Can Graph Transformation Make Aspect Languages for BPEL Redundant?

Roy Grønmo
SINTEF, Oslo, Norway
roy.gronmo@sintef.no

***Abstract*—The aspect language AO4BPEL has been introduced as a way to modularize cross-cutting concerns in Web service compositions that are specified in BPEL. AO4BPEL can be difficult to understand and to write for non-XML experts. This paper explores if algebraic graph transformation rules can be used to specify BPEL aspects at the modeling level, and make new aspect languages like AO4BPEL redundant. Three AO4BPEL examples, taken from the literature, are used to test the suitability of graph transformation as a means to simulate BPEL aspects.**

***Keywords*-Web service composition; graph transformation; aspect-oriented; BPEL; business process; workflow; UML activity model; AO4BPEL;**

I. Introduction

Aspect-orientation has been introduced as a way to modularize cross-cutting concerns in programming, modeling and specification languages. AspectJ [13] is the most famous aspect-oriented language to date. AspectJ allows programmers to specify aspects for Java programs and AspectJ syntax is based on the Java syntax. In this tradition, the AO4BPEL XML language [3] has been proposed as an aspect language to modularize cross-cutting concerns in BPEL [11]. BPEL is an XML language to specify executable Web service compositions.

An aspect language typically defines a *joinpoint* model to denote the possible structures or element types that can be matched by aspects in the target language. An aspect contains a *pointcut* to select joinpoints, and an aspect contains an *advice* to specify the action to take at the joinpoints, i.e., the so called *weaving*.

An XML-based representation can be difficult to understand and to write for non-XML experts. This is why several proposals (e.g., [21], [17]) suggest to specify a Web service composition via a graphical modeling language, e.g., by UML activity models or BPMN, instead of directly in BPEL. Then a BPEL document can be generated from such a model.

In the following we assume that the user has access to such an automated transformation from UML activity models to BPEL. This gives the opportunity to define aspects at the modeling level as an alternative to AO4BPEL, have these aspects statically woven into a model, and then generate a woven BPEL document. From the user perspective, the effect of the model weaving is the same as with weaving at the BPEL level. With a model weaving approach we can freely switch to another execution language or execution engine, without having to change our aspects.

This paper explores if the well-established algebraic graph transformation (GT) [14] can be used to express BPEL aspects at the modeling level. If the GT approach is successful, then it is questionable if there is a need to invent new aspect languages for BPEL. Instead an existing GT tool like AGG [23] can perform the weaving. AGG can also perform a confluence and termination analysis of the weaving process for a fixed set of aspects.

As our case study we examine if and how GT rules can express three AO4BPEL aspects given in a previous paper by Charfi and Mezini [3]. These three examples are intentionally chosen since they together cover the most important parts of AO4BPEL, which are the different advice and joinpoint types. AO4BPEL and GT are compared with respect to expressiveness and user-friendliness as the two main requirements. The case study shows that an aspect that inserts new behaviour *before* some existing behavior may be problematic to simulate by GT in certain situations, and we discuss how this can be solved.

The *expressiveness* requirement is about the kind of structures that can be matched by the language, which is restricted by the pointcut expressiveness and the joinpoints that are allowed. Also, we consider the advice expressiveness, i.e., the kind of structures that can be inserted and the ways that we can change the existing structures.

The *user-friendliness* requirement means that it should be easy to express the most common types of aspects, and such aspect definitions should be easy to maintain and understand. There should be a low risk that the user accidentally expresses incorrect aspects.

This paper is structured as follows. Section II describes the preliminaries about BPEL and UML activity models in a Web service composition context, and about AO4BPEL and graph transformation to express BPEL aspects. Section III investigates three AO4BPEL examples to see if they can be expressed as graph transformation rules to simulate the same effect. Section IV discusses different weave strategies and compares AO4BPEL against graph transformation with respect to the two main requirements of expressiveness and user-friendliness. Section VI describes related work. Finally, section VII concludes the paper.

II. Preliminary

This section is divided into two subsections. The first subsection describes how BPEL and UML activity models both can be used to specify a Web service composition. The second subsection introduces AO4BPEL and GT that both can be used to modularize cross-cutting concerns for a Web service composition specified by BPEL or a UML activity model respectively.

A. BPEL and UML activity models

BPEL is an XML language that is used to specify a Web service composition. The composite Web service takes some input data and produces some output data, which is reflected in its interface and captured in a WSDL description. A BPEL document describes the internal process of a Web service, which consists of control flow, data flow, data transformations and calls to other Web services.

UML 2 activity models can be used to specify the same information as described by a BPEL document. In order to use activity models, we need to define a UML profile, i.e., extensions tailored for Web service information. UML provides standard extension mechanisms as tagged values, stereotypes and OCL constraints. Tagged values for `portType`, `operation`, `WSDL` file etc. can be associated with a UML activity element to precisely denote a call to a Web service operation.

Figure 1 gives an overview of a possible mapping between BPEL and activity models. The composite Web service is represented by the outermost activity in the activity model. UML input pins of the composite Web service correspond to the receive activity in BPEL, and output pins of the composite Web service correspond to the reply activity in BPEL. A *pin* is placed on the boundary of its owner activity. We use arrows to distinguish input pins from output pins. A Web service takes a single XML document as input and returns a single XML document as output. The different parts in an input/output XML document correspond to pins, and the set of input/output pins correspond to the input/output XML document.

UML data flow from one pin/data object to another corresponds to an assign element with a nested copy element in BPEL. Pins and data objects correspond to variables in BPEL. A UML control flow corresponds to sequence in BPEL. Contained activities in the outermost UML activity correspond to internal calls to other Web services, for which BPEL uses the invoke elements.

In Figure 2 we show a travel package service as an activity model, which corresponds to a BPEL example from Charfi and Mezini [3] (the BPEL code is omitted due to space restrictions). The service takes the relevant cities (`fromCity` and `toCity`) and the travel dates as input, and returns information about possible flights and hotels. Internally, this composite Web service calls a flight service and a hotel service, sequentially one after the other. These could also

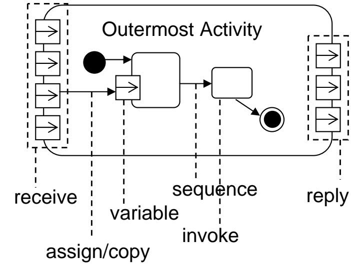


Figure 1. Mapping from UML activity model to BPEL

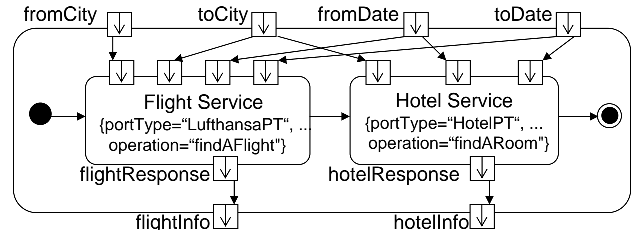


Figure 2. BPEL *travel package* service represented as a UML activity diagram

have been called in parallel, but we keep the same process as given in the original example.

In the following we assume that each activity has exactly one incoming and one outgoing control flow. This can be ensured by using explicit control flow constructs such as fork, join, decision and merge whenever necessary. Such a condition is considered good practice, and it will simplify the specification of appropriate graph transformation rules.

B. AO4BPEL and Graph Transformation

AO4BPEL has a joinpoint model that allows to match receive, reply and invoke activities in addition to messages (soapmessagein and soapmessageout). The pointcut is expressed by using the XPath language, which is a general language to select elements within XML documents.

There are three advice types (before, after and around) in AO4BPEL that correspond to those available in AspectJ. The advice type defines where to place the new advice code in relation to the matched joinpoints. The advice code is specified as pure BPEL.

An algebraic graph transformation rule is typically displayed with a left hand side graph (LHS), a right hand side graph (RHS), and a number of negative application condition graphs (NACs). When using graph transformation to simulate aspect-orientation, we should clarify the terminology. A rule (or possibly a set of rules) correspond(s) to an aspect, a LHS corresponds to a pointcut, a RHS corresponds to an advice, and transformation (a sequence of derivation steps) corresponds to weaving. Below

we provide the known formal foundation of algebraic graph transformation [14].

Definition 1: (**Graph and graph morphism**) A *graph* $G = (G_N, G_E, src, trg)$ consists of a set G_N of nodes, a set G_E of edges, two mappings $src, trg : G_E \to G_N$, assigning to each edge $e \in G_E$ a source node $src(e) \in G_N$ and target node $trg(e) \in G_N$. A *graph morphism* $f : G_1 \to G_2$ from one graph to another, with $G_i = (G_{E,i}, G_{N,i}, src_i, trg_i), (i = 1, 2)$, is a pair $f = (f_E : G_{E,1} \to G_{E,2},\ f_N : G_{N,1} \to G_{N,2})$ of mappings, such that $f_N \circ src_1 = src_2 \circ f_E$ and $f_N \circ trg_1 = trg_2 \circ f_E$ (preserve source and target).

A graph morphism $f : G_1 \to G_2$ is injective if f_N and f_E are injective mappings. Only injective graph morphisms will be relevant in this paper.

Definition 2: (**Rule**) A graph transformation rule $p : L \xleftarrow{l} I \xrightarrow{r} R$ consists of three graphs L(LHS), I(Interface) and R(RHS) and a pair of injective graph morphisms $l : I \to L$ and $r : I \to R$.

Definition 3: (**Match and Dangling Condition**) Given a rule $p : L \xleftarrow{l} I \xrightarrow{r} R$ and a graph G. Then an occurrence of L in G, i.e., an injective graph morphism $m : L \to G$, is called a *match*. A match m for rule p satisfies *the dangling condition* if no node in $m(L \setminus l(I))$ is incident to an edge in $G \setminus m(L \setminus l(I))$.

Definition 4: (**Derivation Step**) Given a graph G, a graph transformation rule $p : L \xleftarrow{l} I \xrightarrow{r} R$, and a match $m : L \to G$, then there exists a *derivation step* from the graph G to the graph H if and only if the dangling condition is satisfied. H is constructed as follows:

1) Remove the image of the non-interface elements of L in G, i.e., $H' = G \setminus m(L \setminus l(I))$.
2) Add the non-interface elements of R into H, i.e., $H = H' \cup (R \setminus r(I))$.

A graph transformation rule can also have an arbitrary number of *negative application condition* graphs [14]. Such a graph is an extension of the LHS which prevents matches from being applied in a derivation step. In addition to the above, we adopt the theory of *typed attributed graphs* [10], where graphs are extended by assigning types to nodes and edges, and by assigning a set of named attributes to each node type. A graph morphism must now also preserve the node and edge types, and the attribute values.

In the graph transformation rules throughout this paper we only explicitly display the LHS and the RHS graphs, while the interface graph is given by shared identifiers of elements in the LHS and the RHS. Such identifiers are displayed next to its element.

We intend to specify the rules based on the concrete syntax of UML activity models, which we believe will make the rules more user-friendly to specify. Such concrete syntax-based rules are mapped to traditional abstract syntax-based rules for transformation in an existing graph transformation tool.

The AO4BPEL weave strategy is to identify all matches in the base BPEL statically, but to apply the advice code at runtime. This means that the number of matches is finite and the weaving process will always terminate. Currently, AO4BPEL does not support multiple aspects that are dependent.

Unlike AO4BPEL a set of GT rules is applied as long as possible. A set of GT rules can be non-terminating such as when the RHS of a rule contains a match of the rule's LHS. The AO4BPEL weave strategy can however be simulated by GT if we mark all the RHS elements to indicate that these can no longer take part in any matches. The LHS elements must then also be extended to check that none of its elements have such a RHS marking. One alternative is to extend all nodes and edges by a meta attribute `exclude` of type boolean. As the initial marking, the value of all the `exclude` attributes are set to false in the source graph and all the LHSs, while the `exclude` attributes are set to true in all the RHSs.

Since our rules are based on the concrete syntax with a mapping to abstract syntax-based rules (`c2a`), the initial marking can be part of this mapping. Hence, the rule specifier does not need to clutter the rules by all these `exclude` attributes. Furthermore, we may switch between different transformation strategies, by defining different `c2a` mappings, without having to change the rules on concrete syntax. We assume that AO4BPEL's weave strategy is simulated by the marking solution discussed above when we in the next section specify GT rules to simulate the same effects as achieved by some AO4BPEL code examples.

We have developed a proof-of-concept implementation of the `c2a` mapping for activity model-based rules available for download at [19], and further described in [8], [9], [20].

III. Simulating BPEL Aspects by Graph Transformation

In this section we investigate three AO4BPEL aspects, given by Charfi and Mezini [3], and we try to re-specify these aspects as GT rules that have the same effect. Due to limited space, we have removed the `partnerLinks` and `variables` sections from the AO4BPEL listings and do not discuss these details.

A. Example: Advice of type 'after'

An aspect to count the number of calls to Lufthansa's `findAFlight` service is taken from Charfi and Mezini [3]. The need to count these calls is because Lufthansa is assumed to charge clients depending on the number of calls to their service. Such calls occur in two different processes and can be counted by adding a counting service after each of these two calls. However, it is better to modularize this counting service into a single aspect. With the counting aspect, we can check if the bills we receive from Lufthansa are correct.

```
<pointcut...>
  //process//invoke[@portType="LufthansaPT" and
                   @operation="findAFlight"]
</pointcut>
<advice type="after">
   <sequence>
     <assign><copy><from expression="1"/>
             <to variable="increaseRequest" part="increaseBy"/>
     </copy></assign>
     <invoke partnerLink="CounterWS" portType="CounterPT"
             operation="increaseCounter"
             inputVariable="increaseRequest"
             outputVariable="increaseResponse"/>
   </sequence>
</advice>
```

Figure 3. The *counting* aspect represented by AO4BPEL

AO4BPEL. An extract of the AO4BPEL aspect is shown in Figure 3. The XPath expression in the pointcut selects a set of all the invoke elements that are used to call the `findAFlight` service.

The advice is of type `after`, which means that the advice content shall execute immediately after the execution of each joinpoint. When the execution of the advice content has terminated, then the BPEL engine executes the behavior that originally came after the joinpoint.

The body of the advice element adds two new BPEL actions. The first action is a copy element which assigns the fixed value 1 to a variable. This variable is used as input parameter by the second action, which is an invoke element for the new counter service.

GT. The counter aspect can also be specified with a single GT rule as shown in Figure 4. The rule's LHS matches an activity with `portType` and `operation` values that correspond to calls to the Lufthansa service. We also need to match the outgoing control flow of the Lufthansa activity (id=2), so that the new behavior can be connected properly to the existing control flow graph.

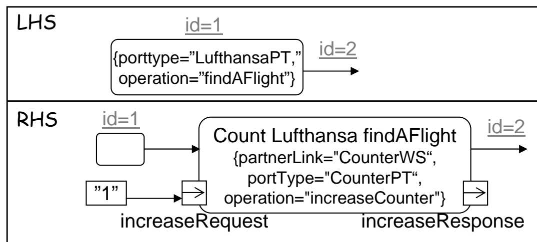


Figure 4. The *counting* aspect represented by GT

The RHS inserts the activity representing the counter service with a new control flow after the Lufthansa service, and uses the matched control flow (id=2) as its outgoing control flow. This ensures that we make a proper control flow graph without dangling activities and where the new activity has exactly one incoming and one outgoing control flow. The copy element from AO4BPEL becomes an incoming data flow for the input parameter of the counter service activity. This is an implicit control flow that can start at any time as long as it happens before the counter service.

B. Example: Advice of type 'around'

The second aspect (also taken from Charfi and Mezini [3]) monitors the execution time of certain services.

AO4BPEL (Figure 5). The services to be monitored are given by the AO4BPEL pointcut that selects all invoke activities which call one of the two operations `findAFlight` or `findARoom`. The aspect uses an advice of type `around`, which allow us to insert a call to a start timer service before the joinpoint, and to insert a call to a stop timer service after the joinpoint. The proceed element (`<proceed/>`) is used to position the execution of the joinpoint in between the new BPEL code.

```
<pointcut...>
   //invoke[@operation="findAFlight"] |
   //invoke[@operation="findARoom"]
</pointcut>
advice type="around">
   <sequence>
      <assign><copy>
          <from variable="ThisJPActivity" part="name"/>
          <to variable="startTimerRequest" part="activityName"/>
      </copy></assign>
      <invoke partnerLink="AuditingWS" portType="AuditingPT"
              operation="startTimer" inputVariable="startTimerRequest"
             outputVariable="startTimerResponse"/>
      <proceed/>
      <assign>...</assign>
      <invoke partnerLink="AuditingWS" portType="AuditingPT"
              operation="stopTimer" inputVariable="stopTimerRequest"
              outputVariable="stopTimerResponse"/>
</sequence></advice>
```

Figure 5. *Monitor execution time* aspect represented by AO4BPEL

GT. Figure 6 shows the first attempt at defining a corresponding GT rule. An `or` expression in the position of an attribute value in a LHS activity is used to express that the operation of the activity is either `findAFlight` or `findARoom`. Our previous paper [8] describes how such `or` expressions in a GT rule can be translated into multiple `or` expression free GT rules.

This first version of a GT rule for the monitoring aspect is not sufficient in some cases where we have data flow from a branch of parallel behavior. We illustrate the problem in Figure 7 which shows a possible woven model extract of the monitor aspect, where the `B` activity is the joinpoint activity. Due to the data flow from `d1` to `d2`, that goes from one of the parallel branches to the other, the `A` activity has to be completed before the `B` activity starts. So there is an implicit control flow dependency from the activity `A` to the activity `B`.

Because of the implicit control flow, there is no guarantee that the start timer activity starts immediately before the `B`

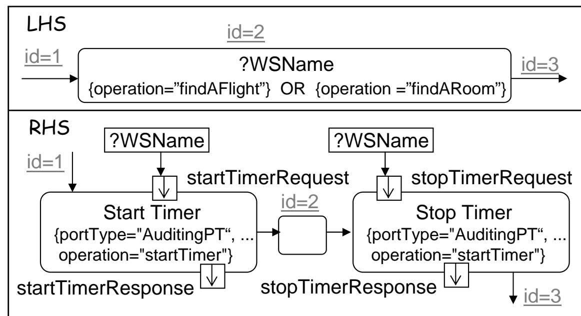
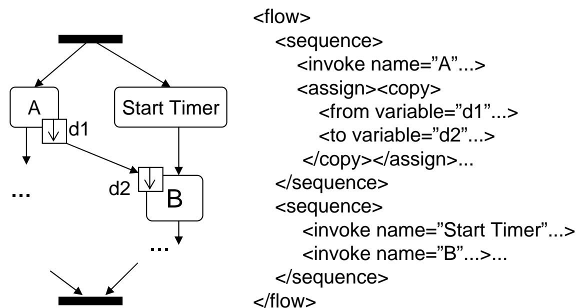


Figure 6. *Monitor execution time* aspect represented by GT - version 1




```
<flow>
  <sequence>
    <invoke name="A"...>
    <assign><copy>
      <from variable="d1"...>
      <to variable="d2"...>
    </copy></assign>...
  </sequence>
  <sequence>
    <invoke name="Start Timer"...>
    <invoke name="B"...>...
  </sequence>
</flow>
```

Figure 7. Data flow represents implicit control flow. Left: UML activity model extract. Right: BPEL document extract

activity, as intended. If the A activity is still running, then the time it takes to finish this activity is incorrectly included in the timer that is supposed to only measure the execution time of the B activity.

The AO4BPEL approach does not have this limitation since the matching of a joinpoint is done at runtime, followed by execution of the corresponding advice. This means that a match is first encountered when the B activity is about to execute.

We now discuss three alternative ways to fix the v.1 GT rule that simulates the aspect to monitor execution time. The three alternative rules are a bit simplified in the figure. We do not include identical parts from version 1 that are not relevant for the discussion.

The first alternative rule is given in the left part of Figure 8. The LHS from v.1 of the GT rule is extended by using our previously proposed *collection operator* for graph transformation [7]. For graph transformation, there exist other proposals for constructs like the collection operator in e.g., [22], [18].

The collection operator matches all the input pins and incoming data flow that goes to the activities we want to match. The RHS introduces a new sub activity with the same input pins as a matched activity and with the incoming data flow moved from the matched activity's input pins. The sub activity must contain a start and end activity.

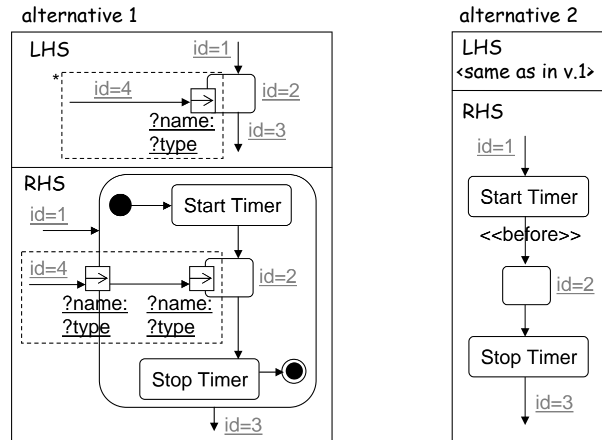


Figure 8. Alternative ways to fix the v.1 GT rule that simulates the aspect to monitor execution time

In addition it contains the three activities `Start Timer`, the joinpoint activity (id=2), and the `Stop Timer` in that sequential control flow order. There is also a data flow from each of the sub activity's input pins to the joinpoint activity's input pins.

A sub activity will not start before all its input pins are available. Hence, this GT rule will ensure that the joinpoint activity in the woven model is started immediately after the start timer activity has completed.

The second alternative rule is given in the middle part of Figure 8. The only change from the v.1 GT rule is that the control flow from the start timer activity to the matched activity has been stereotyped as «before», which has the semantics that the source activity must execute immediately before the target activity.

This new type of control flow, which we will refer to as *BFlow*, is an extension of the UML activity modeling language, and we need to provide special treatment of the BFlow somewhere on the way towards the running composition in a BPEL engine. The alternatives range from: (1) transforming a GT rule with BFlow into a corresponding GT rule without BFlow, (2) transforming a GT rule with BFlow into an AO4BPEL aspect, and (3) extending the existing transformation specification from activity model to BPEL so that activity models with BFlows are properly mapped. In the first alternative, the transformation can introduce a sub activity for each BFlow in a way that would transform our monitor GT rule in the middle part of Figure 8 into the GT rule to the left.

The third alternative rule identically specified as the v.1 of the GT rule. However, now all the new control flows that are added by a GT rule are interpreted as BFlows.

With alternative three, we cannot define a new incoming control flow which is not a BFlow (since all new control flows are implicitly interpreted as BFlows). This can have

a negative impact on the performance, since the source activities of BFlows may have their execution start delayed, also in cases where it is not necessary.

C. Example: joinpoint of type 'reply'

The third aspect (also taken from Charfi and Mezini [3]) adds information about relevant car rental offers as an addition to the existing output information from three different services.

AO4BPEL. The AO4BPEL code in Figure 9 defines a pointcut that matches `reply` activities within three different BPEL processes. For all these three processes, the similar advice of type `before` is used. This means that the advice body executes before the output data is finalized and is a way to add some action at the end of a service and to modify the final output data.

```
<pointcut name="about to reply" contextCollection="true">
//process[@name="travelProcess"]//reply[@operation ="getTravelPackage"]
|  //process[@name="flightProcess"]//reply[@operation="getFlight"]
| //process[@name="hotelProcess"]//reply[@operation="showHotels"]
</pointcut>
<advice type= "before">
    <sequence><assign><copy>
            <from variable="ThisProcess(clientrequest)" part="deptDate"/>
            <to variable="getCarRequest" part="startDate"/>
        </copy>...</assign>
        <invoke partnerLink="CarPortal" portType="CarPT"
                   operation="getCar" inputVariable="getCarRequest"
                   ouputVariable="getCarResponse"/>
        <assign><copy>
            <from
                expression="concat(getVariableData(ThisJPOutVariable),
                                        getVariableData(getCarResponse))"/>
               <to variable="ThisJPOutVariable"/>
        </copy></assign>
</sequence></advice>
```

Figure 9. *Car rental* aspect represented by AO4BPEL

The advice body will call a service to get car rental offers based on the relevant input data that are already available within the existing service composition. The information about car rental offers is concatenated with the original output of the service composition to constitute the final output.

GT. A corresponding GT rule to simulate the AO4BPEL aspect is shown in Figure 10. We use an `or` expression to capture that there are three alternative services that we intend to match. The LHS matches a final node inside the matched service. We also need to match the relevant data pins that shall serve as input to a new car rental offers service which is inserted into the control flow before the final node. The to be matched input pins must be part of the matched service, just as they need to be part of the matched process in AO4BPEL.

IV. Weave Strategies

Figure 11 gives an overview of three alternative ways to specify and weave aspects into an existing Web service composition. The composition has three levels. On top is

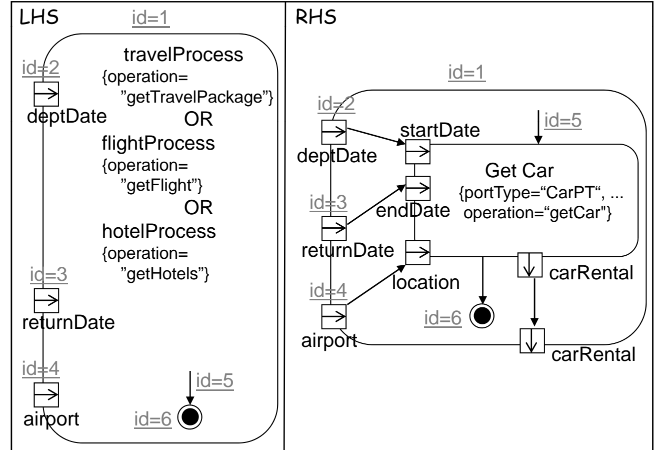


Figure 10. *Car rental* aspect represented by GT

the modeling level where we in this paper use UML activity models. In the middle is the XML level, where a Web service composition is represented by a BPEL document. At the bottom is the runtime level, where a BPEL engine runs the composite Web service.

In the **AO4BPEL weave approach**, a service composition has been specified by a BPEL document, which is given to a BPEL engine for execution. An aspect is specified as an AO4BPEL document and woven at runtime with the running composition. The BPEL engine needs to be extended so that it can weave AO4BPEL aspects, which is the case for the BPEL engine developed by Charfi and Mezini [3]. In the AO4BPEL approach, nothing is expected to be carried out at the modeling level. However, a composition may be specified at the modeling level followed by a transition to BPEL.

There are two main GT approaches to defining aspects, where one alternative uses a static weaving and the other uses a runtime weaving. In both alternatives, a composition is specified as a UML activity model with a graph representation. The user defines the model in the concrete syntax of the modeling language, and an underlying mapping derives the graph representation of the model. Furthermore, an aspect is specified as a GT rule. A GT rule is also based on the concrete syntax of the UML activity modeling language, and again an underlying mapping derives the GT rule in abstract syntax.

In the **static GT weave approach**, we can simply reuse an existing graph transformation tool to perform a model weaving. The transformed graph representation of the UML activity model represents the woven composition. This composition can be transformed to a woven BPEL document by an existing UML2BPEL transformation tool. Finally, this woven BPEL is sent to a BPEL engine for execution.

In the **runtime GT weave approach**, the weaving is not

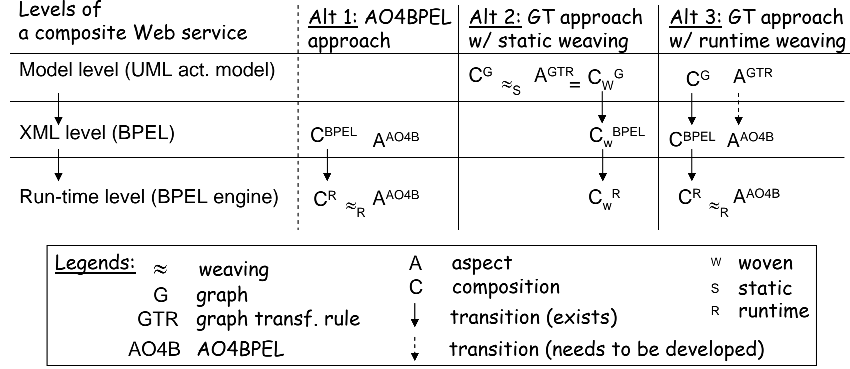


Figure 11. Overview of three alternative weave strategies

performed by an existing graph transformation tool. Instead, the GT rule is transformed into an AO4BPEL document, and this transformation needs to be developed since such a transformation, to our best knowledge, does not exist. With the generated AO4BPEL document, a runtime weaving can be performed similarly to the AO4BPEL approach.

A GT rule can in general be more expressive than an AO4BPEL aspect. This means that some checks on the GT rule is needed, to report if the GT rule can be transformed to AO4BPEL. It should for instance be checked that the LHS does not contain a sequence of activities, and that the LHS does not contain control flow constructs such as decision and merge. If the GT rule is not too expressive, then we believe that it can be feasible to specify a transformation to AO4BPEL by looking for patterns similar to those described in the examples given in the previous section.

V. Comparison

This section compares GT vs. AO4BPEL with respect to our two requirements: expressiveness and user-friendliness. We use the term 'basic GT' to mean the well-known algebraic GT which for instance has tool support by AGG. Extensions like the collection operator, `or` expressions in attribute values and BFlow are not part of basic GT.

A. Expressiveness

Two of the three examples above can be expressed with basic GT. The example with the around advice is a bit more problematic if we try to use basic GT without any extensions. Without `or` expressions and a construct like the collection operator, we can still use GT to express the aspect. However, we need to define two GT rules, one for each `or` operand in the rule defined above. In addition we need to hardcode the exact number of input pins when there is no construct like the collection operator available. In the around advice example above, the input pins are known, and we expect this to be the normal case. Basic GT is not able to express aspects using before or around advices where the number of input pins is not known.

We argue that the possible AO4BPEL types not shown by the three examples above, also are expressible with GT. A before advice is just a simplified version of an around advice. A GT rule for a receive joinpoint can be specified in a similar manner to the reply joinpoint example above where we add behavior after the initial node instead of before the final node.

AO4BPEL has a predefined joinpoint model and three fixed advice types, which gives limited expressiveness compared to GT. In GT any LHS graph pattern can serve as a joinpoint and any RHS graph pattern can serve as advice type. In AO4BPEL only single activities or single messages can be a joinpoint, i.e., a match. GT can for instance specify the matching of two or more activities in a sequence, all activities with exactly two input pins, all activities without an output pin (achieved by a so called *negative application condition*), two activities that are started in parallel, all the incoming control flow of a merge node etc. None of these examples are expressible with AO4BPEL.

In AO4BPEL there are only three advice types (before, after and around). GT is more expressive since a RHS can introduce parallel behavior, alternative behavior, and new sub activities that include all or some of the matched elements. Charfi and Mezini [3] argue that BPEL *links* and the around advice can be used together to simulate some kinds of parallel advice. However, it is quite limited compared to what GT supports.

B. User-friendliness

GT, and specifically GT rules based on some preferred concrete syntax such as UML activity models, is normally considered a benefit compared to specifying in the lower level XML-based AO4BPEL. This is the same benefit as achieved by specifying a composite Web service with UML/BPMN towards using BPEL. In the continued discussion about the user-friendliness we ignore this first benefit that

was in favour of GT.

An advice of type `after` is fairly easy to specify with AO4BPEL, and also with basic GT as long as we ensure that all activities have a single incoming and a single outgoing control flow. In AO4BPEL, the after behavior is ensured regardless of the actual syntactic structure.

An advice of type `before` is difficult to express with basic GT. The problem with before advice also applies to around advices, since an around advice includes a before advice. With basic GT, a `before` advice needs to be specified in a quite cumbersome way, as shown by alternative 1 in Figure 8. By extending basic GT with one of the alternative strategies proposed by alternatives 2 and 3 in Figure 8, then a before advice can be specified as easily as with an after advice.

Without support for `or` expressions within attribute values, then we need to make one almost identical GT rule for each `or` operand. This is the case for the monitor execution time example (Figure 6) and for the add car rental example (Figure 10).

The user-friendliness can also be compared with respect to the risk of producing aspects/rules that lead to incorrect woven results. For GT which is more expressive than AO4BPEL, this risk seems to be higher. Consider the `after` advice from Figure 4. It is easy to forget to match (and move) the outgoing control flow of the LHS activity. This leads to the invalid woven result of producing the new activity as a dangling activity, i.e., the new activity will not have any outgoing control flow.

With AO4BPEL it is not desirable to have a `before` advice combined with a joinpoint of type receive, and it is not desirable to have an after advice combined with a joinpoint of type reply. This is because the receive and reply activities represent the very first and the very last actions of the entire Web service composition. It is therefore unclear what it would mean in general to add behavior before a receive or to add behavior after a reply activity. The receive and reply activities also represent the public interface of the composite Web service, and changes to this interface should be avoided according to Charfi and Mezini [3].

The XPath expression can also lead to incorrect AO4BPEL code, since an XPath expression in general can select any elements in a BPEL document and not only those supported by the joinpoint model. This problem does not occur for GT, since any graph pattern can be an allowed joinpoint.

C. General Comparison

The GT approach with static weaving has several benefits compared to a runtime weaving. Firstly, it can be independent of a Web service composition language and support all those composition languages from which there exists a transformation from UML/BPMN. The AO4BPEL runtime weaving is handled by extending a BPEL engine, which means that AO4BPEL is tied not only to BPEL as the composition language but also to BPEL engines that have been extended to support AO4BPEL. With a runtime weaving, there is also a need for runtime checks which can slow down the execution of the Web service. However, Charfi and Mezini's experiments in [3] indicate that the performance overhead in practice is negligible. Other benefits with a static weaving is that the woven result can be viewed by the designer, such as for documentation or for manual validation purposes. Finally, BPEL tools allow us to perform a static analysis to detect possible errors on the statically woven BPEL.

A drawback of using GT as a BPEL aspect language is that it requires more tool support in order to be used in practice. This includes a mapping from UML (alternatively BPMN) to BPEL, and a mapping from GT concrete syntax-based rules to GT abstract syntax-based rules.

If there are multiple aspects that are dependent, then the weave order is important. Such dependency is currently not supported by AO4BPEL. GT on the other hand, provides a well-established theory and associated tools [23] that can be used to automatically detect dependencies among its rules. For dependent rules, the notion of *layers* can be used to control the transformation order.

VI. Related Work

We investigated using UML activity models with a mapping to BPEL, while Ouyang et al. [17] present a mapping from BPMN diagrams to BPEL. Our results in this paper should also be valid if we use BPMN instead of UML. This is because BPMN and UML activity models use quite similar ways to display control and data flow. The data flow in BPMN can also lead to implicit control flow between activities, and thus using graph transformation to model `before` (and `around`) advices will in general also be difficult for BPMN.

The Padus tool from Braem et al. [2], [1] uses Prolog-based relations to express pointcuts, and it uses BPEL code for the advice. XSLT is used to statically weave the advice with the original BPEL into a woven BPEL that can be executed in a standard BPEL engine. Padus provides a richer joinpoint model than in AO4BPEL, including control flow constructs like while and switch.

Padus has an advice of type `in` as an addition to those available in AO4BPEL. An `in` advice allows us to insert new BPEL elements as children to a number of existing BPEL elements. Many of these can easily be expressed with GT, but adding a child element to a parallel branch (e.g., flow) is not easy with GT. This is because GT has no general way to match the related pairs of a fork and a join in an activity model. If we carefully design the metamodel with an explicit edge that relates the fork and join, then a GT rule on the abstract syntax can simulate an `in` advice also for parallel branches.

Courbis and Finkelstein [5] is quite similar to AO4BPEL, except the advice language is Java instead of BPEL.

Whittle et al. [25] have also used algebraic graph transformation to simulate aspects, but so far the approach has not been configured for UML activity models or BPMN.

Model transformations can be seen as a generalization of aspect weaving. Hence, by making BPEL processes available at the modeling level enable us to specify BPEL aspects also by general model transformation tools like QVT [16] and ATL [12].

Xu et al. [26] defines a UML profile to model aspects for a Web service composition, where the composition is represented by UML activity models. Like other aspect languages it suffers from having a fixed predefined joinpoint model. Their aspects are limited to expressing that a new Web service call shall be inserted before, after or instead of a set of matched Web service calls. Furthermore, they have not specified how to implement the weaving.

Some recent approaches have defined adaptive and distributed Web services that go beyond the capabilities within BPEL [15], [24], [6]. These approaches do not yet have a well-established relation to graphical models, and hence it is non-trivial to see how GT can be helpful.

In parallel with our work, Charfi et al. [4] have presented an aspect-oriented language for BPMN called AO4BPMN. The possible joinpoints are limited to single activities or events, while the advice can be of arbitrary complexity. The authors propose to tag the base model elements that are to be affected by aspects instead of a separate pointcut model. Unfortunately, this early work on AO4BPMN does not include information about how to weave the aspects with the base model. This means that we cannot fully evaluate the capabilities of AO4BPMN with respect to the three AO4BPEL examples we have used in this paper.

VII. Conclusions

This paper has investigated if GT rules can simulate BPEL aspects at the modeling level and make BPEL aspect languages such as AO4BPEL redundant. We focused on expressiveness and user-friendliness of specifying the BPEL aspects, as our two main requirements. There are pros and cons with both alternatives with respect to these requirements.

Since GT rules represent a viable alternative to AO4BPEL, it can be questioned if defining a completely new BPEL aspect language is really needed. With a new aspect language it is necessary to specify an explicit joinpoint model, a pointcut language and a set of advice types. These parts are implicitly supported by the general apparatus of GT. Furthermore, it can be questioned if such an aspect language with a predefined joinpoint model will be expressive enough for unforeseen aspects.

GT allows us to define aspects/rules at a higher level by using graphical models in UML or BPMN. On the other hand, this paper shows that it can be difficult to specify an advice of type `before` (and thus also `around`) with GT. We propose a new control flow, called BFlow, dedicated to handle `before` advices.

Further case studies are needed to see what kinds of aspects are needed in practice, so that we know more about the needed expressiveness. Such experience may reveal the need for extensions in both AO4BPEL and GT.

Acknowledgment

The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project), and by the ENVISION project grant no. 249120 (EU FP7).

References

[1] M. Braem and N. Joncheere. Requirements for Applying Aspect-Oriented Techniques in Web Service Composition Languages. In *Software Composition, 6th International Symposium, SC*, volume 4829 of *Lecture Notes in Computer Science*. Springer, 2007.

[2] M. Braem, K. Verlaenen, N. Joncheere, W. Vanderperren, R. V. D. Straeten, E. Truyen, W. Joosen, and V. Jonckers. Isolating Process-Level Concerns Using Padus. In *Business Process Management, 4th International Conference, BPM*, volume 4102 of *Lecture Notes in Computer Science*. Springer, 2006.

[3] A. Charfi and M. Mezini. AO4BPEL: An Aspect-oriented Extension to BPEL. In *World Wide Web*, pages 309–344, 2007.

[4] A. Charfi, H. Müller, and M. Mezini. Aspect-Oriented Business Process Modeling with AO4BPMN. In *Modelling Foundations and Applications, 6th European Conference, ECMFA*, volume 6138 of *Lecture Notes in Computer Science*, pages 48–61. Springer, 2010.

[5] C. Courbis and A. Finkelstein. Weaving Aspects into Web Service Orchestrations. In *IEEE International Conference on Web Services (ICWS)*. IEEE Computer Society, 2005.

[6] A. Erradi, V. Tosic, and P. Maheshwari. MASC - .NET-Based Middleware for Adaptive Composite Web Services. In *IEEE International Conference on Web Services (ICWS)*. IEEE Computer Society, 2007.

[7] R. Grønmo, S. Krogdahl, and B. Møller-Pedersen. A Collection Operator for Graph Transformation. In *Theory and Practice of Model Transformations, Second International Conference, ICMT*, volume 5563 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2009.

[8] R. Grønmo and B. Møller-Pedersen. Aspect Diagrams for UML Activity Models. In *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Revised Selected and Invited Papers*, volume 5088 of *Lecture Notes in Computer Science*. Springer, 2008.

[9] R. Grønmo, B. Møller-Pedersen, and G. K. Olsen. Comparison of Three Model Transformation Languages. In *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA*, volume 5562 of *Lecture Notes in Computer Science*. Springer, 2009.

[10] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In *Graph Transformation, First Int. Conf., ICGT*, 2002.

[11] D. Jordan and J. Evdemon. Web Services Business Process Execution Language Version 2.0. Committee Specification. OASIS WS-BPEL TC., 2007.

[12] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*. Springer, 2006.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, 2001.

[14] L. Lambers, H. Ehrig, and F. Orejas. Conflict Detection for Graph Transformation with Negative Application Conditions. In *Graph Transformations, Third Int. Conf., ICGT*, Lecture Notes in Computer Science. Springer, 2006.

[15] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. D. Fraine, and D. Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD*. ACM, 2006.

[16] Object Management Group. MOF QVT Final Adopted Specification, OMG Document: ptc/05-11-01, November 2005.

[17] C. Ouyang, M. Dumas, W. M. P. van der Aalst, A. H. M. ter Hofstede, and J. Mendling. From business process models to process-oriented software systems. *ACM Trans. Softw. Eng. Methodol.*, 19(1), 2009.

[18] A. Rensink. Nested Quantification in Graph Transformation Rules. In *Graph Transformations, Third International Conference, ICGT*, Lecture Notes in Computer Science. Springer, 2006.

[19] Roy Grønmo. Proof-of-concept Model Transformation Tool for UML Activity Models with Support for the Collection Operator. http://folk.uio.no/roygr/ECMDA-2009-impl.zip.

[20] Roy Grønmo. *Using Concrete Syntax in Graph-based Model Transformations*. PhD thesis, Faculty of Mathematics and Natural Sciences, Univ. of Oslo, 2010.

[21] D. Skogan, R. Grønmo, and I. Solheim. Web Service Composition in UML. In *Proceedings of the 8th IEEE Intl Enterprise Distributed Object Computing Conf (EDOC'04)*, Monterey, California, September 2004.

[22] G. Taentzer. *Parallel and Distributed Graph Transformation. Formal Description and Application to Communication-Based Systems*. PhD thesis, Technische Universität Berlin, 1996.

[23] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Applications of Graph Transformations with Industrial Relevance, Second International Workshop (AGTIVE)*, 2003.

[24] B. Verheecke, M. A. Cibrán, and V. Jonckers. AOP for Dynamic Configuration and Management of Web Services. In *Web Services - ICWS-Europe 2003, International Conference*, volume 2853 of *Lecture Notes in Computer Science*. Springer, 2003.

[25] J. Whittle, P. Jayaraman, A. Elkhodary, A. Moreira, and J. Araújo. MATA: A Unified Approach for Composing UML Aspect Models based on Graph Transformation. *Transactions on Aspect-Oriented Software Development VI. Special Issue on Aspects and Model-Driven Engineering*, 5560, 2009.

[26] Y. Xu, S. Tang, Y. Xu, and Z. Tang. Towards Aspect Oriented Web Service Composition with UML. In *th Annual IEEE/ACIS International Conference on Computer and Information Science (ICIS)*. IEEE Computer Society, 2007.