ANALYSIS THROUGH REFLECTION: WALKING THE EMF MODEL OF BPEL4WS

Kien Huynh
khuynh@cs.yorku.ca

A thesis submitted to the Faculty of Graduate Studies
in partial fulfilment of the requirements
for the degree of

Master of Science

Graduate Programme in Computer Science
York University
Toronto, Ontario
September 5, 2005

Abstract

In this thesis, we review different approaches to implement analyses for the Business Process Execution Language for Web Services (BPEL4WS) — a language initially proposed by BEA, IBM and Microsoft to describe business process behaviour based on web services. Analyses of a BPEL4WS program often boil down to walking the abstract syntax tree of the program. We exploit the Eclipse Modelling Framework (EMF) to generate a hierarchy of Java classes and interfaces that represents the abstract syntax of BPEL4WS. We review, refine and extend a technique, based on Java's reflection mechanism and introduced by Palsberg and Jay [44], to walk such trees. Such a walker provides us a powerful approach to analyze BPEL4WS programs. Unfortunately, since the walker relies on Java's reflection mechanism, its performance is rather poor. A significant part of our research has focused on improving its performance. Caching the results of Java's reflection as proposed by Bravenboer and Visser [9] and also by Forax and Roussel [25], and generating new code, at run time, to replace calls to Java's reflection mechanism by Grothoff [28] have been shown to improve the performance of the walker considerably. Inspired by the work of Grothoff [28], we propose preprocessing the code that performs an analysis to generate new and more efficient code for it. Our approach inherits most of the benefits of using the walker to implement analyses of BPEL4WS and gives rise to improved performance. However, the fact that the code needs to be compiled again every time the EMF model of BPEL4WS changes is its main drawback.

Acknowledgements

The major credit for the work in this thesis must go to my supervisor Franck van Breugel. I am gratefully appreciative for his many valuable suggestions, tremendous contributions, always prompt feedback, guidance, encouragement, and support during the research. My gratitude is beyond words.

Many thanks to Bill O'Farrell for his valuable suggestions, and Jane Fung for helping me on the technical issues of WASDIE. I would like to thank Frank Budinsky and Ed Merks for valuable discussions on EMF. Also, thanks Julie Waterhouse for providing a very comfortable environment at CAS, and for broadening my knowledge in AOP.

Many thanks to the committee members: Kelly Lyons who is willing to spend her time-off reading my thesis, Jonathan Ostroff who has taught me a lot about software engineering, and Sylvie Morin who has inspired me to pursue this. Last but not least, I would like to thank the Computer Science department of York University and IBM for supporting me throughout my program and research.

Finally, I would like to send my love to my parents, the rest of my family, and James. Thank you for always being supportive.

All of your support and help in very different ways were crucial to the final completion of this thesis and are always greatly appreciated.

Kien Huynh
September 5, 2005

Table of Contents

1 Introduction

This thesis is part of an ongoing effort to develop analysis tools, in Java, for the Business Process Execution Language for Web Services (BPEL4WS) [1]. The development of tools to analyze BPEL4WS programs entails two fundamental issues. They are, firstly how to build a representation of a BPEL4WS program in Java, and secondly how to implement the analysis of this representation.

1.1 The BPEL4WS language

With BPEL4WS, one can create a business process which consists of basic activities that can, for example, invoke an operation of a web service, receive a request, send a reply, and manipulate data. These activities, then, can be combined into structured activities. The structured activities are built using constructs such as sequential control flow constructs (like `sequence`, `switch` and `while`), and concurrency and synchronization constructs (like `flow`, `pick` and `switch`).

Synchronization between activities is achieved by using links. If there is a link from one activity to another, then the target activity can only start once the source activity has completed. Each activity has a join condition. The join condition is a Boolean expression which consists of links combined by Boolean operators. Once all the source activities corresponding to the incoming links of an activity have completed, the join condition of the activity is evaluated. If the join condition evaluates to true, then the activity is started. Otherwise, since the activity will never start, it can be garbage collected using a scheme known as dead-path-elimination (DPE) [1, 40] in BPEL4WS.

Since our work mostly focuses on the analyses of activities and links of BPEL4WS, we will only discuss those in this thesis. We refer the reader to [1] for a complete description of BPEL4WS.

1.2 The Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) [12] is a Java framework which enables users to generate code based on a data model. Models can be specified using annotated Java, XSD schemas, or UML. The specifications then can be imported into EMF to generate the corresponding Java classes and interfaces. EMF supports three code generation facilities. They are the model generation, the adapter generation, and the editor generation facility. The model generator is responsible for generating Java classes and interfaces that represents the data model.

The adapter generator is responsible for generating reusable classes for building editors for EMF models. Finally, the editor generator is responsible for generating a simple tree-based editor that allows one easily produce instances of an EMF model. EMF also generates a so-called EMF-switch utility class, which can be used for walking the EMF model.

For this thesis, we will only focus on EMF's model generator. Based on the XSD specification of BPEL4WS, one can exploit the model generator of the framework to generate a hierarchy of Java classes and interfaces for BPEL4WS. This hierarchy will be exploited to represent the syntax of BPEL4WS in Java.

1.3 Thesis contributions

Our contributions have been in several areas and are briefly outlined below.

1.3.1 Approaches to implement analyses of a BPEL4WS program

Analyses of a BPEL4WS program often involve validating some of the properties of the program. For example, links in a BPEL4WS program must be properly declared before being used. Generally, the analyses boil down to walking the abstract syntax tree (AST) of the program in a suitable order. When visiting a node, based on the type of the node, a corresponding code snippet is executed.

In this thesis, we review and compare a number of techniques which can be used to implement such analyses, including

- adding dedicated methods to the Java classes representing the ASTs,
- applying the syntax separate from interpretations approach [2],
- attaching external methods to nodes of particular types by exploiting the visitor design pattern [27],
- exploiting the EMF-switch mechanism [12],
- tailoring automatically generated tree walkers as supported by compiler kits like SableCC [21],
- using the reflection based techniques, which are based on Java's reflection mechanism, originally proposed by Palsberg and Jay [44] (see also [45, 9, 25, 28]).

In this thesis, we will provide a detailed discussion of the advantages and disadvantages of using each approach to analyze a program.

1.3.2 Reflection based walker

Palsberg and Jay [44] introduced a tree walker, which is based on Java's reflection, to analyze a program. Java's reflection is one of the advanced features of the Java environment that makes information about objects, classes, and interfaces available at runtime. It also allows objects to be manipulated and operated on at runtime.

The tree walker of Palsberg and Jay [44] has some limitations. It does not support walking arrays and walking graphs. Since EMF models often form a graph, this tree walker does not support walking these models. Therefore, we refine and extend this tree walker to build a more powerful reflection based walker which supports walking arrays and graphs. Furthermore, in the reflection based walker we will also consider walking both public fields and non-public ones. This feature is also essential for walking EMF models. The `Walker` class, which provides the implementation of the reflection based walker, has nearly 150 lines of code.

1.3.3 Improving the performance of the reflection based walker

The performance of the reflection based walker is rather poor. A significant part of our thesis will focus on improving its performance. There are two factors that contribute to the poor performance. One of them is the use of Java's reflection mechanism. To minimize the use of Java's reflection mechanism, we propose a technique which requires preprocessing the code (that performs the analysis) and generation of new code. A similar code generation technique had already been proposed by Grothoff [28]. However, since Grothoff [28] focused on generating code at runtime, this approach introduce some overhead. Java's reflection mechanism is not the only factor that contributes to the poor performance of the walker. Some analyses of a BPEL4WS program only require certain parts of its AST to be walked. The fact that the walkers traverse every part of the AST, hence, may also cause the performance of the walkers to be poor. In this thesis, we will also address this issue during the preprocessing step.

1.3.4 Implementing analysis tools for BPEL4WS

A graphical editor of BPEL4WS has been developed and successfully integrated with the WebSphere Studio Application Developer Integration Editor (WSADIE). On the one hand, our aim is to have the analysis tools for BPEL4WS to be pluggable into WSADIE so that the results of analyses can be reflected by the editor. On the other hand, as BPEL4WS is still evolving, the EMF model of BPEL4WS may also change. The analysis tools for BPEL4WS should be implemented in such a way that changes to the EMF model of BPEL4WS will have little impact on these tools.

Given the above constraints, we will show later in this thesis that the reflection based techniques provide the most appropriate way to implement analyses of BPEL4WS. The other approaches will be shown to be less suitable.

Exploiting the reflection based walker, we have developed a number of analysis tools for BPEL4WS, including tools

- to detect control cycles in a BPEL4WS program,
- to check that each link of a BPEL4WS program has a unique source and a unique target,
- to translate a BPEL4WS program into a BPE-process (for the latter properties can be verified using the Concurrency Workbench), and
- to check if dead-path-elimination (DPE) gives rise to side effects in a BPEL4WS program.

At the time when the research started, no such tools were available for BPEL4WS. In particular, the tool that detects side-effects of DPE is the only available tool which enables to check whether DPE gives rise to side effects in a BPEL4WS program.

1.3.5 Performance evaluation

Finally, in this thesis, we will also assess the performance of some approaches, including the walkers with preprocessing, based on the following analyses.

(a) An analysis that finds the activity in a BPEL4WS program with a given identifier (see Chapter 7).

(b) An analysis that translates a BPEL4WS program into a BPEL-calculus process (see Chapter 8).

(c) An analysis that builds a directed graph for a BPEL4WS program (see Chapter 9).

In particular, we want to compare the performance of those approaches that do not rely on Java's reflection mechanism to analyze BPEL4WS programs, to the reflection based approaches. Among the reflection based techniques, the walkers with preprocessing will be shown to obtain the best performance for these analyses.

1.4 Overview

The rest of this thesis is organized as follows. Chapter 2 provides a brief introduction to web services and then to BPEL4WS. Chapter 3 introduces EMF. In this chapter, we concentrate on how EMF generates the hierarchy of Java classes and interfaces for BPEL4WS from the XSD specification of BPEL4WS. Due to the complexity of this language, we will focus on a small fragment of BPEL4WS, that is, the join conditions, to illustrate how such a hierarchy is generated.

In Chapter 4 and 5, we review different ways to implement analyses for BPEL4WS. In Chapter 5, the reflection based walker are described, whereas Chapter 4 discusses some approaches that do not rely on Java's reflection mechanism. In Chapter 6, we present our proposal to improve the performance of the (reflection based) walker.

In Chapter 7, 8, and 9, we sketch the implementations of some analyses for BPEL4WS using different approaches. These analyses include finding the activity with a given ID within a BPEL4WS program, translating a BPEL4WS program into BPEL-calculus process, and building a graph representation for a BPEL4WS program. To give the reader an idea of how one approach performs, compared to another, in Chapter 10, we evaluate the performance of the various approaches. Chapter 11 summarizes, concludes and discusses future work.

2 Business Process Execution Language for Web Services

2.1 Introduction to web services

Recently, web services have gained considerable popularity in both industry and academia. Web services were developed based on many other technologies, one of which is the eXtensive Markup Language (XML). XML standards [10] were developed by the World Wide Web Consortium. XML plays an essential role in encoding data and creating additional markup languages, both of which will be exemplified below. XML has been adopted in many other current technologies, mainly for cross-platform data communication between different web components.

XML is a markup language that is much like HTML. The main differences between XML and HTML are that while HTML is used for displaying data XML is used to describe data; and while HTML has its own set of predefined tags, the XML tag library is defined by the users either using a Document Type Definition (DTD) [10] or an XML Schema Definition (XSD) [48, 6]. For example, one can use an XML document to encode personal information of a consumer as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<consumer id="12">
  <name>Micheal Lee</name>
  <age>25</age>
  <shipping-address>20 Pond Street, Toronto, Ontario, M3J 3P3
  </shipping-address>
</consumer>
```

Figure 2.1: An XML document.

An XML document begins with the header (the first line), followed by the content of the document. The markup tags `consumer`, `name`, `age`, and `shipping-address` and the attribute `id` used in the above document are specified using either a DTD or an XSD. The following shows how to define those tags in a DTD.

```
<!ELEMENT element-name consumer (name,age,shipping-address)>
<!ATTLIST consumer id #CDATA #REQUIRED>
<!ELEMENT element-name name (#PCDATA)>
<!ELEMENT element-name age (#PCDATA)>
<!ELEMENT element-name shipping-address (#PCDATA)>
```

Figure 2.2: A DTD document.

In the above DTD fragment, the `!ELEMENT` tag is used to define new XML library tags such as `consumer`, `name`, `age`, and `shipping-address`. The `!ATTLIST` tag specifies the list of attributes, for example, the attribute `id` in the tag `consumer`. The construct `#REQUIRED` enforces the user to specify the value of id in the XML document. The constructs #CDATA and #PCDATA are generally referring to string typed data. The difference between the two constructs is well-explained in [10].

Alternatively, one can use XSD instead of DTD to define new XML library tags. One of the advantages of using XSD over DTD is that it supports data types. Everything in DTD is defined as a string even for instance age. In contrast, XSD allows the data type for age to be defined as an integer. Moreover, XSD provides a more flexible way to define the document content, to validate the correctness of data, to define data types and so on. The tags defined in the above XML document can be specified in XSD as follows.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="consumer">
    <complexType>
      <sequence>
        <element name="name" type="string"/>
        <element name="age" type="integer"/>
        <element name="shipping-address" type="string"/>
       </sequence>
    </complexType>
    <attribute name="id" type="integer" use="required"/>
  </element>
</schema>
```

Figure 2.3: An XSD document.

The above specification in XSD is very similar to the one in DTD. XSD allows users to specify additional information. For example, XSD makes it possible to control the order of `name`, `age`, and `shipping-address` appearing inside the `consumer` tag in the XML document by means of the `sequence` tag.

Back to web services, a primary goal of web services is to promote the universal interoperability between applications. It allows one application to remotely invoke a method of another application; the former application is technically referred to as a client application and the latter as a server application. These applications may be run on different platforms and even be implemented in different languages. The client application sends a request in the form of a message, the server application responds to this request also using a message. The protocol of the message exchanges is in Simple Object Access Protocol (SOAP).

SOAP is an XML-based communication protocol between applications possibly running on different platforms and using different programming languages and communicating via the Internet (see [17]). The client application will send a SOAP-formatted request to access services provided by the server application. The server application processes the request and responds with another SOAP-formatted message.

Web services can be exploited in many different ways. For example, they can be used to make a purchase online. In such an example, a consumer sends a `buy` request to a producer. The request has three parameters, the consumer identification information `consumerId`, the item `item`, and the quantity `quantity`.

```
<?xml version="1.0"?>
<Envelope xmlns="http://www.w3.org/2001/12/soap-envelope"
       encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <Body xmlns="http://www.producer.com/purchase">
    <buy>
      <consumerId>12</consumerId>
      <item>A</Item>
      <quantity>100</quantity>
    </buy>
  </Body>
</Envelope>
```

Figure 2.4: A SOAP-formatted message.

The producer will process such request, compute the corresponding subtotal and total cost of the purchase based on the provided information, and finally respond with a SOAP-formatted message which confirms the purchase and the result of the payment computation.

```
<?xml version="1.0"?>
<Envelope xmlns="http://www.w3.org/2001/12/soap-envelope"
        encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <Body xmlns="http://www.producer.org/purchase">
    <buyResponse>
      <subtotal>132900.00</subtotal>
      <total>152800.35</total>
    </buyResponse>
  </Body>
</Envelope>
```

Figure 2.5: A SOAP-formatted message.

However, how does the consumer know where to locate the service, what type of service is provided, and what kind of information it should send, and what kind of information it will receive from the producer? All of these are made possible through the Web Service Description Language (WSDL) [14], in which we can describe the provided service. WSDL is an XML-based description language. WSDL provides information to the client, for instance publicly available methods, the type of data for all requests and responses, and so on. An example of a partial WSDL document is given next.

```
<definitions name="Purchase"
             targetNamespace="http://www.producer.org/purchase"
             xmlns="http://schemas.xmlsoap.org/wsdl" >
  ...
  <message name="buyInput">
    <part name="consumerId" type="integer"/>
    <part name="item" type="string"/>
    <part name="quantity" type="integer"/>
  </message>
  <message name="buyOutput">
    <part name="subtotal" type="float"/>
    <part name="total" type="float"/>
  </message>
```

```
    <portType name="PurchaseService">
      <operation name="buy">
        <input message="buyInput"/>
        <output message="buyOutput"/>
      </operation>
    </portType>
    ...
  </definition>
```

Figure 2.6: A partial WSDL document.

In the above example, the `message` elements `buyInput` and `buyOutput` describe the format of the request and response messages. The `portType` element describes the operation `buy`. The format and data type of the input and output for this operation are specified through the `input` and `output` elements, respectively.

2.2 Introduction to BPEL4WS

The Business Process Execution Languages for Web Services (BPEL4WS for short) is an XML-based language that allows one to create complex processes to coordinate interactions between Web services. The published draft of the BPEL4WS specification is available at [1].

Below, we present a simple BPEL4WS snippet that will invoke an operation provided by the producer service by means of `invoke`. This `invoke` is executed by the client application. As a result, a method called `buy` is executed by the server application. The three method parameters are stored in the input container `buyInput` and the result (the costs) is put in the output container `buyOutput`.

```
<invoke
  partner="producer"
  operation="buy"
  inputContainer="buyInput"
  outputContainer="buyOutput">
</invoke>
```

Figure 2.7: A simple BPEL4WS snippet.

The `invoke` is referred to as a basic activity in BPEL4WS and is used to perform web service invocations. Other basic activities include receiving a message using `receive`, replying to a message using `reply`, copying data using `assign`, throwing a fault using `throw`, and terminating a process using `terminate`.

BPEL4WS also allows one to create complex processes by composing different activities. Activities can be composed by means of structured activities such as `sequence, flow, switch, pick` and `while`. Below we will briefly describe the structured activities.

The `sequence` activity consists of one or more activities to be performed sequentially in the lexical order – the order of their occurrences in the BPEL4WS program. In the example,

```
<sequence>
   send order
   confirm order
   pay order
</sequence>
```

the activities `send order`, `confirm order` and `pay order` (the details of these activities are not specified to simplify the example) are performed in the specified order: first send the order, then confirm the order and finally pay for the order.

The `switch` activity consists of an ordered collection of one or more conditional branches defined by the `case` element, optionally followed by a default branch defined by `otherwise`. As in Java's `switch` mechanism, the `case`'s are considered in the order in which they appear. For instance,

```
<switch>
  <case condition="getVariableData(price) < 100">
    buy from A
  </case>
  <case condition="getVariableData(price) > 200">
    buy from B
  </case>
  <otherwise>
    buy from C
  </otherwise>
</switch>
```

specifies that we `buy from A` if the `price` is smaller than 100, we `buy from B` if the `price` is greater than 200, and we `buy from C` otherwise. To extract the value of the variable `price`, we use the predefined function `getVariableData`.

The `while` activity repeatedly executes its body until the specified condition no longer holds true. For example, in

```
<while condition="getVariableData(quantity) > 0">
  sell
</while>
```

the `sell` activity will be executed repeatedly until the variable `quantity` becomes non-positive.

The `flow` activity plays an important role in BPEL4WS. It enables the concurrent execution of activities. For example, in

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

the activities `buy` and `sell` are concurrent. The `flow` activity terminates after both `buy` and `sell` activities have completed.

The `pick` activity imposes a choice of which nested activities to be executed. The selection decision is made by some external events, for instance a message is received through the `onMessage` activity. In a pick, an `onMessage` is equivalent to a `receive` activity. Once an activity is selected, the others will be discarded.

The pick activity finishes when the selected activity completes. Consider the following example.

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

On the one hand, if a message from consumer is received, the activity sell will be executed. In this 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 which activity to be executed depends on the implementation of BPEL4WS.

Links are a key ingredient of BPEL4WS. Synchronization between concurrent activities is achieved by using links. They are declared only within a flow and this forms the scope of the links. For instance, we declare links l1 and l2 as follows.

```
<flow>
  <links>
    <link name="l1"/>
    <link name="l2"/>
  </links>
</flow>
```

Each link has a source activity and a target activity. The source and target activities can be basic or structured activities. For instance, we specify the source and target activities for `l1` and `l2` from the previous example as follows.

```
<flow>
  ...
  <receive>
    <source linkName="l1" ... />
  </receive>
  <receive>
    <source linkName="l2" ... />
  </receive>
  ...
  <reply>
    <target linkName="l1" ... />
    <target linkName="l2" ... />
  </reply>
</flow>
```

With each link a transition condition is associated. The transition condition 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. Again from the previous example, we associate some transition conditions to `l1` and `l2` as shown below.

```
<flow>
  ...
  <receive>
    <source linkName="l1" transitionCondition="false" />
  </receive>
  ...
  <receive>
    <source linkName="l2" transitionCondition="true" />
  </receive>
</flow>
```

To simplify the presentation, we will use, for example

$$\begin{tikzcd} \boxed{a_s} \arrow[r, "true", "\ell"'] & \boxed{a_t} \end{tikzcd}$$

to depict that link ℓ has source a_s and target a_t, and its transition condition is true.

Each activity has a join condition. The join condition consists of incoming links of the activity combined by Boolean operators. In BPEL4WS, the join conditions are specified as XPath expressions [15]. For instance, the `reply` activity can have the following join condition.

```
<flow>
  ...
  <reply joinCondition=
         "!getLinkStatus('l1') && !getLinksStatus('l2')">
    ...
  </reply>
</flow>
```

In the join condition, `getLinkStatus` is another BPEL4WS predefined function that returns the status of the specified link. Only when all the values of the incoming links are defined and the join condition evaluates to true, the `reply` activity can start. As a consequence, if its join condition evaluates to false, the activity will never start. We will use, for example,

$$\begin{tikzcd} a_s^1 \arrow[dr, "\textit{false}", "\ell_1"'] & & a_s^2 \arrow[dl, "\textit{true}"', "\ell_2"] \\ & \overset{\neg\ell_1\wedge\neg\ell_2}{a_t} & \end{tikzcd}$$

to depict that the join condition of activity a_t is $\neg\ell_1 \wedge \neg\ell_2$. Let us modify the above example by adding to it the activities a_s^3 and a_t^2 as follows.

$$\begin{tikzcd} a_s^1 \arrow[dr, "\textit{false}", "\ell_1"'] & & a_s^2 \arrow[dl, "\textit{true}"', "\ell_2"] & + & a_s^3 \arrow[d, "\textit{true}"', "\ell_3"] \\ & \overset{\neg\ell_1\wedge\neg\ell_2}{a_t^1} & & & \overset{\ell_3}{a_t^2} \end{tikzcd}$$

We use + to denote the pick construct which will select either a_s^2 or a_s^3 to execute. If a_s^3 is chosen to be performed, then activity a_t^2 will be executed after activity a_s^3 has terminated. Consequently, a_s^2 can never start and could be garbage collected. In BPEL4WS, such a "garbage collection" scheme is described as dead-path-elimination (DPE) and it can be achieved as follows.

- If a pick or switch activity is executed, all the outgoing links of those branches of the pick or switch activity which are not chosen, are set to false.

- If the join condition of an activity evaluates to false, then the activity is garbage collected after assigning false to its outgoing links.

Let us briefly return to the previous example. At first sight, one may be tempted to conclude that activity a_t^1 will never be executed. However, DPE may trigger the execution of activity a_t^1 as follows. Assume that activity a_s^3 is chosen. By DPE, the value of the link ℓ_2 becomes false. Since the value of the link ℓ_1 becomes false as well, the join condition $\neg\ell_1 \wedge \neg\ell_2$ evaluates to true. Hence, activity a_t^1 can be performed. The above can be paraphrased as DPE may have side effects which may be introduced accidentally.

The side effect in the above example is caused by negative occurrences of links in the join condition. The link ℓ_2 has a negative occurrence in the join condition $\neg\ell_1 \wedge \neg\ell_2$. As Van Breugel and Koshkina have shown in [11], 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 ℓ_1 occurs negatively, but does not give rise to any side effects.

There are many other important aspects of the BPEL4WS language which have not been discussed. Since our work focuses on the analyses of the activities of BPEL4WS, we only discussed those. We refer the reader to the specification of BPEL4WS [1] for complete details of the language.

3 The Eclipse Modelling Framework

3.1 An overview

The Eclipse Modelling Framework[1] (EMF) [12] is a Java modelling framework for Eclipse[2] [16] to help programmers rapidly turn a model into easily customizable Java code. The model is one of the key concepts on which EMF is based. A model is defined as an abstract representation of the data used by an application. In EMF, a model can be expressed in one of the following three technologies: annotated Java interfaces, XML or UML. By allowing the following conversion,

$$\begin{tikzcd}
\texttt{XSD} \arrow[dr, leftrightarrow] & & \texttt{UML} \arrow[dl, leftrightarrow] \\
& \boxed{\texttt{Ecore}} \arrow[d, leftrightarrow] & \\
& \texttt{Java} &
\end{tikzcd}$$

Figure 3.1: EMF unifies three technologies.

[1]EMF is an open source project available at [20].

[2]Eclipse is an open source project available at [19].

EMF indeed unifies these technologies [3]. A model written in one form can be converted into another. The basic idea of the framework is to provide a middle ground between two different extremes: the programming extreme and the modelling extreme. Beginning with a model either expressed in XSD or UML, the programmers can utilize EMF to generate Java classes. Then they can continue their work on these Java classes.

3.2 EMF Ecore model

All the models in EMF are represented in Ecore; a model that describes another model, a meta-model. The complete hierarchy of the Ecore model (extracted from [12]) is given below.

[3]We define the Ecore model in Section 3.2.

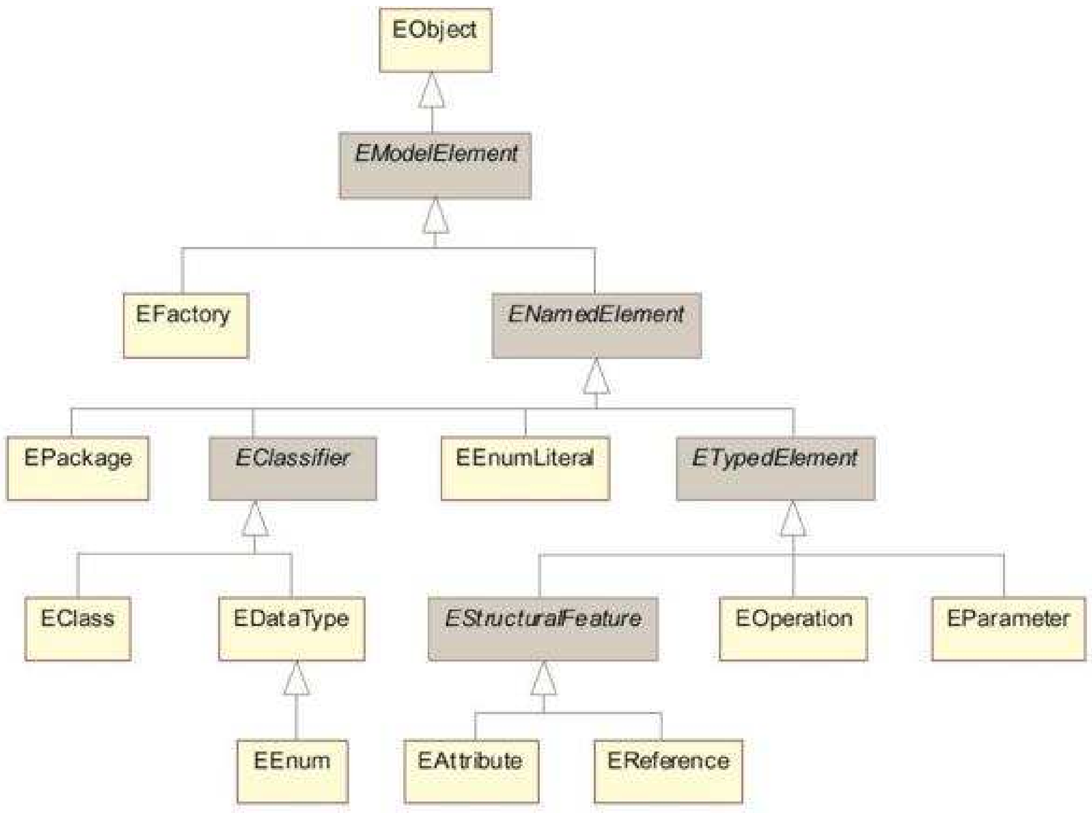


Figure 3.2: The complete hierarchy of the Ecore model.

We will only focus on the key elements in the hierarchy and refer the reader to [12] for a full description. The Ecore model uses `EClass`es to model classes. A modelled class can refer to a number of other classes as its supertypes. The modelled class can also have a number of data members. The data members are classified into two kinds, namely attributes and references. The `EAttributes` model the attributes and the `EReference`s model the references. Attributes and references have two components, its name and its type. The type of an attribute is modelled by an `EDataType` which refers to either a primitive type (`short`, `byte`, `int`, `long`, `float`, `double`, `char` or `boolean`) or to `Object`. The type of a reference is modelled by an `EDataType` which refers to an `EClass`. The behaviors of a modelled class are captured by `EOperation`s. An operation can have zero or more parameters, and a return type, all modelled by `EParameter`s. An instance of the `EParameter` holds information about the parameter name and its type.

Given an input model either in XSD, UML or annotated Java interfaces, EMF will introspect it and create an instance of an Ecore model referred to as the core model. Based on this model, the EMF code generation facility generates the implementation code for it. What kind of code does EMF generate? The EMF generator creates a corresponding interface and a class for each modelled class. The separation of the interface and the class is a design choice imposed by EMF to support multiple inheritance in Java.

Each generated interface extends the `EObject` interface. The `EObject` interface provides behavior common to all the modelled classes.

Besides the Java interfaces and classes for the model, EMF also generates two other packages: EMF.Edit and EMF.Editor. These packages contain generic reusable classes for building editors for the EMF models, and a tree-based editor.

The editor is very basic but provides the user with a convenient tool to construct instances of the model.

3.3 Methods to specify the model

As we already mentioned, the input models can be defined in terms of annotated Java interfaces, XSD, or UML. The annotated Java interfaces are Java interfaces. They are, however, annotated (using Javadoc) with additional information which conforms with EMF's annotation tags. To show some simple annotation tags, we start with an example of a purchase order extracted from [12]. Let every order contain three pieces of information, the "bill to" and "ship to" addresses, and a collection of ordered items. One would simply create the following interface

```
public interface PurchaseOrder
{
  String getShipTo();
  void setShipTo(String value);
  String getBillTo();
  void setBillTo(String value);
  List getItem();
  void setItem(List value);
}
```

Figure 3.3: The `PurchaseOrder` interface.

to represent a purchase order. For EMF to recognize this interface, it needs to be annotated with the `@model` tag. The tag indicates that the interface and the method definitions are part of a modelled class. Below, we illustrate how to annotate the purchase order interface. It is sufficient to include only the accessor methods. At generation time, EMF will automatically insert code into the annotated Java interface to define the mutator methods.

```
/**
  @model
  */
public interface PurchaseOrder
{
  /**
    @model
    */
  String getShipTo();
  /**
    @model
    */
  String getBillTo();
  /**
    @model type="Item" containment="true"
    */
  List getItems();
}
```

Figure 3.4: The annotated `PurchaseOrder` interface.

In addition to the @model tag, line 15 in the above example introduces two other tags. They are type="Item" and containment="true". Since the getItems returns a list of items whose type is Object, the user may want to restrict the type of each element in the list. This can be done through type="Item". At generation time, EMF will create a list that holds only objects of type Item. The flag containment is used to specify the has-a relation of the purchase order and the list of items. A true value indicates a composition, whereas a false value indicates an aggregation. If this flag is omitted, by default it will be understood as containment="false". EMF also recognizes many other annotation tags. Again, we refer the reader to [12] for a full list of available tags.

Returning to the example, EMF then generates the corresponding Java class for the annotated PurchaseOrder interface. Below is a simplified version of the generated class[4].

```
public class PurchaseOrderImpl extends EObjectImpl implements
    PurchaseOrder
{
  protected String shipTo;
  protected String billTo;
  protected EList items;

  protected PurchaseOrderImpl()
  {
```

[4]By convention, the name of the generated Java classes ends with Impl.

```
 9      super();
10      shipTo = null;
11      billTo = null;
12      items = null;
13    }
14
15    public String getShipTo()
16    {
17      return shipTo;
18    }
19
20    public void setShipTo(String newShipTo)
21    {
22       shipTo = newShipTo;
23    }
24    ...
25    public EList getItems()
26    {
27      if (items == null)
28      {
29        items = new EObjectContainmentEList(Item.class, ...);
30      }
31       return items;
32    }
33    ...
34  }
```

Figure 3.5: The generated `PurchaseOrderImpl` class.

Both `shipTo` and `billTo` are declared as `java.lang.String`. In the annotated Java interface, the type of `items` is defined as `java.util.List`. EMF replaces it with `org.eclipse.emf.ecore.EList`. Line 29 illustrates the creation of the list of items. `EList` is an interface that is implemented by the `EObjectContainmentEList` class. The constructor of this class requires several types of data, including the type of elements of the list, which is the `Item` class in this case. In this way, we can create a list of elements in which only objects of type `Item` can be stored. Any attempt to add incompatible objects will cause an exception to be thrown. Moreover, each generated class extends the `EObjectImpl` class. This class provides a default implementation of the behavior common to all the modelled classes.

Similarly, one can open a UML editor and create a static class diagram to describe the purchase order. For example, from this diagram

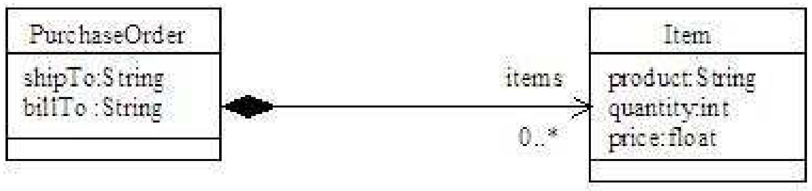


Figure 3.6: UML diagrams for the purchase order.

EMF will generate a similar collection of interfaces and classes. Lastly, just to give a flavor of how one can capture the above example in the form of an XSD, we present the following.

```
<?xml version="1.0" encoding="UTF-8"?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
    <xsd:complexType name="PurchaseOrder">
      <xsd:complexContent>
        <xsd:sequence>
        <xsd:element maxOccurs="1" minOccurs="1"
                     name="billTo" type="xsd:string"/>
        <xsd:element maxOccurs="1" minOccurs="1"
                     name="shipTo" type="xsd:string"/>
        <xsd:element maxOccurs="unbound"
                     name="items" type="Item"/>
        </xsd:sequence>
      <xsd:complexContent>
    </xsd:complexType>
</xsd:schema>
```

Figure 3.7: An XSD specification of the purchase order.

Also from the above XSD, EMF will generate a collection of classes and interfaces that are very similar to the ones we discussed earlier in this section.

3.4 Representation of BPEL4WS in Java

The BPEL4WS syntax is formally specified in terms of an XSD which can be found in [1]. From this specification, EMF generates hierarchies of classes and interfaces for BPEL4WS. These classes represent the abstract syntax of BPEL4WS. The EMF model of BPEL4WS consists of more than 500 classes. Here, we will only show a few (simplified versions) of those classes. In the EMF model of BPEL4WS, activities are represented as objects of type `Activity`. `Activity` is an interface that is extended by a number of interfaces that represent the basic and structured activities. The partial hierarchy of interfaces is presented below.

$$\begin{tikzcd}
 & & \texttt{Activity} & & \\
\texttt{Invoke} \arrow[urr] & \texttt{Receive} \arrow[ur] & \texttt{Flow} \arrow[u] & \texttt{Pick} \arrow[ul] & \cdots
\end{tikzcd}$$

Figure 3.8: The interface hierarchy of the activities.

Correspondingly, we have the following hierarchy of classes.

$$\begin{tikzcd}
 & & \texttt{ActivityImpl} & & \\
\texttt{InvokeImpl} \arrow[urr] & \texttt{ReceiveImpl} \arrow[ur] & \texttt{FlowImpl} \arrow[u] & \texttt{PickImpl} \arrow[ul] & \cdots
\end{tikzcd}$$

Figure 3.9: The class hierarchy of the activities.

In later chapters, we will discuss these interfaces and classes in more detail. In this chapter, we will only have a detailed look at an EMF model for join conditions. In the next section, we will discuss how EMF generates the hierarchies for the join conditions based on an XSD.

3.5 Representation of the join conditions in Java

To ease the presentation we simplify the join conditions and present them in terms of a BNF production as follows.

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

where ℓ is the name of an incoming link. The abstract syntax tree (AST) representing the join condition $\ell_1 \vee (\neg \ell_2 \wedge \ell_3)$ can be depicted as follows.

$$\begin{tikzcd}
 & \texttt{Or} \arrow[dl] \arrow[dr] & & \\
\texttt{Link} \arrow[d] & & \texttt{And} \arrow[dl] \arrow[dr] & \\
\ell_1 & \texttt{Not} \arrow[d] & & \texttt{Link} \arrow[d] \\
 & \texttt{Link} \arrow[d] & & \ell_3 \\
 & \ell_2 & &
\end{tikzcd}$$

Figure 3.10: The AST of the join condition $\ell_1 \vee (\neg \ell_2 \wedge \ell_3)$.

Alternatively, the syntax of the join conditions can also be specified using an XSD. Ideally, each of the join condition constructs in the above production is translated into an XML element. The BNF production

$$c ::= c \wedge c \tag{3.1}$$

is captured by the following XSD.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="Condition" abstract="true"/>
  <xsd:element name="and" type="And"/>
  <xsd:complexType name="And">
```

```
6      <xsd:complexContent>
7        <xsd:extension base="Condition">
8          <xsd:sequence>
9            <xsd:element name="left" type="Condition"/>
10           <xsd:element name="right" type="Condition"/>
11         </xsd:sequence>
12       </xsd:extension>
13     </xsd:complexContent>
14   </xsd:complexType>
15   ...
16 </xsd:schema>
```

Figure 3.11: A partial XSD specification of the join conditions.

Line 3 of the XSD fragment defines a new `Condition` type. Line 4 defines a new type `And` for the production (3.1). Line 5–14 specify other aspects of the production, such as its supertype and the right hand side symbols. For example, line 7 specifies the `Condition` as the supertype of `And`[5]; line 8–11 defines the left and right subtrees of the `And` node, each of which is also of type `Condition`. The `left` and `right` elements in the XSD fragment refer to the first and second c, respectively, in the right hand side of the production (3.1).

The XSD was chosen to specify the join conditions' syntax since we can exploit EMF to quickly generate a hierarchy of Java classes and interfaces representing the ASTs.

[5]From Appendix A, notice that `Condition` is also the supertype of `Or`, `Not`, `True`, `False` and `Link`.

EMF creates an interface for each type defined in the XSD. For example, it creates an interface `And` for the type `And`. Furthermore, it generates a Java class for each type defined in the XSD. For example, it generates a class `AndImpl` for the type `And`. The class `AndImpl` implements the interface `And`. From the complete XSD for the join conditions, which can be found in Appendix A, EMF produces a hierarchy of interfaces and a hierarchy of classes. The interface hierarchy consists of `True`, `False`, `Link`, `Not`, `And`, and `Or`, all of which extend the interface `Condition`.

$$\begin{tikzcd}
 & & \texttt{Condition} & & & \\
\texttt{True} \arrow[urr] & \texttt{False} \arrow[ur] & \texttt{Link} \arrow[u] & \texttt{Not} \arrow[ul] & \texttt{And} \arrow[ull] & \texttt{Or} \arrow[ulll]
\end{tikzcd}$$

Figure 3.12: The interface hierarchy of the join conditions.

The corresponding classes that implement these interfaces are `TrueImpl`, `FalseImpl`, `LinkImpl`, `NotImpl`, `AndImpl`, and `OrImpl`, all of which extend from the class `ConditionImpl`.

$$\begin{tikzcd}
 & & \texttt{ConditionImpl} & & & \\
\texttt{TrueImpl} \arrow[urr] & \texttt{FalseImpl} \arrow[ur] & \texttt{LinkImpl} \arrow[u] & \texttt{NotImpl} \arrow[ul] & \texttt{AndImpl} \arrow[ull] & \texttt{OrImp} \arrow[ulll]
\end{tikzcd}$$

Figure 3.13: The class hierarchy of the join conditions.

The `ConditionImpl` class, in turn, implements the `Condition` interface.

Let us consider the `And` interface and the `AndImpl` class in more detail. The class `AndImpl` has two fields, `left` and `right`, that refer to the left and right subtrees. For both fields, accessors and mutators are introduced. These methods are defined in the `And` interface as follows.

```
public interface And extends Condition
{
  Condition getLeft();
  void setLeft(Condition value);
  ...
}
```

Figure 3.14: The generated `And` interface.

The `AndImpl` class implements the above interface. In this class, the mutator method has been simplified a little.

```
public class AndImpl extends ConditionImpl implements And
{
  protected Condition left;
  protected Condition right;

  protected AndImpl()
  {
    super();
    left = null;
    right = null;
  }

  public Condition getLeft()
  {
    return left;
  }
  public void setLeft(Condition newLeft)
  {
    left = newLeft;
  }
  ...
}
```

Figure 3.15: The generated `AndImpl` class.

The implementation of `OrImpl`, `NotImpl`, `TrueImpl`, `FalseImpl`, and `LinkImpl` is similar. `Condition` is an empty interface extending `EObject` and `ConditionImpl` is an empty class extending `EObjectImpl` and implementing `Condition`.

4 Techniques to implement semantic analyses in Java

4.1 An overview

The semantic analysis is the very last phase of the front-end of a compiler. During this phase, semantic properties of a program are validated. Some typical examples of semantic validations regarding to a BPEL4WS program are: determining whether links in a BPEL4WS program are declared in a flow before being used in its nested activities; checking whether a link has a unique source and a unique target; determining whether a link creates a control cycle; and so on (see [1]).

Generally, the semantic analysis of a program boils down to traversing the AST of the program. When traversing the AST, nodes of the tree are visited. Which nodes are visited and in which order these nodes are visited differ from one analysis to another. When a node is visited, some code is executed to validate its underlying properties. Different code snippets may be associated to different types of nodes. There are a number of techniques to implement such semantic analyses, including

- adding dedicated methods to the Java classes representing the ASTs,
- applying the syntax separate from interpretations approach,
- attaching external methods to nodes of particular types by exploiting the visitor design pattern,
- exploiting the EMF-switch mechanism,
- tailoring automatically generated tree walkers as supported by compiler kits like SableCC,
- using the reflection based techniques.

We will illustrate the first five techniques by implementing the following trivial analysis for the join conditions. Given a join condition c, we want to return a collection v of all the links occurring in c. To perform this analysis, one has to traverse the AST representing c. All the `Link`s encountered during the traversal are added to v. For instance, if $c = \ell_1 \vee (\neg \ell_2 \wedge \ell_3)$ then the collection of `Link` objects corresponding to the links with names ℓ_1, ℓ_2, and ℓ_3, is returned.

The other techniques will be considered in Chapter 5.

4.2 Dedicated methods

This is a classical approach which adds dedicated methods to the Java classes. The dedicated methods provide access to and perform additional actions on the subtrees of a particular node in the AST. In order to find the collection of links `v`[6], a dedicated method, namely `extractLinks`, is inserted into each of the Java classes. The `extractLinks` method accepts `v` as its parameter. The `extractLinks` must be declared in the corresponding interfaces. In this example, it suffices to declare it in the root interface `Condition` and implement it in appropriate Java classes: `AndImpl`, `OrImpl`, `NotImpl`, `TrueImpl`, `FalseImpl`, `LinkImpl` and `ConditionImpl`. The implementation of `extractLinks` for the `AndImpl` class is given below.

```
public class AndImpl extends ConditionImpl implements And
{
  protected Condition left;
  protected Condition right;
  ...
  public void extractLinks(Collection v)
  {
    left.extractLinks(v);
    right.extractLinks(v);
  }
}
```

[6]We use `java.util.Vector` to implement this collection.

Figure 4.1: The `extractLinks` method for the `AndImpl` class.

If this method is invoked on an `And` node, it will simply ask the node's `left` and `right` subtrees to add all the links found in their trees to `v`. The implementation of `extractLinks` for the `Or` node is done in the same way. Since any `True` or `False` node neither has a subtree nor represents an incoming link, their `extractLinks` methods are simply empty. The `Not` node has only one subtree. Therefore, it only needs to extract those links found in its subtree as shown below.

```
public class NotImpl extends ConditionImpl implements Not
{
  protected Condition condition;
  ...
  public void extractLinks(Collection v)
  {
    condition.extractLinks(v);
  }
}
```

Figure 4.2: The `extractLinks` method for the `NotImpl` class.

The `extractLinks` method for a `Link` node simply adds this node to `v`.

```
public class LinkImpl extends ConditionImpl implements Link
{
  ...
  public void extractLinks(Collection v)
  {
    v.add(this);
  }
}
```

Figure 4.3: The `extractLinks` method for the `LinkImpl` class.

In order to implement another analysis, we need to add to the Java classes a new dedicated method. Conversely, if we add a new type of node to the AST hierarchy, the new class for this type of node must also implement all the existing dedicated methods. The Java hierarchy representing the ASTs for the join conditions is considerably small. Imagine that there are more than one hundred classes into which dedicate methods are to be incorporated. Inserting these methods is no longer a trivial task. Manually inserting a method into these classes has two problems: it is tedious and error-prone. Furthermore, since the analysis code is scattered across the Java classes, this makes it difficult to debug the code. As the number of the dedicated methods increases, the classes will grow in size. One should try to refrain from creating large size classes according to the object-oriented programming principles presented in [42, Chapter 21]. Lastly, if the Java classes are automatically generated from a formal specification like, for example, an XSD, then modifying these classes with dedicated methods might not be appropriate. If the specification is changed, the Java classes are generated again, and the added dedicated methods cannot be retained during the code generation; all the dedicated methods must be added again. However, aspect-oriented programming (AOP) can be made use of to address these issues.

AOP is a programming paradigm that addresses the issue of related code that “cuts” across many classes. For instance, the above analysis code to extract links “cuts” across the Java classes representing the join conditions. A detailed discussion of AOP is beyond the scope of this thesis. For more details, we refer the reader to, for example, [34]. However, we will briefly show how one can use AspectJ[7] [39, 33] — a Java extension to support AOP — to specify the additional method `extractLinks` for the Java classes. To achieve such a task, an aspect definition in AspectJ, named `LinkExtractorAspect`, is created.

```
public aspect LinkExtractorAspect
{
  public abstract void ConditionImpl.extractLinks(Collection v);
  public void AndImpl.extractLinks(Collection v)
  {
    getLeft().extractLinks(v);
    getRight().extractLinks(v);
  }
  public void OrImpl.extractLinks(Collection v)
  {
    getLeft().extractLinks(v);
    getRight().extractLinks(v);
  }
  public void NotImpl.extractLinks(Collection v)
  {
    getCondition().extractLinks(v);
```

[7]AspectJ is an open source project available at [3].

```
17   }
18   public void LinkImpl.extractLinks(Collection v)
19   {
20     v.add(this);
21   }
22   public void TrueImpl.extractLinks(Collection v)
23   {
24     // do nothing
25   }
26   public void FalseImpl.extractLinks(Collection v)
27   {
28     // do nothing
29   }
30 }
```

Figure 4.4: The `LinkExtractorAspect` class.

The AspectJ language basically extends the Java language. Line 1 of the above example introduces a new keyword `aspect` which is used to indicate an aspect, named `LinkExtractorAspect`. Line 3 illustrates the syntax of how to specify the abstract method `extractLinks` for `ConditionImpl`. Likewise, we specify and implement the `extractLinks` methods for the `AndImpl`, `OrImpl`, `NotImpl`, `TrueImpl`, `FalseImpl`, and `LinkImpl` classes. The aspect `LinkExtractorAspect` is then compiled by the AspectJ compiler. During the compilation, the bytecode representing these methods is generated and gets woven into the specified Java classes.

Using the AOP approach, we first resolve the problem of manual insertion of the dedicated methods. All the dedicated methods are actually grouped into one aspect. Therefore, the analysis code is no longer scattered. Finally, since the analysis code is preserved separately from the Java classes, regenerating these Java classes due to changes in the specification is also no longer an issue.

4.3 Syntax separate from interpretations

The syntax separate from interpretations approach described by Appel [2, Chapter 4] addresses the issue of separating the ASTs from operations performed on them. An interpretation in fact refers to an analysis to be performed on the tree. An interpretation is implemented as an external method. Generally, these external methods perform appropriate actions based on the type of the current node it is examining. This technique often leads to extensive use of Java's `instanceof`.

For the analysis, `extractLinks` is implemented as an external method and placed in a class, named `LinkExtractor`. The `extractLinks` method is recursive. It does a depth-first traversal on the tree and adds links encountered during the traversal to a `Collection` named `v`. The `extractLinks` accepts one parameter: the node of the AST that is currently visited.

```
public class LinkExtractor
```

```
{
  private Collection v;
  public LinkExtractor()
  {
    v = new Vector();
  }
  public void extractLinks(Condition c)
  {
    if (c instanceof Not)
    {
      Not not = (Not) c;
      extractLinks(not.getCondition());
    }
    else if (c instanceof And)
    {
      And and = (And) c;
      extractLinks(and.getLeft());
      extractLinks(and.getRight());
    }
    else if (c instanceof Or)
    {
      Or or = (Or) c;
      extractLinks(or.getLeft());
      extractLinks(or.getRight());
    }
    else if (c instanceof Link)
    {
      v.add(c);
    }
    else
    {
      // we do nothing if c is a True or False node
    }
  }
}
```

Figure 4.5: The `LinkExtractor` class.

Each interpretation refers to an analysis; and there may be several different interpretations for the same AST. Appel [2] argues this approach is orthogonal to the dedicated method technique in terms of modularity. This orthogonality is illustrated in the table below from which one can construct classes either horizontally or vertically.

	$\texttt{Analysis}_1$	$\texttt{Analysis}_2$	$\cdots$
`And`	●	●	
`Or`	●	●	
`Not`	●	●	
`False`	●	●	
`True`	●	●	
`Link`	●	●	

Figure 4.6: A table illustrating the orthogonal modularity.

Since the operations are external to the nodes in the ASTs, this approach does not require the Java classes to be modified. In contrast, the dedicated methods technique requires a method to be added to each Java class for each analysis.

Hence, this is an advantage over the dedicated methods technique.

In the analysis, to perform an appropriate action based on the type of a node, one needs to use type casting and Java's `instanceof`. Type casting poses a potential problem, since it may cause runtime errors. The use of type casting and Java's `instanceof` often indicates a poor design which can usually be effectively replaced by using polymorphism [29, Chapter 8].

Furthermore, if a new type of node is added to the existing AST hierarchy, one has to modify every interpretation (method) to also account for this new type of node. These modifications violate the open-closed principle of object-oriented programming [27, Chapter 3]. The next section shows a technique that takes the idea of using external methods to attach additional actions, but resolves the problem of type casting and violation of object-oriented programming principles.

4.4 Visitor design pattern

Similar to the syntax separate from interpretations approach, the fundamental idea of using the visitor design pattern [27, Chapter 5] is to separate the analysis code from the AST classes by defining external methods for different types of nodes. It also allows us to attach new operations to the nodes in the ASTs with a minimal alteration of the Java classes. The external methods are typically overloaded methods and are normally named `visit`. The class that hosts these methods is called `Visitor`. Each `visit` method has a parameter whose type is the type of the node to which it attaches. For example,

```
public class Visitor
{
  public visit(Node node)
  {
    // actions
  }
}
```

Figure 4.7: A visitor class.

where `Node` is a type of a node in the ASTs, specifies that the `actions` should be performed when visiting a `Node`. The attachment is achieved by inserting into each Java AST class a method often referred to as the `accept` method. The `accept` method has only one parameter. It is uniformly implemented across all the classes:

```
public void accept (Visitor visitor)
{
  visitor.visit(this);
}
```

The `accept` method acts as a type indicator for the nodes in the ASTs such that the appropriate `visit` method will be invoked once it executes. The `Visitor` and the accept method interact back and forth in such a way that the Visitor asks the node its type by invoking the accept on this node, and the node answers by invoking the appropriate visit method of the Visitor. By using such a mechanism, the type casting and Java instanceof are actually eliminated.

To implement the analysis, we need to add the accept method to the following Java classes: AndImpl, OrImpl, NotImpl, LinkImpl, TrueImpl, FalseImpl, and ConditionImpl. It suffices to declare it in the root interface Condition and make it abstract in the ConditionImpl class. Then we create a LinkExtractor class which acts as a Visitor. The LinkExtractor consists of a number of overloaded visit methods each of which is for a particular type of node.

```
public class LinkExtractor
{
  private Collection v;
  public LinkExtractor()
  {
    v = new Vector();
  }
  public void visit(False node)
  {
    // do nothing
  }
  public void visit(True node)
  {
    // do nothing
  }
  public void visit(Not node)
```

```
    {
      node.getCondition().accept(this);
    }
    public void visit(And node)
    {
      node.getLeft().accept(this);
      node.getRight().accept(this);
    }
    public void visit(Or node)
    {
      node.getLeft().accept(this);
      node.getRight().accept(this);
    }
    public void visit(Link node)
    {
      v.add(node);
    }
  }
```

Figure 4.8: The `LinkExtractor` class.

In this approach, the modification to the generated Java classes is minimized although not completely eliminated. The `accept` method is the same for every class, and thus insertion of this method to the appropriate classes is less complex. Moreover, the `accept` method can be shared among different analyses. Therefore, the method insertion is only done once. A better design of this technique involves defining a `Visitor` interface consisting of an overloaded `visit` method for each type of node in the ASTs. In the example below,

```
public interface Visitor
{
  public void visit(True node);
  public void visit(False node);
  public void visit(And node);
  public void visit(Or node);
  public void visit(Not node);
  public void visit(Link node);
  public void visit(Condition node);
}
```

Figure 4.9: The `Visitor` interface for the join conditions.

the `Visitor` interface defines a generic interface for the analyses. Consequently, different classes serving for different semantic analyses can implement the interface. In the example,

```
public class Analysis1 implements Visitor
{
  // implementation of the interface
}
public class Analysis2 implements Visitor
{
  // implementation of the interface
}
```

Figure 4.10: Analysis classes.

there are two different analyses, namely `Analysis1` and `Analysis2`, each of which provides a different implementation for the `Visitor` interface. Due to this design, switching between different analyses becomes simpler and more convenient. The other advantage of using the visitor design pattern is that we can add a new analysis without changing the AST interfaces and classes, even without re-compiling them.

Watt and Brown [49] suggest modifying the `visit` and `accept` methods to include a return value and an argument. The additional information can be passed down or up the tree. The main advantage of this version over the previous one is that, one can propagate local information from parent/child to child/parent nodes using this argument parameter or the return object. For instance, in validating a BPEL4WS program whether used links are declared and whether declared links are used, the parameter and return value can be used as follows. On the one hand, the parameter passes a collection of declared links from a flow parent to the children nodes. Any link used in the children nodes but not be found in this collection is thereby identified. On the other hand, the return value returns all the links used in the children nodes to the parent flow node where the links are declared. Any link unnecessarily declared in the parent node that is not used in the children nodes is also identified. However, different analyses may require different kinds of information needed to be passed around the tree. Therefore, the return value and the argument must be declared as `Object` types to make the `visit` method as generic as possible. Due to this, they must be cast into appropriate types before any operation can be performed on these objects. Again, the type casting carries a potential problem of runtime errors.

4.5 The EMF-switch mechanism

For every generated EMF model, EMF generates a switch class [12, Chapter 9]. For example, for the EMF model whose package is named `JoinCondition`, EMF generates a class `JoinConditionSwitch`. This class implements a mechanism which can be used to associate certain code snippets to certain types of nodes. Within this class, a set of methods `caseNODE_TYPE`, one for each type of node, is defined. For example, the `JoinConditionSwitch` defines the following method

```
Object caseAnd(And and)
{
  return null;
}
```

for the `And` node, and similar methods for `Or`, `Not`, `True`, `False`, `Link` and `Condition`. The class `JoinConditionSwitch` also contains the following method.

```
Object defaultCase(EObject object)
{
  return null;
}
```

To associate a code snippet with a certain type of node, the `caseNODE_TYPE` method for that node needs to be overridden, as we will show later.

Assume `A` is a subclass of `B` and `B` is a subclass of `C`. For an `object` of type `A`, the following code snippet will be generated.

```
Object result = caseA((A) object);
if (result == null) result = caseB((B) object);
if (result == null) result = caseC((C) object);
if (result == null) result = defaultCase(object);
```

Note that the result of the `caseNODE_TYPE` method is used to determine if other `caseNODE_TYPE` methods are called as well. When one overrides a `caseNODE_TYPE` method, one generally returns a non-`null` result.

EMF also generates the following interface.

```
interface JoinConditionPackage
{
  int AND = 1;
  int OR = 2;
  int LINK = 3;
  int NOT = 4;
  int TRUE = 5;
  int FALSE = 6;
  int CONDITION = 7;
  ...
}
```

Figure 4.11: The `JoinConditionPackage` interface.

This interface contains a constant for each type of node. These constants are used to select the appropriate code snippet as shown below.

```
public class JoinConditionSwitch
{
  public Object doSwitch(EObject object)
  {
    switch (object.eClass().getClassifierID())
    {
      case JoinConditionPackage.AND:
      {
        Object result = caseAnd((And) object);
        if (result == null)
          result = caseCondition((Condition) object);
        if (result == null)
          result = defaultCase(object);
        return result;
      }
      case JoinConditionPackage.OR:        {  ... }
      case JoinConditionPackage.LINK:      {  ... }
      case JoinConditionPackage.NOT:       {  ... }
      case JoinConditionPackage.TRUE:      {  ... }
      case JoinConditionPackage.FALSE:     {  ... }
      case JoinConditionPackage.CONDITION: {  ... }
      default: return defaultCase(object);
    }
  }
  ...
}
```

Figure 4.12: The generated EMF-switch utility class for the join conditions.

The method `doSwitch` takes an `EObject object` as an argument. If this `object` is an instance of one of the classes belonging to the package `JoinCondition`, then it acquires the unique identifier (ID) representing the type of `object`. Based on this ID, it switches to the appropriate code snippet of `caseNODE_TYPE` calls. It executes these `caseNODE_TYPE` methods until one of them returns a non-`null` result.

To use the EMF-switch mechanism, one must declare an instance of the `JoinConditionSwitch` and override some of the `caseNODE_TYPE` methods. For example, in

```
public class LinkExtractor
{
  private Collection v;
  private JoinConditionSwitch s =
    new JoinConditionSwitch()
    {
      Object caseAnd(And and)
      {
        doSwitch(and.getLeft());
        doSwitch(and.getRight());
        return and;
      }
      Object caseOr(Or or)
      {
        doSwitch(or.getLeft());
        doSwitch(or.getRight());
        return or;
      }
```

```
        Object caseNot(Not not)
        {
          doSwitch(not.getCondition());
          return not;
        }
        Object caseLink(Link link)
        {
          v.add(link);
          return link;
        }
      }
    public void extractLinks(Condition c)
    {
      s.doSwitch(c);
    }
  }
```

Figure 4.13: The `LinkExtractor` class.

an instance variable `s` is created and a number of `caseNODE_TYPE` methods are overridden to specify the tree traversal. Notice that within the `caseAnd`, we use `doSwitch` to ensure that corresponding code snippets for the `left` and `right` nodes are called. We return the node itself (a non-`null` value) to stop the `doSwitch` method calling `caseCondition` for this node.

The EMF-switch mechanism is very similar to the syntax separate from interpretations approach. The main difference is that the EMF-switch mechanism uses IDs and a switch statement, whereas the syntax separate from interpretations approach uses Java's `instanceof` and conditionals. Both approaches use casting to invoke the corresponding code snippet for a node type. As a consequence, the EMF-switch mechanism bears the same potential consequences of using type casting as the syntax separate from interpretations approach.

4.6 SableCC's tree walkers

Writing compilers used to be a time-consuming and difficult task. It also required a lot of knowledge of compiler technology. Fortunately, the development of many compiler tools has simplified this task significantly and made it easier to build compilers with a limited knowledge of compilers. These compiler tools are referred to as compiler generators. They normally take a formal specification of a programming language as input and generate a partial front-end compiler (the lexer and/or parser) for it. Over the years, we have seen the development of various compiler generators for Java. To name a few, there are Java Cup [30], JFlex [35], JLex [5], SmartTools [4], ANTLR [46] and SableCC [21]. Some compiler generators are more powerful than the others. For instance, SmartTools, ANTLR and SableCC not only generate a lexer and a parser, but also partial or even complete tree walkers to accommodate the semantic analysis phase. In this section, we will briefly look at the SableCC framework.

SableCC is a Java compiler generator that does the following.

- It generates a lexer.
- It generates a parser.
- It generates depth-first and reversed depth-first tree walker classes which the programmer extends to perform specific analyses on the tree.

In order to exploit SableCC, the join conditions need to be specified in terms of the SableCC framework. Below, we provide a brief discussion of the design pattern imposed on the generated tree walkers, which is the visitor design pattern with some major modifications.

SableCC provides the `Switch` and `Switchable` interfaces.

```
interface Switch {}
interface Switchable
{
  void apply(Switch sw);
}
```

Figure 4.14: The `Switch` and `Switchable` interfaces.

The `Switch` interface is an ancestor of all switch interfaces in the framework. The `Switchable` interface defines a method, named `apply`, that plays a role similar to the `accept` method in the visitor design pattern. All the AST classes must implement this interface and provide an implementation of the `apply` method. For example, the generated interface `And`[8] not only extends the `Condition` interface but also the `Switchable` interface

```
interface And extends Condition, Switchable
{
  ...
}
```

Figure 4.15: The generated `And` interface.

and the generated `AndImpl` class will provide the following implementation of the `apply` method.

```
class AndImpl extends ConditionImpl implements And
{
  ...
  void apply(Switch sw)
  {
      ((Analysis) sw).caseAnd(this);
  }
}
```

Figure 4.16: The generated `AndImpl` class.

The method `caseAnd` is defined in the `Analysis` interface. The `Analysis` interface is also generated by SableCC. It extends the `Switch` interface and consists

[8]SableCC has different naming conventions. However, to make it easier for the reader to follow, we have modified the names of the generated (AST) classes.

of the following methods

```
interface Analysis extends Switch
{
  void caseAnd(And node);
  void caseOr(Or node);
  void caseLink(Link node);
  void caseNot(Not node);
  void caseTrue(True node);
  void caseFalse(False node);
  void defaultCase(Condition node);
}
```

Figure 4.17: The generated `Analysis` interface.

for different types of nodes in the ASTs representing the join conditions. Since all the generated interfaces extend the `Condition` interface, we use the `defaultCase` method to capture the default implementation for all the node types in the ASTs. Additionally, a class implementing `Analysis` called `AnalysisAdapter` is also created. It provides a default implementation for each of the `caseNODE_TYPE` methods. For example,

```
void caseAnd(And node)
{
  defaultCase(node);
}
```

where `defaultCase` is an empty method. Finally, SableCC creates two classes

`DepthFirstAdapter`[9] and `ReversedDepthFirstAdapter`[10] which, in turn, extend the class `AnalysisAdapter`. These tree walkers redefine the `caseNODE_TYPE` methods to specify the tree walking order. Back to the analysis, the `LinkExtractor` class can extend either one of them. Let us choose the `DepthFirstAdapter` and override the `caseLink` method for the `Link` node as follows.

```
public class LinkExtractor extends DepthFirstAdapter
{
  private Collection v;
  public LinkExtractor()
  {
    super();
    v = new Vector();
  }
  public void caseLink(Link node)
  {
    v.add(node);
  }
}
```

Figure 4.18: The `LinkExtractor` class.

The main advantage of using SableCC is that it already provides two predefined tree walkers, thus the implementation of the semantic analysis often becomes much simpler. However, the SableCC tree walkers are limited to walking

[9]The order of traversal is top-down and left to right.

[10]The order of traversal is top-down and right to left.

only the ASTs defined in terms of its framework. The BPEL4WS language already has its own set of AST classes generated by EMF. Furthermore, BPEL4WS is an XML-based language. SableCC does not support the specification of an attributed XML element. Therefore, the framework may not be very appropriate to implement the semantic analysis for some programming languages, such as BPEL4WS.

5 Reflection based techniques to traverse ASTs

5.1 An overview

In this chapter, we focus on the three other techniques to walk ASTs. These techniques are based on Java's reflection mechanism (see, for example, [26]), implemented in the package `java.lang.reflect`. The basic idea of the reflection based techniques was initially put forward by Palsberg and Jay [44] (see also [45]) and refined and improved by Bravenboer and Visser [9] (see also [25]). These techniques exploit Java's reflection mechanism to access (some of) the fields of the objects representing an AST in order to walk the tree. They also use this reflection mechanism to execute the appropriate code snippets when visiting nodes of the tree. As we will see, the reflection based techniques have several advantages over the techniques presented in the previous chapter. Unfortunately, the major concern against using these techniques has been their poor performance.

5.2 Reflection based walker

Originally, the reflection based tree walker was introduced as the Walkabout by Palsberg and Jay [44]. This tree walker has some limitations. The improved version suggested by Bravenboer and Visser [9], and Forax and Roussel [25] addresses some issues which were not dealt with in this tree walker. In this section, we will implement a walker based on the idea of the Walkabout. We introduce the abstract class `Walker` which forms the basis for the walker. It provides the implementation for how to traverse the tree. To implement an analysis, 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) { ... }
```

The algorithm for the `Walker` is as follows.

```
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);
        }
      }
    }
  }
}
```

Figure 5.1: The algorithm for the `Walker` class.

Clearly, the `walk` method is similar to a depth first traversal. Once the `visit` method is found, it will be invoked and the subtree of such a node will not be traversed. Otherwise, it continues to traverse the subtrees via the `walkFields` method.

Let us briefly discuss the differences between the above code and the code presented by Palsberg and Jay [44] and by Bravenboer and Visser [9]. 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 as part of the tree. Bravenboer and Visser [9] also do not consider static fields, but Palsberg and Jay [44] do. Palsberg and Jay [44] and also Bravenboer and Visser [9] 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 ASTs public, hence violating the object-oriented principle of encapsulation. 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 [44] do not walk objects of type `Number`, `Boolean` and `Character`, whereas we follow Bravenboer and Visser [9] 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.

Palsberg and Jay [44] only consider `visit` methods that are part of the class that walks the tree. On the other hand, Bravenboer and Visser [9] 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 one of the examples 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 [9] 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` and assume that the walker has the following `visit` methods

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

In this case it is not evident which of the above two visit methods should be chosen when visiting a node of type `C`. Next, we implement the `LinkExtractor` by means of a reflection based walker.

```
class LinkExtractor extends Walker
{
   private Collection v;
   LinkExtractor()
   {
      super();
      v = new Vector();
   }
   void visit(Link link)
   {
      v.add(link);
   }
}
```

Figure 5.2: The `LinkExtractor` class.

To start the walk, one must invoke the `walk` method on the tree:

```
linkExtractor.walk(c)
```

where `linkExtractor` is an instance of the `LinkExtractor` class and `c` represents the root node of the AST for a join condition. Invoking the `walk` method will initiate the walking process. On the one hand, if a non-`Link` node is encountered, the `Walker` will walk the children of this node. On the other hand, if a `Link` node is encountered, the `visit` method of the `LinkExtractor` will be executed.

5.2.1 Method resolution

In the `walk` algorithm, in order to verify whether "`this class has a visit method that takes o as an argument`" and return this method for invocation, we employ a simple strategy for finding it. Below is a recursive algorithm, called `getVisitMethod` which, given a type `t`, returns a `visit` method. This method has a parameter type, which is either the same as `t` or a superclass of `t`.

```
Method getVisitMethod(Class t)
{
  if (t == null)
  {
    return null;
  }
  else if (exists a visit method whose parameter has type t)
  {
    return this method;
  }
  else
  {
    return getVisitMethod (superclass of t)
  }
}
```

Figure 5.3: The algorithm for the `getVisitMethod` method.

The algorithm first tries to find a `visit` method whose parameter has type `t`. For this, it exploits Java's reflection mechanism. If no such method is found, it will recursively try to find a `visit` method for the superclass of `t`. The interfaces of `t` are not considered due to the ambiguity mentioned earlier. The presented algorithm is inefficient as it depends on Java's reflection mechanism.

5.2.2 Walking arrays

Neither Palsberg and Jay [44] nor Bravenboer and Visser [9] 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]);
      }
    }
  }
```

Figure 5.4: The algorithms for the methods `walk` and `walkArray`.

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
{
  private int sum;
  Adder()
  {
    super();
    sum = 0;
  }
  void visit(Integer i)
  {
    sum += i.intValue();
  }
}
```

Figure 5.5: The `Adder` class.

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
{
  private int product;
  Multiplier()
  {
      super();
      product = 1;
  }
  void visit(Integer[] a)
  {
      sum = 0;
      walkArray(a);
      product *= sum;
  }
}
```

Figure 5.6: The `Multiplier` class.

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

5.2.3 Walking graphs

If we apply the method walk to a collection of objects that forms a graph, rather than a tree, then the traversal unfortunately may not terminate. In the EMF model of the join conditions, every child node also contains a reference to its parent node. For instance, in a tree representing a join condition, an And node will have references to its left and right children nodes. Each of these children nodes, in turn, also contains a reference to its parent node. This two-way reference makes it convenient and efficient to perform top-down as well as bottom-up tree traversals. However, this will form cycles and turn the tree into a graph. Because of the presence of cycles, we have to refine the above introduced walker to also handle graphs. Palsberg and Jay [44] do not walk graphs. A simple solution is either to remember or mark those objects that have already been walked. The first solution does not require any change to the graph. Therefore, we will implement it by keeping all objects that have been walked so far in a container called walked. The algorithm is modified as follows.

```
private Collection 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);
      }
    }
  }
}
```

Figure 5.7: The modified algorithm for the `walk` method.

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 referring to the latter object. Whenever the traversal reaches an object for which a `visit` method has been introduced, then the corresponding code snippet is executed.

5.2.4 Caching the results

The processes of resolving methods and retrieving fields by means of Java's reflection mechanism are expensive. Ideally, we want to cache the results: the `visit` methods and the fields. To enable caching, we use a table that maps a type to its corresponding `visit` method. During a walk, as soon as a `visit` method is found for the currently walked node, it is mapped into the table. If the same type of node is encountered again, the `visit` method can be obtained directly from the table. As a result of that, method resolution of a particular type is only done once.

Let `visits` be an empty hash map consisting of key-value entries. The key represents the type of node and the value represents the `visit` method for this type. The algorithm `getVisitMethod` is adjusted as follows to support caching:

```
Method getVisitMethodWithCaching(Class t)
{
  if (exists a key t in visits)
  {
    m = visits.get(t);
  }
  else
  {
    m = getVisitMethod(t);
    add (t, m) to visits;
  }
  return m;
}
```

Figure 5.8: The algorithm for the `getVisitMethodWithCaching` method.

We also want to cache the (non-static and non-primitive) fields of a particular type. Again, let `fields` be an empty hash map consisting of key-value entries. The key represents the type of a node and the value represents the collection of the fields for this type.

```
Collection getFields(Class t)
{
  if (there exists a key t in fields)
  {
    c = fields.get(t);
  }
  else
  {
    c = empty collection;
    for (each field f of t)
    {
      if (f is not static and f is not of primitive type)
      {
        c.add(f);
      }
    }
    add (t, c) to fields;
  }
  return c;
}
```

Figure 5.9: The algorithm for the `getFields` method.

5.2.5 Benefits

The main advantage of the reflection based approach 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. Furthermore, very often a class hierarchy can have multiple versions of it. This makes it difficult for programmers to make analyses implemented for one version of the class hierarchy also work for the other version. However, with the reflection based approach analyses written for one version of the class hierarchy can be easily adjusted so that they will also work for the other.

The reflection based approach has other advantages as well. The abstract walker contains all the necessary code to mechanically navigate the tree (in arbitrary order) in the `walk` method. Hence, the user needs not to write such code. This also allows the users to focus their attentions on particular types of nodes, therefore, making it easier to write the analysis. The `Walker` class often eases the development of the semantic analysis phase of a programming language.

In addition, the `Walker` class is highly re-usable and flexible. It can be re-used in many different analyses. It not only can walk trees but also graphs; not only one kind of tree/graph but any kind of trees/graphs. Finally, the `Walker` class also helps eliminate the problem of modification to the Java AST classes.

5.2.6 Drawbacks

Unfortunately, the `Walker` class is extremely inefficient. Firstly, it is caused by the poor performance of Java's reflection mechanism. Secondly, it is caused by the process of walking the fields of a node. Remember that the walker will literally scan through each non-static and non-primitive field (including the inherited ones) and check whether there exists a corresponding `visit` method for it. If the `visit` method is not found, this process of walking the fields continues until it reaches a point where the class contains no such field. This process is very expensive. Even with the caching technique in place, the `Walker`'s performance still does not improve significantly. Finally, in the walker the children nodes are walked in no particular order. Therefore, the reflection based walker may not be appropriate for some analyses where the order of visiting the children nodes is important.

5.3 The Runabout

The Runabout is proposed by Grothoff [28]. Fundamentally, it is also based on the original idea of the Walkabout. Like in the Walkabout, Java's reflection mechanism is used to find the `visit` method in the Runabout. However, unlike the Walkabout, it does not use Java's reflection mechanism to invoke the method. Instead, it uses dynamic code generation to create bytecode that in turn uses type casting to correctly invoke the corresponding `visit` method. Writing the link extractor using the Runabout is very similar to writing it using the `Walker` class described in the previous section.

```
class LinkExtractor extends Runabout
{
  private Collection v ;
  LinkExtractor()
  {
    super();
    v = new Vector();
  }
  void visit(And and)
  {
    visitAppropriate(and.getLeft());
    visitAppropriate(and.getRight()) ;
  }
  void visit(Or or)
  {
    visitAppropriate(or.getLeft());
    visitAppropriate(or.getRight()) ;
  }
  void visit(Link link)
  {
    v.add(link);
  }
  void visit(Not not)
  {
    visitAppropriate(not.getCondition());
  }
}
```

Figure 5.10: The `LinkExtractor` class.

Let us briefly describe how the Runabout works. First, all the analysis code must extends the `Runabout` class to use the provided facilities. The `Runabout` class provides method resolution and invocation via `visitAppropriate`. If we perform a `visitAppropriate` on a node of type `And`, this method will look up the proper `visit` method in the `LinkExtractor` for `And` using Java's reflection mechanism. However, once the method is found, instead of reflectively invoking this method, it first generates an instance of `Code`, say `GenCodeAnd`. The `Code` class is central to the implementation of the Runabout, and defined abstractly as follows:

```
public static abstract class Code
{
  public abstract void visit(Runabout r, Object o);
}
```

Figure 5.11: The `Code` class.

The class `GenCodeAnd` is generated on-the-fly and dynamically loaded in the virtual machine using Java's class-loading mechanism. It basically provides the concrete implementation for the `visit` method defined in the `Code` class.

```
class GenCodeAnd extends Code
{
  void visit(Runabout r, Object o)
  {
    ((LinkExtractor) r).visit((And) o);
  }
}
```

Figure 5.12: The generated `GenCodeAnd` class.

Once the `GenCodeAnd` is loaded, the following line of code

```
GenCodeAnd.visit(this, object)
```

is executed. The parameter `this` refers to an instance of `LinkExtractor` and `object` refers to the `And` node. Consequently, the correct `visit` method in the `LinkExtractor` is invoked due to the type casting as illustrated in the above example. The `GenCodeAnd` is then cached for later use.

The Runabout does improve the performance of the reflection based walker (see Chapter 10). However, a disadvantage is that, since the code is generated dynamically, it requires some overhead time. The total number of the generated instances of `Code` will be as many as the total number of the `visit` methods. One should be aware of these generated classes to prevent a situation in which multiple classes have the same name.

Note that the `Runabout` does not provide a mechanism to walk the tree. Its main focus is to improve the method invocation mechanism. Therefore, for an analysis, one must instruct it how to traverse the tree, for example as in line 11 and 12 of the `LinkExtractor` class.

5.4 The multi-methods

In object-oriented languages, multi-methods refer to selecting an appropriate method at run-time based on the dynamic types of the target object and the arguments. Although multi-methods is a powerful feature [13], it is not widely supported in the currently popular object-oriented languages. For example, Java does not support multi-methods. The reason is that "the suitability of multi-methods in practice is in question because, supporting multi-methods, most importantly solving the *multi-method dispatching* problem, is believed to be space- and time-intensive" [22].

Below, we show how multi-methods can be made use of to simplify the implementation of the analyses of programming languages. In particular, we will use the Java multi-methods framework[11] proposed in [23] (see also [25]) to implement the `LinkExtractor` class.

The Java multi-methods framework is a framework that extends the Java language to support multi-methods. This framework also depends on Java's reflection mechanism to resolve and invoke a correct method. To use it, we first define the following class

```java
class JavaMultiMethod
{
  MultiMethod mm;
  JavaMultiMethod()
  {
    mm = MultiMethod.create(this.getClass(), "visit", 1);
  }
  void call(Object object)
  {
    try
    {
      mm.invoke(this, new Object[]{object});
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

Figure 5.13: The `JavaMultiMethod` class.

[11]Available for download at `igm.univ-mlv.fr/~forax/works/jmmf`.

in which an object `mm` of type `MultiMethod` is declared. This object holds a set of overloaded `visit` methods. Generally, these `visit` methods are defined in the analysis code, for example, in `LinkExtractor`. The object `mm` is created by the static method `create`. The `create` method has three parameters. The first parameter is used for specifying the host class of the `visit` methods. The second one is for specifying the method name, which is `visit`. The last one is for specifying the number of method arguments. The `call` method (line 8) takes an `Object` named `object` as its argument. It then tries to find its corresponding method — a `visit` method whose parameter has a type the same as `object` or a supertype of `object`. If such a method is found, it is invoked. An exception is thrown otherwise. The process of method selection and invocation takes place within the `invoke` method in the `MultiMethod` class.

For the link extractor, we define all the `visit` methods in the `LinkExtractor` class as follows.

```
class LinkExtractor extends JavaMultiMethod
{
  private Collection v;
  LinkExtractor()
  {
    super();
    v = new Vector();
  }
  void visit(Link link)
  {
    v.add(link);
  }
  void visit(And and)
  {
    call(and.getLeft());
    call(and.getRight()) ;
  }
  void visit(Or or)
  {
    call(or.getLeft());
    call(or.getRight()) ;
  }
  void visit(Not not)
  {
    call(not.getCondition());
  }
  void visit(Condition condition)
  {
    // empty
  }
}
```

Figure 5.14: The `LinkExtractor` class.

To initiate the walking process on a tree for a join condition, we execute the following line of code

```
  linkExtractor.call(c)
```

where `linkExtractor` is an instance of the above class and `c` represents the join condition. In this example, the `mm` object will hold five overloaded `visit` methods. Depending on the type of `c`, the appropriate `visit` method is invoked. Again, the Java multi-method framework does not provide the tree walking facility. Therefore, in the `LinkExtractor` class we must instruct it how to walk the tree as, for example, in line 15 and 16.

In the multi-methods approach, the type of the parameter of a `visit` is not limited to a class as in the walker, but can also be an interface. This allows more generic code to be written. Moreover, a `visit` method can have an unlimited number of parameters. This becomes useful when one needs to pass some local information up or down the tree using the additional parameters. For instance, to verifying whether a link is declared before used and to check whether a link is used if declared, one can re-write the `visit` method to

```
visit(Flow flow, Link[] declaredLinks, Link[] usedLinks)
```

This allows a parent flow node to check whether a declared link is being used by its children nodes and whether any used link is properly declared.

However, this framework has a large performance penalty, caused by the `create` method. Within this method, a pre-computation step for method resolution is carried out. During this process, an underlying data structure representing relations between types is pre-computed [23].

6 Walker with preprocessing

Our aim is to implement analyses for BPEL4WS. In our implementation we would like to exploit EMF, in particular the hierarchy of classes and interfaces generated by EMF from the XSD of BPEL4WS. Since the BPEL4WS graphical editor has been integrated with the WebSphere Studio Application Developer Integration Editor (WSADIE), we would like our implementation of analyses to be pluggable into WSADIE, so that the results of the analyses can be reflected in the graphical editor. Because (the XSD of) BPEL4WS still changes regularly, we would like our implementation of analyses to be fairly robust with respect to those changes. Given these criteria, let us briefly review the implementation techniques described in Chapter 4 and 5.

Since we want to exploit the classes and interfaces generated by EMF to represent the ASTs, modifying these classes and interfaces may be impractical. Firstly, the modifications will be lost if these classes and interfaces are generated again (because, for example, the XSD of BPEL4WS has changed). Secondly, changes to the Java classes and interfaces may cause changes to the EMF model representing BPEL4WS. Altering the model may have an impact on other tools that manipulate the model and, hence, should be avoided. Because both the dedicated methods approach and the visitor design pattern approach make changes to the classes representing the ASTs, these approaches are not applicable in our setting. The syntax separate from interpretations approach and EMF's switch mechanism are not very suitable either. If the XSD of BPEL4WS changes and, hence, introduces changes to the hierarchy of classes and interfaces, then the code for all those classes that have been affected by the changes may have to be modified. Hence, these approaches may not be most appropriate either. To exploit SableCC, the XSD of BPEL4WS must be translated into a SableCC specification. However, as the XSD of BPEL4WS continues to evolve, it can be time-consuming to maintain the corresponding SableCC specification for BPEL4WS. Furthermore, SableCC does not support the specification of an XML-based language, whereas BPEL4WS is XML-based. The aspect-oriented approach and the multi-methods approach also do not suit our purpose very well. WSADIE currently does not directly support AspectJ. Neither does it support multi-methods. Therefore, the implementation of the analyses cannot be plugged into WSADIE.

The reflection based approaches seem to be the most appropriate choice to implement analyses in our setting. Firstly, it does not require changes to the classes and interfaces generated by EMF. Secondly, any changes to the class hierarchy representing the ASTs have very little effect on the code that performs the analyses. Finally, the implementation can be plugged into WSADIE. Unfortunately the performance of the reflection based approaches is rather poor. In this section, we revisit these approaches. Our goal is to develop a reflection based approach with better performance that still fits our setting.

6.1 Causes of poor performance

Let us discuss the two main causes of the poor performance of the reflection based approaches and give an outline how we attempt to address these causes. First of all, the poor performance of reflection based walkers is caused by the poor performance of Java's reflection mechanism. To address this cause of poor performance, we exploit a technique similar to the one proposed by the Runabout. Recall that the Runabout does not use Java's reflection mechanism to invoke the `visit` methods. Instead, it uses dynamic code generation to create bytecode that invokes the `visit` method. Since the bytecode is generated at runtime, it causes some overhead. Therefore, we do some preprocessing during which the Java code that invokes the `visit` method will be generated. Secondly, in the reflection based walkers a lot of time may be spent on unnecessary walking of subtrees.

Consider, for example, the following AST

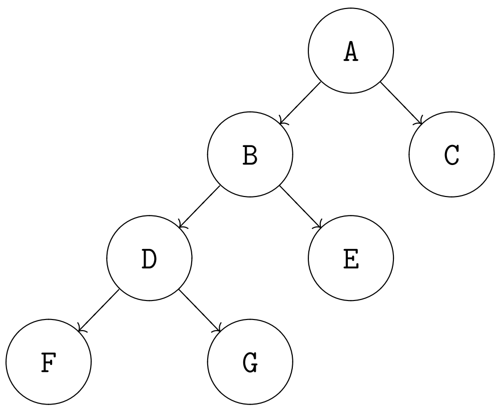


Figure 6.1: An abstract syntax tree.

and the following code in which a `visit` method is introduced for nodes of type `C`.

```
class Analysis extends Walker
{
  public void visit (C c)
  {
    // actions
  }
}
```

Figure 6.2: An analysis.

The walker will exhaustively walk all the nodes from `A` to `G`. But, walking some of the subtrees, such as the one rooted at `B`, is unnecessary. The above analysis is only interested in nodes of type `C`. Obviously, the subtree rooted at `B` need not be walked as it contains no node of type `C`. This second cause can be addressed by not walking all subtrees of each node. As we will see, we can single out some subtrees that need not be walked during preprocessing. The details will be provided in the next section.

6.2 Preprocessing the Java source code of the walker

First of all, to simplify the presentation we restrict ourselves to the following situations.

- Firstly, the class that performs the analysis, say `Analysis`, must directly extend the `Walker` class.
- Secondly, no other classes should be manipulating the `Analysis` object.
- Finally, the methods `walk` and `visit` should not be used anywhere in the code written by the user. The user may freely use the method `walkFields`.

During the preprocessing step, we first examine the hierarchy of Java classes and interfaces representing the ASTs and the `Analysis` class representing the analysis. The main purpose is to identify those classes which may have to be visited and those classes and fields that may have to be walked. Once these have been found, we generate the appropriate Java code based on this information. The generated Java code will contain code to invoke the appropriate `visit` method for a given type of node, and code to walk only certain subtrees of a given type of node.

Below we present simplified algorithms for finding these classes and fields, and for generating the Java code. We will walk the reader through these algorithms and explain how the code will be generated.

6.2.1 Identifying the classes which may have to be visited

Before determining which classes in the AST hierarchy may have to be walked, we first determine the set of classes that may have to be visited. Obviously, all the parameter types of the `visit` methods in the `Analysis` are visitable. Consider, for example, the following Java classes representing the ASTs

```
public class C1  { ... }
public class C2 extends C1 { ... }
```

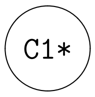
and the following analysis

```
class Analysis extends Walker
{
  int counter;
  Analysis()
  {
    counter = 0;
  }
  public void visit(C1 c1)
  {
    counter++;
  }
  public int getCounter()
  {
    return counter;
  }
}
```

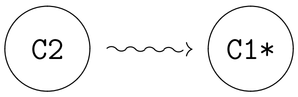
Figure 6.3: The `Analysis` class.

The `Analysis` class keeps track of the number of nodes of type `C1` in an AST. Clearly, the class `C1` is considered to be visitable with respect to this analysis.

To ease the presentation, instead of expressing the examples in Java code we use a graphical notation to illustrate the classes and their relationships. The vertices of the graph represent the classes and interfaces. To indicate that a class that has a `visit` method, we mark the corresponding vertex with an asterisk ($*$). For example,



indicates that the class `C1` has a `visit` method. To denote the is-a relationship, we use an $\rightsquigarrow$ edge. For example,



denotes that class `C2` extends class `C1`.

In the above example, the class `C2` is a subclass of `C1` and `Analysis` does not have a `visit` method defined for it. As a result, an object of type `C2` gives rise to the invocation of the method `visit(C1)`. Hence, the class `C2` is also considered to be visitable and the parameter type of the corresponding `visit` method is `C1`. More generally, every subclass of a visitable class is visitable.

In summary, a class `C` is *visitable* if

- the `Analysis` class contains a `visit` method whose parameter type is `C` or
- the superclass of `C` is visitable.

For each visitable class `C`, *visit*(`C`) is the parameter type of the corresponding `visit` method. If the `Analysis` class contains a `visit` method whose parameter type is `C`, then *visit*(`C`) = `C`. Otherwise, *visit*(`C`) = *visit*(`S`), where `S` is `C`'s superclass.

Those classes for which the `Analysis` class contains a corresponding `visit` method can be easily extracted from the `Analysis` class using Java's reflection. Assume that these classes are collected in the set `visitSet`. The map *visit* can be computed as follows.

```
visitMap = empty map
for each class C in visitSet
  buildMap(C, C, visitMap)

void buildMap(A, C, visitMap)
```

```
add (A, C) to visitMap
for each subclass S of C
  if S not in domain of visitMap and S not in visitSet
    buildMap(A, S, visitMap)
```

Figure 6.4: The algorithms for computing the map `visit`.

We exploit the package `org.eclipse.jdt.core` to find the subclasses of a given class. This package is part of the Eclipse project Java development tools[12] (JDT) [18]. The interface `ITypeHierarchy` contains the method `getSubclasses` which returns the subclasses of a given class within the Eclipse workspace.

6.2.2 Identifying the classes which may have to be walked

Next, we want to approximate the set of classes and fields which may have to be walked. To denote the has-a relationship, we use an $\xrightarrow{\texttt{f}}$ edge, where `f` is the name of the non-static field. For example,

$$\begin{tikzcd}\boxed{\texttt{C2}} \arrow[r, "\texttt{f1}"] & \boxed{\texttt{C1*}}\end{tikzcd}$$

denotes that the class `C2` has a non-static field, named `f1`, of declared type `C1`. This field can either be declared in `C2` or inherited by `C2` from one of its superclasses.

[12]JDT is an open source project [31].

When an object of type `C2` is encountered during a walk its only subtree (to which the field `f1` is referring) will be walked, because `C2` itself is not visitable. Since the root of the subtree is of type `C1`, the method `visit(C1)` will be invoked. The class `C2` is thus considered to be a walkable class because it has a field referring an object for which a `visit` method is defined. Also, the field `f1` is considered to be a walkable field. More generally, a class is walkable if it has a walkable field and a field is walkable if it is non-static and its declared type is visitable.

Assume now that class `C3` has a field named `f2` of type `C2`.

$$\begin{tikzcd}\boxed{\texttt{C3}} \arrow[r, "\texttt{f2}"] & \boxed{\texttt{C2}} \arrow[r, "\texttt{f1}"] & \boxed{\texttt{C1*}}\end{tikzcd}$$

Figure 6.5: Case 1.

The field `f2` is walkable because walking the object (to which `f2` refers) will eventually give rise to the invocation of the method `visit(C1)`. More generally, a field is walkable if it is non-static and its declared type is walkable.

Let us look at some more subtle cases. Consider the following classes and fields.

$$\begin{tikzcd}
 & \texttt{C5} \arrow[dl, "\texttt{f3}"'] \arrow[dr, "\texttt{f2}"] & \\
\texttt{C4} & & \texttt{C3} \\
\texttt{C2} \arrow[u, squiggly] \arrow[rr, "\texttt{f1}"] & & \texttt{C1*} \arrow[u, squiggly]
\end{tikzcd}$$

Figure 6.6: Case 2.

In this case, it is not evident whether the fields `f2` and `f3` are walkable. Although the declared types of both fields are not walkable classes, at runtime they can possibly be assigned to objects of visitable/walkable types, namely, `C1` and `C2`. Due to this possibility, these fields are deemed to be walkable. More generally, a field is walkable if it is non-static and its declared type is a superclass of a class that is visitable or walkable.

To denote that a class implements an interface we use an $\Longrightarrow$ edge. For example,

$$\begin{tikzcd}
\texttt{C1} \arrow[r, Rightarrow] & \texttt{I1}
\end{tikzcd}$$

denotes that the class `C1` implements the interface `I1`.

Next, we focus on fields whose declared type is an interface. Consider, for example, the following classes, interfaces and fields.

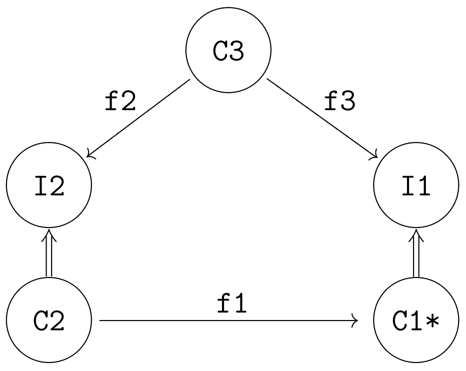


Figure 6.7: Case 3.

Obviously, since at runtime the fields `f2` and `f3` can refer to objects of visitable/walkable type, they are also considered to be walkable. More generally, a field is walkable if it is non-static and its declared type is an interface implemented by a class that is visitable or walkable.

Let us slightly modify the above example.

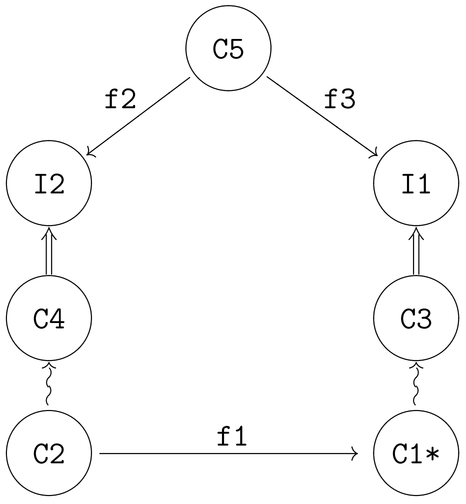


Figure 6.8: Case 4.

The fields `f2` and `f3` are still considered to be walkable because they can possibly refer, at runtime, to objects of visitable/walkable types. More generally, a field is walkable if it is non-static and its declared type is an interface implemented by a superclass of a class that is visitable or walkable.

Similarly, in

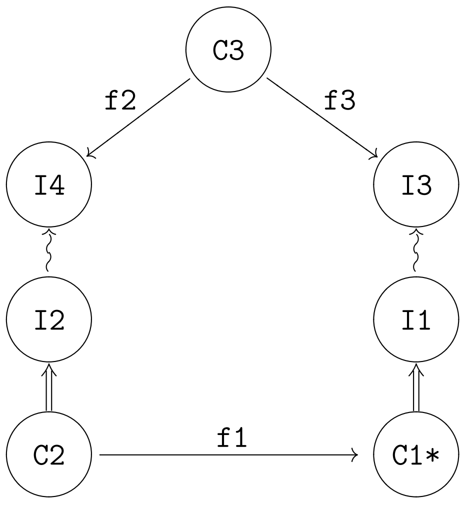


Figure 6.9: Case 5.

the fields `f2` and `f3` are still considered to be walkable because they can possibly refer to objects of visitable/walkable types. More generally, a field is walkable if it is non-static and its declared type is a superinterface of an interface implemented by a class that is visitable or walkable.

Finally, in

$$\begin{tikzcd}
 & \texttt{C5} \arrow[dl, "\texttt{f2}"'] \arrow[dr, "\texttt{f3}"] & \\
\texttt{I4} & & \texttt{I3} \\
\texttt{I2} \arrow[u, rightsquigarrow] & & \texttt{I1} \arrow[u, rightsquigarrow] \\
\texttt{C4} \arrow[u, Rightarrow] & & \texttt{C3} \arrow[u, Rightarrow] \\
\texttt{C2} \arrow[u, rightsquigarrow] \arrow[rr, "\texttt{f1}"] & & \texttt{C1*} \arrow[u, rightsquigarrow]
\end{tikzcd}$$

Figure 6.10: Case 6.

the fields `f2` and `f3` are also considered to be walkable for the similar reason. More generally, a field is walkable if it is non-static and its declared type is a superinterface of an interface implemented by a superclass of a class that is visitable or walkable.

In summary, a class is *walkable* if it has a walkable field (the field is either declared in the class or inherited by the class). A field is *walkable* if it is non-static and its declared type is waitable. A class is *waitable* if

(i) it is visitable,

(ii) it is walkable, or

(iii) it is a superclass of a waitable class.

An interface is *waitable* if

(iv) any of its implementations is waitable, or

(v) it is a superinterface of a waitable interface,

The set of waitable classes and walkable fields can be computed as follows.

```
temp = domain of visitMap
waitable = empty set
walkable = empty set
while temp is nonempty
  for each class C in temp
    remove C from temp and add C to waitable
    for each superclass S of C
      if S not in temp and S not in waitable
        add S to temp
    for each interface I that is implemented by C
      if I not in temp and I not in waitable
         add I to temp
    for each non-static field f of class D whose declared type is
        C
      add D.f to walkable
      if D not in temp and D not in waitable
        add D to temp
  for each interface I in temp
    remove I from temp and add I to waitable
    for each superinterface S of I
```

```
20        if S not in temp and S not in waitable
21          add S to temp
22      for each non-static field f of class D whose declared type is
            I
23        add D.f to walkable
24        if D not in temp and D not in waitable
25          add D to temp
```

Figure 6.11: The algorithm for computing the set of waitable classes.

Note that in line 13 and 22 we consider both the fields declared in class `D` and the fields inherited by it.

Each waitable class or interface is added to the set `temp` and subsequently moved from `temp` to `waitable` (see line 6 and 18). The clauses (i), (ii) and (iii) of the definition of a waitable class are reflected by the lines 1, 15–16 & 24–25, and 7–9, respectively. The clauses (iv) and (v) of the definition of waitable interface are reflected by the lines 10–12, and 19–21, respectively. Lines 14 and 23 add the walkable fields to the set `walkable`.

We again exploit the package `org.eclipse.jdt.core` of JDT to implement line 13 and 22. The interface `ITypeHierarchy` contains the method `getAllClasses` which allow us to collect all classes within the Eclipse workspace. Using Java's reflection we can subsequently inspect the fields of these classes.

6.2.3 Generating Java code for the walker

Once the visitable classes and the walkable fields have been found, the process of generating Java code can begin. Below we describe algorithms of the code generator. Initially, the code generator copies the import statements, the fields and the non-`visit` methods from the original class to the new one. For example, from the `Analysis` class the following class is initially generated.

```
class GenAnalysis
{
  int counter;

  public int getCounter()
  {
    return counter;
  }
}
```

Figure 6.12: A partially generated class `GenAnalysis` for the `Analysis` class.

The constructors are also copied. However, they have to be renamed to match the new class name. For the example, this amounts to

```
GenAnalysis()
{
  counter = 0;
}
```

For each `visit` method

```
public void visit(C c) { ... }
```

of the original class, we add

```
public void visitC(C c) { ... }
```

to the generated class. For example, to the class `GenAnalysis` we add

```
public void visitC1(C1 c1)
{
  counter++;
}
```

Note that, as a result, the `visit` methods are not overloaded any more.

We have left to generate

- the `walk` method,
- the `walkFields` method, and
- a `walkFieldsC` method for each walkable class `C`.

The generated `walk` method, replacing the original one, takes an argument of type `Object`. Based on the dynamic type of this object, it then executes either a corresponding `visitC` method or an appropriate `walkFieldsC` method. We have to generate code that mimicks the behaviour of the following snippet of pseudocode.

```
void walk(Object object)
  C = dynamic type of object;
  if C is visitable
    D = visitMap(C);
    visitD(object);
  else if C is walkable
    walkFieldsC(object);
```

How this is accomplished will be discussed below.

Also the generated `walkFields` method takes an argument of type `Object` and mimicks the behaviour of the following snippet of pseudocode.

```
void walkFields(Object object)
  C = dynamic type of object;
  if C is walkable
    walkFieldsC(object);
```

The implementation details will be provided below.

For each walkable class `C`, we generate a method `walkFieldsC`. This method takes an argument of type `Object` and mimicks the behaviour of the following snippet of pseudocode.

```
void walkFieldsC(Object object)
  for each walkable field f of C
    walk(object.f);
```

Note that the method `walkFieldsC` is only called in `walk` and `walkFields` for `object`s whose dynamic type is `C`.

Using the `Analysis` example, we will discuss the remaining implementation details. Assume that we also have the following classes and interfaces.

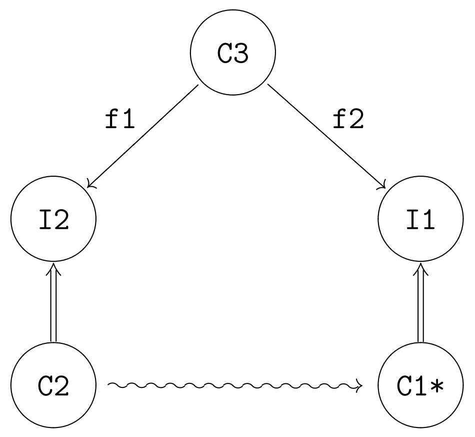


Figure 6.13: A running example.

In `walk` and `walkFields` we need to determine the dynamic type of an `Object` named `object`. This can be done as follows: `object.getClass()`. To associate to each dynamic type the appropriate call of `visitC` and `walkFieldsC`, we introduce a map that assigns to each class a unique natural number. This number acts as a class identifier as we will show below. To the class `GenAnalysis` we add the declaration

```
Map classMap;
```

and we add to the constructor

```
classMap = new HashMap();
classMap.put(C1.class, new Integer(0));
classMap.put(C2.class, new Integer(1));
classMap.put(C3.class, new Integer(2));
```

For this example, the generated `walk` method looks like

```
walk(Object object)
{
  Integer classID = (Integer) classMap.get(object.getClass());
  if (classID != null)
  {
    switch (classID.intValue())
    {
      case 0:
        visitC1((C1) object);
        break;
      case 1:
        visitC1((C1) object);
        break;
      case 2:
        walkFieldsC3((C3) object);
        break;
      default:
        break;
    }
  }
}
```

Figure 6.14: The generated `walk` method.

Similarly, the following `walkFields` method will be generated.

```
walkFields(Object object)
{
  Integer classID = (Integer) classMap.get(object.getClass());
  if (classID != null)
```

```
    {
      switch (classID.intValue())
      {
        case 2:
          walkFieldsC3((C3) object);
          break;
        default:
          break;
      }
    }
  }
```

Figure 6.15: The generated `walkFields` method.

Finally, we show the method `walkFieldsC3`. Since the class `C3` has walkable fields `f1` and `f2`, the method `walkFieldsC3` walks these fields.

```
walkFieldsC3(C3 c3)
{
  Field field1 = C3.class.getField("f1");
  walk(field1.get(c3));
  Field field2 = C3.class.getField("f2");
  walk(field2.get(c3));
}
```

Figure 6.16: The generated `walkFieldsC3` method for the class `C3`.

6.2.4 Walking arrays

Before giving details on how an array of objects will be handled, we first revisit the definition of visitable class and walkable field. To ease the presentation, we introduce the notation `C[]`^{n} for an array of dimension n with base type `C`.

Recall that a class is a subclass of the class `C[]`^{n} if and only if it is of the form `D[]`^{n} and `D` is a subclass of `C`. Consider the method `visit(C1[]`^{n} `c1)` and assume that the class `C2` extends the class `C1`. Since `C2` is a subclass of `C1`, a walk on an array object of type `C2[]`^{n} should lead to the invocation of this `visit` method. Hence, the definition of visitable class stays the same. Therefore, the algorithm to compute the map *visit* can be adjusted as follows. In the algorithm, we write `C`^{0} instead of `C`.

```
visitMap = empty map
for each class $C^n$ in visitSet
  buildMap($C^n$, $C^n$, visitMap)

void buildMap($A^n$, $C^n$, visitMap)
  add ($A^n$, $C^n$) to visitMap
  for each subclass S of C
    if $S^n$ not in domain of visitMap and $S^n$ not in visitSet
      buildMap($A^n$, $S^n$, visitMap)
```

Figure 6.17: The modified algorithms for computing the map `visit`.

Next, we tackle the problem of identifying those walkable fields that are declared as an array type. Again, we will first look at some examples. Consider the following Java class

```
public class C1 { ... }
```

and assume that a `visit` method has been introduced for `C1`. Then walking the following field

```
private C1[] f1;
```

will result in each object (of type `C1`) in the array being walked. Furthermore, walking this field

`private C1[]`^{n} `f1;`

will also result in each object (of type `C1`) in the n-dimensional array to be walked. More generally, a field declared as `C[]`^{n} is walkable if it is non-static and `C` is visitable.

If the class `C1` extends a class `C2`,

```
public class C1 extends C2 { ... }
```

the following field

`private C2[]`^{n} `f1;`

is still walkable since at runtime the field `f1` can be assigned to an object of type `C1[]`^{n}. Therefore, walking each object (of type `C1`) in the array will give rise to the invocation of the method `visit(C1 c1)`. More generally, a field declared as `C[]`^{n} is walkable if it is non-static and `C` is a superclass of a visitable class.

Given a `visit` method for `C1` and the following classes.

```
public class C1 { ... }
public class C2
{
  private C1 f1;
}
```

Clearly, `C2` is a walkable class because it contains the walkable field `f1`. Walking the following field

```
private C2[]^n f2;
```

will lead to the field `f1` of each object (of type `C2`) in the array to be walked. Subsequently, it gives rise to the invocation of the method `visit(C1 c1)`. Therefore, the field `f2` is a walkable field. More generally, a field declared as `C[]`^{n} is walkable if it is non-static and the class `C` is walkable.

If the class `C2` extends a new class `C3`,

```
public class C2 extends C3
{
  private C1 f1;
}
```

the following field

```
private C3[]^n f2;
```

may have to be walked because at runtime the field `f2` can be assigned to an object of type `C2[]`^{n}. More generally, a field declared `C[]`^{n} is walkable if it is non-static and the class `C` is a superclass of a walkable class.

Note that the same reasoning can be applied when considering the fields declared as interfaces. For example, assuming the following class and interfaces,

```
public interface I1 { ... }
public interface I2 extends I1 { ... }
public class C1 implements I2  { ... }
```

and suppose that a `visit` method has been introduced for the class `C1`. The following fields

```
private I1[]ⁿ f1;
private I2[]ⁿ f2;
```

may have to be walked because at runtime, both of them can possibly refer to objects of type `C1[]`^{n}. More generally, a field declared as `I[]`^{n}, where `I` is an interface, is walkable if `I` is an interface or a superinterface of an interface implemented by a visitable class.

In class `C2`,

```
public interface I3 { ... }
public interface I4 extends I3 { ... }
public class C2 implements I4
{
  private C1 f1;
}
```

the field `f1` is obviously walkable because `C1` is visitable. Then the following fields

```
private I3[]ⁿ f1;
private I4[]ⁿ f2;
```

may also have to be walked because at runtime they can be assigned to objects of walkable type `C2[]`^{n}. More generally, a field declared as `I[]`^{n} is walkable if `I` is an interface or a superinterface of an interface implemented by a walkable class.

Consider a single `visit` method whose parameter is of an array type, for example `visit(C1[] c)`. Clearly, the following field

```
private C1 f1;
```

is not walkable. However, the following fields

```
private C1[] f1;
private C1[][] f2;
...
private C[]ⁿ fn;
```

do constitute walkable fields. This can be generalized as follows. Given a method `visit(C1[]`^{m} `c1)` and the following fields.

```
private C1[] f1;
private C1[][] f2;
...
private C1[]ᵐ fm;
...
```

```
private C1[]$^n$ fn;
```

Only the fields from `fm` to `fn` are walkable, the other fields are not. More generally, a field declared as `C[]`^{n} is walkable if `C[]`^{m} is visitable for some $m \leq n$.

Consider the method `visit(C1[]`^{m} `c1)`, the class `C1` extending the class `C2` as illustrated,

```
public class C1 extends C2 { ... }
```

and the following fields.

```
private C2[] f1;
private C2[][] f2;
...
private C2[]$^m$ fm;
...
private C2[]$^n$ fn;
```

Since `C2` is a superclass of the class `C1`, at runtime the fields `fm`, ... , `fn` can possibly be assigned to objects of types `C1[]`^{m}, ... , `C1[]`^{n}, respectively. Hence, the fields `fm`, ... , `fn` may have to be walked. More generally, a field declared as `C[]`^{n} is walkable if `S[]`^{m} is visitable for some subclass `S` of `C` and some $m \leq n$.

Assume that the method `visit(C1[]`^{m} `c1)` has been introduced, and the following interfaces and class.

```
public interface I1 { ... }
public interface I2 extends I1 { ... }
public class C1 implements I2  { ... }
```

The following sets of fields

```
private I2[]^m fm;
...
private I2[]^n fn;
```

and

```
private I1[]^m fm;
...
private I1[]^n fn;
```

may have to be walked because at runtime these fields can possibly be assigned to objects of types `C1[]`^{m}, ... , `C1[]`^{n}, respectively. More generally, a field declared as `I[]`^{n} is walkable if `I` is an interface or a superinterface of an interface implemented by some class `S` and `S[]`^{m} is visitable for some $m \leq n$.

Recall that a class is *walkable* if it has a walkable field (the field is either declared in the class or inherited by the class). A field is *walkable* if it is non-static and its declared type is waitable. Again, we use the convention that `C[]`^{0} represents `C` and `I[]`^{0} represents `I`.

A class `C[]`^{n} is *waitable* if

(i) it is visitable,

(ii) `C` is walkable,

(iii) `C[]`^{m} is waitable for some $m \leq n$, or

(vi) it is a superclass of a waitable class.

An interface `I[]`^{n} is *waitable* if

(v) `C[]`^{n} is waitable for some implementation `C` of `I`, or

(vi) `S[]`^{n} is waitable for some subinterface `S` of `I`.

The main problem with adjusting the algorithm for computing the waitable classes and interfaces and the walkable fields is clause (iii). `C[]`^{n} being waitable implies that `C[]`^{m} is waitable for each $m \geq n$. Hence, the set of waitable classes may be infinite. However, the set of walkable fields is still finite, provided that we restrict our attention to the classes in the Eclipse workspace. To keep the set `waitable` finite, we use `C[]`^{n} to represent all `C[]`^{m} with $m \geq n$. That is, if `C[]`^{n} is in (`temp` or) `waitable` then we know that `C[]`^{m} is waitable for each $m \geq n$. Moreover, we use `I[]`^{n} to represent all `I[]`^{m} with $m \geq n$.

Below, we provide a revised version of the algorithm for computing the waitable classes and interfaces, and the walkable fields.

```
temp = domain of visitMap
waitable = empty set
walkable = empty set
while temp is nonempty
  for each class C[]$^n$ in temp
    remove C[]$^n$ from temp and add C[]$^n$ to waitable
    for each superclass S of C
```

```
8         if for all $m \leq n$, S[]$^m$ not in temp and S[]$^m$ not in
              waitable
9           add S[]$^n$ to temp
10      for each interface I that is implemented by C
11        if for all $m \leq n$, I[]$^m$ not in temp and I[]$^m$ not in
              waitable
12          add I[]$^n$ to temp
13      for each non-static field f of class D declared as C[]$^m$ for
            some $m \geq n$
14        add D.f to walkable
15        if D not in temp and D not in waitable
16          add D to temp
17    for each interface I[]$^n$ in temp
18      remove I[]$^n$ from temp and add I[]$^n$ to waitable
19      for each superinterface S of I
20        if for all $m \leq n$, S[]$^m$ not in temp and S[]$^m$ not in
              waitable
21          add S[]$^n$ to temp
22      for each non-static field f of class D declared as I[]$^m$ for
            some $m \geq n$
23        add D.f to walkable
24        if D not in temp and D not in waitable
25          add D to temp
```

Figure 6.18: The modified algorithm for computing the set of waitable classes.

In the above algorithm, clauses (i), (ii), (iv), (v) and (vi) are reflected by lines 1, 15–16 & 24–25, 7–9, 10–12, and 19–21, respectively.

Since a waitable class `C[]`^{n} is used to represent all waitable classes `C[]`^{m} (for each $m \geq n$), fields that are declared as `C[]`^{m} are also walkable. Consequently, they need to be added to the set `temp` as done in lines 13–16. Similarly, fields that are declared as `I[]`^{m}, where `I[]`^{n} is waitable for some $n \leq m$, are waitable as well (see lines 22–25).

Now, we give some details on how to generate code, which handles an array of objects, via the following simple example.

```
public class C1 { ... }
public class C2 extends C1 { ... }
public class C3 extends C1 { ... }
public class C4
{
  private C1[] f1;
}
```

If a method `visit(C2[] c2)` has been defined, the classes `C2[]`, `C1`, and `C4` are visitable, waitable and walkable, respectively.

Recall that when generating code for the new walker, the code generator initially copies the `visit` methods from the original walker to the new one. For each `visit` method

```
public void visit(C[]^n c) { ... }
```

of the original walker, we add

```
public void visitArray_n_C(C[]^n c) { ... }
```

to the generated class. For example, we add `visitArray_1_C2(C2[] c2)` for the method `visit(C2[] c2)`. Notice that we have used a slightly different naming convention for `visit` methods for arrays.

Consider the following lines of code to initialize the array `f1` of an object of type `C4`.

```
f1 = new C1[size];
for (int i = 0; i < size; i++)
{
  if (i % 2 == 0)
    f1[i] = new C2();
  else
    f1[i] = new C3();
}
```

In the example, the type of the array `f1` is the waitable class `C1[]`. Objects in the array are either of type `C2` or `C3`. Therefore, if this array is walked, not every object in the array needs to be walked. However, for simplicity we will walk all the objects using the method `walkArray`.

```
walkArray(Object object)
{
  for (int i = 0; i < Array.getLength(object); i++)
  {
    walk(Array.get(i, object));
  }
}
```

Figure 6.19: The `walkArray` method.

The code generator also needs to generate this method for each new walker. Note that the implementation of `walkArray` is the same as the one found in the original walker.

We modify the algorithm for generating the method `walk` to handle array objects. The generated method mimicks the behaviour of the following snippet of pseudocode.

```
void walk(Object object)
  C = dynamic type of object;
  if C is visitable
    D = visitMap(C);
    visitD(object);
  else
    if C is an array
      if C is waitable
        walkArray(object);
    else
      if C is walkable
        walkFieldsC(object);
```

Figure 6.20: The algorithm for the `walk` method.

In the above code snippet, if the type of an array object is waitable, then we will walk each object in the array. However, since it is not clear if the condition "`if C is waitable`" can be validated efficiently, we simplify matters by considering all array objects (not just the waitable ones). The quest for an efficient implementation of “`if C is waitable`” is left for future research.

Since the classes `C2[]` and `C4` are visitable and walkable, respectively, they are assigned a unique identifier as follows.

```
classMap.add(C2[].class, new Integer(0));
classMap.add(C4.class, new Integer(1));
```

The generated `walk` method for the example looks like

```
void walk(Object object)
{
  Integer classID = (Integer) classMap.get(object.getClass())
  if (classID != null)
  {
    case 0:
      visitArray_1_C2((C2[]) object);
      break;
    case 1:
      walkFieldsC4((C4) object);
      break;
    default:
      break;
  }
  else if (object.getClass().isArray())
  {
    walkArray(object);
  }
}
```

Figure 6.21: The generated `walk` method.

The generated method `walkFields` looks like

```
void walkFields(Object object)
{
  Integer classID = (Integer) classMap.get(object.getClass())
  if (classID != null)
  {
    case 1:
      walkFieldsC4((C4) object);
      break;
    default:
      break;
  }
}
```

Figure 6.22: The generated `walkFields` method.

Finally, the generated method to walk the fields of the walkable class `C4` looks like

```
void walkFieldsC4(C4 c4)
{
  Field field1 = C4.class.getField("f1");
  walk(field1.get(c4));
}
```

Figure 6.23: The generated `walkFieldsC4` method for the class `C4`.

6.2.5 Walking graphs

To walk a graph, we employ the same strategy as we used for the original walker. That is, we keep track of those objects that have already been walked. This will ensure the same object not being walked again.

7 A simple analysis of BPEL4WS

7.1 Introduction

The recent release of WSADIE (version 5.1) provides a very powerful editing tool for creating and manipulating a BPEL4WS program. This graphical editor for BPEL4WS allows one to easily construct a complex BPEL4WS program. The editor then serializes it as an XML-based document. A snapshot of the editor is presented below.

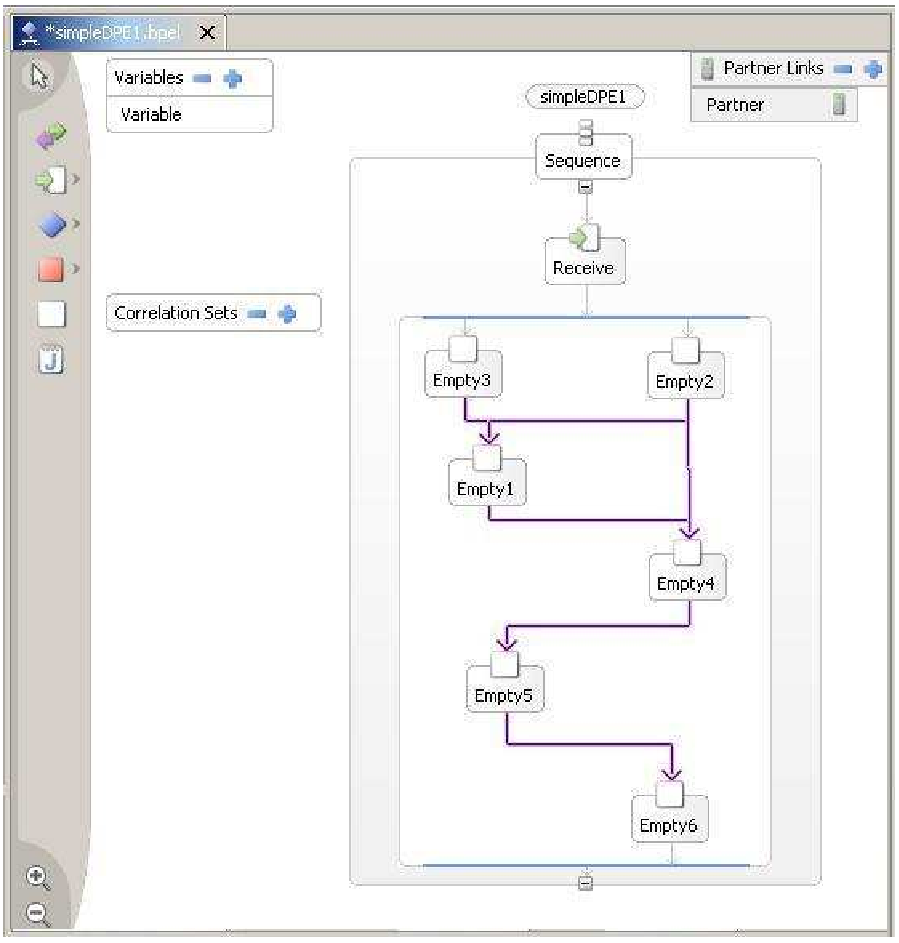


Figure 7.1: A snapshot of the BPEL4WS editor.

In the editor, each created activity is assigned a unique identifier. Often, given an identifier, one needs to find the corresponding activity. Such a task can be implemented using the approaches described in the previous chapters. We will show various implementations using these approaches. However, we will limit ourselves to the following:

- the dedicated methods approach,
- the EMF-switch, and
- the reflection based techniques.

Details of the implementations are outlined in the next sections.

7.2 Dedicated methods

To find the corresponding activity for a given identifier, we start from the root of the tree representing the BPEL4WS program and search the AST for this activity. Once the activity has been found, it is returned.

To accomplish such a task using the dedicated methods approach, one must add a method, named, for example, `findActivity`, to all the Java classes of the AST hierarchy. This method will return the activity which has the matching identifier. The hierarchy of Java classes representing BPEL4WS is complex. Furthermore, the implementation of the `findActivity` method differs from one class to another. We will only show the `findActivity` method for classes representing BPEL4WS activities because the `findActivity` method for other classes can be implemented in a similar way. Within the activities, the implementation of the `findActivity` method is also different for basic activities and structured activities. The following method

```
public Activity findActivity(Id id)
{
  Id i = BPELUtility.getId(this);
  return i.equals(id) ? this : null;
}
```

has been added to the class `ActivityImpl`. Technically, this method will be inherited by classes extending the `Activity` class. The method `findActivity` will return the current activity if its identifier matches the given identifier. Note that we have introduced a utility class `BPELUtility` and a static method `getId`. The method `getId` extracts and returns the identifier of "`this`" activity. In the editor, the identifier of an activity is implemented by the class `com.ibm.etools.ctc.wsdl.Id`.

A structured activity possibly contains nested activities. Hence, the method `findActivity` for the structured activities must also consider these nested activities as well. Consequently, the classes for structured activities must override the above method. Note that we only show the `findActivity` method for the flow activity.

```
public Activity findActivity(Id id)
{
  Activity activity = super.findActivity(id);
  for (int i = 0; activity == null &&
      i < this.getActivities().size(); i++)
  {
    activity = this.getActivities().get(i).findActivity(id);
  }
  return activity;
}
```

In the above example, we first call super.findActivity to check whether “this” activity is the one. If it is not, we start searching in the nested activities (the method getActivities returns all nested activities in the flow). As soon as it is found, it is returned.

7.3 EMF-switch mechanism

Let us first create a class, namely ActivityFinder, which implements the task. In this class, we define the two global variables i and a to store the given identifier and the found activity, respectively. We also declare a new instance variable s of type BPELSwitch. Recall that the class BPELSwitch implementing the EMF-switch is generated by EMF.

To find the activity which has an identifier that matches with i, we must override the appropriate case methods in the BPELSwitch. Note that we only show two case methods, namely for basic activities and the flow activity.

```
public class ActivityFinder
{
  Id i;
```

```
Activity a;
BPELSwitch s =
  new BPELSwitch()
  {
    Object caseActivity(Activity activity)
    {
      if (a == null && BPELUtility.getId(activity).equals(i))
      {
        a = activity;
      }
      return activity;
    }
    Object caseFlow(Flow flow)
    {
      if (a == null && BPELUtility.getId(flow).equals(i))
      {
        a = flow;
      }
      for (int j = 0; a == null && j < flow.getActivities().
          size(); j++)
      {
        doSwitch((EObject) flow.getActivities().get(j));
      }
      return flow;
    }
    ...
  }

public Activity findActivity(Id id, EObject bpel)
{
  i = id;
  s.doSwitch(bpel);
  return a;
}
```

```
}
```

Figure 7.2: The `ActivityFinder` class.

In the above example, we have added a method `findActivity`. This method takes an identifier `id` and an object `bpel` representing a BPEL4WS program as parameters. It then performs a `doSwitch` on the object `bpel`, causing the appropriate `case` methods to be invoked. Finally, the activity to which `a` is referring, is returned (if there is any).

7.4 The reflection based techniques

Recall that the implementation of the `ActivityFinder` is almost the same for all reflection based techniques. To use a specific walker, the `ActivityFinder` extends the appropriate walker class. For example, it can extend the class `Walker` which provides the implementation for the reflection based walker.

```
public class ActivityFinder extends Walker
{
  Id i;
  ActivityImpl a;
  ActivityFinder()
  {
    i = null;
    a = null;
  }
```

```
    public void visit(ActivityImpl activity)
    {
      if (a == null)
      {
        Id id = BPELUtility.getId(activity);
        if (id.equals(i))
          a = activity;
        else
          walkFields(activity);
      }
    }

    public Activity findActivity(Id id, Object bpel)
    {
      i = id;
      walk(bpel);
      return a;
    }
  }
```

Figure 7.3: The `ActivityFinder` class.

In the `ActivityFinder` class, the instance variables `i` and `a` are introduced to store the given identifier and the found activity. Since we are only interested in nodes of type `Activity`, we only introduce a `visit` for this class. In the `visit` method, if the identifier of `activity` does not match the given one, we will invoke the method `walkFields` to walk the fields of `activity`.

For this particular task, the amount of code written using the reflection based techniques is astonishingly less than that using the dedicated methods approach and the EMF-switch. However, as we will see later, the implementations using reflection based techniques are rather inefficient. Before comparing the performance of these implementations, we will show more examples of analyses of BPEL4WS in the next chapters.

8 BPEL-calculus translator

8.1 Overview of BPEL-calculus

The BPEL-calculus is a small language introduced by Koshkina and Van Breugel (see [37, 38]) to capture the control flow in BPEL4WS. A tool has been developed to verify properties like, for example, dead-lock freedom, of BPE-calculus processes (see also [37, 38]). This tool can also be exploited to verify properties of BPEL4WS programs, provided that BPEL4WS programs can be translated to BPE-calculus processes. Such a translator has already been implemented by Ramay [47] using the syntax separate from interpretations approach. This chapter will consider other alternatives to implementing the translator, such as the dedicated methods approach, the EMF-switch, and the reflection based techniques.

Details of the syntax of the BPEL-calculus, fully discussed in [37, 38], is beyond the scope of this thesis. We refer the reader to these sources for further details. Below we present the `Activity` grammar relevant and central to the translation.

```
Activity =  "nil"                           (Empty)
```

```
| "end"                      (Terminate)

| Name                       (Basic)

| Activity "||" Activity     (Flow)

| Activity ";"  Activity     (Sequence)

| Activity "++" Activity     (Switch)

| Activity "+"  Activity     (Pick)

| Activity "*"               (While)

| "out" Link TC Activity     (Outgoing link)

| JC "=>" Activity           (Join condition)
```

Figure 8.1: The `Activity` grammar.

In the BPEL-calculus, basic and structured activities are represented differently. For instance, an empty activity is represented as the string "`nil`", and a terminate activity represented as "`end`", whereas the activity names are used to represent other basic activities. The name of an activity is captured by the nonterminal `Name`.

For the structured activities, the BPEL-calculus uses "`||`", "`;`", "`++`", "`+`", and "`*`" to represent a flow, sequence, switch, pick and while activity, respectively.

The nonterminal `TC` refers to the transition condition associated with `Link`. It is either true, false, or undefined. This is captured by the grammar below.

```
TC = true | false | undefined
```

The nonterminal JC refers to the join condition of Activity. Chapter 3 has provided a detailed discussion of the JC grammar. Therefore, we refer the reader to this chapter for more details.

The nonterminal Link refers to the name of an outgoing link. Since the BPEL-calculus does not support scopes, link names in a BPEL-calculus process must be unique. Hence, for links to have an unique name, it may be necessary to rename some links.

In the next sections, we first briefly describe the implementation of such a link renamer before describing the implementation of the translator.

8.2 Renaming links

A simple solution to rename the links can be implemented by means of the reflection based techniques. Since one is only interested in modifying the names of the links, a visit method responsible for such a task is introduced.

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

The variable `counter` is an integer defined in the link renamer class. It keeps track of the number of links which have been encountered so far. In this example, because the objective of the walker is to change the model, the `visit` method must modify the `Link` objects.

8.3 Dedicated methods

Without loss of generality, we may assume that link names in a given BPEL4WS program are unique. We only focus on how to translate activities. To translate activities, we start from the root of the tree representing a BPEL4WS program and search the AST for activity nodes. Whenever an activity node is encountered, it is translated according to the `Activity` grammar.

To perform such a task using the dedicated methods approach, a method, named, for example, `translate`, is added to the Java classes of the AST hierarchy. This method returns a `String` representing the result of the translation. For convenience, let us create a utility class `TranslatorUtil` which consists of two static methods.

```
public static String getOutgoingLinks(Activity activity);
public static String getJoinCondition(Activity activity);
```

These methods return the BPEL-calculus representations of the outgoing links and the join condition, respectively, for an activity. For example, given an activity with two outgoing links `l1` and `l2`, if the values of the transition conditions associated with these links are `true` and `false`, respectively, then the method `getOutgoingLinks` will return the string "`out l1 true out l2 false`" for this activity. Moreover, if the activity has a join condition "`!getLinkStatus('l3') && getLinksStatus('l4')`", then the method `getJoinCondition` will return the string "`(not l3) and l4`" for this activity.

Now to translate a basic activity we add

```
public String translate()
{
  return TranslatorUtil.getOutgoingLinks(this) +
         TranslatorUtil.getJoinCondition(this) +
         " => " + this.getName();
}
```

to the class `ActivityImpl`. This method captures the translation for all basic activities different from the empty and terminate activity. For the special cases of the empty and terminate activity, we override this method in the corresponding classes. We add

```
public String translate()
{
  return TranslatorUtil.getOutgoingLinks(this) +
         TranslatorUtil.getJoinCondition(this) +
         " => nil ";
}
```

to the class `EmptyImpl`. Similarly, we add

```
public String translate()
{
  return TranslatorUtil.getOutgoingLinks(this) +
         TranslatorUtil.getJoinCondition(this) +
         " => end ";
}
```

to the class `TerminateImpl`.

A structured activity may contain nested activities. These nested activities also need to be translated. The structured activity must then combine the results of the translation returned by its nested activities. Therefore, those classes representing the structured activities must also override the `translate` method. For instance, to translate a flow activity, we add

```
public String translate()
{
  String delimiter = "";
  String str = "";
  for (int i = 0; i < getActivities().size(); i++, delimiter = "
     ||")
  {
    str += delimiter +
           ((Activity) getActivities().get(i)).translate();
  }
  return TranslatorUtil.getOutgoingLinks(this) +
         TranslatorUtil.getJoinCondition(this) +
            "(" +  str + ")";
    }
```

to the class `FlowImpl`. In this method, we first invoke the method `translate` for each of the nested activities. The results of the translation are then combined. The method `translate` for other structured activities is similar to the above one.

8.4 EMF-switch mechanism

Similar to the analysis presented in Section 7.3, we need to override the appropriate `case` methods in the class `BPELSwitch`. Furthermore, in this analysis, we also make use of the return value of the `case` methods to return the result of the translation. For instance, the `case` method for a basic activity looks as follows.

```
public Object caseActivity(Activity activity)
{
  return TranslatorUtil.getOutgoingLinks(activity) +
         TranslatorUtil.getJoinCondition(activity) +
         " => " + activity.getName();
}
```

Again, to handle the special cases of the empty and terminate activity, we override the corresponding `case` methods for these classes.

```
public Object caseEmpty(Empty empty)
{
  return TranslatorUtil.getOutgoingLinks(empty) +
          TranslatorUtil.getJoinCondition(empty) +
          " => nil ";
}
public Object caseTerminate(Terminate terminate)
{
  return TranslatorUtil.getOutgoingLinks(terminate) +
         TranslatorUtil.getJoinCondition(terminate) +
         " => end ";
}
```

Finally, we also need to override the `case` methods for structured activities. Below is an example of the `case` method for the flow activity.

```
public Object caseFlow(Flow flow)
{
  String str = "" ;
  String delimiter = "";
  for(int i = 0; i < flow.getActivities().size(); i++, delimiter
     = "||")
  {
    Object obj = doSwitch((EObject) flow.getActivities().get(i));
    str += delimiter + (String) obj;
  }
  return TranslatorUtil.getOutgoingLinks(flow) +
         TranslatorUtil.getJoinCondition(flow) +
         "(" +  str + ")");
}
```

The `case` methods for other structured activities are similar to that for the flow activity.

8.5 The reflection based techniques

To implement the translator using the reflection based techniques, appropriate `visit` methods performing the appropriate translation are defined. However, because a `visit` method does not support returning a value, the translation cannot be returned as done in the previous implementations. To address this problem, a stack, implemented by the Java class `java.util.Stack`, is used. The translation, instead of being returned, is pushed onto the stack and popped off the stack as needed. For example, in the following `visit` methods for basic activities,

```
public void visit(EmptyImpl empty)
{
  stack.push(TranslatorUtil.getOutgoingLinks(empty) +
             TranslatorUtil.getJoinCondition(empty) +
             " => nil ");
}
public void visit(TerminateImpl terminate)
{
  stack.push(TranslatorUtil.getOutgoingLinks(terminate) +
             TranslatorUtil.getJoinCondition(terminate) +
             " => end ");
}
public void visit(ActivityImpl activity)
{
  stack.push(TranslatorUtil.getOutgoingLinks(activity) +
             TranslatorUtil.getJoinCondition(activity) +
             " => " + activity.getName());
}
```

the results of the translation are pushed onto the stack.

Since a structured activity may contain nested activities, these nested activities must be translated. The results of the translation for these nested activities are pushed onto the stack. As a result, the structured activity may need to pop the stack. For example, in the `visit` method for a flow activity,

```
public void visit(FlowImpl flow)
{
  String str = "";
  String delimiter = "";
  final int SIZE = stack.size();
  walkFields(flow);
  while (stack.size() != SIZE)
  {
    str += delimiter + stack.pop().toString();
    delimiter = "||";
  }
  stack.push(TranslatorUtil.getOutgoingLinks(flow) +
             TranslatorUtil.getJoinCondition(flow) +
             "(" +  str + ")");
}
```

invoking the method `walkFields` gives rise to the nested activities of the flow activity being translated. The translations for the nested activities are then popped off the stack and combined. Finally, the result of the translation for the flow activity is pushed onto the stack. The constant variable `SIZE` is used to ensure that the stack is only popped as many times as needed.

Again, the amount of code written for this analysis using the reflection based techniques is a lot less than that using the other approaches including the one implemented by Ramay. For example, Ramay's implementation requires approximately 2000 lines of code, whereas the implementation by means of the reflection based techniques only requires approximately 200 lines of code.

9 A graph representation for BPEL4WS

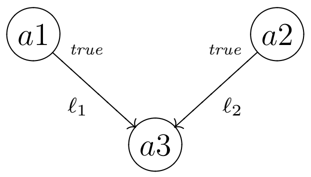
9.1 Introduction

A BPEL4WS program can also be viewed as a directed graph. The vertices of the graph are the activities and the edges of the graph are links. For instance, the following BPEL4WS snippet

```
<empty name="a1">
  <source link="l1" transitioncondition="true">
</empty>
<empty name="a2">
  <source link="l2" transitioncondition="true">
</empty>
<empty name="a3" >
  <target link="l1">
  <target link="l2">
</empty>
```

Figure 9.1: A simple example of BPEL4WS snippet.

can be represented as



Such a graph has been proven to be useful in some analyses of BPEL4WS (see Sections 9.5). In the mean time, we focus on the implementation that builds the graph. Below we present some approaches to implementing such a graph builder.

9.2 Dedicated methods

The class `Graph` implementing the directed graph provides the following methods.

```
public void addSource(Activity source, Link link);
public void addTarget(Activity target, Link link);
public Activity getSource(Link link);
public Activity getTarget(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. It adds the vertex `target` and the edge `link` to the graph, and sets the vertex `target` to be the target of the edge `link`. The methods `getSource` and `getTarget` return the source and target activity of the edge `link`.

Given an AST representing a BPEL4WS program, we build the graph as follows. We traverse and search the AST for the activity nodes. Whenever a source activity is encountered, the activity and its corresponding outgoing link are added to the graph using the method `addSource`. Whenever a target activity is encountered, the activity and its corresponding incoming link are added to the graph using the method `addTarget`.

To implement this by means of the dedicated methods approach, we add a dedicated method, named, for example `build`, to all Java classes of the AST hierarchy. This method takes a `Graph` object as a parameter. Initially, the `Graph` object has no vertices and no edges.

The class `Source`, which represents the declaration of the source of a link, we add

```
build(Graph graph)
{
  graph.addSource(this.getActivity(), this.getLink());
}
```

The method `getLink` returns the link that is being declared and the method `getActivity` returns activity of which the declaration is part. Similarly, we add

```
build(Graph graph)
{
  graph.addTarget(this.getActivity(), this.getLink());
}
```

to the class `Target` which represents the declaration of the target of a link. Finally, we add

```
build(Graph graph)
{
  EList sources = this.getSources();
  for (int i = 0; i < sources.size(); i++)
  {
    ((Source) sources.get(i)).build(graph);
  }
  EList targets = this.getTargets();
  for (int i = 0; i < targets.size(); i++)
  {
    ((Target) targets.get(i)).build(graph);
  }
}
```

to the class `ActivityImpl`. The methods `getSources` and `getTargets` respectively return a list of links of which "`this`" activity is declared to be the source and target.

Again, since structured activities may contain nested activities, one must override the `build` method in the corresponding classes. Below we show an example of the method `build` for a flow activity. This method must be added to the class `FlowImpl`.

```
build(Graph graph)
{
  super.build(graph);
  for (int i = 0; i < this.getActivities().size(); i++)
  {
    ((Activity) this.getActivities().get(i)).build(graph);
  }
}
```

Again, we did not show the method `build` for other classes.

9.3 EMF-switch mechanism

Again, the class `BPELSwitch` is exploited in this analysis. Appropriate `case` methods in the `BPELSwitch` must be overridden. For example, we override the `case` methods for the `Source`, `Target`, `Activity`, and `Flow` classes as follows.

```
public Object caseSource(Source source)
{
  graph.addSource(source.getActivity(), source.getLink());
  return source;
}
public Object caseTarget(Target target)
{
  graph.addTarget(target.getActivity(), target.getLink());
  return target;
}
public Object caseActivity(Activity activity)
{
  EList sources = activity.getSources();
  for (int i = 0; i < sources.size(); i++)
  {
    ((Source) sources.get(i)).builds(graph);
  }
  EList targets = activity.getTargets();
  for (int i = 0; i < targets.size(); i++)
  {
     ((Target) targets.get(i)).builds(graph);
  }
  return activity;
}
public Object caseFlow(Flow flow)
{
  for (int i=0; i < flow.getActivities().size(); i++)
  {
    doSwitch((EObject) flow.getActivities().get(i));
  }
  return null;
}
```

Recall from Section 4.5, a return value of `null` (in the method `caseFlow`) will cause the method `caseActivity` to be invoked for `flow`.

The variable `graph`, an instance of the class `Graph`, is defined in the graph builder class. It initially contains no vertices and no edges. As the `case` methods for the classes `Source` and `Target` are invoked, new vertices and edges are added to `graph`.

9.4 The reflection based techniques

By means of the reflection based techniques, we only need to define two `visit` methods for the classes `Source` and `Target` as shown below.

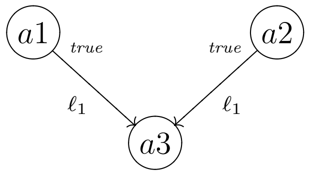
```
  public void visit(Source source)
  {
    graph.addSource(source.getActivity(), source.getLink());
```

```
}
public void visit(Target target)
{
  graph.addTarget(target.getActivity(), target.getLink());
}
```

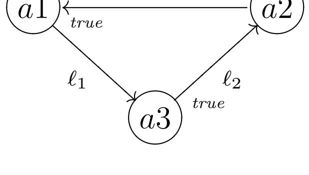
The code written using the reflection based techniques is remarkably less and considerably simpler than the code written using any of the other approaches.

9.5 Use of the graph

Representing BPEL4WS programs as directed graphs can be useful. According to the BPEL4WS definition, each activity should have a unique source activity and a unique target activity and a BPEL4WS program should not contain any (control) cycles. For example, in the BPEL4WS program represented by the graph



the link ℓ_1 has two source activities and, hence, this program is disallowed. The BPEL4WS program represented by the graph



is also disallowed, since the links ℓ_1, ℓ_2 and ℓ_3 form a cycle. Using the graph, we can easily check whether these two situations occur.

The graph has also been exploited in an analysis tool implemented for BPEL4WS to detect side effect caused by dead-path-elimination. Dead-path-elimination (DPE) [1, Section 2.2] (see also [40]) provides a mechanism to discard parts of a BPEL4WS program that will never be activated. As has been shown in [11], DPE may have side effects which are caused by negative occurrences of links in join conditions. 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>
```

The above snippet concurrently assigns 0 and 1 to the variable `v`. Hence, since the value of the variable `v` could be either 0 or 1, a transition condition defined as `bpws:getVariableData(‘v’)=‘0’` can be either true or false. A link ℓ to which this transition condition is associated, as a result, either gets the value true or false. It then follows that the join condition ℓ 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 ℓ, the Boolean $\mathit{dpe}(\ell)$ tells us whether ℓ may be set to false due to DPE. Given a join condition c, $\mathit{value}(c)$ captures the possible values of c. Given a transition condition b, $\mathit{value}(b)$ approximates the possible values of b. The function dpe is defined by

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

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

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

where b is the transition condition of the link ℓ. Note that if the link ℓ 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 ℓ is some times true and other times false. The *value* of a transition condition b is defined as 1 if $b = \mathit{true}$, -1 if $b = \mathit{false}$, and 0 otherwise. Clearly, there is room for improving the precision here. We leave that for future research.

At the implementation level, we introduce a utility class `DPEUtil` which consists of the following static methods.

```
public static boolean dpe(Link link, Graph graph);
public static int value(Activity activity, Graph graph);
```

These methods correspond to the functions *dpe* and *value*. We implement the *dpe* function as follows.

```
public static boolean dpe(Link link, Graph graph)
{
  Activity source = graph.getSource(link);
  return (source is part of a pick or a switch or
          value(source, graph) != 1);
}
```

Note that we have exploited the `Graph` object to efficiently determine the source activity of `link`. Also, the condition "`source is part of a pick or a switch`" can be evaluated by checking whether the parent node of `source` is a pick or a switch.

Computing the function *value*(c), for c a join condition of an activity, involves evaluating the join condition itself. Such an evaluator can be implemented by means of the reflection based techniques and a stack (the use of a stack in implementing an analysis has already been discussed in Section 8.5) as follows.

```
public class Evaluator extends Walker
{
  Graph graph;
  Stack stack;
  public Evaluator(Graph g)
  {
    super();
    stack = new Stack();
    graph = g;
  }
  private void push(int i)
```

```
{
  stack.push(new Integer(i));
}
private int pop()
{
  return ((Integer) s.pop()).intValue();
}
public void visit(TrueImpl t)
{
  push(1);
}
public void visit(FalseImpl f)
{
  push(-1);
}
public void visit(NotImpl not)
{
  walkFields(not);
  push(-pop());
}
public void visit(OrImpl or)
{
  walkFields(or);
  push(Math.max(pop(), pop());
}
public void visit(AndImpl and)
{
  walkFields(and);
  push(Math.min(pop(), pop());
}
public void visit(LinkImpl link)
{
  int v = value of transition condition of link
  if (dpe(link, graph))
```

```
      v = Math.min(v, 0);
    push(v);
  }
  public int getValue(Condition c)
  {
    walk(c);
    return pop();
  }
}
```

Figure 9.2: The `Evaluator` class.

Now, the implementation of the function *value* becomes straightforward.

```
public static boolean value(Activity activity, Graph graph)
{
  return ((new Evaluator(graph))
         .getValue(activity.getJoinCondition());
}
```

Computing the *value* of a join condition may cause that the values of other join conditions need to be computed as well. To refrain from multiple computations of the same join condition, one can store the computed value of the join condition in the `Graph` object (details are not shown).

Finally, we implement a mechanism that detects side effects caused by DPE. Basically, for each negative occurrence of a link in a join condition, we check whether the link may be set to false due to DPE and whether the join condition may evaluate to true. If so, in the graphical editor for BPEL4WS we will mark the corresponding target activity of the link (once again the `Graph` object becomes helpful) and the dead-path of which this link is part. This can also be easily implemented by means of the reflection based techniques. In the class below, while extracting negative links, we also detect any possible side effect.

```
public class NegativeLinkExtractor extends Walker
{
  boolean odd;
  Graph graph;

  public NegativeLinkExtractor(Graph g)
  {
    odd = false;
    graph = g;
  }

  public void visit(NotImpl not)
  {
    odd = !odd;
    walkFields(not);
    odd = !odd;
  }
  public void visit(LinkImpl link)
  {
    if (odd)
    {
      Activity target = graph.getTarget(link);
      if (DPEUtil.dpe(link, graph) &&
          DPEUtil.value(target, graph) == 1)
      {
        mark target and link on the editor;
      }
      }
    }
  }
```

Figure 9.3: The `NegativeLinkExtractor` class.

The variable `odd` is used to determine whether the path from the `Link` object to the root of the AST has an odd number of `Not` objects. If so, the link represented by the `Link` object has occurred negatively in the join condition.

If our tool finds no such occurrences then we know that the BPEL4WS program is free of the side effects. If, however, our tool finds some negative occurrences of links that may give rise to side effects, we should check whether such side effects can really occur and, if so, whether these side effects are intentional. The tool can easily be plugged into WSADIE. In the screenshot below, the links marked with a cross form the dead paths. The activities marked with an exclamation mark may be executed as a side effect of DPE.

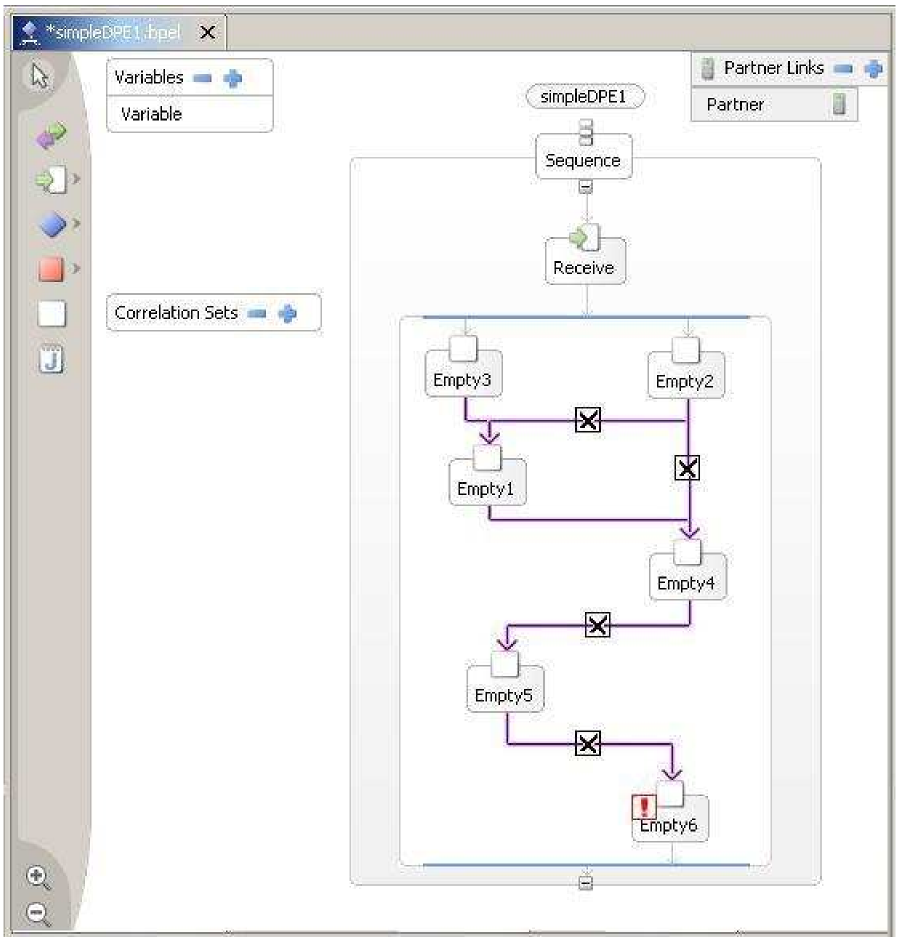


Figure 9.4: The tool that detects side-effects of DPE.

10 Performance evaluation

In this chapter, we will investigate, however only to some extent, the performance aspects of some approaches for some analyses of BPEL4WS. To be specific, we will consider

(i) the dedicated methods approach,

(ii) the EMF-switch mechanism,

(iii) the reflection based walker,

(iv) the walker with caching of fields and methods proposed by Bravenboer and Visser [9],

(v) the walker of Grothoff [28], and

(vi) the walker with preprocessing.

Recall that we have discussed the implementations, by means of these approaches, of the following analyses.

(a) An analysis that finds a corresponding activity for a given identifier (see Chapter 7).

(b) An analysis that translates a BPEL4WS program to a BPEL-calculus process (see Chapter 8).

(c) An analysis that builds a directed graph for a BPEL4WS program (see Chapter 9).

Based on these analyses we will evaluate the performance of each approach.

10.1 Benchmarks

The ASTs of BPEL4WS programs are generated randomly. The size of the ASTs ranges from 1000 nodes to 2000 nodes. We use these random ASTs as BPEL4WS benchmarks to measure the running times of different implementations for analyses (a), (b) and (c). There are in total 6 implementations for each analysis.

Since these implementations have been developed within the Eclipse environment, we will also use the Java compiler, which is distributed as part of Eclipse, to compile them. No particular optimization has been specified while compiling these implementations. The compiled code is then run on Sun's Java virtual machine version 1.4. We evaluate the performance of these implementations on an IBM machine, Intel® Pentium 4, CPU 2.40GHz, 1GB of RAM. Obviously, the running time of an implementation of an analysis would depend on, for example the speed of the machine or optimizations of the compiler. However, since we are only interested in the relative performance of one implementation to another, these factors may not play important roles in our evaluation.

Given these settings, the running time of an implementation for an analysis is then measured as follows. For each generated benchmark, we compute the duration of the execution, d, of the implementation as follows:

$$d = t_e - t_s,$$

where t_s is the system time recorded when the implementation starts and t_e is the system time recorded when the implementation terminates. Ten data points are collected for each implementation and benchmark. We compute their mean, μ, as shown,

$$\mu = \frac{1}{10} \sum_{1 \leq i \leq 10} d_i$$

The value of the mean is used to represent the (mean) running time of an implementation. In addition, we also compute the standard deviation, σ, which measures the spread of data from the mean, as follows.

$$\sigma = \frac{1}{10} \sqrt{\sum_{1 \leq i \leq n} (d_i - \mu)^2}$$

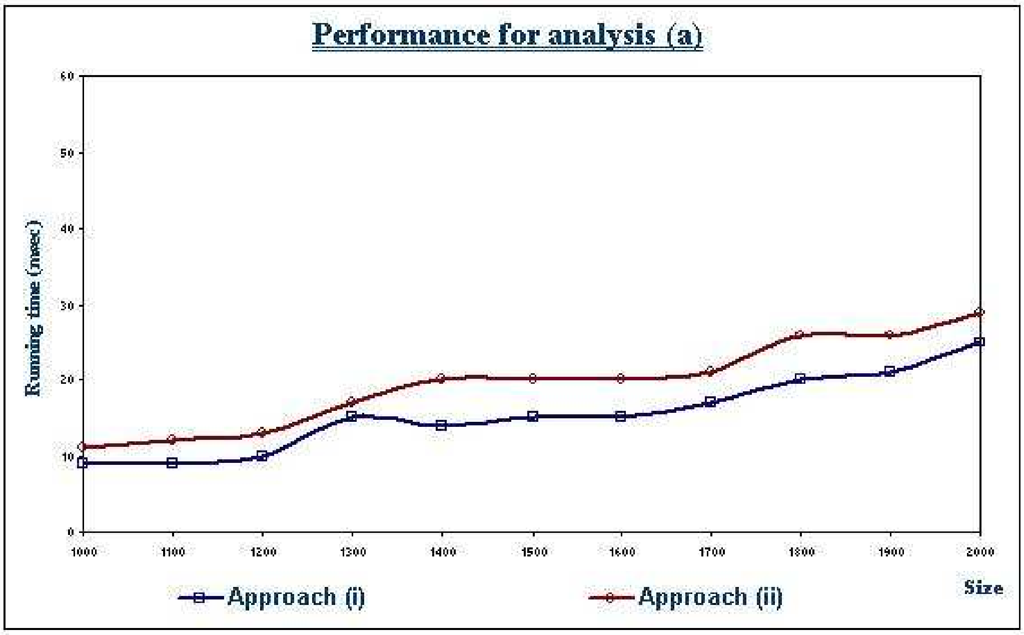
The value of the standard deviation plays an important role to determine whether the computed mean is reliable. The lower the standard deviation, the more reliable the mean. Finally, all the computed means are then analyzed and plotted. To compare the performance of two implementations for an analysis, we compute the ratio of their (mean) running times. The ratio measures how fast one implementation is compared to the other.

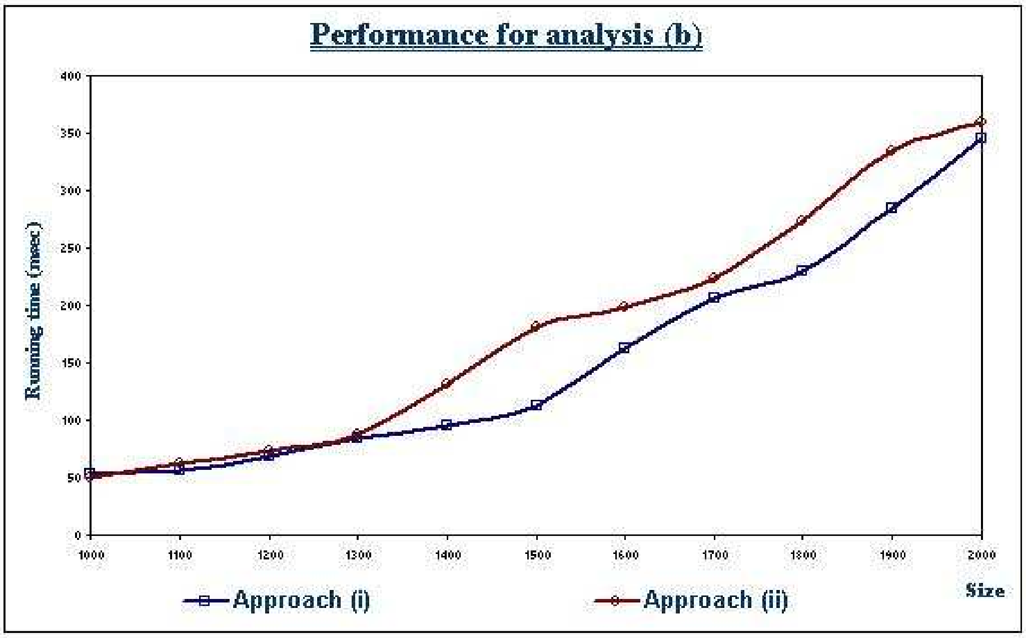
We expect that the approaches (i) and (ii), which do no rely on Java's reflection mechanism, will give rise to faster implementations for these analyses. In contrast, the approaches (iii), (iv) and (v), the reflection based walkers, will give rise to much slower implementations. The approaches (iv) and (v) are expected to improve the performance of the reflection based walker. We expect preprocessing the walker will further improve the performance of the reflection based walker to a greater extent.

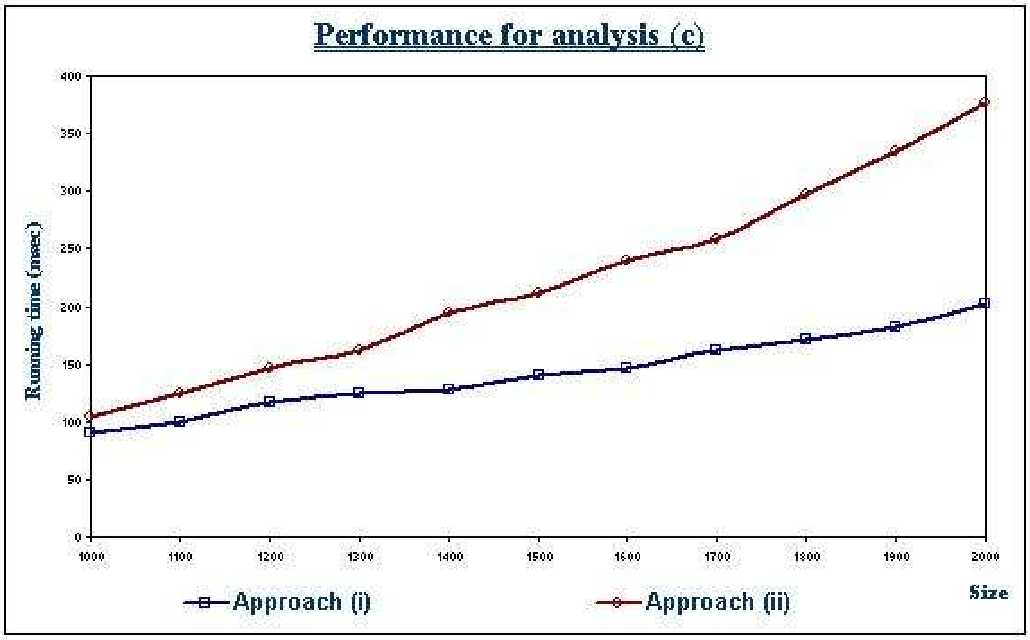
We refer the reader to the Appendix B for complete performance data sets. Below, we summarize and analyze the results of the performance of these approaches.

10.2 The non-reflection based approaches

Let us first look at the performance of those implementations by means of the approaches (i) and (ii), both of which do not depend on Java's reflection mechanism. Below we plotted the (mean) running times of the implementations by means of these two approaches for the analyses (a), (b) and (c).



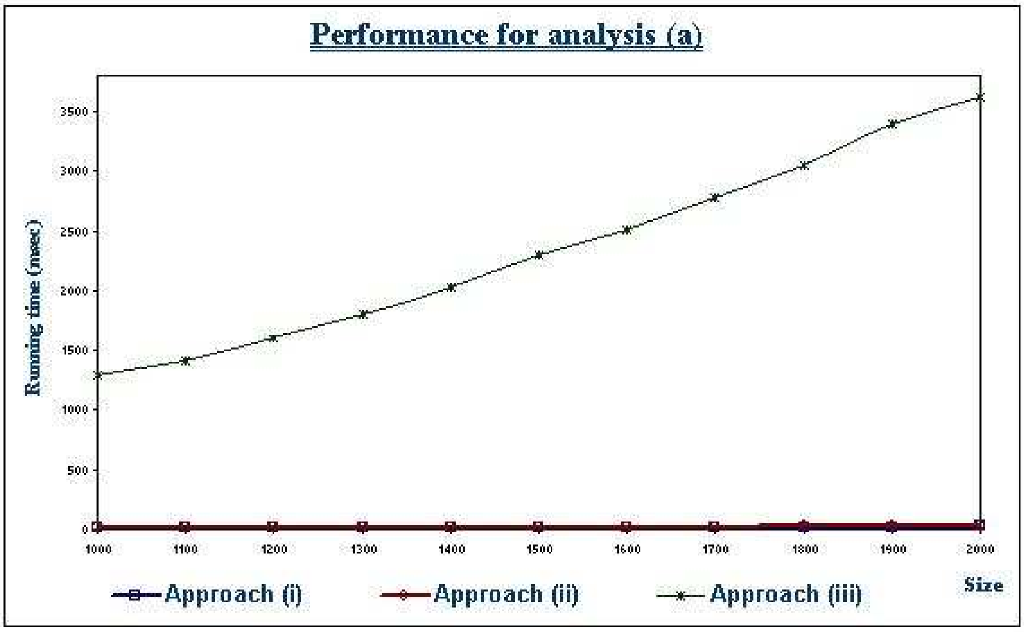


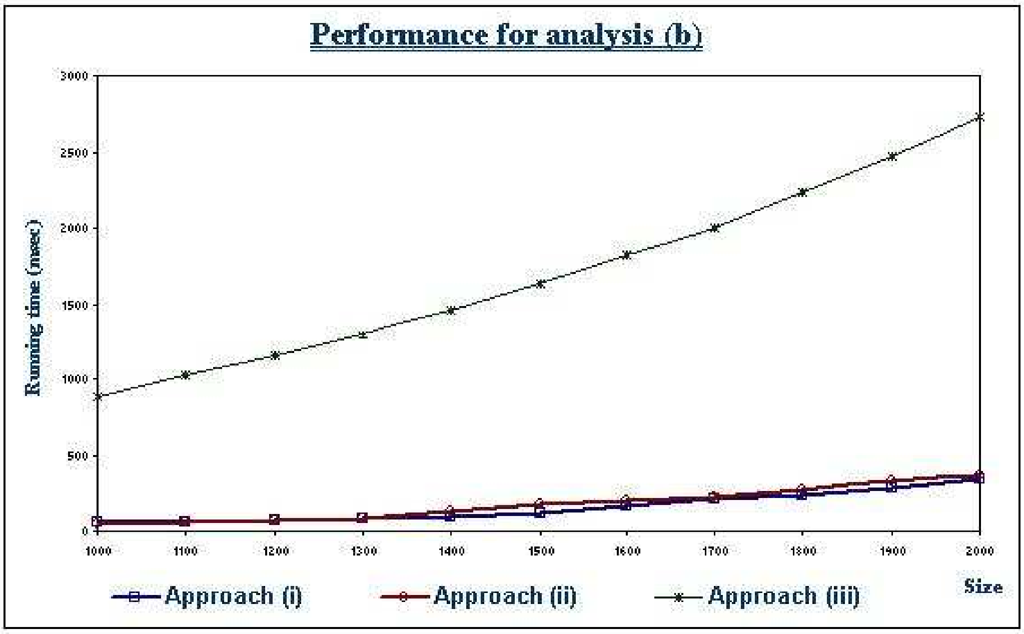


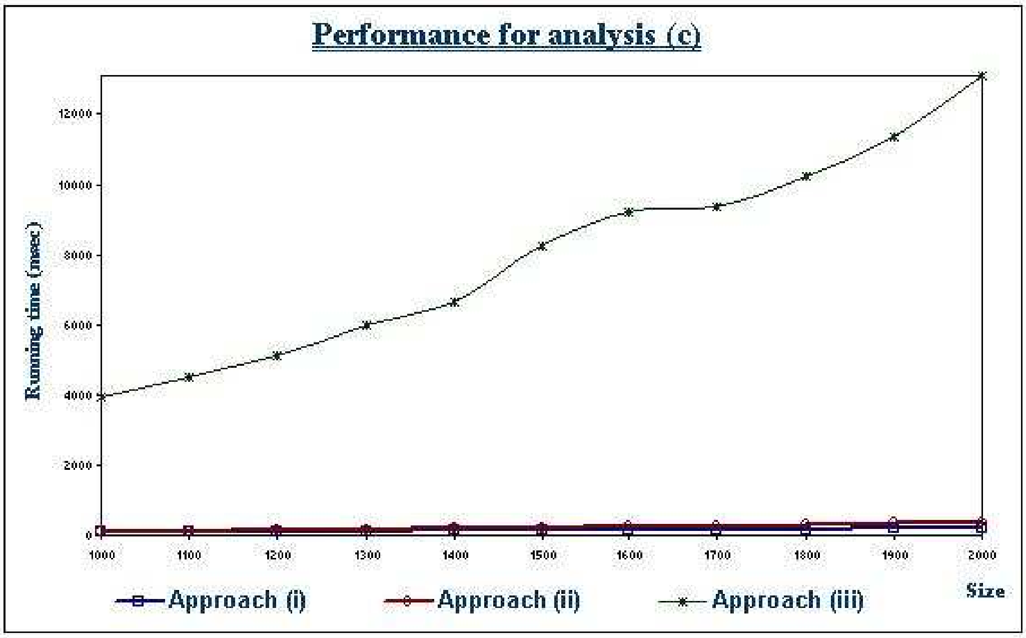
From the plots, the approach (i), the dedicated methods approach, seems to yield slightly faster implementations for these analyses. The poorer performance of the approach (ii) may have been caused by the method `doSwitch` in the EMF-switch class. Recall from Chapter 3 that this method is responsible for selecting and invoking the appropriate `case` methods for a given object. To select the appropriate `case` methods, the type of the object must be determined. EMF has its own mechanism to determine the type of the object. This mechanism may impose a small performance penalty on any implementation that exploits the EMF-switch class.

10.3 The reflection based approaches

Now, let us compare the performance of the previous implementations to the corresponding implementations by means of the reflection based walker.





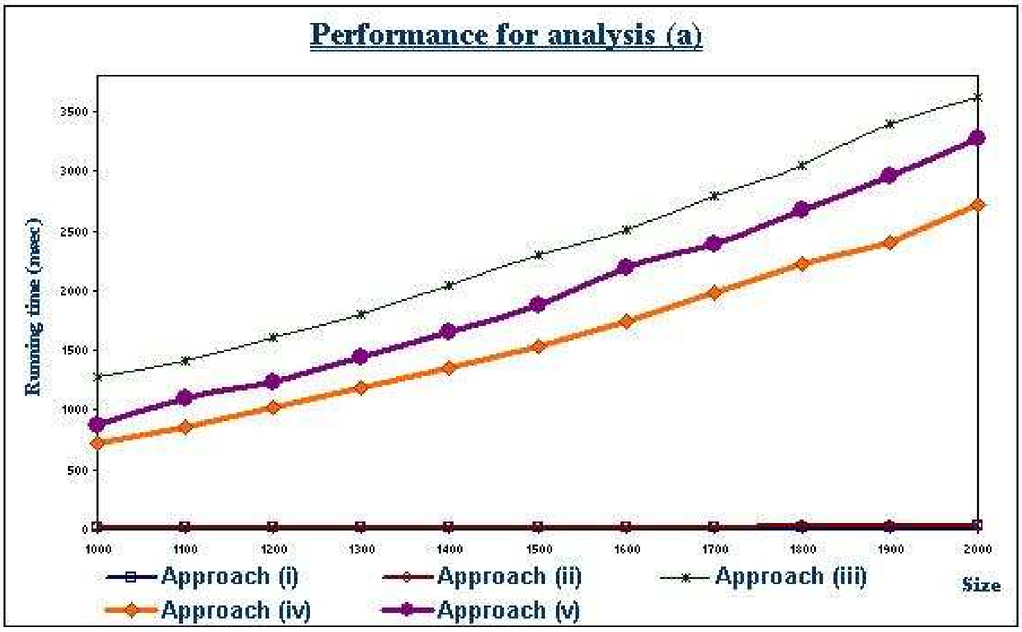


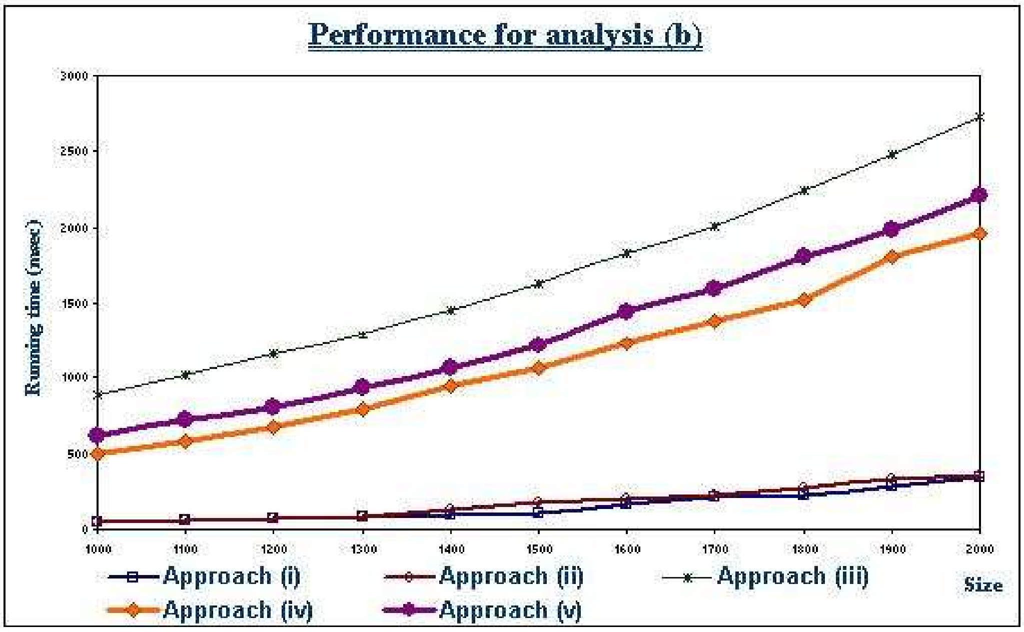
The approaches (i) and (ii), as clearly shown in the plots, give rise to implementations that are much faster. To be precise, we compute the ratio of their running times for all three analyses. The results are summarized in the table below.

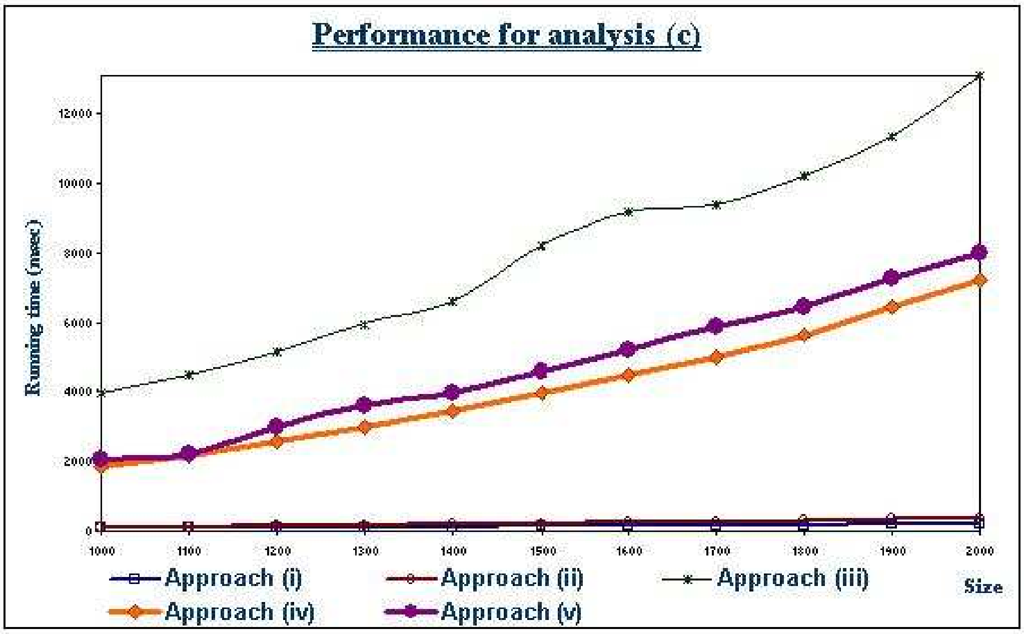
	(a)	(b)	(c)
(iii)/(i)	152	13	55
(iii)/(ii)	119	11	36

For example, the implementation of the `ActivityFinder` by means of the dedicated methods is, on average, roughly 152 times faster than the implementation by means of the reflection based walker.

Forax and Roussel [25] (also Bravenboer and Visser [9]) have shown that the performance of the walker can be improved considerably by caching fields and methods. Also, Grothoff [28] achieves this by using runtime code generation techniques. In the plots below, we observe considerable improvement in the performance.


Performance for analysis (a)
Running time (msec)
Size
Approach (i)
Approach (ii)
Approach (iii)
Approach (iv)
Approach (v)


Performance for analysis (b)
Running time (msec)
Size
Approach (i)
Approach (ii)
Approach (iii)
Approach (iv)
Approach (v)



However, as to how much it will improve the performance, may also depend on the types of the analyses. The ratios of their running times, captured in the table below,

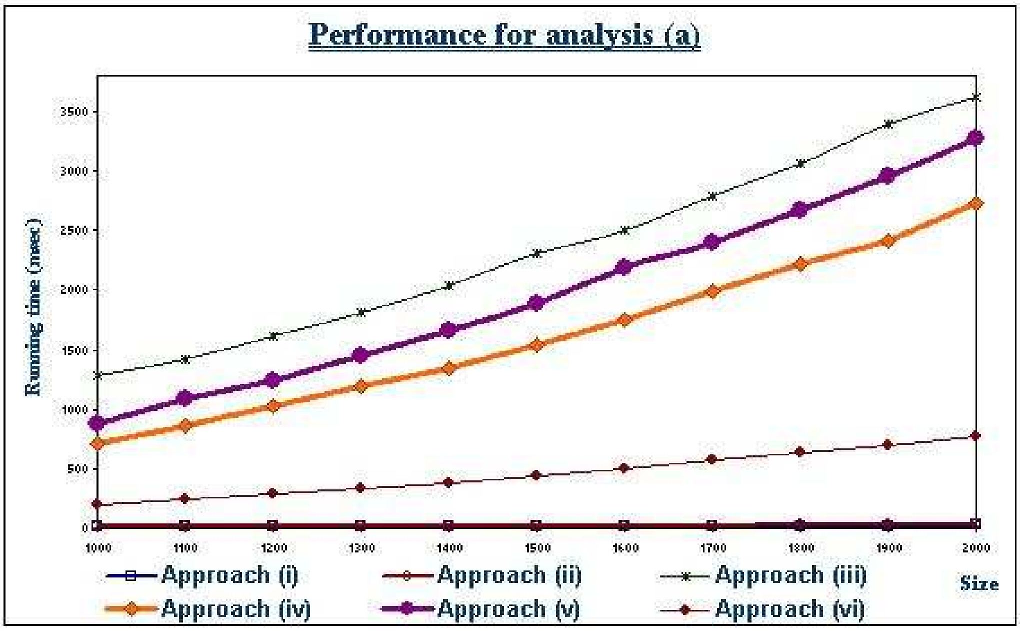
	(a)	(b)	(c)
(iii)/(iv)	1.5	1.5	2
(iii)/(v)	1.2	1.3	2

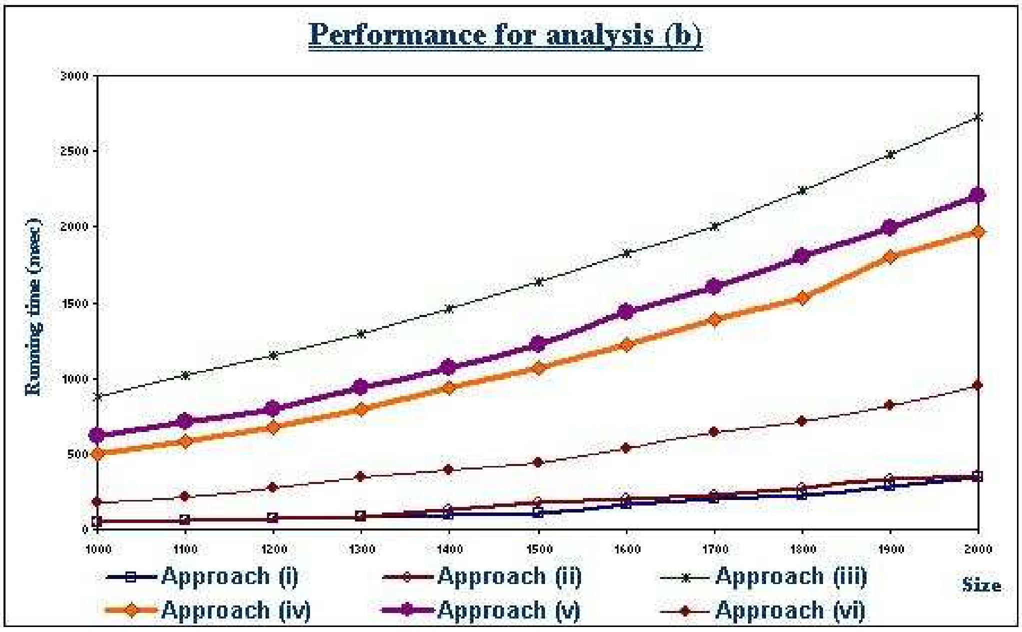
show that the implementations by means of the approach (iv) and (v) are twice as fast as the implementation by means of approach (iii) for the analysis (c). There is only a small improvement in the performance for the analyses (a) and (b).

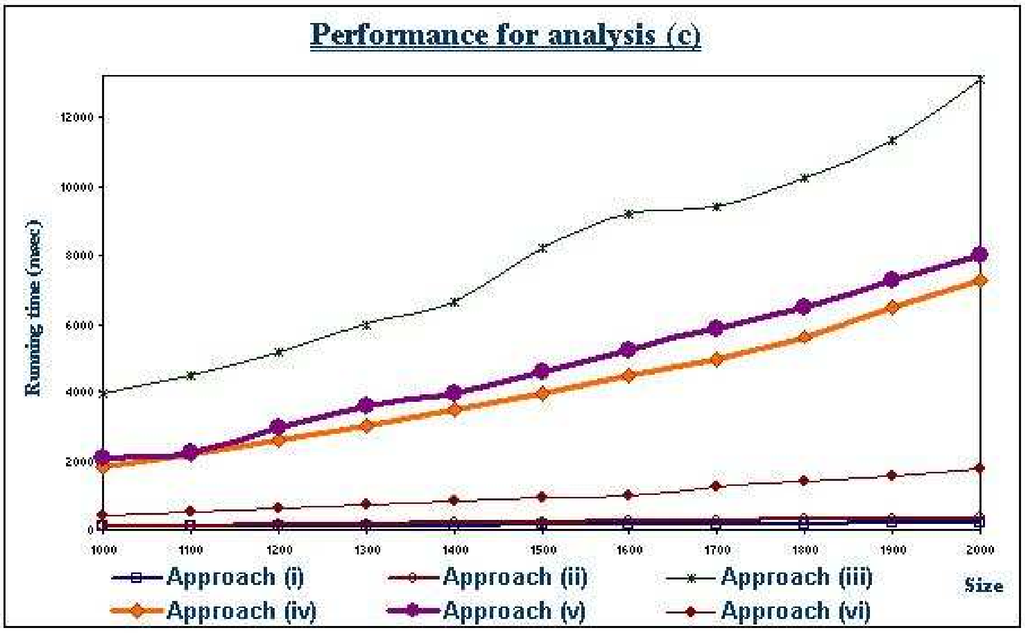
We also notice that those implementations by means of the approach (v) are slightly slower than those by means of the approach (iv) for all three analyses. The overhead, which is introduced due to code generation at runtime (see Chapter 5), may have been the main contributor to the poor performance of the walker of Grothoff [28].

10.4 The walker with preprocessing

The walkers with preprocessing for the analyses (a), (b) and (c) no longer depends on Java's reflection mechanism to select and invoke a `visit` method. Also, in the walkers with preprocessing, not all the subtrees will be walked. Therefore, we expect that they will outperform other walkers. The plots below, without a question, have confirmed our expectation.


Performance for analysis (a)
Running time (msec)
Size
Approach (i)
Approach (ii)
Approach (iii)
Approach (iv)
Approach (v)
Approach (vi)


Performance for analysis (b)
Running time (msec)
Size
Approach (i)
Approach (ii)
Approach (iii)
Approach (iv)
Approach (v)
Approach (vi)



Let us compute the ratio of the running time of the approaches (iii), (iv), and (v) and the running time of our approach for the analyses (a), (b), and (c).

	(a)	(b)	(c)
(iii)/(vi)	5	4	8
(iv)/(vi)	4	2	4
(v)/(vi)	4	3	5

The implementations by means of our approach are at least 4 times faster than the implementation by means of the reflection based walker. For analysis (c), the implementation by our approach is even 8 times faster. Furthermore, the walker with preprocessing is also faster than both the walker with caching of fields and methods, and Grothoff's walker [28].

Now let us compute the ratio of the running time of our approach to the running time of the approaches (i) and (ii), those which do not rely on Java's reflection mechanism, for the analyses (a), (b), and (c).

	(a)	(b)	(c)
(vi)/(i)	29	3	7
(vi)/(ii)	23	3	4

On average, the implementation of the BPEL-calculus translator by means of our approach is only 3 times slower than the implementations of the translator by means of the dedicated methods approach, and the EMF-switch class.

Also, recall from Section 10.3 that by means of the reflection based walker, the implementation of the `ActivityFinder` is 152 times slower than the implementation of the `ActivityFinder` by means of the dedicated methods approach. The walker with preprocessing for the `ActivityFinder` is only 29 times slower.

10.5 Summary

In general, the dedicated methods approach provides an efficient way to implement these analyses of BPEL4WS. Exploiting the EMF-switch class to implement these analyses also gives rise to roughly the same performance.

The reflection based walker, although allowing the resulting code to be produced much quicker, unfortunately gives rise to implementations which are considerably slower. We sometimes observe even more than 100 times slower (for example, the analysis (a)). For a better performance, one can exploit the walker with caching of fields and methods or alternatively the walker of Grothoff [28] to implement these analyses. These walkers are shown to provide slightly faster implementations for these analyses.

Among the walkers, our approach has achieved the best performance. Our approach, sometimes, gives rise to implementations which are comparable to the implementations by means of the dedicated methods approach. For example, the implementation of the BPEL-calculus translator by means of the walker with preprocessing is only 3 times slower than its implementation by means of dedicated methods. In general, our approach allows the resulting code for the analyses to be quickly produced with a reasonable performance. However, the main drawback of our approach is that the Java code needs to be generated and compiled again if the EMF model of BPEL4WS changes.

11 Conclusion

11.1 Summary and discussion

As a part of the effort to develop analysis tools for BPEL4WS, we have presented different approaches to implementing analyses of BPEL4WS. Typically, an analysis mainly consists of code that focuses on how to walk the ASTs of BPEL4WS. The approaches presented in this thesis outline different techniques to walk the ASTs. One first needs to build a representation of the ASTs of BPEL4WS. Since EMF already provides a hierarchy of Java classes and interfaces for BPEL4WS, we exploited this hierarchy to represent the ASTs.

Some of the approaches to implement analyses are found to be not very suitable for BPEL4WS. For example, the dedicated methods approach is not very suitable because this approach requires extensive modifications to the classes and interfaces of the hierarchy. Several problems can arise when modifying them. Firstly, there may already be other analyses implemented for BPEL4WS that also exploit the hierarchy. If one modifies the classes and interfaces one must be certain that the modifications will not have an unexpected impact on the other analyses. Secondly, if the BPEL4WS language changes, thereby causing the hierarchy of classes and interfaces to be generated again by EMF, these inserted methods may be lost. Finally, since these methods are scattered across different classes, it can be difficult to debug the code and assure the correctness of the implementation for an analysis.

Alternatively, we presented the visitor design pattern to walk the trees. This design pattern provides us a way to attach new methods to an existing hierarchy of classes and interfaces with a minimal modification to the hierarchy. However, since one should be refrained from altering the EMF model of BPEL4WS as it may be used by other analyses as well, this approach is still not ideal to implement analyses of BPEL4WS.

Other approaches, that we have discussed, were the syntax separate from interpretations approach and the EMF-switch approach. Both approaches do not modify the hierarchy in any way, but share a different disadvantage. If the XSD specification of BPEL4WS changes and, hence, introduces changes to the hierarchy of classes and interfaces, then the code for all those classes that have been affected by the changes may have to be modified.

One can also exploit compiler toolkits to generate walkers that walk the trees in some order. In this thesis, we discussed how one can make use of SableCC to generate such walkers. However, for us to exploit SableCC, the XSD specification of BPEL4WS must be translated into the specification formalism used by the toolkit. As the language evolves, maintaining two separate specifications may be unnecessarily time-consuming.

Most of these problems can be resolved with the reflection based walker. This walker uses Java's reflection mechanism to walk the trees. As we have shown, the implementations by means of reflection based walkers do not require any modification to the EMF model of BPEL4WS. It also allows one to focus on the analysis itself, rather than on walking the trees. Bravenboer and Visser [9] refined and improved the walkers. In particular, they do not walk fields that are 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 thesis, we extended the 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.

The main drawback of the reflection based walker is its rather poor performance. In this thesis, we addressed the two main causes of the poor performance of the walker. On the one hand, it is caused by the poor performance of Java's reflection mechanism. On the other hand, the fact that a reflection based walker may traverse parts of the tree that do not need to be walked can also contribute to its poor performance. We reviewed two techniques which addressed the first cause. These techniques involve caching fields and methods (proposed by Bravenboer and Visser [9] also by Forax and Roussel [25]), and generating code at runtime (proposed by Grothoff [28]). In particular, Bravenboer and Visser [9] cache `Method` objects, which represent `visit` methods, and `Field` objects, which represent fields. Grothoff [28], instead of using Java's reflection mechanism to invoke the `visit` methods, generates code to invoke these methods. Generating code at runtime induces some overhead. Inspired by Grothoff's work, we propose preprocessing the code that implements an analysis of BPEL4WS and generating new Java code for it. During the preprocessing step, we compute the set of the so-called visitable classes. For each visitable class, we find and generate code to invoke the corresponding `visit` method. The code generated to invoke the `visit` methods is now free of any use of Java's reflection mechanism. Consequently, the performance of the walker with preprocessing improved significantly.

During the preprocessing step, we also single out some subtrees that need not be walked by approximating those fields that may have to be walked. Once these fields have been found, we can address the second cause of the poor performance by generating traversal code that tries to minimize the unnecessary walking. We generate Java code that guides the traversal which does not iterate over all fields. In the reflection based walker, all the fields are walked, whereas in the walker with preprocessing only the so-called walkable fields are walked. Consequently, the performance of the this walker improved further.

Exploiting reflection based walkers of the EMF model of BPEL4WS, we have implemented a number of analyses. In this thesis, we discussed and evaluated the performance of three analyses of BPEL4WS by means of various approaches, namely, the activity finder, a BPEL-calculus translator, and a BPEL4WS graph builder. We have shown that the non-reflection based approaches, such as the dedicated methods approach and the EMF-switch mechanism give rise to very fast implementations for these analyses. Unfortunately, the performance of implementations by means of the reflection based walker are incomparable to the implementations by means of the non-reflection based approaches. However, the walker with caching of fields and methods and the walker of Grothoff [28] have been shown to give rise to implementations which are approximately twice as fast as the original walker. More importantly, the walker with preprocessing has been shown to further improve the performance. The implementations by means of this walker are at least 4 times faster than the implementations by means of the reflection based walker.

Also, exploiting walkers, tools have been developed

- to detect control cycles in a BPEL4WS program,
- to check that each link of a BPEL4WS program has a unique source and a unique target,
- to translate a BPEL4WS program into a BPE-process (for the latter properties can be verified using the Concurrency Workbench), and
- to check if dead-path-elimination gives rise to side effects in a BPEL4WS program.

11.2 Future work

11.2.1 Enhancing the performance of the walker with preprocessing

Currently, the walker with preprocessing will walk all the array objects. The fact that the generated method `walk` may walk array objects that need not be walked, leaves room for further improvement in the performance. By limiting ourselves to walk only those array objects whose base type is waitable, we can further eliminate unnecessary walking.

The walker with preprocessing is not completely free of Java's reflection mechanism. The generated traversal code still depends on Java's reflection mechanism to obtain the so-call walkable fields of an object. If the fields can be obtained without using Java's reflection mechanism, the performance will be expected to be improved even more.

11.2.2 Exploiting EMF-switch in the walker with preprocessing

Chapter 10 has shown that exploiting the EMF-switch results in much faster implementations. We believe that once the visitable classes and walkable fields have been found, one can also generate from a walker an implementation that exploits the EMF-switch. The EMF-switch class can be exploited in different ways.

Firstly, we can exploit the `case` methods for traversal code and secondly for invoking the `visit` methods. For these two separate tasks, we can create two instances of the EMF-switch class.

The first instance defines a set of `case` methods consisting of traversal code. Once the walkable fields have been found, we can generate the appropriate `case` methods which contain the appropriate traversal code. Exploiting the `doSwitch` method will give rise to appropriate `case` methods which guide the traversal to be invoked.

The second instance defines a set of `case` methods for those objects that have a `visit` method introduced in the walker. Once the set of visitable classes has been found, for each `visit` method body found in the reflection based walker, we generate a corresponding `case` method. Again, exploiting the `doSwitch` method will give rise to appropriate `case` methods which correspond to the appropriate `visit` methods in the reflection based walker, to be invoked.

The fact that the code must be generated and compiled again every time the EMF hierarchy changes is also a disadvantage of this approach. Another disadvantage of this approach is that a walker, which relies on the EMF-switch mechanism, will only walk the objects of the EMF model. The walker with or without preprocessing is not limited to walking the EMF model, but can also walk other objects.

Bibliography

[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/library/ws-bpel, May 2003.

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

[3] The AspectJ Project. Available at www.eclipse.org/aspectj.

[4] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, and Claude Pasquier. SmartTools: a generator of interactive environment tools. In Reinhard Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 355–360, Genova, Italy, April 2001. Springer–Verlag.

[5] Elliot Berk. JLex: A lexical analyzer generator for Java. Available at www.cs.princeton.edu/~appel/modern/java/JLex, October 1997. JLex's manual.

[6] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes. Available at www.w3.org/TR/xmlschema-2, October 2004.

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

[8] John Boyland and Giuseppe Castagna. Parasitic methods: an implementation of multi-methods for Java. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 66–76, Atlanta, Georgia, United States, October 1997. ACM.

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

[10] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible markup language (XML). Available at www.w3.org/TR/REC-xml, February 2004.

[11] Franck van Breugel and Mariya Koshkina. Dead-path-elimination in BPEL4WS. In *Proceedings of the 5th International Conference on Application of Concurrency to Systems Design*, pages 192–201, St Malo, June 2005. IEEE.

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

[13] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56, Utrecht, July 1992. Springer-Verlag.

[14] Erik Christensen, Francisco Curbera, Grey Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. Available at www.w3.org/TR/wsdl, March 2001.

[15] James Clark and Steve DeRose. XML path language (XPath), version 1.0. Available at www.w3.org/TR/xpath, November 1999.

[16] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-Ins*. The Eclipse Series. Addison-Wesley, 2004.

[17] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, March 2002.

[18] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthey. *The Java Developer's Guide to Eclipse.* Addison-Wesley, 2004.

[19] The Eclipse Project. Available at www.eclipse.org.

[20] The Eclipse Modelling Framework Project. Available at www.eclipse.org/emf.

[21] Étienne Gagnon. SableCC, an object-oriented compiler framework. Master's thesis, McGill University, School of Computer Science, Montreal, March 1998. Available at www.sablecc.org.

[22] Paolo Ferragina, S. Muthukrishnan, and Mark de Berg. Multi-method dispatching: a geometric approach with applications to string matching problems. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 483–491, Atlanta, Georgia, United States, May 1999. ACM.

[23] Rémi Forax, Etienne Duris, and Gilles Roussel. Java multi–method framework. In *Proceedings of International Conference on Technology of Object–Oriented Languages and Systems*, pages 45–56, Sydney, Australia, November 2000. IEEE.

[24] Rémi Forax, Etienne Duris, and Gilles Roussel. Reflection-based implementation of Java extensions: the double-dispatch use-case. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 1409–1413, Santa Fe, New Mexico, March 2005. ACM.

[25] Rémi 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, Germany, 2000. Springer-Verlag.

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

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[28] Christian Grothoff. Walkabout revisited: the Runabout. In Luca Cardelli, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 103–125, Darmstadt, July 2003. Springer-Verlag.

[29] Cay S. Horstmann. *Practical Object-Oriented Development in C++ and Java.* John Wiley & Sons, May 1997.

[30] Scott Hudson. CUP Parser Generator for Java. Available at www.cs.princeton.edu/~appel/modern/java/CUP, March 1998. Java Cup's manual.

[31] The Java Development Tools Project. Available at www.eclipse.org/jdt.

[32] Brian W. Kernighan and Rob Pike. *The Practice of Programming.* Professional computing series. Addison–Wesley, 1999.

[33] Gregor Kiczales, Eric Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2071 of *Lecture Notes in Computer Science*, pages 327–353, Budapest, June 2001. Springer-Verlag.

[34] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, June 1997. Springer-Verlag.

[35] Gerwin Klein. JFlex – The Fast Scanner Generator for Java. Available at www.jflex.de, April 2004. JFlex's manual.

[36] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction in: *Mathematical Systems Theory* 5(1):95–96, 1971.

[37] 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.

[38] 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.

[39] Ramnivas Laddad. *AspectJ in Action.* Manning Publications Co., 2003.

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

[41] John R. Mashey. War of the benchmark means: time for a truce. *SIGARCH Computer Architecture News*, 32(4):1–14, September 2004.

[42] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[43] Paul Muschamp. An introduction to Web Services. *BT Technology Journal*, 22(1):9–18, March 2004.

[44] 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.

[45] 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.

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

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

[48] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML schema part 1: Structures. Available at www.w3.org/TR/xmlschema-1, October 2004.

[49] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java: compilers and interpreters*. Prentice Hall, 2000.

[50] David S. Wile. Abstract syntax from concrete syntax. In *Proceedings of the 19th International Conference on Software Engineering*, pages 472–480, Boston, Massachusetts, United States, May 1997. ACM.

A XML Schema for the join conditions

Below we present the XSD specification of the join conditions of BPEL4WS. This specification has been used in Section 3.5 to generate a hierarchy of Java classes and interfaces that represents the syntax of the join conditions.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="Condition"/>
  <xsd:element name="and" type="And"/>
  <xsd:complexType name="And">
    <xsd:complexContent>
      <xsd:extension base="Condition">
         <xsd:sequence>
            <xsd:element name="left" type="Condition"/>
               <xsd:element name="right" type="Condition"/>
         </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:element name="not" type="Not"/>
  <xsd:complexType name="Not">
     <xsd:complexContent>
        <xsd:extension base="Condition">
           <xsd:sequence>
              <xsd:element name="condition" type="Condition"/>
           </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:element name="or" type="Or"/>
  <xsd:complexType name="Or">
     <xsd:complexContent>
        <xsd:extension base="Condition">
           <xsd:sequence>
              <xsd:element name="left" type="Condition"/>
                 <xsd:element name="right" type="Condition"/>
           </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:element name="link" type="Link"/>
  <xsd:complexType name="Link">
     <xsd:complexContent>
        <xsd:extension base="Condition">
           <xsd:sequence>
              <xsd:element name="name" type="xsd:string"/>
           </xsd:sequence>
        </xsd:extension>
     </xsd:complexContent>
  </xsd:complexType>

  <xsd:element name="true" type="True"/>
  <xsd:complexType name="True">
     <xsd:complexContent>
        <xsd:extension base="Condition"/>
     </xsd:complexContent>
  </xsd:complexType>

  <xsd:element name="false" type="False"/>
```

```
  <xsd:complexType name="False">
     <xsd:complexContent>
        <xsd:extension base="Condition"/>
     </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

B Performance data

The following tables describe sets of data collected from executing different implementations for analyses (a), (b) and (c) as explained in Chapter 10. Column 1 of the tables lists the size of the program inputs. The size of a program is measured by the number of nodes of the tree representing the BPEL4WS program. Column 2 lists the approaches. Column 3-12 list the 10 data points representing the running times, measured in milliseconds, for each implementation. Column 13 calculates the mean of the running times. These means were used for the plots in Section 10. Finally, column 14 calculates the spread of the data points.

B.1 Analysis (a)

N	A	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	μ	σ
	(i)	16	0	15	15	0	0	16	0	15	15	9	7
	(ii)	16	15	0	16	16	16	0	15	0	16	11	7
	(iii)	1297	1219	1297	1297	1281	1297	1281	1281	1281	1281	1281	22
1000	(iv)	718	703	703	719	719	719	719	703	719	719	714	7
	(v)	875	874	875	875	875	890	891	875	875	875	878	6
	(vi)	204	203	203	203	187	218	187	203	203	203	201	8
	(i)	0	15	0	16	15	0	16	0	16	15	9	7
	(ii)	15	16	16	15	0	16	15	16	16	0	12	6
	(iii)	1485	1406	1406	1406	1406	1406	1390	1406	1390	1406	1410	25
1100	(iv)	859	859	859	859	859	874	859	859	874	859	862	6
	(v)	1062	1063	1062	1062	1078	1094	1063	1093	1156	1157	1089	35
	(vi)	235	250	250	234	250	235	250	250	235	235	242	7
	(i)	16	15	16	15	0	16	15	16	0	0	10	7
	(ii)	15	16	15	0	16	15	16	15	15	16	13	4
	(iii)	1609	1609	1610	1609	1609	1609	1594	1609	1593	1609	1606	6
1200	(iv)	1015	1141	1000	1000	1016	1016	1000	1000	1000	1000	1018	41
	(v)	1235	1281	1234	1219	1250	1234	1234	1234	1234	1235	1239	15
	(vi)	282	281	281	281	265	281	281	281	282	282	279	4
	(i)	16	15	16	15	16	15	16	15	16	16	15	0
	(ii)	16	16	16	16	16	16	16	16	31	16	17	4
	(iii)	1813	1796	1813	1812	1812	1797	1797	1812	1813	1797	1806	7
1300	(iv)	1188	1187	1188	1171	1172	1172	1171	1172	1281	1187	1188	31
	(v)	1437	1453	1437	1437	1437	1453	1453	1485	1437	1452	1448	14
	(vi)	328	328	328	329	328	328	329	328	312	328	326	4
	(i)	15	16	16	15	16	0	15	16	15	16	14	4
	(ii)	32	31	15	16	15	31	16	15	16	15	20	7
	(iii)	2031	2047	2031	2031	2046	2046	2031	2031	2031	2047	2037	7
1400	(iv)	1344	1343	1359	1344	1343	1343	1359	1344	1343	1344	1346	6
	(v)	1656	1656	1656	1671	1656	1625	1656	1656	1656	1656	1654	10
	(vi)	390	375	375	390	375	375	375	375	375	375	378	6
	(i)	16	16	16	15	16	16	15	16	16	15	15	0
	(ii)	31	16	16	32	15	16	16	15	15	32	20	7
	(iii)	2265	2265	2265	2265	2265	2327	2265	2265	2266	2594	2304	98
1500	(iv)	1531	1531	1531	1546	1547	1547	1531	1531	1531	1547	1537	7
	(v)	1875	1875	1891	1875	1875	1875	1875	1922	1875	1890	1882	14
	(vi)	437	438	437	438	422	437	437	438	438	437	435	4
	(i)	16	16	16	16	16	15	15	16	16	15	15	0
	(ii)	16	16	31	15	15	32	16	15	31	16	20	7
	(iii)	2515	2515	2516	2515	2500	2515	2515	2500	2500	2515	2510	6
1600	(iv)	1766	1750	1749	1749	1749	1734	1734	1734	1734	1734	1743	10
	(v)	2203	2218	2188	2172	2157	2234	2203	2171	2141	2188	2187	26
	(vi)	500	485	500	500	500	500	500	500	500	500	498	4
	(i)	16	16	16	16	31	16	16	16	16	15	17	4
	(ii)	15	16	16	16	16	31	31	31	31	16	21	7
	(iii)	2953	2781	2766	2765	2781	2781	2765	2765	2766	2781	2790	54
1700	(iv)	1953	1953	2000	1953	1937	1937	1968	2125	2047	1968	1984	56
	(v)	2406	2391	2390	2390	2390	2390	2391	2437	2390	2374	2394	15
	(vi)	563	546	609	562	578	562	563	563	656	563	576	30
	(i)	16	16	16	15	15	31	31	15	15	31	20	7
	(ii)	15	31	31	31	32	16	31	32	32	16	26	7
	(iii)	3046	3031	3046	3109	3046	3046	3046	3031	3031	3109	3054	28
1800	(iv)	2218	2375	2188	2218	2203	2203	2297	2187	2171	2187	2224	59
	(v)	2672	2687	2672	2671	2672	2672	2656	2672	2703	2672	2674	11
	(vi)	703	641	624	640	641	640	640	625	625	625	640	22
	(i)	15	16	31	15	16	31	31	15	16	31	21	7
	(ii)	31	31	31	15	31	31	15	16	31	31	26	7
	(iii)	3343	3343	3328	3687	3327	3328	3328	3656	3328	3312	3398	137
1900	(iv)	2421	2406	2406	2421	2406	2406	2406	2406	2421	2406	2410	6
	(v)	2953	2953	2984	2984	2969	2968	2937	2953	2968	2968	2963	14
	(vi)	688	703	703	703	703	703	688	703	688	687	696	7
	(i)	32	16	31	31	31	16	15	31	16	31	25	7
	(ii)	31	31	31	31	31	32	31	32	32	16	29	4
	(iii)	3625	3624	3625	3625	3609	3624	3625	3609	3609	3624	3619	7
2000	(iv)	2734	2719	2734	2719	2734	2718	2719	2719	2718	2719	2723	7
	(v)	3265	3265	3249	3265	3280	3265	3265	3281	3265	3281	3268	9
	(vi)	781	765	765	781	765	781	765	765	782	765	771	7

B.2 Analysis (b)

N	A	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	μ	σ
	(i)	63	47	47	47	47	47	62	62	63	47	53	7
	(ii)	47	63	62	46	47	47	47	47	47	47	50	6
	(iii)	828	875	891	890	890	891	891	891	875	890	881	18
1000	(iv)	469	484	484	469	485	516	485	562	485	563	500	33
	(v)	515	547	563	578	562	578	609	922	609	640	612	108
	(vi)	188	188	172	172	188	203	187	171	172	172	181	10
	(i)	63	62	46	62	63	63	62	62	47	47	57	7
	(ii)	62	63	79	47	62	62	63	63	62	62	62	7
	(iii)	968	1016	1031	1031	1015	1016	1031	1047	1031	1015	1020	19
1100	(iv)	625	578	593	610	562	578	562	578	578	579	584	18
	(v)	750	688	688	718	828	672	688	672	766	687	715	48
	(vi)	219	219	219	219	219	219	219	218	218	219	218	0
	(i)	78	63	62	79	78	79	62	62	63	63	68	7
	(ii)	62	78	78	78	78	62	79	78	78	78	74	6
	(iii)	1109	1156	1172	1156	1156	1172	1156	1156	1156	1172	1156	17
1200	(iv)	687	672	672	672	672	672	672	672	672	672	673	4
	(v)	766	812	796	797	812	797	797	812	797	797	798	12
	(vi)	360	265	250	312	250	250	250	265	266	250	271	34
	(i)	94	79	93	93	94	94	78	78	78	78	85	7
	(ii)	125	93	78	79	78	78	94	94	78	79	87	14
	(iii)	1250	1281	1296	1281	1296	1312	1281	1328	1297	1297	1291	19
1300	(iv)	828	782	781	797	781	781	797	797	781	797	792	14
	(v)	875	921	938	922	922	937	1000	937	937	937	932	28
	(vi)	297	344	344	343	344	344	328	343	344	359	339	15
	(i)	110	109	93	109	94	94	78	78	94	93	95	10
	(ii)	125	125	125	125	125	125	140	141	141	141	131	7
	(iii)	1375	1453	1500	1453	1453	1453	1469	1453	1469	1469	1454	30
1400	(iv)	938	938	953	938	937	938	937	938	937	938	939	4
	(v)	1015	1078	1063	1078	1079	1078	1063	1078	1063	1078	1067	18
	(vi)	390	390	390	391	375	391	391	391	391	391	389	4
	(i)	156	109	109	109	109	109	109	94	109	109	112	15
	(ii)	156	187	188	188	188	188	188	187	172	172	181	10
	(iii)	1547	1672	1641	1625	1625	1640	1672	1625	1625	1656	1632	33
1500	(iv)	1078	1062	1063	1063	1062	1062	1063	1062	1078	1063	1065	6
	(v)	1187	1375	1203	1203	1219	1203	1203	1203	1203	1203	1220	52
	(vi)	438	437	438	437	438	438	438	438	438	438	437	0
	(i)	156	156	157	172	172	157	172	156	156	172	162	7
	(ii)	203	203	203	203	188	203	187	204	203	188	198	7
	(iii)	1719	1828	1844	1844	1844	1828	1859	1828	1843	1828	1826	37
1600	(iv)	1250	1218	1219	1218	1235	1234	1218	1203	1235	1218	1224	12
	(v)	1422	1422	1437	1407	1437	1422	1422	1578	1437	1407	1439	47
	(vi)	500	547	546	532	531	547	532	531	531	531	532	12
	(i)	203	203	218	203	203	203	219	203	204	218	207	7
	(ii)	235	234	235	219	218	235	203	219	218	219	223	10
	(iii)	1922	2015	2016	2015	2000	2015	2031	2015	2000	2015	2004	28
1700	(iv)	1391	1359	1391	1390	1390	1422	1375	1374	1375	1375	1384	16
	(v)	1515	1578	1609	1703	1609	1593	1594	1578	1593	1594	1596	43
	(vi)	641	656	641	640	656	640	641	641	640	656	645	7
	(i)	234	234	234	234	234	219	234	234	218	234	230	6
	(ii)	281	266	281	266	281	281	266	266	282	266	273	7
	(iii)	2109	2234	2249	2250	2328	2265	2250	2234	2234	2250	2240	51
1800	(iv)	1531	1516	1515	1531	1547	1515	1531	1531	1531	1515	1526	10
	(v)	1735	1812	1812	1813	1796	1813	1812	1844	1813	1812	1806	26
	(vi)	719	719	719	703	703	703	719	703	719	703	711	8
	(i)	281	281	281	281	281	282	281	297	297	281	284	6
	(ii)	359	328	328	328	328	328	328	344	328	344	334	10
	(iii)	2344	2484	2484	2500	2500	2500	2484	2484	2484	2499	2476	44
1900	(iv)	1734	1922	1734	2000	1734	2000	1734	1750	1734	1734	1807	110
	(v)	1890	2000	2000	1984	2016	2031	2000	1984	2000	1984	1988	35
	(vi)	844	813	828	828	828	812	828	828	812	828	824	9
	(i)	344	344	344	343	344	359	344	344	343	344	345	4
	(ii)	390	359	359	360	375	344	343	359	360	359	360	12
	(iii)	2562	2734	2734	2734	2735	2765	2766	2734	2734	2781	2727	57
2000	(iv)	1984	1922	1968	1969	1953	1968	1984	1953	1969	1999	1966	20
	(v)	2157	2234	2235	2203	2202	2188	2203	2219	2218	2203	2206	21
	(vi)	937	938	953	953	937	953	953	938	937	953	945	7

B.3 Analysis (c)

N	A	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	μ	σ
	(i)	94	78	94	78	94	93	94	94	93	94	90	6
	(ii)	94	109	94	109	94	110	109	110	110	109	104	7
	(iii)	3906	4047	4062	4062	4062	4046	3843	3827	3828	3843	3952	105
1000	(iv)	1719	1828	1812	1891	1844	1844	1828	1828	1859	1828	1828	41
	(v)	1953	2078	2094	2093	2078	2078	2093	2094	2078	2062	2070	40
	(vi)	422	422	421	422	422	453	422	422	422	421	424	9
	(i)	93	94	93	109	94	94	94	110	109	109	99	7
	(ii)	125	125	125	125	125	125	125	125	125	125	125	0
	(iii)	4312	4500	4499	4468	4516	4484	4515	4578	4500	4547	4491	66
1100	(iv)	2172	2172	2187	2172	2188	2203	2172	2172	2171	2203	2181	12
	(v)	2125	2250	2234	2234	2235	2250	2250	2234	2281	2250	2234	38
	(vi)	516	500	500	515	500	500	500	515	500	500	504	7
	(i)	109	125	110	125	125	110	109	109	125	125	117	7
	(ii)	156	140	156	141	141	140	156	156	141	141	146	7
	(iii)	4968	5140	5172	5156	5172	5156	5140	5172	5156	5140	5137	57
1200	(iv)	2499	2593	2578	2594	2594	2578	2593	2593	2593	2577	2579	27
	(v)	2796	2969	2969	2984	2952	2969	2984	2969	2968	2984	2954	53
	(vi)	609	594	610	610	609	609	593	594	610	609	604	7
	(i)	125	125	125	140	125	109	125	125	125	125	124	6
	(ii)	171	156	172	157	157	172	172	156	156	156	162	7
	(iii)	6703	5891	5890	5906	5891	5890	5890	5890	5890	5891	5973	243
1300	(iv)	2890	3000	3015	3016	3015	3016	3031	3031	3015	3016	3004	39
	(v)	3374	3687	3578	3641	3640	3578	3594	3578	3625	3593	3588	79
	(vi)	703	766	734	703	688	687	687	688	703	688	704	24
	(i)	125	125	140	141	125	125	125	124	125	125	128	6
	(ii)	203	203	188	187	187	187	188	204	219	188	195	10
	(iii)	6422	6672	6703	6656	6656	6656	6640	6718	6671	6687	6648	78
1400	(iv)	3250	3484	3484	3484	3484	3484	3468	3485	3500	3484	3460	70
	(v)	3735	4031	3969	3985	3984	3968	3984	3984	3985	3984	3960	77
	(vi)	812	829	812	796	812	812	797	812	812	812	810	8
	(i)	125	140	141	141	140	140	141	140	157	140	140	7
	(ii)	219	219	203	218	219	203	219	219	203	203	212	7
	(iii)	7874	8202	8234	8234	8265	8202	8218	8203	8687	8233	8235	184
1500	(iv)	3688	3984	3969	3984	3968	3969	3969	3968	3985	3968	3945	86
	(v)	4312	4593	4625	4594	4562	4578	4563	4578	4577	4562	4554	82
	(vi)	953	938	938	953	938	953	937	937	937	922	940	9
	(i)	156	156	140	141	156	141	141	156	141	141	146	7
	(ii)	234	235	250	250	234	234	250	234	234	250	240	7
	(iii)	9171	9186	9187	9203	9234	9187	9186	9202	9202	9202	9196	16
1600	(iv)	4187	4563	4499	4484	4500	4500	4484	4499	4500	4500	4471	97
	(v)	4890	5187	5203	5203	5265	5187	5249	5187	5422	5171	5196	123
	(vi)	984	969	953	969	969	953	984	968	969	969	968	9
	(i)	172	156	156	156	156	157	172	156	172	172	162	7
	(ii)	266	250	250	266	265	249	250	266	266	265	259	7
	(iii)	8968	9875	9874	9312	9296	9312	9296	9312	9311	9281	9383	264
1700	(iv)	4656	5031	4984	4984	4984	4984	4984	4999	5000	5000	4960	102
	(v)	5562	5875	6500	5921	5641	5906	5859	5859	5844	5906	5887	233
	(vi)	1234	1234	1219	1235	1219	1234	1218	1234	1219	1218	1226	7
	(i)	187	171	156	172	172	171	172	172	172	172	171	6
	(ii)	297	297	297	297	297	297	297	297	297	297	297	0
	(iii)	9858	10249	10249	10281	10265	10264	10265	10250	10296	10265	10224	122
1800	(iv)	5249	5703	5671	5687	5672	5749	5688	5655	5641	5655	5637	132
	(v)	6109	6516	6547	6531	6531	6547	6531	6531	6531	6547	6492	128
	(vi)	1391	1375	1391	1391	1390	1390	1391	1391	1390	1390	1389	4
	(i)	187	172	172	171	172	203	172	187	188	203	182	12
	(ii)	328	328	328	344	344	328	328	328	359	328	334	10
	(iii)	10890	11296	11280	11297	11311	11296	11296	11281	11702	11827	11347	242
1900	(iv)	5859	6343	6281	6312	6281	6624	6281	6265	6265	8203	6471	602
	(v)	6796	7874	7312	7280	7093	7312	7265	7265	7250	7265	7271	249
	(vi)	1562	1578	1531	1562	1546	1546	1562	1547	1563	1562	1555	12
	(i)	203	187	204	203	188	203	203	203	219	219	203	9
	(ii)	391	375	359	359	374	375	375	391	406	359	376	14
	(iii)	12952	13577	13546	13514	13546	13484	12561	12562	12530	12546	13081	466
2000	(iv)	10249	6937	6937	6906	6922	6922	6921	7030	6953	6921	7269	993
	(v)	7562	8249	8015	8030	8046	8062	8062	8015	8093	8031	8016	164
	(vi)	1765	2031	1734	1734	1750	1750	1734	1765	1735	1735	1773	86