Analysis and Transformation of Workflows Using Region Trees. *IEEE Transactions on Systems, Man, and Cybernetics - Part C* 2008; **38**(3): 347-359. DOI: 10.1109/TSMCC.2008.919178.

[53] Hauser R. Analysis and Transformation of Behavioral Models Containing Overlapped Patterns. *Journal of Object Technology* 2010; **9**(3): 105-124. http://www.jot.fm/issues/issue_2010_05/article4/ [1 May 2010].

[54] Koehler J, Hauser R, Küster JM, Ryndina K, Vanhatalo J, Wahler M. The Role of Visual Modeling and Model Transformations in Business-driven Development. GT-VMT 2006, 5th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006). ENTCS 211: 5-15. DOI: 10.1016/j.entcs.2008.04.025.

[55] Küster JM, Abd-El-Razik M. Validation of Model Transformations - First Experiences using a White Box Approach. 3rd Workshop on Model Design and Validation (MoDELS 2006), October 2006. LNCS 4364: 193-204. DOI: 10.1007/978-3-540-69489-2.

[56] IBM Insurance Application Architecture. http://www.ibm.com/industries/insurance/us/detail/solution/P669447B27619A15.html [1 May 2010].

[57] Gardner T, Griffin C, Koehler J, Hauser R. A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. MetaModelling for MDA Workshop, November 2003. http://www.omg.org/cgi-bin/doc?ad/2003-08-02 [1 May 2010].

[58] OMG RFP: MOF 2.0 Query / Views / Transformations RFP. http://www.omg.org/cgi-bin/doc?ad/2002-4-10 [1 May 2010].