Compiling Process Graphs into Executable Code

Rainer Hauser and Jana Koehler

IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
{rfh,koe}@zurich.ibm.com
http://www.zurich.ibm.com/csc/ebizz/bpia.html

Abstract. Model-driven architecture envisions a paradigm shift as dramatic as the one from low-level assembler languages to high-level programming languages. In order for this vision to become reality, algorithms are needed that compile models of software systems into deployable and executable implementations. This paper discusses two algorithms that provide such transformations for process graph models in a business process or workflow environment and produce executable programs based on Web services and orchestration languages. The reverse transformations back from executable programs to process graphs are also described.

1 Introduction

The model-driven architecture (MDA) initiative introduced by the Object Management Group (OMG) [1] is slowly becoming mature and is being given appropriate tool support [2]. However, there is still a long way to go before complex software systems can be completely described as models and deployed automatically [3]. In this respect, software engineering is far behind hardware development.

Model-driven development (MDD) for arbitrary software systems is still not possible, and – even if it were – the danger is that creating the complete set of models for a complex system may turn out to be more difficult than realizing it with traditional programming methods. Thus, MDD may not be considered worth the effort despite potential savings of maintenance costs. Applied to the field of business process engineering, the problems of MDD are simpler than those for general software systems thanks to the componentization and composition structure where the parts that are difficult to describe in models have already been implemented by other means. The basic building blocks for business processes (i.e., the components) are either services directly implemented as Web services or legacy systems wrapped as a Web service [4]. These building blocks are combined to form complex business logic (i.e., the composition) using orchestration languages such as BPEL4WS [5] and datatype definitions specified using WSDL [6]. Therefore, applying MDD to business processes means modeling a complex business process in terms of available Web services and transforming this model (or set of models) into executable and deployable orchestrations.

Complete business process models consist of a process model to describe the execution logic, an information model for the datatypes used in the process model, an organizational model with a role or authorization model, and possibly other models. The compilation from a model describing a complex business process to a deployable BPEL4WS implementation has to transform the process model into BPEL4WS activities, the role or authorization model into BPEL4WS partners, and the information model into BPEL4WS variables specified using WSDL. In this paper, we will concentrate on transforming the process model's control flow into BPEL4WS activities. We specify process models using a subset of UML 2.0 activity diagrams [7], which is sufficiently rich to allow concurrency and arbitrary cycles in the control flow. For the orchestration, we simplified the BPEL4WS specification to contain only those elements needed to describe the execution logic extracted from the process model. UML 2.0 activity diagrams and BPEL4WS have been selected because both are widely accepted, de-facto standards.

UML activity diagrams (and most of the other modeling languages for business processes) allow specification of cyclic behavior. However, these cycles are unstructured, and BPEL4WS only allows structured cycles in the form of while-loops. Therefore, when going from the UML process model to the BPEL4WS implementation, unstructured cycles have to be resolved into structured cycles. Our discussion traces such transformations of process graphs into executable code, and we present two algorithms in detail. We also show the transformation in the other direction, which is important for reconciliation if the process graph and the executable code can be modified independently[1].

This paper is structured as follows: We define the subset of UML 2.0 activity diagrams used to model business processes (process graphs) in Section 2, and the simplified BPEL4WS used to describe the orchestration (executable code) in Section 3. We discuss in Sections 4 and 5 the transformation (compilation) from process graphs into executable code and the transformation (decompilation) of executable code back into a process graph, respectively. Finally, we conclude the paper in Section 6 with a discussion of this approach and an outlook.

2 Process Graph Model

Several modeling languages for business processes (such as BPMN [8]) have been proposed. We selected a subset of the UML 2.0 activity diagram metamodel [7] as the basis for this work. (BPMN and UML 2.0 activity diagrams are similar and may converge in the future [9].) In order to remove redundancy

[1] It is not yet clear how important reconciliation in MDA/MDD will turn out to be. Software engineers these days rarely look at the machine code produced by the compiler of a high-level programming language. When MDA/MDD has become reality and the compilation of models into deployable code has become mature, there will be no need to modify the deployed code except through recompilation and redeployment. However, there may still be a need for decompiling deployed code into a business process model.

and architectural elements not needed for modeling the control flow of a business process or workflow, we restrict the UML 2.0 activity diagrams to a subset shown in Figure 1.

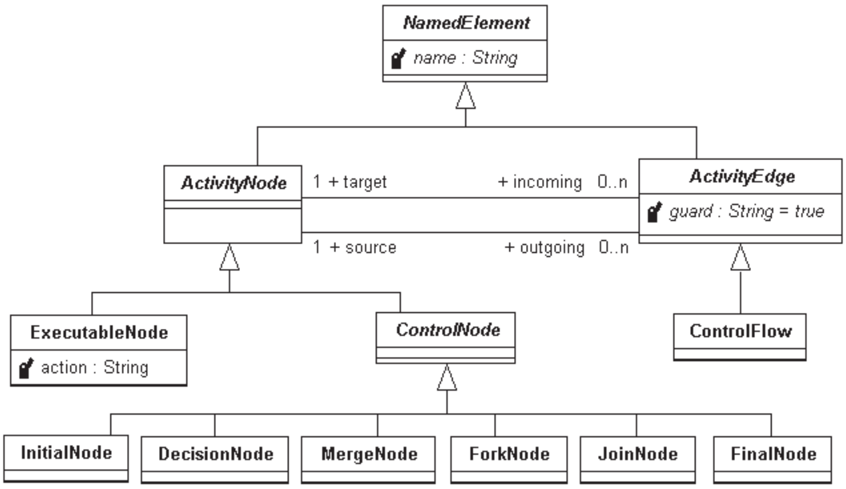

Fig. 1. Abstract syntax of process graph models.

According to the abstract syntax, a process graph consists of nodes and edges represented as instances of the abstract classes ActivityNode and ActivityEdge. A node is either an ExecutableNode or a ControlNode. An executable node is supposed to perform an action represented as a string that is not specified further. An edge can only be a ControlFlow whose guard is also represented as a string. The relation between nodes and edges is modeled as bidirectional associations. A node can have multiple incoming and outgoing edges, but an edge has always exactly one source and one target node. The start node of a process graph is an InitialNode, and the end node is a FinalNode. The XOR-splits and -joins (alternatives) are represented as DecisionNodes and MergeNodes. The AND-splits and -joins (concurrency) are modeled as ForkNodes and JoinNodes.

Basically, a process graph is a connected, directed graph with a single start and a single end node. Such a graph is called reducible if (informally speaking) "there are no jumps into the middle of loops from outside" [10]. A node n_1 is said to dominate a node n_2 if all paths from the start node to n_2 go through n_1 [10]. A fork-join pair is properly nested if (again informally speaking) there are no jumps into and out of the corresponding concurrent threads nor any interactions between the threads[2]. A formal definition will be given in Section 4.4.

For a process graph to be valid, there are further well-formedness constraints not shown in the class diagram:

[2] It is not possible to design more sophisticated synchronization mechanisms between threads in this simplified model.

1. The process graph is finite.
2. A process graph must have exactly one InitialNode and one FinalNode. (A process graph has a single entry and a single exit.)
3. A process graph and every thread contains at least one ExecutableNode.
4. For every node, there is a path from the start node to the end node going through this node. (This avoids unreachable areas of the process graph, which are detectable through static analysis of the graph.)
5. All nodes except DecisionNodes, ForkNodes and FinalNodes have exactly one outgoing edge. DecisionNodes and ForkNodes have at least two outgoing edges, and FinalNodes have no outgoing edge.
6. All nodes except MergeNodes, JoinNodes and InitialNodes have exactly one incoming edge. MergeNodes and JoinNodes have at least two incoming edges, and InitialNodes have no incoming edge.
7. All ExecutionNodes have a unique name, which can be used to identify a specific ExecutionNode.
8. The guards for all nodes except DecisionNode are *true*. If n edges leave a DecisionNode with $expr_1, \ldots, expr_n$ as guards, then
   $expr_1 \vee expr_2 \vee \ldots \vee expr_n = true$ complete (PM1)
   $expr_i \wedge expr_j = false$ (for $i \neq j$) deterministic (PM2)
   must hold[3].
9. All fork-join pairs are properly nested (see Section 4.4).

When transforming a process graph with the algorithms introduced below, these constraints can be checked automatically except for PM1 and PM2 in Constraint 8 which would require domain knowledge of the model's data types[4].

The above list of constraints expresses a set of necessary conditions for a process graph to be valid, but the list is not sufficient. It is still possible to define valid process graphs that do not make sense.

Figure 2 shows the simple and artificially created example of a process graph, which will be used throughout this paper, because it contains all the features needed to explain the transformation algorithms. It introduces at the same time the graphical notation for the types of nodes in UML 2.0 activity diagrams used in this paper. The InitialNode on the left side is connected to a ForkNode, and the corresponding JoinNode on the right side leads to the FinalNode. There are two threads between fork and join. The upper one is more complex and contains DecisionNodes with guards (e.g., $exprSA$) and MergeNodes. The ExecutableNode A is contained in a cycle. The lower thread is a simple sequence of the two ExecutableNodes C and D. If $exprSB = \neg exprSA$ is true (to enforce PM1 and PM2), this example process graph is valid according to the above well-formedness constraints. There is exactly one start and one end node, the

[3] We also assume that each $expr_i$ is sometimes true in order to avoid areas of the process graph whose unreachability cannot solely be detected by static analysis of the process graph.

[4] We can always enforce PM1 by adding an 'else'-edge going to the FinalNode (or its corresponding MergeNode), and PM2 by modifying the guards with $expr'_i = expr_i \wedge \neg expr_1 \wedge \ldots \wedge \neg expr_{i-1}$.

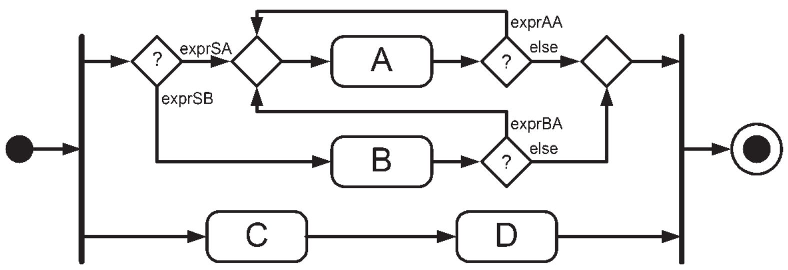

Fig. 2. Process graph example.

fork-join pair is properly nested, and the two sequential threads satisfy the other constraints.

3 Executable Code Model

The orchestration language BPEL4WS [5] describes the execution logic for business processes composed of Web services. Figure 3 shows the simplified subset used for the purpose of demonstrating compilation and decompilation.

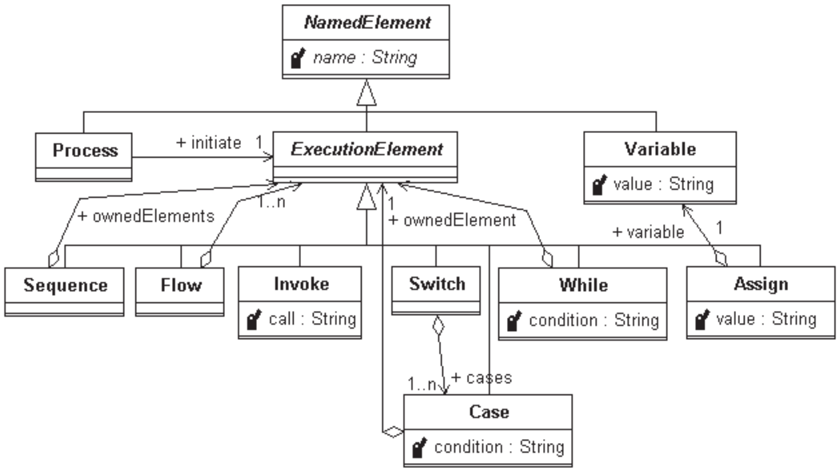

Fig. 3. Abstract syntax of executable code models.

A model contains a Process with one ExecutionElement initiating the execution. ExecutionElements can be Sequences (sequential), Flows (concurrent), the structural elements Switch (case-statement) and While (loop), Assign (variable assignment) and Invoke. The Invoke is merely a placeholder for synchronous and asynchronous Web service invocations with an action attribute (call). For

```
<process>
  <flow>
    <sequence>
      <assign 'SA:=exprSA' />
      <assign 'SB:=exprSB' />
      <switch>
        <case condition= 'SB'>
          <sequence>
            <invoke B />
            <assign 'BA:=exprBA' />
          </sequence>
        </case>
      </switch>
      <switch>
        <case condition= 'SA | (SB & BA)'>
          <sequence>
            <assign 'loopA:=true' />
            <while condition= 'loopA'>
              <invoke A />
              <assign 'AA:=exprAA' />
              <assign 'loopA:=AA' />
            </while>
          </sequence>
        </case>
      </switch>
    </sequence>
    <sequence>
      <invoke C />
      <invoke D />
    </sequence>
  </flow>
</process>
```

Fig. 4. Executable code example.

simplicity, we do not allow the BPEL4WS 'otherwise' element in a Switch but assume that all cases are coded as Case elements fulfilling similar constraints as PM1 and PM2 for process graphs.

For a more concise textual representation (concrete syntax) of this simplified BPEL4WS, we use the XML representation of BPEL4WS with less verbose assignments instead of copy specifications [5], and we encode negation, disjunction and conjunction in the conditions with "!", "|" and "&", respectively. We also assume that the corresponding WSDL definitions have been defined.

A possible transformation of the example in Figure 2 is shown in Figure 4. Using the transformation algorithms described in Section 4, this BPEL4WS skeleton code can be derived automatically from the given process graph. The resulting BPEL4WS contains a Flow with two Sequences representing the two concurrent threads. The upper thread contains two switch elements, where the second one consists of a while-loop. The second thread contains simply two invocations called one after the other without further control logic.

4 Compilation of Process Graphs

Among the model transformation approaches [11], graph transformation methods are quite popular [12, 13]. Work on signal flow graph compilers is relevant as well [14]. The problem we need to solve can – independent of the actual transformation approach – use techniques from compiler theory [15] because a sequential process graph can be compiled into a program with gotos. After this initial transformation, goto-elimination methods for sequential programming languages can be applied. We look at non-concurrent process parts first and examine concurrency later.

4.1 Initial Transformation for a Sequential Part

The general pattern of an activity in a sequential part of a process graph (either a completely sequential process graph or a sequential thread) is – as shown in Figure 5 – a MergeNode followed by an ExecutableNode followed by a DecisionNode, where the MergeNode collects the incoming edges (XOR-join), and the DecisionNode routes the control flow to the next activity (XOR-split). However, MergeNodes with only one incoming edge and DecisionNodes with only one outgoing edge are eliminated. (The lower thread in Figure 2, for example, shows the two activities C and D without MergeNodes and DecisionNodes.) Similarly, the InitialNode is followed by a DecisionNode, and the FinalNode is preceeded by a MergeNode. InitialNodes and FinalNodes (or the ForkNodes and JoinNodes as the start and end nodes of a thread) can be interpreted as no-op action.

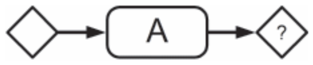

Fig. 5. General pattern of an activity.

Thus, a sequential process graph (or a sequential thread) can be translated into an initial program with guarded gotos for the edges using the rules:

1. A MergeNode – even if eliminated – becomes a label in the program. (Especially, the always missing MergeNode of the start node – interpreted as a no-op action – becomes the start label of the program.)
2. A node representing an action – usually an ExecutionNode – becomes an executable action that is not further specified. (In particular, the start node becomes a no-op action, and the end node an exit action.)
3. A DecisionNode – even if eliminated – becomes a set of guarded gotos (one per edge).

If a MergeNode is directly connected to a DecisionNode, we also insert a no-op action.

The result of this initial transformation applied to the upper thread of Figure 2 is shown in Figure 6 in a simple programming language[5] and with the labels S and T used for the invisible start and end node of the thread.

[5] Instead of extending the executable code model with a goto ExecutionElement or defining an intermediate model, we use a familiar but not formally specified programming language with simple if-statements and repeat-while-loops instead of the more complicated switch-construct and the while-loop in BPEL4WS.

```
S: if (exprSA) goto A;             // left-most DecisionNode
   if (exprSB) goto B;
A: invoke A;                       // MergeNode and ExecutionNode A
   if (exprAA) goto A;             // DecisionNode after A
   if (!exprAA) goto T;            // instead of "else goto T;"
B: invoke B;                       // ExecutionNode B
   if (exprBA) goto A;             // DecisionNode after B
   if (!exprBA) goto T;            // instead of "else goto T;"
T: exit;                           // right-most MergeNode
```

Fig. 6. Initial transformation result.

This initial program consists of blocks, where a block starts with a label and ends either before the next label or at the program's end. Because of PM1, there is no implicit flow of control from one block to the next, and the blocks (except for the first one) can be arbitrarily reordered. Because of PM2, the order of the guarded goto-statements in a block is irrelevant, and they can be arbitrarily reordered as well[6]. Both properties are important as will be shown below.

4.2 Finite State Machine Transformation for a Sequential Part

There is a straightforward translation of this initial program into a single loop, which basically transforms the sequential process graph into a finite state machine. Figure 7 shows the result for the upper thread of the example shown in Figure 2. The gotos become assignments to the variable *nextNode*, the initial block becomes the initialization, the terminal block dissolves into an implicit exit at the program end, and the other blocks become cases in the switch.

This compilation method, which obviously could be directly applied to a sequential process graph instead of going through the initial transformation first, has the advantage that it is easy to implement. However, it has two disadvantages. First, performance is an issue for process graphs with a large number of nodes because the switch testing of which node comes next has to test $n/2$ guards on average if n is the number of nodes in the graph. Second, the program does not show the inherent program structure with the different phases of a business process, the nesting of cyclic activities and the order of sequential activities[7]. For

[6] Without PM2, there are three semantics possible for the DecisionNode: (1) The guards are tested in a certain order, and the first true one gets control. (2) The set of guards is executed nondeterministically (e.g., as a set of Dijkstra's guarded commands). (3) All edges whose guards are enabled are followed concurrently. Note that for the second case, the initial transformation preserves the behavior if we interpret the set of guarded gotos in a block as a set of Dijkstra's guarded commands.

[7] It is rather subjective to determine which program structure best represents a process graph. However, we believe that business processes are often designed as a sequence of various phases. A shopping application, for example, starts with an authentication phase, continues with a product configuration and selection phase, and ends with a negotiation of terms and conditions. These phases should be made visible in the executable code model and should not be merged into a single loop.

```
<sequence>
  <switch>
    <case condition= 'exprSA'>
      <assign 'nextNode:=A' />
    </case>
    <case condition= 'exprSB'>
      <assign 'nextNode:=B' />
    </case>
  </switch>
  <while condition= 'nextNode!=T'>
    <switch>
      <case condition= 'nextNode=A'>
        <sequence>
          <invoke A />
          <switch>
            <case condition= 'exprAA'>
              <assign 'nextNode:=A' />
            </case>
            <case condition= '!exprAA'>
              <assign 'nextNode:=T' />
            </case>
          </switch>
        </sequence>
      </case>
      <case condition= 'nextNode=B'>
        <sequence>
          <invoke B />
          <switch>
            <case condition= 'exprBA'>
              <assign 'nextNode:=A' />
            </case>
            <case condition= '!exprBA'>
              <assign 'nextNode:=T' />
            </case>
          </switch>
        </sequence>
      </case>
    </switch>
  </while>
</sequence>
```

Fig. 7. Finite state machine transformation result.

these two reasons we do not consider the result of this transformation method satisfactory. Note, however, that this transformation would allow nondeterminism to be preserved if we relax PM2 and if the underlying platform supports nondeterminism (e.g., in the form of Dijkstra's guarded commands).

4.3 Goto-Elimination Method for a Sequential Part

The method by Ammarguellat [16] for eliminating gotos in program languages to achieve single-entry, single-exit while-loops has two main steps called "derecursivation" for removal of self-references (self-loops) and "substitution" for merging two activities into one activity. Note that these two steps correspond to the T1 and T2 rule, respectively, in the T1-T2 analysis [17]. The substitution step may need "if-distribution" and "factorization" as additional steps at the end. We adapted this method for process graphs and describe the different steps in the following.

Pre-calculation: We start from the initial program and introduce additional variables in order to save the current state of the expressions. The reason for this step is explained below. The input to this step (and the other steps later on) is shown on the left, the output on the right:

```
L: invoke L;                   L: invoke L;
   if (exprLM1) goto M1;          if (exprLM1) nextNodeFromL:=M1;
   ...                            ...
   if (exprLMm) goto Mm;          if (exprLMm) nextNodeFromL:=Mm;
                                  if (nextNodeFromL=M1) goto M1;
                                  ...
                                  if (nextNodeFromL=Mm) goto Mm;
```

Common Block Structure: After every transformation step, we bring each block in the program back into a structure where the structured elements appear before the unstructured guarded goto-statements. After pre-calculation, the program is in this structure. The *invoke L* and the first set of if-statements with assignments for *nextNodeFromL* are the structured part. We will call the structured part of block L *bodyL* in the following steps.

Substitution and Elimination: Because there is no implicit control flow from one block to another, we can replace all occurences of *goto M* with the complete block M if block M does not itself contain a *goto M*:

```
L: bodyL;                     L: bodyL;
   if (exprL1) goto M1;          if (exprL1) goto M1;
   ...                           ...
   if (exprLi) goto M;           if (exprLi) {
   ...                             bodyM;
   if (exprLm) goto Mm;            if (exprM1) goto N1;
...                                ...
M: bodyM;                          if (exprMn) goto Nn;
   if (exprM1) goto N1;          }
   ...                           ...
   if (exprMn) goto Nn;          if (exprLm) goto Mm;
```

If-Distribution: With the substitution step, the label *M* together with all *goto M* statements have been eliminated. However, we have lost the common block structure. In order to regain it, we have to apply another step:

```
if (exprLi) {                  if (exprLi) {
  bodyM;                          bodyM;
  if (exprM1) goto N1;         }
  ...                          if (exprLi & exprM1) goto N1;
  if (exprMn) goto Nn;         ...
}                              if (exprLi & exprMn) goto Nn;
```

This step is only allowed if *exprLi* is not changed by *bodyM*. (Ammarguellat introduces an additional variable because of the general applicability of her method, but we will show below how to ensure in the special context of our transformation that the logic is not changed by if-distribution.)

After substitution and if-distribution, we can move the structured parts of the source and the target block of the substitution together because the guards of the if-statements are mutually exclusive owing to PM2. Note also that the resulting guarded gotos are still mutually exclusive and complete.

Factorization: If the unstructured part of a block contains two guarded gotos to the same label, the two statements can be combined into one:

```
L: bodyL;                      L: bodyL;
   if (exprL1) goto M1;           if (exprL1) goto M1;
   ...                            ...
   if (exprLi) goto M;            if (exprLi | exprLj) goto M;
   ...                            ...
   if (exprLj) goto M;            if (exprLm) goto Mm;
   ...
   if (exprLm) goto Mm;
```

Derecursivation: Self-references have to be resolved through derecursivation after moving the self-referencing goto to the top of the unstructured part:

```
L: bodyL;                     L: repeat {
   if (exprLs) goto L;               bodyL;
   if (exprL1) goto M1;          } while (exprLs);
   ...                           if (exprL1) goto M1;
   if (exprLm) goto Mm;          ...
                                 if (exprLm) goto Mm;
```

The complete repeat-while-loop becomes the new structured part *bodyL* for the next step. Note also here that the guards of the unstructured gotos are still complete and mutually exclusive, but may no longer form a logical tautology, even if they did before. (After the loop, *exprLs* must be false, and therefore $exprL1 \vee \ldots \vee exprLm$, without *exprLs*, must be true.)

A repeat-while-loop with the test at the end can be converted into an ordinary while-loop with the test at the beginning – as needed for BPEL4WS – but requires the introduction of one additional variable per loop:

```
repeat {                       loopL:=true;
  bodyL;                       while (loopL) {
} while (exprLs);                bodyL;
                                 loopL:=exprLs;
                               }
```

Obsolete-Guard-Removal: Because of PM1 the set of guarded gotos in any block of the initial program is complete in the sense that every possible case is covered. This property is preserved by all the transformation steps discussed above. Therefore, if only one guarded goto is left in a block, the guard must be true and can be removed:

```
L: bodyL;                     L: bodyL;
   if (expr) goto M;             goto M;
```

The Complete Algorithm for Reducible Process Graphs: These transformation steps can be applied in different order. Before we describe in which sequence the goto-elimination algorithm will apply them, we make some observations: (1) The derecursivation step can be applied to a block with self-references at any time, but if it is applied right before it is substituted, the number of while-loops is minimized. (2) Substituting a block for more than one goto-statement leads to code duplication. (3) Blocks can be substituted in any order.

Ammarguellat proves that there is a substitution sequence for reducible graphs where each block is only substituted once. This property together with the observations above determines the sequence in which the various steps are applied for reducible process graphs. The pre-calculation is the first step. Next, one block after the other (except the blocks corresponding to the start and end node in the graph) is selected in a loop for substitution such that there is only one goto to the label of this block. In the loop, when we have selected a block, we apply derecursivation if necessary and perform the substitution. With if-distribution we bring the target block back into the common block structure, and with factorization we reduce the number of gotos in a block to at most one for each remaining label. If at this point only one guarded goto is left in a block, it can be removed using the obsolete-guard-removal step. After elimination of all labels and gotos with exception of the ones corresponding to the start and end block, the label for the start node can be removed[8]. Next, the guard for the remaining *goto T* can be removed with obsolete-guard-removal, the block corresponding to the end node dissolves into an implicit *exit* at the end of the program, and *bodyS* becomes the resulting program:

```
S: bodyS;                        bodyS;
   if (expr) goto T:
T: exit;
```

The algorithm in pseudo-code is shown in Figure 8. Note that in a final step, *bodyS* has to be converted into the simplified BPEL4WS encoding.

The result of the transformation algorithm applied to the upper thread in Figure 2 is shown in Figure 9. For space reasons we cannot show larger examples, but we applied this algorithm to real-world business processes with very promising results for the program structure. The method of Ammarguellat nicely

[8] A start node has no incoming edges. If this restriction is relaxed, a derecursivation step would remove all remaining gotos to the label of the start node.

```
$blocks \leftarrow preCalculation(processGraph)$
while $blocks \backslash \{startBlock, endBlock\} \neq \emptyset$ do
  $sourceBlock \leftarrow selectBlockForSubstitution(blocks \backslash \{startBlock, endBlock\})$
  $blocks \leftarrow blocks \backslash \{sourceBlock\}$
  if $sourceBlock \in sourceBlock.gotoTargets$ then
    $derecursivation(sourceBlock)$
  end if
  for all $targetBlock \in blocks$ do
    if $sourceBlock \in targetBlock.gotoTargets$ then
      $substitution(sourceBlock, targetBlock)$
      $ifDistribution(targetBlock)$
      $factorization(targetBlock)$
      if $\mid targetBlock.gotoTargets \mid = 1$ then
        $obsoleteGuardRemoval(targetBlock)$
      end if
    end if
  end for
end while
```

Fig. 8. Goto-elimination algorithm.

structures the execution logic of the original graph but has the disadvantage of duplicating code if the graph is irreducible.

Note also that this method – unlike the finite state machine translation – does not preserve nondeterminism, because the guarded gotos (initially interpretable as a set of Dijkstra's guarded commands) get separated by substitution and derecursivation. We finally mention without further elaboration that the two transformation methods can be combined in various ways (e.g., by applying the goto-elimination method until code duplication would be necessary, and completing with the finite state machine method.)

Discussion of Additional Variables: The method described above works only for reducible process graphs because the if-distribution can be dangerous as mentioned when we introduced the if-distribution step. In Ammarguellat's original method, if-distribution is safe but always adds a new variable to the program. Additional variables must be meaningful for a reader of the program if the program is intended to be read by a human user. The variable *nextNode* in the finite state machine transformation method (Figure 7) and the variables *nextNodeFromX* as well as *loopX* in the goto-elimination transformation method (Figure 9) have an obvious meaning[9].

We outline a proof that if-distribution is legal if every substitution step replaces only a single goto with a block (which is always possible for reducible graphs). We define for each block of the initial program after the pre-calculation step a set called $Modifies(block, step)$ that shows which variables can be modi-

[9] Instead of a variable $nextNodeFromX$, we used a set of Boolean variables XY in Figure 4 to express $nextNodeFromX = Y$ in an alternative, more compact form.

```
<sequence>
  <switch>
    <case condition= 'exprSA'>
      <assign 'nextNodeFromS:=A' />
    </case>
    <case condition= 'exprSB'>
      <assign 'nextNodeFromS:=B' />
    </case>
  </switch>
  <switch>
    <case condition= 'nextNodeFromS=B'>
      <sequence>
        <invoke B />
        <switch>
          <case condition= 'exprBA'>
            <assign 'nextNodeFromB:=A' />
          </case>
          <case condition= '!exprBA'>
            <assign 'nextNodeFromB:=T' />
          </case>
        </switch>
      </sequence>
    </case>
  </switch>
  <switch>
    <case condition= 'nextNodeFromS=A |
                     (nextNodeFromS=B & nextNodeFromB=A)'>
      <sequence>
        <assign 'loopA:=true' />
        <while condition= 'loopA'>
          <invoke A />
          <switch>
            <case condition= 'exprAA'>
              <assign 'nextNodeFromA:=A' />
            </case>
            <case condition= '!exprAA'>
              <assign 'nextNodeFromA:=T' />
            </case>
          </switch>
          <assign 'loopA:=(nextNodeFromA=A)' />
        </while>
      </sequence>
    </case>
  </switch>
</sequence>
```

Fig. 9. Goto-elimination transformation result.

fied by the body of block *block* in step *step*. Initially, the set is $Modifies(L, 0) = \{nextNodeFromL\}$ for each block L. If block M is substituted in block L in step n, the set becomes $Modifies(L, n) = Modifies(L, n-1) \cup Modifies(M, n-1)$. Because the problematic expression in the if-condition references only variables in $Modifies(L, n-1)$ and *bodyM* modifies only variables in $Modifies(M, n-1)$, if-distribution is allowed if $Modifies(L, n-1) \cap Modifies(M, n-1) = \emptyset$. This intersection can only be nonempty if a block N previously has been substituted in blocks L and M.

The easiest solution to make if-distribution safe in the case of irreducible graphs is what is called node-splitting in compiler theory. For each *goto* M in the initial program, a copy of the block M is created with new labels M_1, M_2 and so on. This leads to different additional variables $nextNodeFromM_i$.

4.4 Separation of Sequential and Concurrent Parts

The compilation of a single pair of ForkNode and JoinNode in the process graph model into a Flow element in the executable code model is trivial if each thread between them contains only one ExecutableNode. Based on this observation, we outline an algorithm to decompose a process graph into sequential and concurrent subgraphs.

Areas of a process graph with only one edge leading from the outside into the area and one edge leading from the area to the outside are of special interest because they can be abstracted into a single node as a structured activity. The incoming and the outgoing edge define the interfaces to such an abstractable area or subgraph. A fork-join pair is called properly nested if all threads are such abstractable areas, and their incoming edges come from the same fork, and the outgoing edges lead to the same join. In the following paragraphs, we give formal definitions that a reader may skip if satisfied with the informal descriptions.

We define a process graph $G(N, E, n_i, n_f)$ with a set of nodes N, a set of edges E, the InitialNode n_i and the FinalNode n_f as usual. We write $e(n_1, n_2)$ for the edge from $n_1 \in N$ to $n_2 \in N$. A subgraph $G'(N', E')$ with $N' \subseteq N \setminus \{n_i, n_f\}$ and $\forall e = e(n_1, n_2) \in E\,(n_1 \in N' \vee n_2 \in N' \leftrightarrow e(n_1, n_2) \in E')$ is called abstractable if there exist two edges $e(n_1, n_1')$ and $e(n_2, n_2')$ in E' such that $n_1 \in N \setminus N'$, $n_1' \in N'$, $n_2 \in N'$ and $n_2' \in N \setminus N'$, whereas all the other edges in E' lead from a node in N' to a node in N'. In other words, the subgraph contains all edges between its own nodes, and there is exactly one edge coming in from outside and exactly one edge going out. (For all process graphs $G(N, E, n_i, n_f)$, the subgraph $G(N \setminus \{n_i, n_f\}, E)$ is an abstractable subgraph because there is one edge with n_i as source and one edge with n_f as target.)

The source of the incoming edge of an abstractable subgraph $G(N', E')$ dominates all nodes in N'. Note also that if the original graph was valid, an abstractable subgraph is also valid when completed with an additional InitialNode and FinalNode to replace the source and target of the incoming and outgoing edges, respectively.

To define when fork-join pairs are properly nested, we select one ForkNode f, which must have at least two outgoing edges to be valid. We follow one of them, come to node n_i (the first node of thread i) and compute N_i, the set of nodes dominated by n_i. We determine further the set of edges where source and/or target belong to N_i and call this set E_i. We do this for all threads. If $G(N_i, E_i)$ for all threads i are abstractable subgraphs whose outgoing edges lead to the same JoinNode j, which does not have any other incoming edge, then the corresponding fork-join pair is properly nested.

The abstractable subgraph $G(N_i, E_i)$ for each thread i can be combined into an abstract sequential node s_i hiding all nodes in N_i and all edges in E_i except for the incoming edge from f and the outgoing edge to j. The subgraph $G(N_\|, E_\|)$ with $N_\| = \{f, j\} \cup \bigcup_i \{s_i\}$ and $E_\| = \bigcup_i \{e(f, s_i), e(s_i, j)\}$ is also an abstractable node, which can be combined into an abstract concurrent node.

By recursively combining threads into abstract sequential nodes and fork-join pairs with already abstracted threads into abstract concurrent nodes, a complete valid process graph can be transformed into sequences and flows in the simplified version of BPEL4WS.

5 Reverse Engineering of Process Graphs

The result of the two transformation methods can be decompiled back into a process graph if its structure has not been modified manually. For the flows, this is trivial. Thus, we only have to outline the decompilation of a sequence without flows but possibly with abstract concurrent nodes hiding flows.

5.1 Reverse Finite State Machine Transformation

The initialization before the while-loop determines the start node and its sucessor or successors with transition conditions. The loop-condition determines the end node. The cases in the switch of the loop determine the successor or successors for each node with transition conditions. Thus, we can obtain all edges with their transition conditions and create the process graph. This is not very surprising because a process graph can be interpreted as a graphical representation of a finite state machine.

5.2 Reverse Goto-Elimination Method

Decompilation after the goto-elimination method is based on two observations. First, substitution (together with if-distribution) adds an if-statement to the end of the target block's structured part, and derecursivation wraps a structured part into a while-loop. Factorization affects only the unstructured part. Second, if the conditions in the loops and if-statements are brought into disjunctive normal form $C_1 \vee \ldots \vee C_n$, all C_i are of the form $nextNodeFromX_1 = X_2 \wedge nextNodeFromX_2 = X_3 \wedge \ldots \wedge nextNodeFromX_{n-1} = X_n$, and they all lead from the same X_1 to the same X_n. Thus, these conditions encode possible paths in the process graph from X_1 to X_n. Not all paths may be shown explicitly because some conditions may have been eliminated by obsolete-guard-removal.

To reverse the goto-elimination method, the statements *invoke* X and the variables $nextNodeFromX$ are used to determine the ExecutableNodes with their names, and the program is recursively brought back into the form it had after the initial transformation and the pre-calculation step, which obviously can be transformed into a process graph. The program – after conversion from the simplified BPEL4WS back into the simple programming language with if-statements and repeat-while-loops – becomes:

```
S: bodyS;
   if (expr) goto T;
T: exit;
```

We set *expr* to true and mark it as temporary because it may have to be modified in order to fulfill PM2 (reverse obsolete-guard-removal).

Also for the reverse transformation, we keep the blocks in the common block structure, and a block therefore has the form:

```
L: bodyL;
   if (expr1) goto L1;
   ...
   if (exprn) goto Ln;
```

The last statement in *bodyL* is either an *invoke X*, an if-statement or a repeat-while-loop. If the last statement is not the only statement in *bodyL*, we split the block into two blocks, one with all the statements except for the last one (*rest*), and one with only the last statement (*last*):

```
L: rest;
   if (expr) goto M;
M: last;
   if (expr1) goto L1;
   ...
   if (exprn) goto Ln;
```

We set *expr* again to true and mark it as temporary. The body of block M now contains only one statement. If it is an *Invoke X*, we are done. If it is a loop, we undo derecursivation:

```
M: repeat {                        M: body;
     body;                            if (cond) goto M;
   } while (cond);                    if (expr1) goto L1;
   if (expr1) goto L1;                ...
   ...                                if (exprn) goto Ln;
   if (exprn) goto Ln;
```

If the body of block M is an if-statement, we undo substitution:

```
N: ...                             N: ...
   if (expr) goto M;                  if (expr & cond) goto M;
   ...                                ...
M: if (cond) {                     M: body;
     body;                            if (expr1') goto L1;
   }                                  ...
   if (expr1) goto L1;                if (exprn') goto Ln;
   ...
   if (exprn) goto Ln;
```

In order to determine *expr*$1'$, we have to undo if-distribution (i.e., remove the conjunction with *cond*) and factorization (i.e., split disjunctions).

After each step, we have to rename previously introduced labels such that the label for *Invoke L* is L, and the logical expressions marked as temporary must be resolved if possible. For space reasons, we will not go into more details of the reverse goto-elimination method.

6 Summary and Outlook

This paper describes two methods for compiling process graphs into executable code and vice versa. We concentrated on the algorithmic aspects of the transformations between business process models and executable models with an emphasis on the implementation. Other aspects such as some background from theoretical computer science and an alternative algorithm based on continuation semantics are discussed in [15].

Transformation methods like the ones discussed here are a step in the direction of OMG's MDA vision. However, the simplified models used to demonstrate the transformations have to be extended to the full power of control and data flow available in BPEL4WS, including fine-grained synchronization between concurrent threads. Organizational models for roles and authorizations as well as information models have to be included, together with additional deployment information. Only then will the completely automatic translation and deployment of business processes encoded as BPEL4WS be possible.

As an intermediate step on the way to fulfilling the complete MDA vision, the executable model may have to be refined manually before deployment. In this case, reconciliation of business process models and executable models is important in order to preserve manual changes on the side of the executable models when the business process models are modified. The transformations from the executable models back into the business process models is only one part, but keeping track of the mapping between the elements in the two model sets is also crucial. This has been recognized by the community discussing the upcoming OMG Q/V/T standard for MOF 2.0 query, views and transformations [18].

The two models shown in Figures 1 and 3 as well as the not formally specified intermediate model (the simple programming language used to describe the two transformation methods) can be represented as MOF 2.0 models [19]. The Object Constraint Language OCL [20], which is regarded as an important part of the Q/V/T standard, is powerful enough – though rather verbose and difficult to read – to specify the well-formedness constraints on a process graph to be valid. Therefore, we can imagine that the final Q/V/T language will be capable of defining complex transformations such as the ones described in this paper. There are two reasons why this may not be the right way to go.

First, the examples in the Q/V/T proposals are all very simple, and there is a big class of transformations needed in the MDA-space with a similar complexity. A Q/V/T language designed for this class of problems can be made simple, easy to use and purely declarative. If a Q/V/T language tries to solve all possible transformation problems, it may become just another imperative, general-purpose programming language. In the goto-elimination method, defining the mapping for process graph and executable code elements as well as the transformation logic itself is far from being trivial.

Second, we are not sure how much of the transformations as described in this paper will still be needed for MDA in the long run. To explain what we mean we have to step back and examine why we are today where we are. Software projects in general and business process or workflow projects in particular require bridging the gap between the customer (business people) and the IT team. These two groups speak different languages and think in terms of different concepts. Informal drawings have helped and still help them find a common understanding to design such software systems. These graphical aids evolved into modeling languages and have become quite mature by now. The fact that we can define algorithms such as the ones discussed in this paper, which compile business process models automatically into executable code, makes it clear that these business process model languages are – or can be made – precise, complete and formal enough, although there is no formally defined semantics for UML 2.0 yet.

Thus, why not develop formally defined semantics for these modeling languages – if not yet available – together with execution engines that allow deployment and execution of business models directly [21]? If a model contains all information needed to simulate it and to compile it automatically into orchestration languages, an appropriate execution engine should also be able to run it. Efficiency may be a problem, and there are other open issues, but they may be solved in the future. Thus, algorithms to transform unstructured cycles into structured loops may in the long run disappear or only remain needed for analysis and validation purposes, and orchestration languages such as BPEL4WS may turn out to be only an interim step on the way to directly deployable and executable models.

Acknowledgments

We thank the anonymous reviewers for their encouraging comments and valuable suggestions, and Jochen Küster, Shane Sendall, Markus Stolze and Michael Wahler for their advice, which helped to improve this paper significantly.

References

1. OMG: Model-Driven Architecture (MDA). http://www.omg.org/mda/.
2. Uhl, A.: Model Driven Architecture Is Ready for Prime Time. IEEE Software 20(5), September/October 2003, pp. 70-73.
3. Ambler, S.: Agile Model Driven Development Is Good Enough. IEEE Software 20(5), September/October 2003, pp. 71-73.
4. W3C: Web Services Activity. http://www.w3.org/2002/ws/.
5. OASIS: Business Process Execution Language for Web Services (BPEL4WS) 1.1. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/. May 5, 2003.
6. W3C: Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl. March 15, 2001.
7. OMG: Unified Modeling Language 2.0. http://www.omg.org/uml/.
8. BPMI: BPMN 1.0 working draft. http://www.bpmi.org/.
9. White, S.: Process Modeling Notations and Workflow Patterns. In The Workflow Handbook 2004. Fischer, L. (Ed.). Future Strategies Inc., Lighthouse Point, FL, USA, 2004.
10. Aho, A. et al.: Compilers. Principles, Techniques, and Tools. Addison-Wesley, 1986.

11. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. Report of 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture, Anaheim, California, October 2003. http://www.softmetaware.com/oopsla2003/czarnecki.pdf.
12. Karsai, G., Agrawal, A.: Graph Transformations in OMG's Model-Driven Architecture. Proc. Applications of Graph Transformations with Industrial Relevance, Charlotsville, Virginia, September 2003.
13. Heckel, R. et al.: Towards Automatic Translation of UML Models into Semantic Domains. Proc. APPLIGRAPH Workshop on Application of Graph Transformation (AGT 2002), Grenoble, France, April 2002, pp. 11-22.
14. Wess, B.: Optimizing Signal Flow Graph Compilers for Digital Signal Processors. Proc. 5th International Conference on Signal Processing Applications and Technology, Dallas, Texas, October 1994.
15. Koehler, J., Hauser, R.: Untangling Unstructured Cyclic Flows - A Solution based on Continuations. Submitted for publication, 2004.
16. Ammarguellat, Z.: A Control-Flow Normalization Algorithm and Its Complexity. Software Engineering 18(3), pp. 237-251, 1992.
17. Hecht, M.S., Ullman, J.D.: Flow Graph Reducibility. SIAM J. Comput. 1(2), pp. 188-202, 1972.
18. Gardner, T. et al.: A Review of OMG MOF 2.0 Query / Views / Ttransformations Submissions and Recommendations Towards the Final Standard. Workshop on MetaModelling for MDA, York, England, November 2003.
19. OMG: Meta Object Facility 2.0. http://www.omg.org/docs/ad/03-04-07.pdf.
20. Warmer, J., Kleppe, A.: The Object Constraint Language – Second Edition. Getting Your Models Ready for MDA. Addison-Wesley, 2003.
21. Rumpe, B.: Executable Modeling with UML. A Vision or a Nightmare? In: Issues & Trends of Information Technology Management in Contemporary Associations, Seattle. Idea Group Publishing, Hershey, London, pp. 697-701. 2002.