RZ 3624 (# 99634) 08/08/2005
Computer Science   17 pages

Research Report

Transforming Unstructured Cycles to Structured Cycles in Sequential Flow Graphs

R.F. Hauser

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

E-mail: rfh@zurich.ibm.com

Transforming Unstructured Cycles to Structured Cycles in Sequential Flow Graphs

Rainer Hauser
IBM Zurich Research Laboratory, Switzerland
rfh@zurich.ibm.com
`http://www.zurich.ibm.com/csc/bit/bpia.html`

July 18, 2005

Abstract

Methods that allow transforming of unstructured cyclic models to functionally equivalent specifications having only structured cycles are crucial for automatically deploying graphical business process or workflow models, e.g., in the form of Universal Modeling Language (UML) activity diagrams, to an underlying platform based on a structured programming language, e.g., the Business Process Execution Language for Web Services (BPEL4WS). We present a method based on two simple transformation rules that can be applied to any sequential model without node-splitting in case of irreducibility. We further explore the effect of different rule-selection strategies based on these two basic rules as well as three other rules which are specializations of one of them.

1 Introduction

With the trend towards model-driven architecture [1] and development, graphical modeling languages such as the Universal Modeling Language (UML) gain importance as can be seen from the many new packages added to UML from its early days to version 2.0, its latest version [2]. The package for activity diagrams is such an extension for dynamic behavior [3].

UML 2.0 activity diagrams, like many other graphical modeling languages for business processes and workflows, use one or another form of flow graphs (directed graphs) and model a behavior with nodes and edges, where nodes represent activities and edges model the potential continuations from one activity to other activities. An edge can be seen as a visual representation of a goto as available in some programming languages.

Flow graphs can become cyclic when certain activities are executed more than once in a single run of the behavioral model. These cycles are called unstructured to distinguish them from the well-structured cycles modeled as explicit while- or repeat-loops. If unstructured cycles in business process or workflow models have to be eliminated and replaced by structured constructs, cycle-removal algorithms can be applied [4, 5].

One area where cycle-removal algorithms are needed is the field of transformations from a source metamodel that allows unstructured cycles to a target metamodel that only allows structured constructs. An example of such a transformation is the compilation of UML 2.0 activity diagram models to the Business Process Execution Language for Web Services (BPEL4WS) [6]. Another area where cycle-removal algorithms are useful is management of complexity, where they can help analyze large flow graphs in order to understand the intended behavior. As in structured programming, structured loops are easier to understand than unstructured cycles.

Different algorithms can be used to remove cycles in sequential flow graphs, two are discussed in [4]. The *state-machine controller method* basically interprets the nodes as states and the edges as transitions. It executes the activities in a single while-loop and keeps track of what the next activity is going to be. This method can be applied to any flow graph. The *goto-elimination method* on the other hand extracts the intended cycle-structure and replaces the cycles found with structured cycles. This method can only be applied, without additional aids, to flow graphs that are well-structured enough.

The connectedness of nodes in a flow graph with several nodes is a kind of measure for how well-structured the flow graph is. If we start with an acyclic graph, it is well-structured and can be transformed into a structured program with only if-statements. We can add more edges, and at a certain point in time, the flow graph becomes cyclic. The flow graph is still well-structured and can be transformed into a structured program with if-statements and one repeat-loop. If we progressively add edges, the flow graph evolves until it eventually becomes unstructured. When every node has an edge leading to every other node, the flow graph has become completely unstructured and can be traversed in any possible way. Any set of nodes with two or more nodes contains cycles.

Compiler theory came up with the concept of *reducibility* to define when a flow graph is well-structured [7]. The T1-T2 analysis is one algorithm to determine whether a flow graph is reducible [8]. The concept of reducibility has proved valuable over the past three decades as many goto-elimination in programming languages can only be applied to programs whose control flow graph is reducible [9, 10]. If a flow graph is irreducible, auxiliary methods such as node-splitting can be applied to regain reducibility, but the consequence is that nodes, i.e., pieces of code, have to be duplicated and this is not always desirable [11]. Since node-splitting increases the size of a flow graph, techniques have been suggested for the optimal strategy to split the nodes [12].

Various definitions for reducibility have been suggested, and the complexity of the corresponding algorithms to determine reducibility of a flow graph has been studied [13], but all these definitions of reducibility are equivalent. Despite this strong empirical evidence[1] that "well-structured" means "reducible" for flow

[1] The evidence may in the end not be as strong as it seems because the different definitions were not proposed completely independent of each other.

graphs, the class of flow graphs, for which cycle-removal can be applied without node-splitting or other similar methods, can be extended in a natural way to any sequential flow graph. This has the advantage firstly, that all sequential flow graphs can be transformed using a single algorithm, and secondly, that no auxiliary methods such as node-splitting are needed for irreducible flow graphs and therefore no code has to be duplicated.

To do so, we introduce two simple transformation rules, called L and C, which can be applied in any order to any sequential flow graph in order to produce a structured program that is functionally equivalent. For rule C, we further define three specializations in order to guide the transformation by selecting priorities for the different rules. This leads to more fine-grained control over the resulting program and allows the transformation to be optimized for specific needs. (Based on flow graphs, the transformation is restricted to input models without unstructured jumps into and out of structured elements such as while-loops as allowed by other goto-elimination algorithms [14]. This helps keeping the rules simple.)

In Section 2, the transformation rules are introduced and their effect is shown, while we discuss how the selection and application strategy of the rules influence the result of the transformation in Section 3. Finally, the paper concludes with a summary and outlook in Section 4.

2 The Transformation Algorithm

The transformation is based on two rules using a pattern with an edge and the nodes at either end. To formulate them, some basic definitions are needed.

2.1 Basic Definitions

A *flow graph* $G(N, E, n_i, n_f)$ is a directed graph with a set of nodes N, a set of edges E, an initial node n_i, and a final node n_f. For an edge e, leading from node n_1 to node n_2, the notation $n_1 \to n_2$ is used. With $e = n_1 \to n_2 \in E$, node n_1 is called the *source* of the edge e and the *predecessor* of node n_2, and node n_2 is called the *target* of the edge e and the *successor* of node n_1. An edge $n \to n$ from a node to itself is called a *self-loop*. In a flow graph, there are no edges leading to the initial node n_i, and no edges leaving the final node n_f. A *path* from n_s to n_t is a sequence of edges $n_s^i \to n_t^i \in E$ $(i = 1, ..., m)$ such that $n_s = n_s^1$, $n_t^i = n_s^{i+1}$ $(1 \leq i < m)$ and $n_t^m = n_t$. In the following it is assumed that there is always a path from the initial node n_i to all other nodes and a path to the final node n_f from all other nodes. The *complementary* flow graph $G'(N', E', n_i', n_f')$ of a flow graph results when the direction of all edges are swapped and the initial and final nodes are exchanged, i.e., time is reversed. Formally, $N' = N$, $n_i' = n_f$, $n_f' = n_i$ and $e' = n_2 \to n_1 \in E'$ if and only if $e = n_1 \to n_2 \in E$.

In order to represent business processes and workflows, we extend the definition of a flow graph. First, a node is assumed to be annotated with a *behavior* that describes what the node as an activity does[2]. Initially, the behavior of node A will be written as `invoke A`, and the behavior of the initial and final node is empty. Secondly, a condition called *guard* is assigned to every edge such that the guards of all edges leaving a node are mutually exclusive and exhaustive. The guards state which edge will be fired by the node when the execution of the node terminates. As a convention, the guard for an edge $A \rightarrow B$ will be written in this paper as `AB`, and the names S and T are used for the start (initial) and the termination (final) node, respectively. Figure 1 shows an example of a flow graph with these annotations.

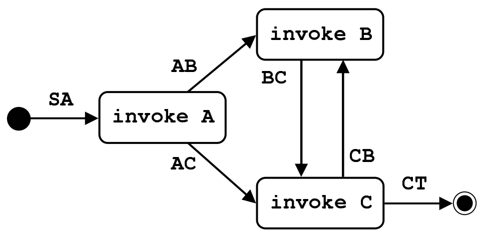


Figure 1: Sample flow graph.

2.2 Initial Transformation Step

The guards in a flow graph are originally expressions over variables within the scope of a node that tests whether the guards are true. Because the setting of these variables and the tests may become separated during the transformation, additional variables have to be introduced and maintained in order to keep the intended execution logic correct. (Similar variables are needed in other goto-elimination methods [9, 14].)

In an *initial transformation step* one single variable, called `next`, in the scope of all nodes, i.e., readable and writable by all nodes, in the flow graph is first introduced. It is used to store the next node to be executed. Secondly, the initial behavior `invoke N` for a node N is replaced by

```
invoke N;
if (c1) { next := N1 }
...
if (cn) { next := Nn }
```

where `ci` is the original guard of the edge $N \rightarrow Ni$ (for $1 \leq i \leq n$). Lastly, the initial guard `ci` for every edge $N \rightarrow Ni$ is replaced with `next = Ni`.

The simple schema introduced here in the initial transformation step basically uses the variable `next` to store the goto-target similarly to the schema used in the *state-machine controller method* [4].

[2]The behavior is supposed to be described with a behavioral metamodel, but a simple string representation will be used in this paper for compactness of the representation.

2.3 The Two Transformation Rules

The two transformation rules are based on a pattern that consists of an edge $M \rightarrow N$ with its source M and its target N. An edge is either a self-loop or not. Thus, the only two patterns that are possible are the ones shown in Figure 2. One of the transformation rules, called L, removes the self-loop in Figure 2 a, and the other, called C, merges the two nodes in Figure 2 b and simultaneously removes the edge. Because a flow graph contains only a finite number of nodes and edges and each rule removes at least one edge, the algorithms eventually terminates. The result is a flow graph with a single node and no edges, because otherwise one of the rules could still be applied.

The application of a rule in a transformation step modifies the flow graph by merging nodes and removing edges, updates the behavior of nodes, and may change the guard of edges. The result of the transformation is the behavior of the single remaining node when the rules can no longer be applied.

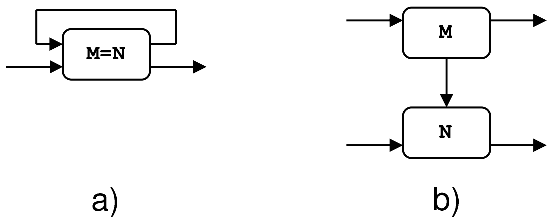


Figure 2: Possible patterns.

Figure 3 depicts the transformation rule L that is applicable to an edge $M \rightarrow N$ if $M = N$. The self-loop with guard `c1` is removed and the node with behavior `bodyN` before the transformation step is modified.

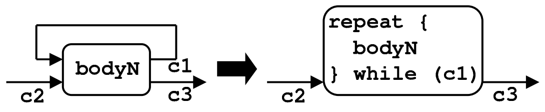


Figure 3: Transformation rule L.

The behavior of node N is wrapped into a repeat-loop with the guard of the removed edge as its loop-condition. All other edges of node N remain unchanged. This rule is responsible for introducing *loops* into the behaviors of the nodes.

Figure 4 depicts the transformation rule C that is applicable to an edge $M \rightarrow N$ if $M \neq N$. The nodes M and N with behavior `bodyM` and `bodyN` before the transformation step are merged, the edge leading from M to N is removed, and the other edges with their guards are adapted appropriately.

The behavior of the resulting node executes `bodyM` only if `c2|c3`, the disjunction of all guards of incoming edges of M, is true while `bodyN` is executed only if `c1|c4|c5`, the disjunction of all guards of incoming edges of N, is true. The

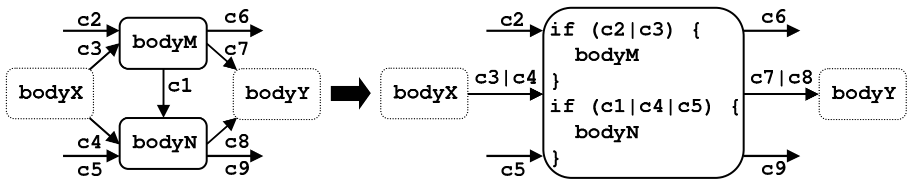


Figure 4: Transformation rule C.

symbol "|" represents the logical *or*. The new node inherits all incoming and outgoing edges of M and N except for the removed edge. The guards of incoming and outgoing edges remain unchanged except if both nodes originally had an edge from or to a third node (here nodes X and Y), the two edges are merged and their guards are combined with a disjunction. This rule is responsible for introducing *conditionals* into the behavior of the nodes.

Rule C wraps the original behavior of both nodes into if-statements. This is not always necessary, because if the source node has only one outgoing edge or the target node has only one incoming edge, and this edge obviously has to be the edge that will be removed by the transformation step, the rules can be simplified as shown in Figure 5.

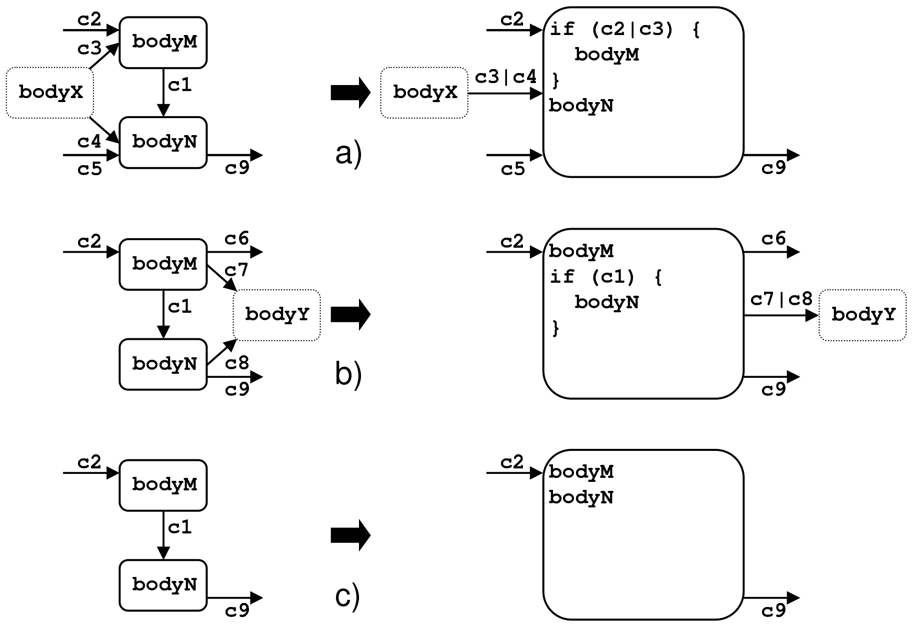


Figure 5: Specializations of rule C.

We introduce the following specializations for rule C. Rule C_s in Figure 5 a is applicable if node M has only one outgoing edge, the edge $M \rightarrow N$. The guard of this edge must always be true, and the second if-statement is not necessary, because `bodyN` must be executed when the merged node is reached. Rule C_t in Figure 5 b is applicable if node N has only one incoming edge, the edge $M \to N$. Node N is only reachable through node M and the first if-statement is therefore not necessary. Rule C_{st} shown in Figure 5 c specializes both rules C_s and C_t further and is applicable when the two nodes M and N build a simple sequence. No if-statement is therefore necessary.

2.4 Categorization of Flow Graphs

The transformation rule C and its three specializations C_s, C_t, and C_{st} form a lattice as shown in Figure 6 a. These rules together with rule L can, by ignoring the behaviors of the nodes and the guards of the edges, also be seen as reduction rules as used in the T1-T2 analysis to determine whether a flow graph is *reducible* [8]. Because rule L corresponds to T1 and C_t corresponds to T2[3], rules L and C_t define the set of reducible flow graphs. Similarly, for completeness of terminology, we define a flow graph as *complementarily reducible* if and only if its complementary flow graph is reducible, and we also introduce the term *symmetrically reducible* through L, C_s, and C_t the same way as reducibility is defined through L and C_t. A flow graph can be *reduced* by a set of rules if at the end, when the rules in the set can no longer be applied, it contains only one node and no edge. This leads to the categorization of flow graphs shown in Figure 6 b, indicating how well-structured a flow graph is.

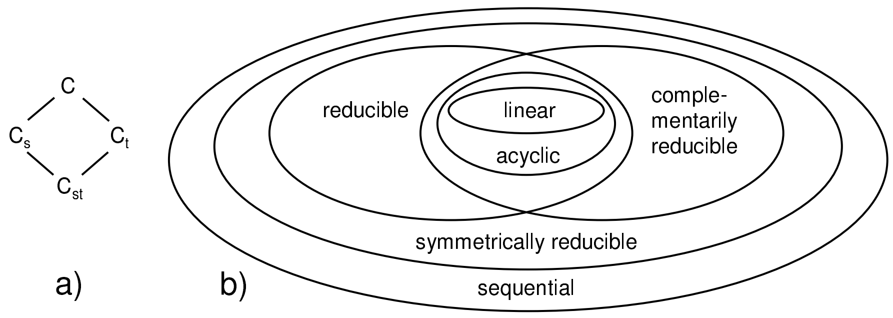


Figure 6: The lattice of the C rules and categorization of flow graphs.

We describe each of the categories and discuss the main properties of their members:

1. A flow graph is *linear* if and only if it can be reduced by the rule C_{st}. In other words, there is only one path from the initial to the final node in a linear flow graph, and each node is visited exactly once. Linear flow graphs are sequences.

[3]These two rules as transformation rules only use the guard of the removed edge in the condition of the repeat-loop and the if-statements, respectively, but no guards of other come-from-edges. This is the distinguishing feature of reducibility.

2. A flow graph is *acyclic* if and only if it can be reduced by one of the rules C_s, C_t, or C. Because flow graphs have a single initial and final node, tree structures are not possible. Thus, all three rules can reduce the same flow graphs.

3. A flow graph is *reducible* if and only if it can be reduced by the two rules L and C_t. This set obviously contains the linear and acyclic flow graphs, but also certain cyclic flow graphs: the cyclic flow graphs where each cycle has a single entry.

4. A flow graph is *complementarily reducible* if and only if it can be reduced by L and C_s. Note that L is symmetric with respect to time, and that C_s and C_t complement each other when time is reversed. The set therefore contains the cyclic flow graphs where each cycle has a single exit.

5. A flow graph is *symmetrically reducible* if and only if it can be reduced by L, C_s, and C_t. There exist symmetrically reducible flow graphs that are neither reducible nor complementarily reducible.

6. A flow graph is *sequential* if and only if it can be reduced by L and C. (Because the definition of flow graphs used in this paper does not allow to express concurrency, every flow graph is sequential.)

The reduction algorithms based on these rules are always confluent, i.e., lead to the same result independent of the sequence in which the rules are applied and the selection of the node to which a rule is applied. The proof for the T1-T2 analysis in [15] can easily be extended to C_s, and the rules L and C reduce any sequential flow graph and therefore always lead to the same result.

The sample flow graph in Figure 1 is not reducible because none of the ordinary, neither initial nor final, nodes has a unique predecessor that is not the initial node. It is, however, complementarily reducible because node B, i.e., the node with behavior `invoke B`, has a unique successor. By applying C_s, the nodes B and C are merged and get a self-loop that can be removed with L. In the rest of the steps, C_{st} can be applied to reduce the remaining edges.

Figure 7 shows two symmetrically irreducible flow graphs, without initial and final node, where Figure 7 b results from Figure 7 a when nodes A and D are the same node[4]. If any of the edges in either flow graph is removed, the resulting flow graph becomes symmetrically reducible.

2.5 Final Transformation Step

The transformation steps may lead to nested if-statements that can be flattened in a *final transformation step*. Generally, the conditions in guards and if-statements can often be simplified. When a node has only one outgoing edge,

[4]Turned into a pattern, the flow graph Figure 7 a with the restriction that nodes A, B, and C must be different nodes, but A and D can be the same node, can be used similarly to the subgraph (*) in [8, 15].

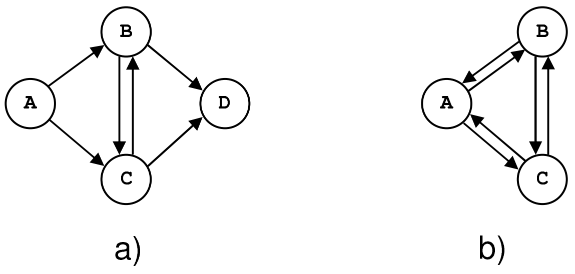


Figure 7: Symmetrically irreducible flow graphs.

the guard must be true and can be eliminated. Further, the transformation rules may lead to disjunctions such as `next = N | next = N` that contain the same term more than once and can be simplified.

In the following examples, we do this silently and also write, to keep behaviors compact, `next = A | B` instead of `next = A | next = B`.

2.6 Correctness of the Transformation

The behavior of the flow graph is not changed through the introduction of the variable `next` in the initial transformation step. To prove that the whole transformation leads to a program that is functionally equivalent to the original flow graph, it is sufficient to show that L and C preserve the behavior.

We first observe that for all transformation rules the guards of all outgoing edges of a new node are still mutually exclusive and exhaustive. They may no longer form a tautology because one edge is removed by L, but its guard can never be true after the repeat-loop.

We next show that the transformation rules do not change the execution logic. The proof for rule L is trivial because the guards of the outgoing edges, i.e., the conditions `c1` and `c3` in Figure 3, stay correct for the old and the new behavior and these guards are mutually exclusive. The proof for C must show that the behavior of the new node is correct, that the guards of the incoming edges cannot trigger unintended behavior inside the new node, and that the behavior of the new node cannot enable the guards of the wrong outgoing edges, i.e., the conditions `c6` to `c9` in Figure 4. The first two points are obvious and the last point is guaranteed because the guard of an edge $A \rightarrow B$ can only contain a disjunction of terms `next = Ci` where each Ci is either B or has been merged in a previous step into B. Note that a node cannot be merged into more than one node.

2.7 An Example

Figure 8 shows the transformation with the rules L, C_s, and C_t applied to the sample flow graph in Figure 1 after the initial transformation step to set and use the newly introduced variable `next`. In a first step, application of C_s merges

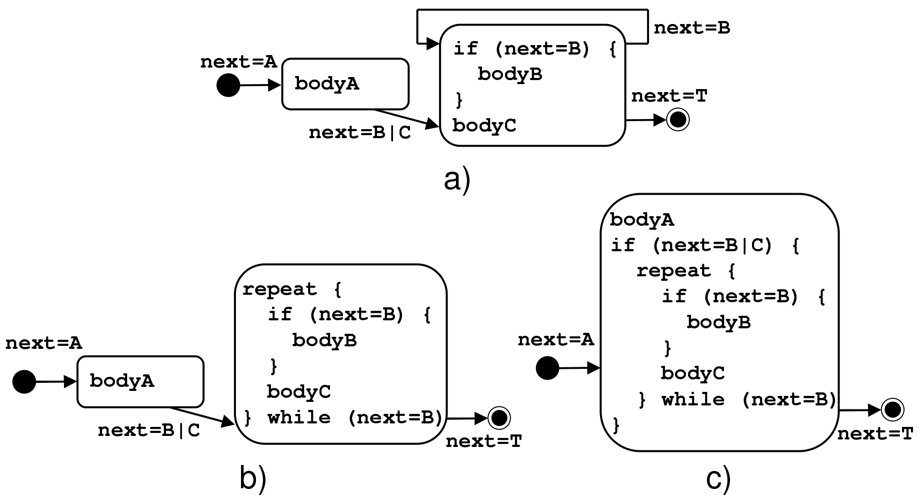


Figure 8: Sample transformation result.

node B with C and leads to the flow graph with a self-loop in Figure 8 a. The self-loop is eliminated next using L as shown in Figure 8 b where the repeat-loop replaces the self-loop. In the last step resulting in Figure 8 c, rule C_t is applied to demonstrate how the guard of the edges are used. C_{st} could have been applied instead to avoid the unnecessary if-statement with condition `next = B | C`. To complete the transformation, the initial node can be removed and becomes the initialization, and the final node disappears into implicit termination:

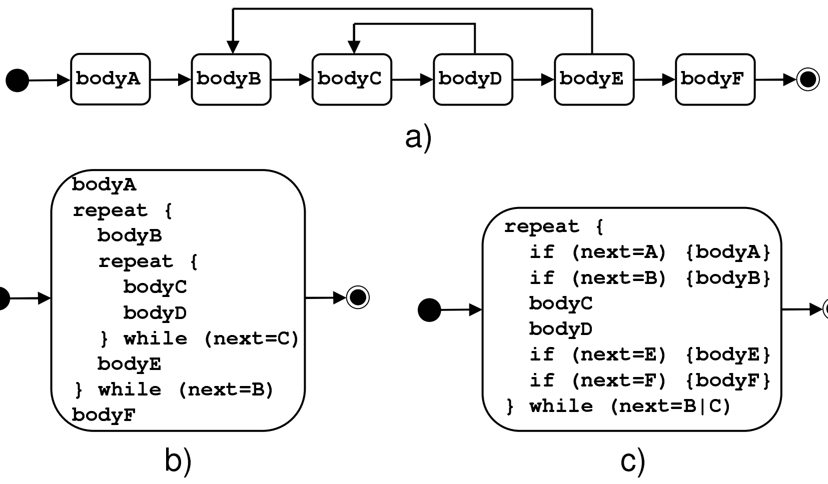
```
next := A
invoke A
if (AB) {next := B}
if (AC) {next := C}
repeat {
  if (next = B) {
    invoke B
    next := C
  }
  invoke C
  if (CB) {next := B}
  if (CT) {next := T}
} while (next = B)
```

If there would have been more than one edge leaving the initial node, the initialization would have become a set of if-statements that initialize the variable `next`. If the final node is resolved last, only a single edge remains in the flow graph and rule C_{st} can be applied. It is clear that the empty behavior of the final node has no influence on the program. Terms `next = T` never appear in conditions of if-statements or repeat-loops.

3 Rule-Application Strategy

As already seen in Figure 8, transformations based on some or all five rules are not confluent. Though the resulting programs are not equal, they are functionally equivalent. Consequently, different strategies can be used to run the transformation in order to shape the resulting program. In other words, the transformations defined by the five rules build a family of transformations, and the strategy determines the family member. The family of transformations is outlined as pseudo-code in the Appendix.

Figure 9 shows an example. The flow graph in Figure 9 a contains two nested loops. To transform them, we used all rules and applied two different strategies. Giving L highest priority, leads to the result shown in Figure 9 b, and giving L lowest priority leads to the result shown in Figure 9 c.



```
bodyA
repeat {
  bodyB
  repeat {
    bodyC
    bodyD
  } while (next=C)
  bodyE
} while (next=B)
bodyF
```

```
repeat {
  if (next=A) {bodyA}
  if (next=B) {bodyB}
  bodyC
  bodyD
  if (next=E) {bodyE}
  if (next=F) {bodyF}
} while (next=B|C)
```

Figure 9: Transformation for nested cycles.

3.1 Rule-Selection Strategy

There are two decisions to be made when choosing the strategy: 1) The subset of rules to be used, and 2) the priorities of the rules. The first decision determines the set of flow graphs that can be transformed completely to a structured program. The second decision influences the quality of the transformation result. In general, if L has highest priority, the resulting program contains many repeat-loops, and these loops contain only code that is part of the cycle. Therefore, the number of if-statements is minimal. If, on the other hand, L has lowest priority, the resulting program contains a low number of repeat-loops, but unnecessary code is moved into these loops protected by if-statements to ensure that the code is only executed when necessary. Therefore, a loop may contain code that is only executed once as can be seen in Figure 9 c for nodes A and F.

The case where L has lowest priority is worth exploring a bit further. If the rule-set contains C_t, this strategy has the tendency to move nodes to the right of a cycle into the loop. Similarly, if the rule-set contains C_s, nodes to the left of the cycle may get moved into the loop. Because C generalizes these two rules, nodes on both sides of the cycle are consumed before the loop is resolved, and – after flattening of nested if-statements – the result for cyclic flow graphs is similar to the the result of the state-machine controller approach, although rule C orders the nodes according to the connectivity in the flow graph, while the state-machine controller approach is completely unordered. In general, rule C without the help of any other rule can transform any flow graph into a flow graph with a single node. If this node has a self-loop, the flow graph was cyclic and a single application of rule L at the end of the transformation can remove the self-loop. Otherwise, the flow graph was acyclic.

Rule-sets containing C_s and C_t not only merge nested but also overlapping cycles as shown in Figure 10. Because the flow graph in Figure 10 a contains node A with only one outgoing edge and node D with only one incoming edge, nodes A and B as well as nodes C and D can be merged using C_s and C_t, respectively. These two transformation steps merge the two cycles into one. Thus, as shown in Figure 10 b, overlapping cycles result in a single loop in case the rule-set contains L, C_s, and C_t, and the strategy gives L lowest priority.

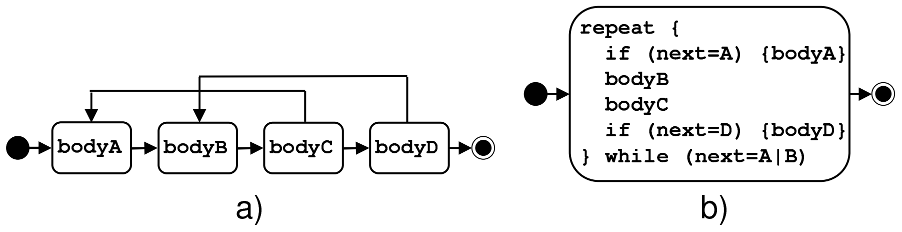


Figure 10: Transformation for overlapping cycles.

Disjoint cycles such as the ones in Figure 11 a, on the other hand, can only be resolved into a single loop if the rule-set contains C, because nodes A and B both have two incoming and two outgoing edges. Transformed with L and either C_s or C_t lead independent of the priorities to the two repeat-loops in Figure 11 b, while transformed using C with high priority and L with low priority produce the behavior in Figure 11 c. A strategy to keep disjoint loops separate may thus use all rules but give C_s and C_t high, L medium and C low priority.

3.2 Node-Selection Strategy

Generally, we always resolve initial and final node last, because they become the initialization and implicit termination of the resulting program. Otherwise, the nodes are rather indistinguishable from the transformation's point of view, as long as only local properties of the flow graph are considered. The only

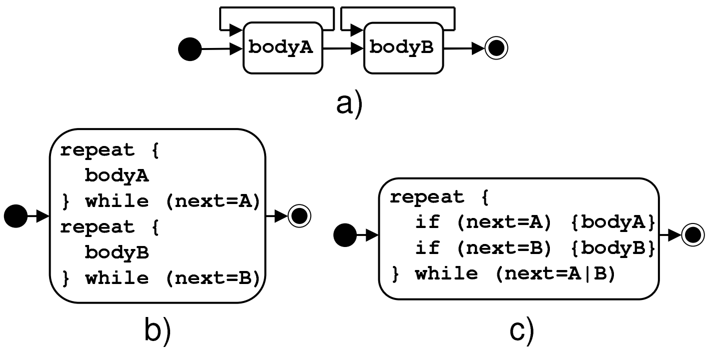


Figure 11: Transformation for disjoint cycles.

distinguishing features are the number of incoming and outgoing edges of the source and target node, whether source and target node are the same, and whether source and/or target node have a self-loop.

The preconditions of the five transformation rules describe local patterns in flow graphs. To determine whether L is applicable, a pattern with a single node and a self-loop has to be matched. To determine whether one of the other four rules is applicable, a pattern with two distinct nodes connected through an edge and, with possibly some additional rule-specific constraints has to be matched.

The strategy can be further refined through additional and more complex patterns than the preconditions of the rules in the rule-set. For example, we have implemented the following strategy: The rule-set contained L and C_t, and C_t had higher priority than L. Instead of applying L blindly, when C_t was no longer applicable, we used a pattern that preferred nodes with a self-loop and only one other incoming edge over other nodes with a self-loop. This strategy minimized the number of repeat-loops, because the node and its predecessor became applicable for C_t after removal of the self-loop.

The fact that the three rules C_s, C_t, and C have the tendency to move nodes unnecessarily into loops if L has lowest priority, may not always be desired. To improve this situation in particular, and to fine-tune the strategy in general, patterns involving more than an edge with its source and target node or even global properties of the flow graph may have to be used, though we prefer to stay with local properties as long as possible for performance reasons.

4 Summary and Outlook

The paper presents a family of algorithms that allow the transformation of sequential behavioral models with unstructured continuations, e.g., gotos, into behavioral models with only the structured constructs for loops, e.g., repeat-loops, and conditionals, e.g., if-statements. The algorithms can be applied as an update-transformation or as a transformation that creates a new model either to textual models, e.g, programming languages, in order to eliminate gotos, or to graphical models, e.g., business processes or workflows, in order to remove unstructured cycles, if representable as flow graphs, although we only demonstrated the concepts throughout this paper for an update-transformation on a graphical model where each node has a textual model attached[5].

The family of algorithms is based on two general transformation rules, called L and C, and three specializations of C, called C_s, C_t, and C_{st}, that have been derived from generalizations of the reduction rules provided by the T1-T2 analysis [8]. The rule L is responsible for the introduction of structured loops by resolving gotos that go "upstream", and the rule C is responsible for the introduction of conditionals by resolving gotos that go "downstream".

A transformation algorithm, i.e., a member of this family, can be defined by selecting a subset of rules and by choosing a rule-application strategy. Depending on the rule-set selected, any source model or only either acyclic, reducible, complementarily reducible or symmetrically reducible source models can be completely transformed. This has the advantage that no auxiliary methods such as node-splitting are needed in order to transform the desired set of source models.

The transformations are preceded by an initial transformation step to save the continuation conditions. We used a single variable, called `next`, to store this information. A more elegant schema is shown in [4], but it cannot be used for the rules C_s and C without modification[6]. This schema nicely shows the possible paths in the conditions, and it is well possible, that there are schemas for the initial transformation step that provide additional insight into the loop structure and are thus more meaningful than the simple schema used in this paper.

More serious is the fact that, depending on the requirements for the transformation, the rule-application strategy cannot always be defined accurately enough only based on local properties of the source model. Giving L highest priority leads to many loops. Giving L lowest priority minimizes the number of loops, but moves nodes into the loops that do not belong to any cycle. Further research may lead to more fine-tuned rule-application strategies.

The rule-application strategy is also the weak point of the goto-elimination method in [9]. This method corresponds to the rule-set containing L and C_t[7]. Its rule-application strategy, i.e., the order of resolution, is based on a topological sort that analyses global properties of the flow graph, but its results are not always convincing. Using L and C_t only, corresponding to the T1 and T2

[5]Rule L could produce a LoopNode and rule C a StructuredActivityNode containing two ConditionalNodes in UML 2.0 activity diagrams as has been done for the rules L and C_t in [5].

[6]Note that the schema for L and C_t used in [4] with one variable per node N (e.g., called `nextFromN`) instead of the single variable for the whole flow graph (e.g., called `next`) as used in this paper would change the conditions in the result for C_t in Figure 5. The guard `c9` would become `c1 & c9`, and the guard `c7 | c8` would become `c7 | (c1 & c8)` with "`&`" indicating the logical *and*.

[7]L is called *derecursivation*, and C_t is called *substitution*. The merging of two edges leading to the same third node after merging two nodes with C_t corresponds to *factorization*. The initial transformation step is the *pre-calculation* step, and the flattening of nested if-statements is called *if-distribution*.

rule of the T1-T2 analysis, respectively, the method also has to deal with code duplication strategies in case the flow graph is irreducible.

The family of transformation algorithms presented in this paper can only handle sequential flow graphs. Concurrency, however, is crucial for business processes and workflows. The question of how to extend this approach to handle concurrency in a similar way with transformation rules based on local patterns is a challenging research topic.

Concurrency can range from simple, nicely nested fork-join pairs to arbitrary cycles in concurrent regions. Using token-flow semantics, nicely nested fork-join pairs allow static determination of the number of tokens flowing in a system at any point in time, while concurrent cycles may lead to an unbound number of tokens. The well-known workflow patterns give a good overview of what kind of concurrency can be expected in business process and workflow systems [16].

Independent of the current and future restrictions of business and workflow systems, it would be valuable to understand unrestricted concurrency to the same degree as sequential flow graphs are understood. If a theory handles sequential and concurrent constructs uniformly, all the better.

Acknowledgments

We thank Jochen Küster for his advice, which helped to improve this paper significantly.

References

[1] OMG: Model-Driven Architecture (MDA). http://www.omg.org/mda/.

[2] OMG: Unified Modeling Language 2.0. http://www.omg.org/uml/.

[3] OMG: UML 2.0 Superstructure Final Adopted Specification. Document pts/03-08-02, August 2003.

[4] Hauser, R., Koehler, J.: Compiling Process Graphs into Executable Code. Proc. 3rd International Conference on Generative Programming and Component Engineering, LNCS 3286, pp. 317–336, Vancouver, Canada, October 2004.

[5] Koehler, J. et al.: Declarative Techniques for Model-Driven Business Process Integration. IBM Systems Journal, 44(1), pp. 47-65, 2005.

[6] OASIS: Business Process Execution Language for Web Services 1.1. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/. May 5, 2003.

[7] Aho, A. et al.: Compilers. Principles, Techniques, and Tools. Addison-Wesley, 1986.

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

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

[10] Koehler, J., Hauser, R.: Untangling Unstructured Cyclic Flows – A Solution Based on Continuations. On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004, LNCS 3290, pp. 121–138, Agia Napa, Cyprus, October 2004.

[11] Carter, L., Ferrante, J., Thomborson, C.: Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger. Proc. 30th Annual Symposium on Principles of Programming Languages, pp. 106–114, New Orleans, Louisiana, USA, January 2003.

[12] Unger, S., Mueller, F.: Handling Irreducible Loops: Optimized Node Splitting versus DJ-Graphs. ACM Transactions on Programming Languages and Systems, 24(4), pp. 299–333, July 2002.

[13] Ryder, B.G., Paull, M.C.: Elimination Algorithms for Data Flow Analysis. ACM Computing Surveys, 18(3), pp. 277–316, September 1986.

[14] Erosa, A.M., Hendren, L.J.: Taming Control Flows: A Structured Approach to Eliminating Goto Statements. Proc. International Conference on Computer Languages, pp. 229–240, Toulouse, France, May 1994.

[15] Hecht, M.S.: Flow Analysis of Computer Programs. Elsevier North-Holland, Amsterdam, 1977.

[16] Van der Aalst, W.M.P. et al.: Workflow Patterns. BETA Working Paper Series, WP 47, Eindhoven University of Technology, Eindhoven, 2000.

Appendix

The algorithm outlined as pseudo-code works as follows, where *input* is the input model passed into the transformation and *output* the result produced by the transformation:

```
$graph \leftarrow initialTransformationStep(input)$
repeat
  $applied \leftarrow false$
  for all $rule \in strategy$ while $\neg applied$ do
    for all $edge \in graph.edges$ while $\neg applied$ do
      if $rule.pattern.match(edge)$ then
        $rule.update.apply(edge)$
        $applied \leftarrow true$
      end if
    end for
  end for
until $graph.edges = \emptyset \vee \neg applied$
if $graph.edges = \emptyset \wedge \mid graph.nodes \mid = 1$ then
  $node \in graph.nodes$
  $output \leftarrow finalTransformationStep(node.behavior)$
end if
```

The algorithm gets the input model (i.e., the *graph*) and repeats the transformation steps until either no edge is left in the graph or none of the rules can be applied anymore. If no edge is left and the graph therefore only contains a single node, the output of the transformation is the behavior of the single remaining node. Otherwise, the output is undefined. If the set of input graphs is limited to a category as shown in Figure 6 b and the strategy contains all rules needed to process this category, the transformation always terminates with a defined result.

Inside the repeat-loop, the algorithm loops through the rules in the strategy selecting rules with higher priority before rules with lower priority. For each rule, the algorithm goes through all the edges[8] and tests whether the pattern of the rule matches. If it does, the rule updates the graph and sets the variable *applied* to true. At this point, the algorithm leaves both for-all-loops and starts again with the rule that has highest priority.

The strategy can be defined by the user or by the application. If the user can select the rules and their respective priorities, the result of different strategies can be compared.

[8]The pseudo-code leaves out details such as the fact that edges leaving the initial node and edges leading to the final node are only handled when none of the rules can be applied to any of the other edges anymore.