Untangling Unstructured Cyclic Flows - A Solution Based on Continuations

Jana Koehler Rainer Hauser

IBM Zurich Research Laboratory
CH-8803 Rueschlikon
Switzerland
email: {koe | rfh}@zurich.ibm.com

Abstract. We present a novel transformation method that allows us to map unstructured cyclic business process models to functionally equivalent workflow specifications that support structured cycles only. Our solution is based on a continuation semantics, which we developed for the graphical representation of a process model. By using a rule-based transformation method originally developed in compiler theory, we can untangle the unstructured flow while solving a set of abstract continuation equations. The generated workflow code can be optimized by controlling the order in which the transformation rules are applied.
We then present an implementation of the transformation method that directly manipulates an object-oriented model of the Business Process Execution Language for Web Services BPEL4WS. The implementation maps abstract continuation equations to the BPEL4WS control-flow graph. The transformation rules manipulate the links in the graph such that all cycles are removed and replaced by equivalent structured activities. A byproduct of this work is that, if a continuation semantics is adopted for BPEL4WS, its restriction to acyclic links can be dropped.

1 Introduction

Unstructured cycles in business process modeling usually cause hot debates. Do business consultants and customers really need to express cyclic business process flows? What do they try to express and specify with these cycles? Isn't it the case that different people interpret these cycles differently and that this is not good? Isn't a good business consultant able to resolve these problems when reviewing the process model with the customer and map it to a process model that has controlled, well-structured cycles only?

We do not know the best answer to all these questions and we can easily imagine that different needs and points of view may lead to very different answers. Rather, we are interested in the technical problems behind the discussion:

- Is there a formal semantics for graphically represented business process models containing unstructured cycles, which facilitates their transformation into a structured representation?
- Given a business process model containing unstructured cycles, can it be transformed into an equivalent specification in the Business Process Execution Language for Web Services (BPEL4WS) [1] that supports only structured cycles?

An answer to these questions is important for our work, where we investigate the suitability of graphical business process models as a means for requirement specification and develop methods that allow us to automatically generate executable workflow code from such models. On the one hand, we are interested in models that allow users to express business requirements without being constrained by the limitations of IT systems. On the other hand, we need automatic algorithms that can transform such models into performant code tailored to a specific IT platform.

In this paper, we describe a method that we developed to synthesize BPEL4WS code from business process models containing unstructured cycles. Section 2 introduces an example of an electronic purchasing process that contains unstructured cycles. A *continuation semantics* is proposed to capture the intended meaning of the cycles. Section 3 presents an efficient rule-based transformation method originating from compiler theory that takes a model with unstructured cycles and transforms it into a *functionally equivalent* model with structured cycles only. Section 4 discusses the possibilities to optimize the generated workflow code by controlling the application order of the transformation rules. In Section 5, we discuss how this transformation method can be implemented as an update transformation that manipulates an initially invalid BPEL4WS model. We conclude in Section 6 with a summary and outlook on future work.

2 Unstructured Cyclic Flows

We start with the graphical representation of a business process model that describes the possible flow of activities by adopting a UML Activity Diagram-like notation [2]. The choice of the representation language does not matter as long as we can assign the semantics to its graphical elements that we introduce below. Figure 1 shows the example of an electronic purchasing business process, which we will use throughout this paper. The process describes how a user buys products via an online purchasing system.[1]

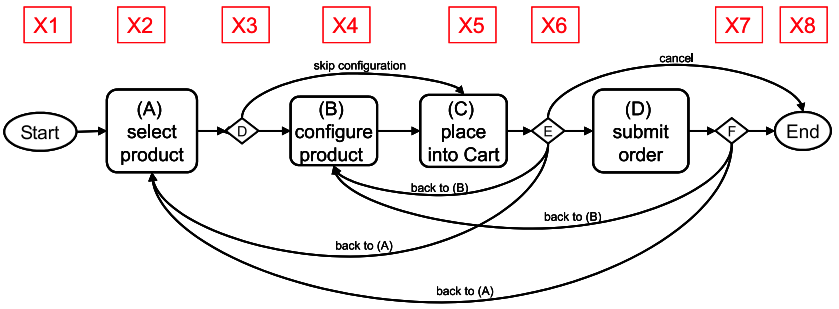

Fig. 1. Purchasing business process showing unstructured cycles.

[1] The role of the boxed variables, which are vertically aligned with selected nodes in the process model, will become clear in the next section.

Once the process has started, activity *(A) select product* is executed. After the *select product* activity has been completed, the process branches. The user can either decide to configure the product executing activity *(B) configure product* or to place the product directly into the shopping cart using activity *(C) place into Cart*. Note that we consider a nonconcurrent process model in which the branching is *exhaustive* and *disjoint*, i.e. after each decision exactly one of the possible branches is selected. After these activities have been completed, the user submits the order by executing activity *(D) submit order*. This sequence of activities describes the "normal" purchasing process. For a successful implementation, however, this process must allow the user to navigate freely between the various activities. For example, after a product is placed into the cart, the user may want to revisit its configuration and perhaps change it. Furthermore, a user may want to select several products before submitting an order. After an order has been submitted, the user may also want to revisit the configuration of the ordered product and/or change the set of selected products. Finally, a user may want to delay or cancel the placement of an order and leave the process without executing the *submit order* activity. This freedom in the process execution is described by the various back links from decisions E and F to one of the possible activities A or B.

The example illustrates that arbitrary, unstructured cycles may easily occur in the graphical representation of business processes. *Unstructured* cycles are characterized by *more than one entry or exit* point. Consider the example above and the cycle containing A, B, C, and D. This cycle can be entered in A by coming from *Start*, E, or F. It can also be entered in B by coming from E or F, and left via F and E. These multiple entry and exit points are the characteristic features of unstructured (sometimes also called wild or arbitrarily nested) cycles. In contrast to unstructured cycles, a *structured* cycle has *exactly one entry and one exit* point. On the one hand, unstructured cycles have even been identified as a pattern that frequently occurs in a business process model [3]. On the other hand, they are often the source of semantic problems [4], which explains why commercial workflow systems usually only implement structured cycles.

2.1 Continuation Semantics for Unstructured Graphical Flows

In order to transform a business process model with unstructured cycles into workflow code, which supports structured cycles with uniquely defined entry and exit points only, we assign a *continuation semantics* to the graphical model. Continuation semantics is a special form of a denotational semantics for programs with jumps. It has its origins in the Theory of Computation, where it has been discussed extensively in the context of functional and imperative languages [5]. A *continuation* describes "the rest of the program that has yet to be evaluated".

> *The key to achieving such a semantics is to make the meaning of every command a function whose result is the final result of the entire program, and to provide an extra argument to the command meaning, called a* continuation, *that is a function from states to final results describing the behavior of the "rest of the program" that will occur if the command relinquishes control.* [6], page 116.

The continuation semantics partitions the graphical flow into the past, present, and future and allows us to describe the intended execution of a process model. For example, given the activity A, we consider A the present state of the process, Start as its past and B or C as its future. We developed a method that assigns a continuation semantics to graphical models describing sequential flows. First, we assign continuation variables to the Start and End nodes as well as to all other nodes, in which sequential flows branch or merge, i.e., each activity or decision node in the flow that has more than one incoming or outgoing link is assigned a continuation variable. The resulting assignment is shown in Figure 1, which vertically aligns the boxed continuation variables with their corresponding nodes in the flow model.

Second, we have developed a method that allows us to extract *continuation equations* from the graphical process model. On the left-hand side of the equation symbol, we put the continuation variable that we consider the present. On the right-hand side of the equation, we describe the possible continuations that can follow this variable. A continuation can either be another variable or it can be an activity, which we denote by *invoke A*, *invoke B*, etc. A linear continuation can be described using the sequence operator ";". A branching of the continuation is described using a conditional statement *if* ⟨*condition*⟩ *then x*. Each link leaving a decision node in the process model is mapped to a branching. The ⟨*condition*⟩ can be derived from the process model if its graphical representation is annotated by branching conditions for the decision nodes. We introduce fresh Boolean variables to capture these conditions, but abstract from any concrete value in the following. For example, the condition that drives the continuation from process step A to process step B is denoted by the variable AB, the condition to continue from A to C is denoted by AC. Once a continuation variable has been added to each branch at the right-hand side of an equation, this equation is complete and a new equation begins. For the example under consideration, we obtain the following eight equations.

(1) x_1 = Start; x_2;
(2) x_2 = invoke A; x_3;
(3) x_3 = if AB then x_4;
if AC then x_5;
(4) x_4 = invoke B; x_5
(5) x_5 = invoke C; x_6
(6) x_6 = if CD then invoke D; x_7
endif;
if CEnd then x_8;
if CA then x_2;
if CB then x_4;
(7) x_7 = if DB then x_4;
if DA then x_2;
if DEnd then x_8;
(8) x_8 = End ;

The ordering of the conditional statements in the equations is arbitrary, because we consider nonconcurrent business process models with exhaustive and disjoint branching that do not need to specify an explicit ordering in which the branches are tried.

3 Transformation Method

We are now in the position to answer our second question: can a graphical model containing unstructured cycles be mapped into an equivalent program permitting structured

cycles only? The answer was given almost forty years ago [7]: any concurrent or sequential *flow diagram* can be translated into a functionally equivalent program containing a single while-loop and new conditional statements. Unfortunately, the proof in [7] is nonconstructive, i.e., it does not give us a method of computation. Soon after this fundamental result, the problem of transforming unstructured loops into a well-structured form became known as the *GOTO-elimination problem* in compiler theory. Several algorithmic solutions, which permit an arbitrary number of well-structured loops but also focus on the optimization of the transformed code [8–10], have been developed based on the famous T1-T2 transformations [11]. We describe the application of these techniques to the problem of business-to-IT model transformation in more detail in the next section.

3.1 Solving the System of Equations

Our transformation method is based on the transformation rules presented in [8]. Whereas Ammarguellat presents her transformation rules using a Lisp-like notation, we have developed a representation based on the abstract mathematical equations introduced above, from which implementations for very different model representations can be easily derived. A derived implementation, which works on an object-oriented model of the Business Process Execution Language for Web services [1] is discussed in the second part of this paper. The soundness of the rules follows from the observation that each of them preserves the possible continuations in the encoded flow model.

Substitution: This rule reduces the number of variables and thereby also the number of equations. Given the occurrence of a variable on the right-hand side of an equation, it replaces this variable with its defining equation.

$$\begin{array}{l} x_0 = \textit{invoke } A; \boxed{x_1}; \\ \boxed{x_1} = \textit{invoke } B; \end{array} \quad \longrightarrow \quad \begin{array}{l} x_0 = \textit{invoke } A; \\ \qquad \textit{invoke } B; \end{array}$$

Factorization: This rule is applied to a continuation equation that contains several disjoint and exhaustive branches, which all lead to the same continuation x. The multiple occurrences of x are replaced by a single occurrence of x at the end of the equation, which is guarded by a new Boolean condition assembled from the governing conditions of the various branches. Fresh variables have to be used to capture the "state" of governing conditions in the case that different executions of the flow can modify their value in different ways, cf. [12] for more details. In the following, we will omit these variables in order to keep the example transformations more easily readable.

$$\begin{array}{l} x_0 = \textit{invoke } A; \\ \quad \textit{if } c \textit{ then invoke } B; \boxed{x_1} \\ \quad\quad \textit{else if } d \textit{ then } \boxed{x_1}; \\ \quad \textit{endif}; \end{array} \quad \longrightarrow \quad \begin{array}{l} x_0 = \textit{invoke } A; \\ \quad \textit{if } c \textit{ then } \textit{invoke } B; \\ \quad \textit{pred} := c \vee (\neg c \wedge d); \\ \quad \textit{if } \textit{pred} \textit{ then } \boxed{x_1}; \end{array}$$

Derecursivation: This rule eliminates cycles. It is applied to equations that mention the same continuation variable x at their left-hand and right-hand sides. The occurrence

of x in the right-hand side is eliminated by a repeat-while statement ranging from the beginning of the right-hand side until x occurs. The termination condition for the loop is obtained from the conditions on the execution path that leads to the continuation variable. This rule can be applied if no other continuation variables occur between the equation sign and the recursive continuation variable. Otherwise, the continuations have to be reordered first using a variant of the if-distribution rule below.

$$\boxed{x_0} = \textit{invoke A};\ \textit{if } c \textit{ then } \boxed{x_0}; \longrightarrow x_0 = \textit{repeat}\ \textit{invoke A};\ \textit{while } c;$$

If-Distribution: This rule rewrites nested branching continuations into a sequence of branches that can be arbitrarily ordered. This rule may occur in many different forms. A variant used in this paper is shown below:

$$x_0 = \textit{if } \boxed{c1} \textit{ then } x_1\ \textit{else if } \boxed{c2} \textit{ then } x_2;\ \textit{endif}; \longrightarrow x_0 = \textit{if } \boxed{\neg c1 \wedge c2} \textit{ then } x_2;\ \textit{if } c1 \textit{ then } x_1;$$

These rules are maintained and organized by a transformation engine that operates in the following steps:

1. Select a rule that is applicable to an equation;
2. Apply the rule and compute the modified set of equations;
3. Goto step 1 until only a single equation remains in the set.

3.2 Solving the Example

In the following, we describe how the example equations are solved. The order in which the rules are selected for application determines the quality of the generated workflow code. For our purposes, we developed various application orders that enable our transformation engine to produce code of different quality. We discuss our optimization heuristics in Section 4.

Pass 1: Only the substitution rule is applicable. The derecursivation rule is not applicable, because no equation contains the same variable on both sides. The factorization rule is not applicable, because no equation contains multiple occurrences of the same continuation variable on the right-hand side. The transformation engine decides to apply the substitution rule to variable x_3 in Equation (2), then to variable x_6 in Equation (5), and finally to variable x_7 in the (transformed) Equation (5).

(1) x_1 = Start; x_2;

(2) x_2 = invoke A;
if AB then x_4;
if AC then x_5;

(4) x_4 = invoke B; x_5

(8) x_8 = End;

(5) x_5 = invoke C;
if CD then invoke D;
if DB then x_4;
if DA then x_2;
if DEnd then x_8;
endif ;
if CEnd then x_8;
if CA then x_2;
if CB then x_4;

Pass 2: The transformation engine works on the complex Equation (5) by applying the factorization rule to the continuation variables x_2, x_4, x_8, which each occur twice on the right-hand side of this equation. Then, the variable x_8 is eliminated by substituting Equation (8).

(5) x_5 = invoke C;
if CD then invoke D;
if CA ∨ (CD ∧ DA) then x_2;
if CB ∨ (CD ∧ DB) then x_4;
if CEnd ∨ (CD ∧ DEnd) then End;

Pass 3: The variable x_4 is substituted in Equations (2) and (5). Then, multiple occurrences of x_5 in Equation (2) are eliminated by applying the factorization rule again.

(2) x_2 = invoke A;
if AB then invoke B;
if AB ∨ AC then x_5;

(5) x_5 = invoke C;
if CD then invoke D;
if CA ∨ (CD ∧ DA) then x_2;
if CB ∨ (CD ∧ DB) then invoke B; x_5
if CEnd ∨ (CD ∧ DEnd) then End;

Pass 4: The transformation engine eliminates the recursion in Equation (5). Variable x_2 occurs inside the continuation that the repeat-while loop will spawn, i.e., x_2 has to be moved using if-distribution prior to creating the loop such that it succeeds x_5.

(5) x_5 = repeat
invoke C;
if CD then invoke D;
if CB ∨ (CD ∧ DB) then invoke B;
while CB ∨ (CD ∧ DB);
if CA ∨ (CD ∧ DA) then x_2;
if CEnd ∨ (CD ∧ DEnd) then End;

Pass 5: Variable x_5 is substituted in Equation (2).

(2) x_2 = invoke A;
if AB then invoke B;
if AB ∨ AC then
repeat
invoke C;
if CD then invoke D;
if CB ∨ (CD ∧ DB) then invoke B;
while CB ∨ (CD ∧ DB);
if CA ∨ (CD ∧ DA) then x_2;
if CEnd ∨ (CD ∧ DEnd) then End;
endif;

Pass 6: The transformed Equation (2) is recursive. Variable x_2 occurs inside the conditional branch governed by AB ∨ AC, which would be incorrectly interrupted if derecursivation were applied immediately. Therefore if-distribution is applied first to rearrange the branching continuations. The transformed Equation (2) is inserted into Equation (1)

to replace the last remaining occurrence of x_2. These last transformation steps solve the equational system. Only a single equation defining the variable x_1 is left, which contains no other continuation variables on its right-hand side.

```
(1)  x1 = Start;
       repeat
           invoke A;
           if AB then invoke B;
           if AB ∨ AC then
                       repeat
                           invoke C;
                           if CD then invoke D;
                           if CB ∨ (CD ∧ DB) then invoke B;
                       while CB ∨ (CD ∧ DB);
           endif;
       while ( AB ∨ AC) ∧ (CA ∨ (CD ∧ DA));
       if ( AB ∨ AC) ∧ (CEnd ∨ (CD ∧ DEnd)) then End;
```

Any applied transformation rule preserves the possible continuations of the process model. The flows described by the business process model and the flow described by the remaining equation (or any intermediate form of the equation set) are functionally equivalent, i.e., when invoked on the same input, both flows will produce exactly the same output.

4 Optimizing the Generated Workflow

We have developed two techniques to further simplify and optimize the generated workflow code, which we describe in the following:

1. The resulting normalized process model, in particular the governing conditions, can be simplified by exploiting the fact that the branching is disjoint and exhaustive.
2. The transformation engine can influence various structural properties of the generated code by applying the transformation rules in a specific order.

4.1 Simplifying the Normalized Process Model

Several of the governing transitions in the solved equation form a tautology, because they describe all possible paths to reach a particular continuation. Activity C has to be executed in any possible execution path. It will either follow activity A directly or it will follow activity B, but it cannot be skipped. This means that (AB ∨ AC) is a tautology, because the transitions to B or C are the only ones possible from A. Thus, the condition that governs the inner loop is unnecessary. From B, only a single, unguarded transition to C is possible. The same argumentation applies to (CEnd ∨ (CD ∧ DEnd)). It follows that the condition governing the reachability of End can be skipped.

Furthermore, the Start and End activities can be removed from the equational representation, because they do not describe business-relevant data manipulations. In the flow

model, these nodes indicate where the business process starts and ends. They determine the initial entrance into the flow and how the continuation equations are systematically built from the graphical model. The variable assigned to the start node also determines which continuation variable is left after the equational system has been solved. The result of these simplification steps is:

x_1 = repeat
    invoke A;
    if AB then invoke B;
    repeat
        invoke C;
        if CD then invoke D;
        if CB ∨ (CD ∧ DB) then invoke B;
    while CB ∨ (CD ∧ DB);
while CA ∨ (CD ∧ DA);

We observe that this equation contains two properly nested loops. The inner loop captures the forward and backward flow between the activities B, C, and D. The outer loop captures the backward flow to activity A from either C or D.

4.2 Controlling Rule Application Order

The second opportunity for optimizing the generated code lies in computing the "correct" order for the application of rules by the transformation engine. We note that each transformation rule guarantees that the transformation will *terminate*, but the rule application is *not confluent*, i.e., different application orders produce syntactically different transformation results. The method in [8] uses a topological sorting of the nodes in the control-flow graph to determine the order in which variables should be eliminated from the equations. We found this method to be insufficient for our purposes. Instead, we developed a control scheme that keeps information about how often variables occur on the right-hand side of the equations. We also added rule priorities. Factorization has a higher priority than derecursivation. Derecursivation is only applied directly prior to a substitution step or as a last step of the transformation if the remaining equation is recursive. If-distribution is only applied if required, which happens in two situations: First, to move any continuation of the flow towards the end activity to the very end of an equation. Second, to move continuation variables outside the scope of applicability of the derecursivation rule. To explain how the rule application order is controlled, let us revisit the example.

In the first pass, only the substitution rule was applicable. The following occurrences of continuation variables on the right-hand side of the equations are counted: $x_2 = 3, x_3 = 1, x_4 = 3, x_5 = 2, x_6 = 1, x_7 = 1$, and $x_8 = 2$. We note that the variables x_3, x_6, x_7 occur only once. Whenever such single-occurrence variables exist, the transformation engine will apply the substitution rule to eliminate them from the equation set. In the second pass, the factorization rule was applicable, because the variables x_2, x_4, and x_8 occured twice in the same right-hand side of an equation. Owing to its higher priority, this rule was applied. Then the substitution rule was considered

again, controlled by the occurrence of the continuation variables: $x_2 = 2, x_4 = 2, x_5 = 2, x_8 = 1$. Only the variable x_8 occurs a single time and thus the substitution rule was applied to it.

For the third pass, all remaining continuation variables occur exactly two times. Only the substitution rule is applicable. The transformation engine has no unique choice to continue. This phenomenon reflects the fact that the flow graph encoded in the business process model is *non-reducible*, which is a widely studied phenomenon in compiler theory [13]. Using the transformation rules from [8], code duplication is unavoidable, e.g., variable x_4 is substituted in Equations (2) and (5) in Pass 3. In contrast to programming languages, non-reducibility of the underlying flow graph seems to be quite common for business process models. To avoid code duplication for such non-reducible flows, we have developed an alternative code generation method that synthesizes a state-machine encoded in BPEL4WS, cf. [12].

The transformation engine selects the variable that occurs the minimum number of times. If no such choice exists (as is the case in our example), it selects the variable that has the *smallest* right-hand side in its equation. *Small* can be defined in different ways depending on the goal of the code optimization. It can be the number of *invoke* statements, the number of conditions tested or any other user-defined criterion or combination thereof. In our case, we try to minimize the number of *invoke* statements followed by the number of tested conditions, because we want to minimize the number of Web service invocations generated for the workflow code, and we want to keep the branching logic as simple as possible. Consequently, the transformation engine selects variable x_4 in the third pass. Eliminating x_4 transforms Equation (5) into a recursive equation and thus, in Pass 5, the derecursivation rule is applied. It requires applying the if-distribution rule first, because another continuation variable occurs in the scope for applying this rule. In Pass 5, the only variables left are x_2 (which occurs twice) and x_5 (which occurs once). Consequently, x_5 is substituted first. In Pass 6, derecursivation preceded by if-distribution is applied because of the higher rule priority. Finally, in Pass 7, a last application of the substitution rule is possible.

4.3 Mapping to BPEL4WS

The single equation computed by the transformation engine contains only two well-structured cycles in the form of repeat-while statements as well as a few conditional branches. It can be directly mapped to an XML representation of the standardized language BPEL4WS. Each invocation of an activity is mapped to the invocation of a Web service. A repeat-while loop is mapped to a while-do loop combined with an assignment:

```
<sequence>
  <assign newcondition := true />
  <while  newcondition>
    <assign newcondition := condition/>
  </while>
</sequence>
```

A conditional statement is mapped to a `<switch>`:

```
<switch>
  <case condition = guard-expression/>
</switch>
```

We show an abstract specification in simplified BPEL4WS syntax that defines the control-flow for the workflow, but omits all details that relate to partners, messages, and Web services as well as fresh variables that may have been introduced during the transformation to capture the values of guard conditions.

```
<process>
  <sequence>
    <assign cond1: = 'true'/>
    <while cond1>
      <sequence>
        <invoke A/>
        <switch>
          <case condition = 'AB'>
            <invoke B>
          </case>
        </switch>
        <assign cond2 := 'true'/>
        <while cond2>
          <sequence>
            <invoke C/>
            <switch>
              <case condition = 'CD'>
                <invoke D/>
              </case>
            </switch>
            <switch>
              <case cond = '(CD & DB) or CB'/>
                <invoke B>
              </case>
            </switch>
            <assign cond2 := '((CD & DB) or CB)'/>
          </sequence>
        <while/>
      </sequence>
    </while>
      <assign cond1 := '(((CD & DA) or CA) & (AB or AC))'/>
  </sequence>
</process>
```

The XML representation can also be graphically displayed by mapping it, for example, to the UML Profile for BPEL [14], which is sketched in Figure 2.

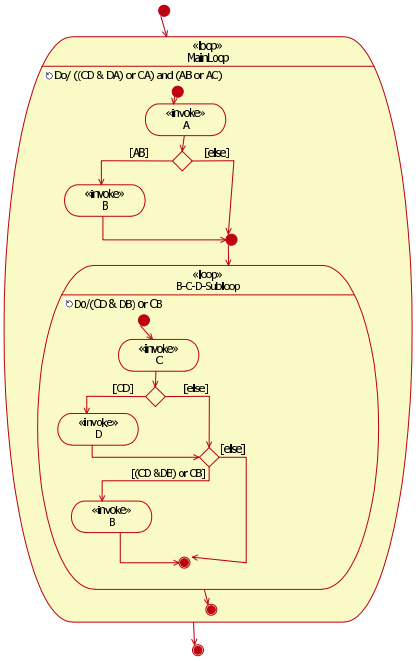

Fig. 2. Resulting BPEL4WS diagram.

5 Working Directly on a BPEL4WS Model

Based on the abstract description of the transformation rules, very different implementations can be imagined. First, one can refactor the graphical business process model, e.g., by replacing cyclic links with appropriate loop nodes in UML 2. The advantage of this approach is that different, but equivalent views on the process model can be offered to the modeler and that a view of the process model is available, which is structurally very similar to the generated code. The disadvantage lies in the need to add many additional variants to the four basic transformation rules that deal with the various graphical modeling elements. Second, one can map an unstructured cyclic flow directly to an executable specification in some workflow language, which supports unstructured cycles or refactor the workflow specification until it contains only structured cycles. In the following, we discuss an implementation that maps an unstructured cyclic business process model directly into BPEL4WS. The process model can also contain concurrency, which may be introduced by fork or join activities in UML 2, for example.

Figure 3 gives a more complete overview of the BPEL4WS language in the form of an object-oriented model, which we have developed. We adopt the view of a model as a containment hierarchy, starting with one root object, where each node may have any number of property settings in the form of name-value pairs. A property value is either a simple data value or a reference to another object in the same model. A convenient way to represent such models is the UML class diagram [2]. Objects are represented

as classes, properties are represented as attributes, and references to other objects that express non-simple values can be expressed with the help of associations.

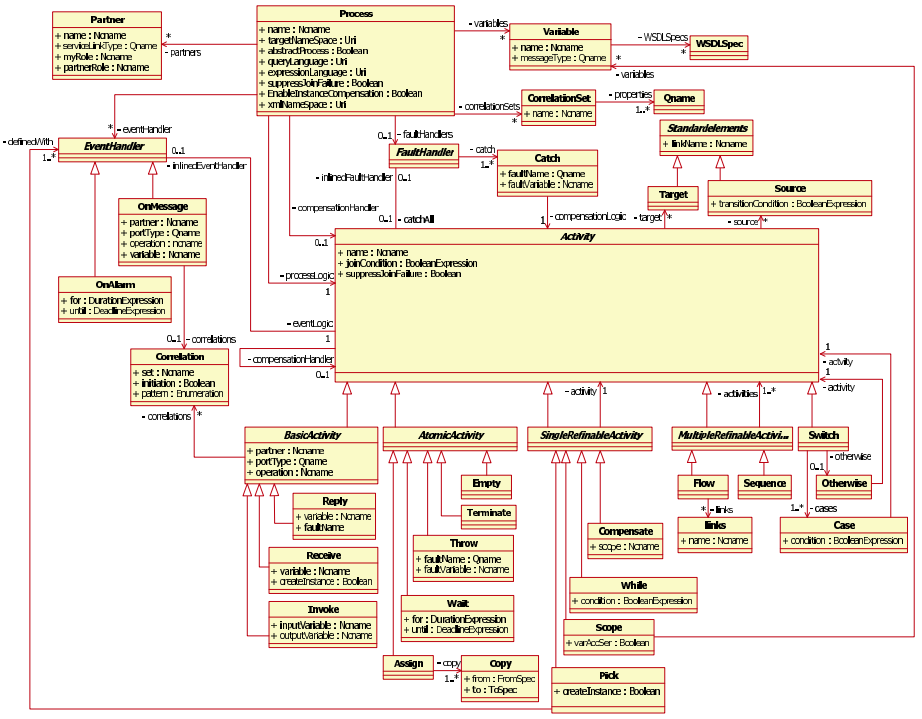

Fig. 3. Class diagram for the BPEL4WS language.

Some details of the BPEL4WS language have been omitted from this class diagram: We have not further refined the definitions of the types used for the *From* and *To* attributes, which we simply called *FromSpec* and *ToSpec*. Similarly, the types *Ncname*, *Qname*, *BooleanExpression*, *DurationExpression*, and *Deadline Expression* remain undefined. We do not (yet) care about the visibility of associations, which is simply set to *private*. Furthermore, we do not make explicit whether certain associations are aggregations or compositions. Some attributes may not occur together. For example, the *for* and *until* attributes of a *Wait* activity occur exclusively. We have again abstracted from this detail and assume that in this case, an attribute may be set to *unknown* or the Object Constraint Language OCL [15] would be used to add constraints to our model. As for associations, we have not made a distinction between whether a set of associated classes is ordered or unordered, which distinguishes a *sequence* activity from a *flow*. The multiplicity of the associations has been derived from the BPEL schema definitions. For example, a process may define 0 to n event handlers. However, if the *EventHandler* element is used in the XML definition, at least one *OnMessage* or *OnAlarm* handler

should be present. Again, this is best expressed with the help of OCL constraints that are added to the class diagram.

5.1 Encoding Continuations in BPEL4WS

The continuations in the process model can be alternatively represented as a control-flow graph. Figure 4 illustrates this representation for Equations (4) to (8). Each equation corresponds to a subtree contained in the graph. The root node of each tree encodes the continuation variable that occurs on the left-hand side of the equation. Each statement occurring on the right-hand side of the equation is mapped to a child node. Solid edges represent the possible continuations. A path from the root to a leaf node encodes a sequential continuation. Several branching child nodes of the same node encode conditional continuations. An edge from a node to one of its children can be annotated with the variable encoding the transition condition. For the example we consider here, these edges denote alternative continuations and reflect the exhaustive and disjoint branching that we postulated for the business process model. Dashed edges encode continuations that link the various subtrees with each other. However, concurrent flows can be easily captured in an AND-OR tree and graph, respectively.

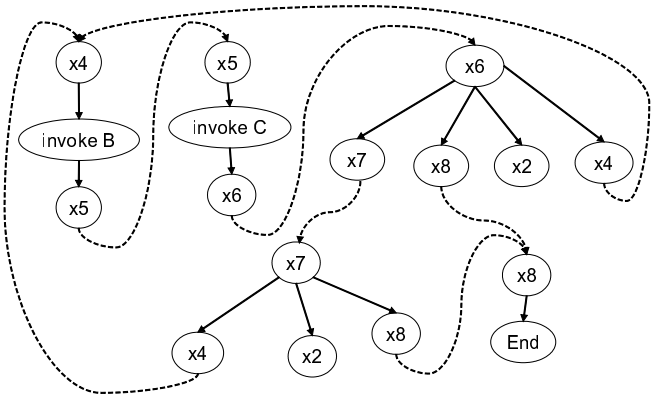

Fig. 4. Forest of trees capturing the semantics of continuation equations.

The encoding in BPEL4WS works as follows: The control-flow graph is mapped to a BPEL4WS `flow` containing a sequence for each subtree. Tree nodes, which contain continuation variables, are mapped to `empty` BPEL4WS activities. Tree nodes, which contain activity invocations, are mapped to `invoke` activities. Depending on the semantics of Start and End nodes in the business process model, these nodes are either mapped to `empty` or `invoke` activities. The names of the `empty activities` are set to the names of the continuation variables they encode, the names of the invoke activities are set to the names of the activities. The edges between the tree nodes are mapped to `links` and the activities define whether they are the `source` or `target`

of a link. The transition conditions are captured in the `transitionCondition` attribute of a `source` element. If a node has more than one child node, another `flow` is introduced. Alternatively, we could map these alternative, nonconcurrent branches to a `switch`, but in using a `flow` we adopt a unique encoding for all edges and can immediately capture concurrent branching. The example encoding for the control flow that was captured in Equations (4) and (6) is sketched below.

```
<flow>
  <links>
    <link name='x2link'>
    <link name='x3link'>
    <link name='x4link1'>
    <link name='x4link2'>
     ...
  </links>

  <sequence>
    <empty name='x4'/>
      <target linkName='x4link1'/>
      <target linkName='x4link2'/>
    </empty>
    <invoke name='B'/>
    <empty name ='x5'>
      <source linkName='x5link'/>
    </empty>
  </sequence>

  <sequence>
    <empty name='x6'/>
      <target linkName='x6link'/>
      <source linkName='x6-x7link'
         transitionCondition='CD'/>
      <source linkName='x6-x8link'
         transitionCondition='CEnd'/>
      <source linkName='x6-x2link'
         transitionCondition='CA'/>
      <source linkName='x6-x4link'
         transitionCondition='CB'/>
    </empty>
    <flow>
      <empty name='x7'>
        <target linkName='x6-x7link'/>
      </empty>
      <empty name='x8'>
        <target linkName='x6-x8link'/>
      </empty>
      <empty name='x2'>
        <target linkName='x6-x2link'/>
      </empty>
      <empty name='x4'>
```

```
        <target linkName='x6-x4link'/>
      </empty>
    </flow>
  </sequence>
</flow>
```

We could almost hand over this BPEL4WS specification to a BPEL4WS engine. Unfortunately, our process definition violates a major constraint of the specification, namely that the `links` must form an acyclic graph. As we can see in Figure 4, the encoding links form a control cycle.

We observe that the continuation semantics paves the way to allowing cyclic links between activities in BPEL4WS, because the semantics clearly defines which instructions in the BPEL4WS program should be executed next, and rescheduling activities for another execution has a clear meaning. This is in contrast to the current semantics of BPEL4WS links, where cyclic links cause activities to wait for each other to complete execution, while none of them can start. Although the current limitation in the specification can be overcome and an execution semantics for cyclic BPEL4WS flows is within reach, such an extension would make it easy to write BPEL4WS programs that contain runtime errors such deadlocks and livelocks, which require verification technology for detection.

5.2 Transforming the BPEL4WS Model

The cyclic, concurrent BPEL4WS model can be transformed into valid acyclic BPEL 1.1 code if all control cycles encoded in the links are sequential and properly nested within a single concurrent execution branch, i.e., there should be no control link from one sequential cycle to another running in a different concurrent thread. Each sequential cycle can be transformed by replacing cyclic links with appropriate BPEL4WS `while` activities. However, no links are allowed between two different `while` activities in the BPEL4WS specification, i.e., the language forbids any form of synchronization of concurrent cyclic processes in order to avoid problems of possible deadlocks or livelocks, etc. We do not propose the sequentialization of concurrent BPEL4WS as a possible solution to transform unstructured concurrent cycles, although it is a theoretical possibility [7], because we do not consider it to be practically relevant. Instead, the direct execution of cyclic BPEL4WS, as sketched above, seems to make more sense.

The transformation rules can be reformulated as a manipulation of links and their sources and targets.

Substitution works on `empty` activities that are the target of exactly one link. Consider the trees for Equations (4) and (5). The substitution deletes the root node x_5 of tree (5) and the leaf node x_5 of tree (4). A new link (or associated activity in our class model) is created from the parent node of the deleted leaf node in tree (4) (`invoke B`) to the child node of the deleted root node (`invoke C`) in tree (5).

Factorization is applied to trees (BPEL4WS `flows`) that contain multiple occurrences of an `empty` activity with the same name. No additional Boolean variables are required, but instead the transition conditions are assembled from the links when multiple occurrences of the same node are merged. Consider the substitution of x_7, for example. This requires joining the transition condition *CD* of the link to x_7 with the transition

conditions *DB*, *DA*, *DEnd* of the links from x_7. We obtain $CD \wedge (DB \vee DA \vee DEnd)$, which is transformed into disjunctive normal form and leads to three new links with transition conditions $CD \wedge DB$, $CD \wedge DA$, and $CD \wedge DEnd$, which replace the old links. Multiple paths to the same leaf node can be merged by disjunctively joining their transition conditions. For example, when merging the two `empty` activities labeled x_8, we obtain the new transition condition $CEnd \vee (DEnd \wedge CD)$. Figure 5 illustrates this transformation.

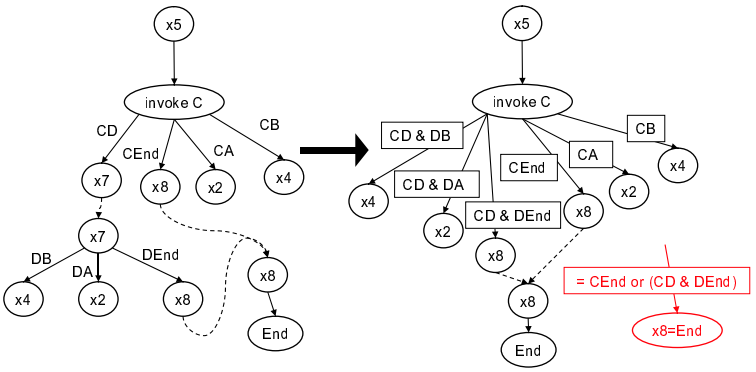

Fig. 5. Factorization working on links.

Derecursivation directly introduces a new `while` activity instead of a repeat-while loop. It is applied to trees that contain links from an `empty` activity node back to the root of the same tree.

One can imagine using OCL [15] or any other expression language to describe the pre- and postconditions of the transformation rules by using the types, associations, and attributes of the BPEL4WS class diagram. The precondition of a rule describes when the rule is applicable to the model, whereas the postcondition describes the required update. The computation of the update, often called *model reconciliation*, is a nontrivial computation that goes beyond the focus of this paper and is the subject of our current work. Expressions in the postcondition should be limited such that the update is unique and can be computed efficiently, i.e., they must be functional. This requirement translates into restrictions on the expression language, which we are currently investigating. Furthermore, our transformation rules all have a natural inverse interpretation, although we only described them in an unidirectional and not in a bidirectional way. In the case of bidirectional transformations, the pre- and postconditions must be limited such that the reconciliation of the model is computable in both directions.

6 Conclusion

We discuss the transformation of business process models with unstructured cycles into workflow languages that support only well-structured cycles based on a continuation

semantics. We present a rule-based transformation method that works on a set of continuations that captures the semantics of cyclic models. From this abstract representation, various implementations of the transformation method, which can be tailored to different model representations, can be derived. For example, we discuss the implementation of the transformation as an update of an object-oriented model for the Business Process Execution Language BPEL4WS. A byproduct of our work is that we can show that, if a continuation semantics is defined for BPEL4WS links, the requirement of acyclicity can be dropped and executable cyclic workflows could be permitted. The small set of required transformation rules, the interesting opportunities to control the order of rule application as well as the ability to apply the rules in a bidirectional manner make this transformation particularly appealing.

References

1. Andrews, T., et al.: Business process execution language for web services. www-106.ibm.com/developerworks/webservices/library/ws-bpel/ (2002)
2. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley (1898)
3. van der Aalst, W., van Hee, K.: Workflow Management Models, Methods, and Systems. MIT Press (2004)
4. van der Aalst, W., Desel, J., Kindler, E.: On the semantics of EPCs: A vicious circle. In: Proceedings of the Workshop on EPK. (2003) 7–18
5. Reynolds, J.: The discoveries of continuations. LISP and Symbolic Computation **6** (1993) 233–247
6. Reynolds, J.: Theories of Programming Languages. Cambridge University Press (1998)
7. Böhm, C., Jacopini, G.: Flow diagrams, turing machines and languages with only two formation rules. Communications of the ACM **9** (1966) 366–371
8. Ammarguellat, Z.: A control-flow normalization algorithm and its complexity. Software Engineering **13** (1992) 237–251
9. Erosa, A., Hendren, L.: Taming control flow: A structured approach to eliminating goto statements. In: Proceedings of the International Conference on Computer Languages (ICCL, IEEE Press (1994) 229–240
10. Peterson, W., Kasami, T., Tokura, N.: On the capabilities of while, repeat, and exit statements. Communications of the ACM **16** (1973) 503–512
11. Hecht, M., Ullman, J.: Flow graph reducibility. SIAM Journal of Computing **1** (1972) 188–202
12. Hauser, R., Koehler, J.: Compiling process graphs into executable code. In: Third International Conference on Generative Programming and Component Engineering. LNCS, Springer (2004) forthcoming.
13. Aho, A., Sethi, R., Ullman, J.: Compilers–Principles, Techniques, and Tools. Addison-Wesley (1986)
14. Amsden, J., Gardner, T., Griffin, C., Iyengar, S., Knapman, J.: UML profile for automated business processes with a mapping to BPEL 1.0. IBM Alphaworks http://dwdemos.alphaworks.ibm.com/ wstk/ common/ wstkdoc/ services/ demos/ uml2bpel/ docs/ UMLProfileForBusinessProcesses1.0.pdf (2003)
15. Warmer, J., Kleppe, A.: The Object Constraint Language - Precise Modeling with UML. Addison-Wesley (1999)