# Analysis through Reflection
# Walking the EMF model of BPEL4WS*

Kien Huynh and Franck van Breugel

York University, Department of Computer Science
4700 Keele Street, Toronto, M3J 1P3, Canada

khuynh@cs.yorku.ca and franck@cs.yorku.ca

## Abstract

The Eclipse modelling framework provides a hierarchy of Java classes that represent the abstract syntax of BPEL4WS. Many analyses of a BPEL4WS program boil down to walking its abstract syntax tree. We review, refine and extend a technique, based on Java's reflection mechanism, to walk such trees. We apply this technique to implement two non-trivial analyses of BPEL4WS programs.

### Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*object-oriented design methods*; D.2.4 [**Software Engineering**]: Software/Program Verification—*validation*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*

### General Terms

Languages

### Keywords

Abstract syntax tree traversal, Java's reflection mechanism, Eclipse modelling framework, BPEL4WS

*Supported by IBM and the Natural Sciences and Engineering Council of Canada.

## 1 Introduction

The analysis of a program usually involves walking the abstract syntax tree of the program. When walking the syntax tree, nodes of the tree are visited. Which nodes are visited and in which order these nodes are visited may differ from one analysis to another. When a node is visited, some code is executed. Different code snippets may be associated to different types of nodes.

There are a number of different ways to implement such analyses in Java:

- adding dedicated methods to the Java classes representing the abstract syntax,
- the syntax separate from interpretation approach [2, Section 4.2],
- exploiting the visitor design pattern [14],
- tailoring automatically generated tree walkers as supported by compiler kits like ANTLR [25] and SableCC [12, 13], and
- reflection based tree walkers.

In this paper, we focus on reflection based tree walkers. This approach was put forward by Palsberg and Jay [23] (see also [24]) and refined and improved by Bravenboer and Visser [4]. This approach is based on Java's reflection mechanism (see, for example, [11]), implemented in the package java.lang.reflect. Reflection is both used for walking the tree and for executing the appropriate code snippet when visiting a node. The main advantage of reflection based tree walkers over all other approaches is

that changes to the class hierarchy representing the abstract syntax have very little effect on the code that performs the analyses. As we will see, it has other advantages as well. Its main disadvantage is its relatively poor performance.

Neither Palsberg and Jay nor Bravenboer and Visser walk array objects. In this paper, we show how to extend reflection based tree walkers so that we can also deal with arrays.

The Eclipse modelling framework (EMF) [6] generates a hierarchy of Java classes from the specification of an XML based language. These classes represent the abstract syntax of the language. In this paper, we will focus on the EMF model of the language BPEL4WS [1]. This language allows a programmer to compose web services. We present the implementation of a number of analyses of BPEL4WS programs by walking the EMF model by means of reflection based tree walkers. For example, we present the implementation of an analysis that detects if dead-path-elimination, a key ingredient of BPEL4WS, may cause side effects in a BPEL4WS program. Furthermore, we show how to translate BPEL4WS programs into the BPE-calculus — the input language for a verification tool for web service orchestration.

Since the EMF model of BPEL4WS represents BPEL4WS programs as directed graphs, rather than trees, some care is needed to ensure that the walkers terminate. As we will see, walking the EMF model of BPEL4WS can be viewed as a depth first traversal of a directed graph.

Combining EMF and reflection based graph walkers provides us with a powerful approach to analyze programs written in XML based languages. As we will see, the amount of code one needs to write is often considerably less than when exploiting any of the other approaches mentioned above. Since the resulting code can be produced rather quickly but is relatively slow, we believe that this approach is ideal for prototyping.

The rest of this paper is organized as follows. In Section 2, we provide a very brief introduction to BPEL4WS. We focus only on those concepts of BPEL4WS that play a role in the rest of this paper. Reflection based tree walkers are reviewed and refined in Section 3. In Section 4, we show how to walk array objects. Furthermore, we provide two simple examples of array walkers. The EMF model for BPEL4WS and reflection based walkers of this model are presented in Section 5. We provide two simple examples that walk the EMF model for BPEL4WS. In Section 6 and 7 we discuss two more elaborate examples of graph walkers. A translation from BPEL4WS to the BPE-calculus is sketched in Section 6 and detection of side effects of dead-path-elimination is the topic of Section 7. Section 8 concludes and discusses related and future work.

# 2 BPEL4WS

The business process execution language for web services (BPEL4WS) [1] represents the uniting of two previously competing standards: the web services flow language (WSFL) [19] from IBM and Microsoft's XLANG [28]. Like WSFL and XLANG, BPEL4WS has been designed to compose web services. For an introduction to web services, we refer the reader to, for example, [26].

In BPEL4WS, the basic activities include assignments, invoking web service operations, receiving requests, and replying to requests. These basic activities are combined into structured activities using ordinary sequential control flow constructs like sequencing, switch constructs, and while loops.

Concurrency is provided by the flow construct. For example, in

```
<flow>
  buy
  sell
</flow>
```

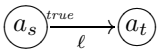
the activities `buy` and `sell`, whose behaviour has been left unspecified to simplify the example, are concurrent. The pick construct allows for selective communication. Consider, for example,

```
<pick>
  <onMessage partner="consumer">
    sell
  </onMessage>
  <onMessage partner="producer">
    buy
  </onMessage>
```

</pick>

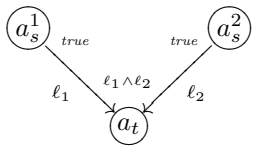
On the one hand, if a message from `consumer` is received then the activity `sell` is executed. In that case, the `buy` activity will not be performed. On the other hand, the receipt of a message from `producer` triggers the execution of the `buy` activity and discards the `sell` activity. In the case that both messages are received almost simultaneously, the choice of activity to be executed depends on the implementation of BPEL4WS.

Synchronization between concurrent activities is provided by means of links. Each link has a source activity and a target activity. Furthermore, a transition condition is associated with each link. The latter is a Boolean expression that is evaluated when the source activity terminates. Its value is associated to the link. As long as the transition condition of a link has not been evaluated, the value of the link is undefined. In this paper, we will use, for example




to depict that link $\ell$ has source $a_s$ and target $a_t$ and transition condition *true*.

Each activity has a join condition. This condition consists of incoming links of the activity combined by Boolean operators. Only when all the values of its incoming links are defined and its join condition evaluates to true, an activity can start. As a consequence, if its join condition evaluates to false then the activity never starts. We will use, for example,




to depict that the join condition of activity $a_t$ is $\ell_1 \wedge \ell_2$. In the above example, activity $a_t$ can only start after activities $a_s^1$ and $a_s^2$ have finished.

# 3 Walking Trees

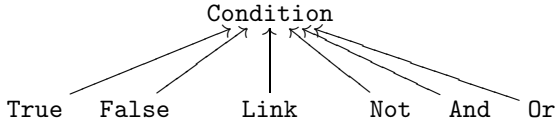
Below, we introduce the reflection based tree walkers of Palsberg and Jay [23] by means of simple examples. In the examples, we focus on join conditions of BPEL4WS. We introduce a small hierarchy of Java classes that represents the abstract syntax of join conditions. We show how a reflection based tree walker can be used to traverse an abstract syntax tree of a join condition to extract (some of) the links of the join condition.

As we already mentioned above, a join condition in BPEL4WS consists of links combined by Boolean operators. To simplify the presentation, we consider join conditions defined by the following BNF production

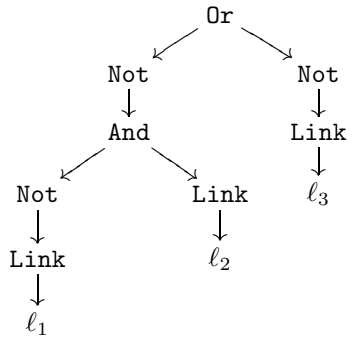
$$c ::= \mathit{true} \mid \mathit{false} \mid \ell \mid \neg c \mid c \wedge c \mid c \vee c \mid (c)$$

where $\ell$ is the name of a link. For example, $\neg(\neg\ell_1 \wedge \ell_2) \vee \neg\ell_3$ is a join condition.

To represent the abstract syntax of join conditions in Java, we introduce the following class hierarchy.




The class `Condition` is abstract and the other classes are not. The class `Not` has a single field of type `Condition` and the classes `And` and `Or` have two such fields. The class `Link` has a field of type `String`. The classes `True` and `False` have no fields. The abstract syntax tree of the join condition $\neg(\neg\ell_1 \wedge \ell_2) \vee \neg\ell_3$ is represented by a `Condition` object that can be depicted as the following tree.




To distinguish the three `String` objects, these objects are represented by their value. The nodes of the tree are objects. An object is a parent of another object in the tree if the former object has a non-static field the value of which is the latter object.

A reflection based tree walker uses Java's reflection mechanism to access (some of) the fields of the objects representing an abstract syntax tree in order to walk the tree. Furthermore, the reflection mechanism is also exploited to execute the appropriate code snippets when visiting nodes of the tree. The abstract class `Walker` forms the basis for a reflection based tree walker. To implement an actual walker, one has to extend this class. The subclass contains the code snippets that need to be executed when visiting nodes of the tree. We can associate different code snippets to different types of nodes. For example, to associate a code snippet to nodes of type `Link`, we introduce a method

```
void visit (Link link) { ... }
```

Next, we give an informal presentation of the class `Walker`.

```
abstract class Walker
{
  void walk(Object o)
  {
    if (o != null)
      if (this class has a visit method
          that takes o as an argument)
        visit(o);
      else
        walkFields(o);
  }

  void walkFields(Object o)
  {
    if (o != null)
      for (each field f of o)
        if (f is not static and
            f is not of primitive type)
          walk(o.f);
  }
}
```

Clearly, the `walk` method is similar to a depth first traversal. Whenever we encounter a node for which we have introduced a `visit` method, the corresponding code snippet is executed. Note that the subtree of such a node is not traversed.

Let us briefly discuss the differences between the above code and the code presented by Palsberg and Jay [23] and by Bravenboer and Visser [4]. When walking the tree, we do not consider static fields. These fields contain very generic information about a class and not specifically about an object. Therefore, these fields are not considered part of the tree. Bravenboer and Visser also do not consider static fields, but Palsberg and Jay do.

Palsberg and Jay and also Bravenboer and Visser only consider the public fields, including inherited public fields. In contrast, we consider all non-static fields, including inherited ones. If only public fields were walked, it would force us to make the fields in the classes representing the syntax trees public, hence violating the object-oriented principle of encapsulation. Furthermore, tree walkers would not be applicable to EMF models since most fields of EMF models are not public. However, by walking non-public fields, one should in general refrain from changing the object that is provided as an argument to the `visit` methods. For example, in the body of the method

```
void visit(Link link) { ... }
```

the object `link` should not be modified in general. Otherwise, we may be violating the object-oriented principle of encapsulation, since the object `link` may be the value of a private field.

Palsberg and Jay do not walk objects of type `Number`, `Boolean` and `Character`, whereas we follow Bravenboer and Visser and do not consider fields of primitive type. If one were to remove the condition `f is not of primitive type`, then the `walk` method would not terminate for objects of type `Number`, `Boolean` and `Character`. For example, consider an object `o` of type `Boolean`. The class `Boolean` has a private and non-static field named `value` of type `boolean`. Java's reflection mechanism automatically wraps values of primitive type in an object. Hence, `walk(o)` would gives rise to `walk(new Boolean(o.value))`, where the `Boolean` objects `o` and `new Boolean(o.value)` represent the same Boolean value, and hence `walk(o)` would give rise to infinite recursion.

On the one hand, Palsberg and Jay only consider `visit` methods that are part of the class that walks the tree. On the other hand, Bravenboer and Visser consider `visit` methods in the class that walks the tree, but also in

any of its superclasses. We also take the latter approach. This approach allows us to exploit inheritance as we will show in the second example presented below.

The type of the parameter of the `visit` methods we restrict to classes. That is, we disallow the use of an interface as the type of the parameter of a `visit` method. Bravenboer and Visser allow interfaces. Although interfaces allow more generic code in some cases, the use of interfaces in this setting may lead to ambiguity. For example, consider that class `C` implements the interfaces `I` and `J`. Furthermore, assume that the walker has

```
void visit(I i) { ... }
void visit(J j) { ... }
```

but no `visit` method with parameter of type `C`. In this case, when visiting a node of type `C`, it is not evident which of the above two visit methods should be chosen.

Next, we realize the following analysis by means of a reflection based tree walker. Given a join condition $c$, return the collection of all links that are part of $c$. That is, given a `Condition` object `c`, return the collection of all `Link` objects that are part of `c`. Let us implement the collection by a `Vector`. To perform this analysis, one has to traverse the tree of `c`. Whenever a `Link` node is visited, this object has to be added to the vector.

```
class LinkExtractor extends Walker
{
  Vector links;

  LinkExtractor()
  {
    super();
    links = new Vector();
  }

  Vector getLinks()
  {
    return links;
  }

  void visit(Link link)
  {
    links.add(link);
  }
}
```

The `Vector` object `links` keeps track of the `Link` objects that have been encountered during the traversal.

Given a `Condition` object `c`,

```
(new LinkExtractor()).walk(c).getLinks();
```

gives us a `Vector` object containing the `Link` objects that are part of `c`.

As a second example, we show how to find all negative occurrences of links in a join condition. We will exploit this analysis in Section 7. A `Link` object occurs negatively in a `Condition` object if the path from the `Link` object to the root of the abstract syntax tree has an odd number of `Not` objects. For example, in the join condition $\neg(\neg\ell_1 \wedge \ell_2) \vee \neg\ell_3$ the links $\ell_2$ and $\ell_3$ occur negatively and the link $\ell_1$ does not.

```
class NegativeLinkExtractor
  extends LinkExtractor
{
  boolean odd;

  NegativeLinkExtractor()
  {
    super();
    odd = false;
  }

  void visit(Link link)
  {
    if (odd)
      links.add(link);
  }

  void visit(Not not)
  {
    odd = !odd;
    walkFields(not);
    odd = !odd;
  }
}
```

The field `odd` tells us whether the number of nodes of type `Not` from the currently visited node to the root of the tree is odd. In order to walk the subtree rooted at a node of type `Not`, we use the method `walkFields`.

# 4 Walking Arrays

Neither Palsberg and Jay nor Bravenboer and Visser walk arrays. To handle array objects we modify the `walk` method and add the `walkArray` method to the class `Walker`.

```
void walk(Object o)
{
  if (o != null)
    if (this class has a visit method
        that takes o as an argument)
      visit(o);
    else
      if (o is an array)
        walkArray(o);
      else
        walkFields(o);
}

void walkArray(Object o)
{
  if (o != null and
      component type of o is not primitive)
    for (int i = 0; i < o.length; i++)
      walk(o[i]);
}
```

Now let us walk some arrays. For example, the sum of the values of an ($n$-dimensional) array of `Integer` objects can be computed as follows.

```
class Adder extends Walker
{
  int sum;

  Adder()
  {
    super();
    sum = 0;
  }

  int getSum()
  {
    return sum;
  }

  void visit(Integer i)
  {
    sum += i.intValue();
  }
}
```

Consider a two-dimensional array `a` of `Integer` objects with $m$ rows and $n$ columns. We can compute

$$\prod_{1 \le r \le m} \sum_{1 \le c \le n} \mathtt{a}[c][r]$$

as follows.

```
class Multiplier extends Adder
{
  int product;

  Multiplier()
  {
    super();
    product = 1;
  }

  int getProduct()
  {
    return product;
  }

  void visit(Integer[] a)
  {
    sum = 0;
    walkArray(a);
    product *= sum;
  }
}
```

Note that the class `Multiplier` extends the class `Adder`. As a consequence, the method call `walkArray(a)` results in summing the values of the array `a`.

# 5 Walking Graphs

The Eclipse modelling framework (EMF) [6] generates a hierarchy of Java classes from the specification of an XML based language. These classes represent the abstract syntax of the language. In this paper, we will focus on the EMF model of BPEL4WS.

Note that if we apply the method `walk` to a collection of objects that forms a graph, rather than a tree, then the traversal may not terminate. The objects representing a BPEL4WS program often form a graph. For example, a `Flow` object `flow`, which represents a flow, refers to a `Links` object, say, `links`, which represents the collection of links that are declared

within the flow. The `Links` object `links` in turn refers to the `Flow` object `flow`. Because of the presence of cycles, we have to refine the above introduced tree walker so that it can also deal with graphs.

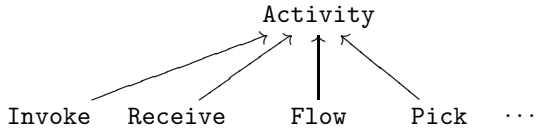
Palsberg and Jay do not walk graphs, whereas Bravenboer and Visser do. They propose two solutions to address the termination problem that arises when walking graphs. The first is to remember which objects have already been walked. The second is to mark those objects that have been walked. We have implemented the first solution as it does not require any changes to the graph. We introduce a collection `walked` that contains all the objects that have been walked so far.

```
Collection walked;

Walker()
{
  walked = empty collection;
}

void walk(Object o)
{
  if (o != null and
      walked does not contain o)
    walked.add(o);
    if (this class has a visit method
        that takes o as an argument)
      visit(o);
    else
      if (o is an array)
        walkArray(o);
      else
        walkFields(o);
}
```

Note that a walk is similar to a depth first traversal of a directed graph. The vertices of the graph are objects. There is a directed edge from one object to another if the former object has a field the value of which is the latter object. Whenever the traversal reaches an object for which a `visit` method has been introduced, then the corresponding code snippet is executed. In that case, the objects that are reachable from the visited object are not walked.

The EMF model of BPEL4WS consists of more than 500 classes. Here, we will only discuss a few (simplified versions) of those classes. However, all the graph walkers presented below work for the EMF model of BPEL4WS (provided that some class names, that we abbreviated, are expanded and some pseudocode is replaced with Java code).

In the EMF model of BPEL4WS, activities are represented as objects of type `Activity`. `Activity` is an abstract class that is extended by a number of concrete classes.




In BPEL4WS, a link is declared within a flow. The scope of the link is the flow. For example, in the BPEL4WS snippet

```
<flow>
  <links>
    <link name="l">
  </links>
  ...
  <flow>
    <links>
      <link name="l">
    </links>
    ...
  </flow>
  ...
</flow>
```

two links, both named `l`, are declared. It is sometimes useful to rename the links so that all links have a different name.

In the EMF model of BPEL4WS, a `Flow` object has a collection of `Link` objects. These `Link` objects represent the links that are declared within the flow.

The following walker of the EMF model of BPEL4WS renames links so that all links have a different name.

```
class LinkRenamer extends Walker
{
  int counter;

  LinkRenamer()
  {
    super();
    counter = 0;
```

```
  }

  void visit(Link link)
  {
    link.setName("l" + counter);
    counter++;
  }
}
```

Note that the `visit` method changes the `link` object. In this case, the objective of the walker is to change the model. We will exploit this `LinkRenamer` in a tool described in Section 6.

In Section 7, we will present a tool to detect side effects of dead-path-elimination. The tool makes use of a directed graph. This graph is extracted from an `Activity` object. The vertices of the graph are activities and the edges of the graph are links. The directed graph is represented by a `Graph` object. The class `Graph` includes the methods

```
void addSource(Activity source,
               Link link);
void addTarget(Activity target,
               Link link);
```

The method `addSource` adds the vertex `source` and the edge `link` to the graph provided that the graph does not already contain them. Furthermore, it sets the vertex `source` to be the source of the edge `link`. The method `addTarget` has a similar effect. The graph can be built as follows.

```
class GraphBuilder extends Walker
{
  Graph g;

  GraphBuilder()
  {
    super();
    g = new Graph();
  }

  Graph getGraph()
  {
    return g;
  }

  void visit(Source s)
  {
    g.addSource(s.getActivity(),
                s.getLink());
  }

  void visit(Target t)
  {
    g.addTarget(t.getActivity(),
                t.getLink());
  }
}
```

Once we have built the (hyper)graph, we can easily check if each link has a unique source and target activity. We can also verify the absence of (control) cycles. A valid BPEL4WS program has to satisfy both conditions as specified in the BPEL4WS definition [1, Section 12.5].

# 6 BPE-Calculus

The BPE-calculus is a small language based on BPEL4WS proposed by Koshkina and Van Breugel in [18] (see also [17]). In the BPE-calculus they focus on the control flow in BPEL4WS. They abstract from many details. In particular, they do not consider data, time, and fault and compensation handlers.

The concurrency workbench (CWB) is a generic and customizable verification tool developed by Cleaveland et al. [7, 8]. Originally, the CWB was designed for the verification of Milner's calculus of communicating systems (CCS) [22]. However, the CWB can be customized to support languages other than CCS. In [18], Koshkina and Van Breugel show how to customize the CWB so that it also supports the BPE-calculus. In this way, they obtain a verification tool for the BPE-calculus. Furthermore, Ramay [27] implemented a tool that translates a BPEL4WS program into a BPE-process. Using this tool, Koshkina and Van Breugel have verified a number of BPEL4WS programs.

Next, we sketch how the translation from BPEL4WS to the BPE-calculus can be implemented by walking the EMF model of BPEL4WS. First, we use the walker as described in Section 5 to rename the links in such a way that all links have a different name. Next, we show how a `Condition` object, representing a BPEL4WS join condition, can be translated into the `String` representation of

the corresponding join condition in the BPE-calculus.

```
class Translator extends Walker
{
  Stack s;

  Translator()
  {
    super();
    s = new Stack();
  }

  String getTranslation()
  {
    return (String) s.pop();
  }

  void visit(True t)
  {
    s.push("true");
  }

  void visit(False f)
  {
    s.push("false");
  }

  void visit(Link link)
  {
    s.push(link.getName());
  }

  void visit(Not not)
  {
    walkFields(not);
    s.push("(not " +
           (String) s.pop() + ")");
  }

  void visit(And and)
  {
    walkFields(and);
    String right = (String) s.pop();
    String left = (String) s.pop();
    s.push("(" + left +
           " and " + right + ")");
  }

  void visit(Or or)
  {
    walkFields(or);
    String right = (String) s.pop();
    String left = (String) s.pop();
    s.push("(" + left +
           " or " + right + ")");
  }
}
```

We use a `Stack` object `s` to temporarily store the translation of subtrees of the syntax tree of the join condition.

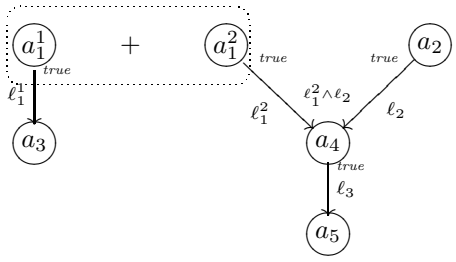
Similarly, we can develop a walker that translates an `Activity` object, representing a BPEL4WS activity, into a `String` representation of its counterpart in the BPE-calculus.

Ramay used the syntax separate from interpretation approach to implement the translation from BPEL4WS to the BPE-calculus. For this translation the syntax separate from interpretation approach needed more than five times as much code as our walker of the EMF model. We conjecture that the other approaches mentioned in the introduction also need considerably more code to implement the analysis.

# 7 Dead-Path-Elimination

In this section, we present a tool to detect side effects of dead-path-elimination (DPE). Also this tool relies on walkers of the EMF model of BPEL4WS. Before we present these walkers, we first briefly discuss DPE.

Let us consider the following activities and links.



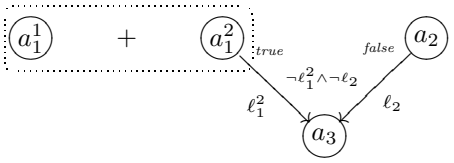

In the above picture, we use + to depict the pick construct. Note that the choice between the activities $a_1^1$ and $a_1^2$ determines which of the activities $a_3$, $a_4$ and $a_5$ are performed. For example, if $a_1^1$ is chosen then $a_3$ is executed. In that case neither $a_4$ nor $a_5$ can ever occur. As a consequence, the activities $a_4$ and $a_5$ could be "garbage collected." We use the term "garbage

collection" to denote the process of automatically reclaiming activities (rather than memory). This can be achieved as follows.

- If a pick construct is executed, then we also assign false to all the outgoing links of those branches of the pick construct that are not chosen.
- If the join condition of an activity evaluates to false, then the activity is "garbage collected" after assigning false to its outgoing links.

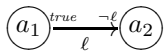
This "garbage collection" scheme is named dead-path-elimination (DPE) [1, 20]. As an aside, DPE not only "garbage collects" activities but also simplifies termination detection of structured activities. Let us briefly return to the above example. Assume that activity $a_1^1$ is chosen. Then, as a result of DPE, the value of the link $\ell_1^2$ becomes false. When activity $a_2$ terminates, the link $\ell_2$ gets the value true. At this point, the join condition of activity $a_4$ can be evaluated. Since its value is false, by DPE, false is assigned to link $\ell_3$ and activity $a_4$ is "garbage collected." Subsequently, again exploiting DPE, activity $a_5$ can be "garbage collected" as well.

Now let us consider another example.




At first sight, one may be tempted to conclude that activity $a_3$ will never be executed. However, DPE may trigger the execution of activity $a_3$ as follows. Assume that activity $a_1^1$ is chosen. By DPE, the value of the link $\ell_1^2$ becomes false. Since the value of the link $\ell_2$ is false as well, the join condition $\neg\ell_1^2 \wedge \neg\ell_2$ evaluates to true. Hence, activity $a_3$ can be performed. The above can be paraphrased as DPE may have side effects. We believe that side effects as in the above example may be introduced accidentally.

The side effect in the above example is caused by negative occurrences of links in the join condition. The links $\ell_1^2$ and $\ell_2$ both have a negative occurrence in the join condition $\neg\ell_1^2 \wedge \neg\ell_2$. As Van Breugel and Koshkina have shown in [5], by disallowing negative occurrences of links in join conditions, side effects like the one in the above example can be eliminated. However, not every negative occurrence of a link in a join condition gives rise to side effects. For example, the link $\ell$ in




occurs negatively, but does not give rise to any side effects.

Note that, in the second example of this section, activity $a_3$ can only be executed once the execution of activity $a_1^1$ has started and the execution of activity $a_2$ has terminated. Since BPEL4WS is Turing complete, the problem of determining whether a given negative occurrence of a link in a join condition may give rise to side effects can be reduced to the halting problem and, hence, is undecidable.

Below, we present a tool that can narrow down the negative occurrences of links in join conditions that may be troublesome. If our tool finds no such occurrences then we know that the BPEL4WS program is free of such side effects. If, however, our tool finds some negative occurrences of links that may give rise to side effects, then we should check whether such side effects can really occur and, if so, whether these side effects are intentional.

If a negative occurrence of a link in a join condition gives rise to a side effect, then

- the value of the link is set to false due to DPE and
- the join condition evaluates to true.

Next, we introduce two (mutually recursive) functions to capture these two conditions. But before presenting these functions, we first consider the following BPEL4WS snippet.

```
<flow>
  <assign>
    <copy>
      <from expression="0">
      <to variable="v">
    </copy>
  </assign>
  <assign>
```

```
    <copy>
      <from expression="1">
      <to variable="v">
    </copy>
  </assign>
</flow>
```

After executing the above snippet, the value of the variable `v` can either be 0 or 1. As a consequence, the transition condition

```
bpws:getVariableData('v')='0'
```

can be either true or false. Hence, the link $\ell$ to which this transition condition is associated either gets the value true or false. Therefore, the join condition $\ell$ either evaluates to true or false. Consequently, if we want to predict the value of a transition condition, a link, or a join condition, then we can make three different predictions: its value is always true (which we represent by 1), its value is always false (which we represent by $-1$) or its value is some times true and other times false (represented by 0).

Given a link $\ell$, the Boolean $dpe(\ell)$ tells us whether $\ell$ may be set to false due to DPE. Given a join condition $c$, $value(c)$ captures the possible values of $c$. Given a transition condition $b$, $value(b)$ approximates the possible values of $b$. The function $dpe$ is defined by

$$dpe(\ell) = (s \text{ is part of a pick}) \vee (value(c) \neq 1)$$

where $s$ is the source activity of link $\ell$ and $c$ is the join condition of activity $s$. The function $value$ on join conditions is defined by

$$\begin{array}{rcl} value(true) & = & 1 \\ value(false) & = & -1 \\ value(\ell) & = & \begin{cases} value(b) \min 0 & \text{if } dpe(\ell) \\ value(b) & \text{otherwise} \end{cases} \\ value(\neg c) & = & -value(c) \\ value(c_1 \wedge c_2) & = & value(c_1) \min value(c_2) \\ value(c_1 \vee c_2) & = & value(c_1) \max value(c_2) \end{array}$$

where $b$ is the transition condition of the link $\ell$. Note that if the link $\ell$ can be set to false due to DPE and the value of $b$ is either always true or some times true and other times false, then the value of $\ell$ is some times true and other times false. The function $value$ on transition conditions is defined by

$$value(b) = \begin{cases} 1 & \text{if } b = true \\ -1 & \text{if } b = false \\ 0 & \text{otherwise} \end{cases}$$

Note that the approximation of the possible values of a transition condition is not very precise. This approximation could easily be improved considerably. We leave this for future work.

To compute $dpe(\ell)$, we have to determine the source activity of the link $\ell$. For this purpose, we exploit the `Graph` object built by the walker described in Section 5.

While computing the value of a join condition, we may have to compute the values of other join conditions as well. To refrain from computing the value of a join condition multiple times, we store these values in the `Graph` object. Therefore, we augment the `Graph` class with the following methods.

```
boolean hasValue(Activity activity);
int getValue(Activity activity);
void setValue(Activity activity,
              int value);
```

The method `hasValue` checks if the value of the join condition of the activity `activity` has already been set. The method `getValue` returns the value of the join condition of the activity `activity`. The method `setValue` sets the value of the join condition of the activity `activity` to `value`.

We implement the function $dpe$ as follows.

```
boolean dpe(Link link, Graph graph)
{
  Activity a = graph.getSource(link);
  if (!graph.hasValue(a))
    Evaluator e = new Evaluator(graph);
    Condition c = a.getCondition();
    e.walk(c);
    int v = e.getValue();
    graph.setValue(a, v);
  return (a is part of a pick or
          graph.getValue(a) != 1);
}
```

In the above snippet, we use an `Evaluator` object to compute the possible values of the join condition of the source activity of the link `link`. The value of a `Condition` object is computed by walking the tree of the `Condition` object as follows.

```
class Evaluator extends Walker
{
```

```
Stack s;
Graph g;

Evaluator(Graph graph)
{
  super();
  s = new Stack();
  g = graph;
}

int getValue()
{
  return pop();
}

void visit(True t)
{
  push(1);
}

void visit(False f)
{
  push(-1);
}

void visit(Link link)
{
  int v = value of transition
          condition of link;
  if (dpe(link, g))
    v = Math.min(v, 0);
  push(v);
}

void visit(Not not)
{
  walkFields(not);
  push(-pop());
}

void visit(And and)
{
  walkFields(and);
  push(Math.min(pop(), pop()));
}

void visit(Or or)
{
  walkFields(or);
  push(Math.max(pop(), pop()));
}

  void push(int i)
  {
    s.push(new Integer(i));
  }

  int pop()
  {
    return ((Integer) s.pop()).intValue();
  }
}
```

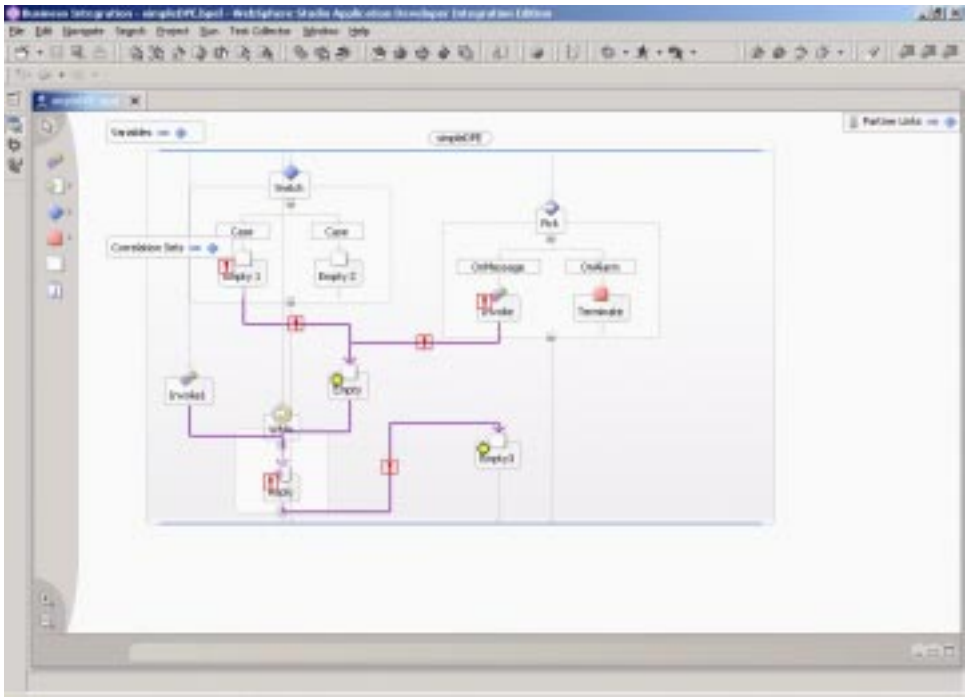
To check if DPE may give rise to side effects, for each link $\ell$ that occurs negatively in a join condition $c$, we compute $dpe(\ell)$ and $value(c)$. This can be accomplished by modifying the walker `NegativeLinkExtractor` by replacing the `visit` method for `Link` objects with

```
void visit(Link link)
{
  if (odd)
  {
    Activity a = graph.getTarget(link);
    if (!graph.hasValue(a))
      Evaluator e = new Evaluator(graph);
      Condition c = a.getCondition();
      e.walk(c);
      int v = e.getValue();
      graph.setValue(a, v);
    if (dpe(link, graph) &&
        graph.getValue(a) == 1)
      links.add(link);
  }
}
```

Our tool to detect side effects of DPE can easily be plugged into IBM's WebSphere Studio Application Developer Integration Edition. In the screenshot below, the activities marked with an exclamation mark may be "garbage collected" due to DPE. The links marked with an exclamation mark may be set to false due to DPE. And the activities marked with a sun may be executed due to DPE.

# 8 Conclusion

Palsberg and Jay [23] (see also [24]) introduced reflection based tree walkers. Blosser [3] also used Java's reflection mechanism to walk trees. Bravenboer and Visser [4] refined and improved



the tree walkers of Palsberg and Jay. In particular, they do not walk fields that static or primitive. Furthermore, they allow `visit` methods in superclasses of the walker and they also allow interfaces as the type of the parameter of a `visit` method. They not only walk trees but also walk graphs. In this paper, we extended reflection based walkers by also walking array objects. When walking trees and graphs, we not only consider public fields but also non-public ones. This is essential when walking EMF models, since most fields in the Java classes generated by EMF are not public.

Exploiting reflection based walkers of the EMF model of BPEL4WS, we have implemented a number of analyses. For example, we developed a tool that translates a BPEL4WS program into a BPE-process (which can subsequently be verified using the CWB) and a tool that detects if DPE may give rise to side effects. Furthermore, we used a walker to build a graph from which we can easily derive whether each link has a unique source and a unique target and whether there are (control) cycles.

The performance of the reflection based tree walkers of Palsberg and Jay is relatively poor in comparison with the other approaches mentioned in the introduction. Forax and Roussel [10] and also Bravenboer and Visser have shown that the performance can be improved considerably by caching fields and methods. Grothoff [15] demonstrated that the performance can be improved even more by using runtime code generation techniques. EMF provides it own reflection mechanism for the classes generated by EMF. Since EMF reflection is more efficient than Java reflection [21], we plan to exploit EMF reflection in combination with the techniques mentioned above to improve the performance of our walkers.

Another topic for further research is the improvement of our tool to detect side effects of DPE. In particular, we are interested to improve the approximation of the possible values of transition conditions. Again, we hope to exploit walkers, this time to approximate the values of variables.

We plan to exploit reflection based walkers to implement other analyses of BPEL4WS programs. In particular, we are interested to develop type systems to detect deadlocks and race conditions a la [9, 16] and to implement these type systems by means of reflection based walkers of the EMF model of BPEL4WS.

## Acknowledgements

The authors would like to thank Martin Bravenboer, Frank Budinsky, Bill O'Farrell, Jane Fung and Ed Merks.

## About the Authors

Kien Huynh is a graduate student in computer science at York University in Toronto. Franck van Breugel is an associate professor in computer science and an adjunct professor in mathematics at York University and an IBM CAS faculty fellow.

## References

[1] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services, version 1.1. Available at `www.ibm.com/developerworks/webservices/library/ws-bpel/`, May 2003.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[3] Jeremy Blosser. Java tip 98: reflect on the visitor design pattern. Available at `www.javaworld.com`, July 2000.

[4] Martin Bravenboer and Eelco Visser. Guiding visitors: separating navigation from computation. Technical Report UU-CS-2001-42, Utrecht University, November 2001.

[5] Franck van Breugel and Mariya Koshkina. Does dead-path-elimination have side effects? Technical Report CS-2003-04, York University, Toronto, April 2003.

[6] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2003.

[7] Rance Cleaveland, Tan Li, and Steve Sims. The Concurrency Workbench of the New Century user's manual. Available at `www.cs.sunysb.edu/~cwb`, July 2000.

[8] Rance Cleaveland and Steve Sims. The NCSU concurrency workbench. In Rajeev Alur and Thomas Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, NJ, July 1996. Springer-Verlag.

[9] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, Vancouver, June 2000. ACM.

[10] Remi Forax and Gilles Roussel. Recursive types and pattern matching in Java. In Krzysztof Czarnecki and Ulrich W. Eisenecker, editors, *Proceedings of the 1st International Symposium on Generative and Component-Based Software Engineering*, volume 1799 of *Lecture Notes in Computer Science*, pages 147–164, Erfurt, September 1999. Springer-Verlag.

[11] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications, 2004.

[12] Etienne M. Gagnon. SableCC, an object-oriented compiler framework. Master's thesis, McGill University, Montreal, 1998.

[13] Etienne M. Gagnon and Laurie J. Hendren. SableCC, an object-oriented compiler framework. In *Proceedings of the 26th International Conference on Technology of Object-Oriented Languages and Systems*, pages 140–154, Santa Barbara, August 1998. IEEE.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[15] Christian Grothoff. Walkabout revisited: the runabout. In Luca Cardelli, editor,

*Proceedings of the 17th European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 103–125, Darmstadt, July 2003. Springer-Verlag.

[16] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–141, London, January 2001. ACM.

[17] Mariya Koshkina. Verification of business processes for web services. Master's thesis, York University, Toronto, October 2003. Available at `www.cs.yorku.ca/~franck/students`.

[18] Mariya Koshkina and Franck van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *ACM SIGSOFT Software Engineering Notes*, 29(5), September 2004. To appear.

[19] Frank Leymann. Web services flow language. Available at `www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`, May 2001.

[20] Frank Leymann and Wolfgang Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 33(2):326–348, 1994.

[21] Ed Merks. Personal communication. June 2004.

[22] Robin Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[23] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference*, pages 9–15, Vienna, August 1998. IEEE.

[24] Jens Palsberg, C. Barry Jay, and James Noble. Experiments with generic visitors. In Roland Backhouse and Tim Sheard, editors, *Proceedings of the Workshop on Generic Programming*, pages 81–84, Marstrand, Sweden, June 1998.

[25] Terence Parr. ANTLR reference manual. Available at `www.antlr.org/doc/index.html`, May 2004.

[26] *Queue*, 1(1), March 2003.

[27] Fatima Ashfaq Ramay. Translating BPEL4WS into the BPE-calculus. Unpublished project report, August 2003.

[28] Satish Thatte. Xlang: Web services for business process design. Available at `www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`, 2001.