Towards Aspect Weaving Applications

Carine Courbis
University College London
Department of Computer Science
Adastral Park - Martlesham Heath
IP5 3RE, UK
c.courbis@cs.ucl.ac.uk

Anthony Finkelstein
University College London
Department of Computer Science
Gower Street, London
WC1E 6BT, UK
a.finkelstein@cs.ucl.ac.uk

ABSTRACT

Software must be adapted to accommodate new features in the context of changing requirements. In this paper, we illustrate how applications with aspect weaving capabilities can be easily and dynamically adapted with unforseen features. Aspects were used at three levels: in the context of semantic analysers, within a BPEL engine that orchestrates Web Services, and finally within BPEL processes themselves. Each level uses its own tailored domain-specific aspect language that is easier to manipulate than a general-purpose one (close to the programming language) and the pointcuts are independent from the implementation.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: General; D.2.10 [**Design**]: Methodologies

General Terms

Design, Languages

Keywords

Software adaptability, Aspect-Oriented Programming (AOP), weaver, domain-specific aspect language, Business Process Execution Language (BPEL)

1. INTRODUCTION

Applications need to be designed in such a way that enables them to easily adapt as new requirements emerge. Aspect weaving at the application level provides a mechanism for delivering such adaptability and extensibility. To this end, there is a need for Domain-Specific Aspect Languages that are simpler to use and understand than general-purpose ones (close to the implementation). To demonstrate this, we look at the use of aspects at three levels:

1. In the semantic analysers implemented with our development toolkit, named SmartTools [15];
2. Within an adaptable business process engine constructed on top of the toolkit semantic analysers;
3. And within extensible business processes built on top of that engine.

Our key contribution is to demonstrate an approach to achieving application adaptation and extension by providing integrated support for aspect weaving. We show the benefits of bringing this programming technology to the application level. This approach has been proven to work on applications that are of industrial significance (such as our example). The key ideas of Aspect-Oriented Programming (AOP) [10] are applied to different settings.

This paper is organised as follows. For each of the levels, the three next sections describe respectively *i)* what adaptability and extensibility are, *ii)* what aspect weaving is and how it delivers adaptability and extensibility, and finally *iii)* how domain-specific aspect languages suit our needs better than a general-purpose one. Section 5 provides architectural details. Then Section 6 presents the related work and we conclude the paper in Section 7.

2. ADAPTABILITY AND EXTENSIBILITY

Due to fierce competition between companies, the time-to-market for new products must be shortened and any new emerging feature or requirement quickly integrated. In this context, software should have the ability to be quickly modified to evolve and to accomodate new requirements while remaining easy to maintain.

Adaptability is an important non-functional requirement that needs to be taken into consideration very early in the life cycle when the software architecture is designed. It improves software reuse, but may impede performance. Hence, there is a trade-off between adaptability and performance. Software can be adapted on different axes [20], depending on its domain. Adaptations can be *i)* static or dynamic, *ii)* manual or automatic, and *iii)* proactive or retroactive. Static adaptations involve software code modifications whereas dynamic ones only modify its run-time behavior. Manual adaptations can be constrasted with automatic ones which are performed by the software itself when a certain condition is reached (adaptive software). Proactive adaptations are triggered before a change occurs in the environment whereas retroactive ones are performed after, as a consequence.

Extensibility can be seen as a static adaptation that extends the business logic of an application (the functional part). For example, an interpreter can be extended to deal with the addition of a new instruction into the language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.
Copyright 2005 ACM 1-58113-963-2/05/0002 ...$5.00.

Using a dynamic code loader, dynamic extensions by hot-swapping modules are possible. Extensibility can be classified into three forms [27]: the white-box, the gray-box, and the black-box. With white-box extensibility, the least restrictive and most flexible form, applications are modified or new code embedded into them. Two sub-categories exist [2]: the open-box and the glass-box. With open-box extensibility, changes are directly performed into the original source code, mixing the extension code and the original application code. With glass-box extensibility, the application code can be viewed but it is separated from the extension code. With black box extensibility, typically the easiest to use but the least flexible, the original code is encapsulated and explicitly contains the extensibility mechanism. Gray-box extensibility is a trade-off between the two other approaches as the original code cannot be examined but its binary can be extended.

We now describe what adaptability and extensibility mean in the context of our three examples: the semantic analysers built using the SmartTools toolkit, a business process engine more specifically a BPEL engine, and its business processes. The first two examples focus on static glass-box extensibility and manual dynamic adaptations, whereas the third one on dynamic extensibility in a multi-thread context.

2.1 Semantic analysers described in SmartTools

It is important that "systems infrastructure" software such as application servers, virtual machines, middleware, compilers, and operating systems are open and adaptable; otherwise no user-specific feature can be added after implementation time. Integrating new functionalities requires the redevelopment of the whole software.

To tackle this problem, SmartTools, a tool factory [9], provides a way to build extensible and adaptable semantic analysers dedicated to Domain-Specific Languages (DSLs). These analysers can be easily refined and updated when DSL definitions change, and their behaviors can be enhanced. For example, the core logic of an interpreter can be implemented as a semantic analyser and the visualisation of the environment as an adaptation to that analyser. The code of the interpreter is easier to write and to maintain as well as the code of the visualisation that is usually scattered. Another example of adaptation for this interpreter is constraint checking (design by contracts) on the environment values, prior to interpreting any instruction so as to ensure a safe execution. Also at development or maintenance stage, the interpreter can be enhanced with debugging information. As the architecture of the interpreter is flexible, the visualisation code or any other adaptation code is transparent to its core business logic. Adding and removing such adaptations, on the fly, are two important operations, particularly in the context of system infrastructures or legacy systems.

The designers of DSLs may not possess a deep knowlegde of computer science. The solution provided by SmartTools delivers semantic analysis frameworks that are easy to use and build on established techniques.

2.2 BPEL engine

With Web Services [23], there is a layer of abstraction above the components that makes it possible to orchestrate interactions between different services. The most well established orchestration technology for Web Services is BPEL (*Business Process Execution Language*) [3]. This XML-based language is rather small [12] but sufficient to handle variables with scopes, loops, conditional branches, synchronous and asynchronous communications, concurrent activities with correlated messages, transactions, and exceptions. With this language, a business process can be described by gluing different Web Services together thereby creating a new Web Service. This process description is interpreted by a BPEL engine.

In our view, the core logic of a BPEL engine should be minimal and compliant with the specifications but also easy to adapt and extend to cope with new requirements and features. As BPEL is an extensible language (that is new instructions can be used in a process description to cope with user-specific needs), its engine also needs to be extensible to integrate new behaviours for user-specific instructions. Examples of such extensions may be to introduce a new instruction to launch an executable or to replace a Web Service.

Enhancing the engine with orthogonal functionalities such as execution monitoring can be useful to manage the running process. Such functionalities should, however, be easy to disable, at any stage of the process execution, as they are themselves performance-inefficient. With such capabilities, process executions can be debugged during development stage, monitored, and even driven by agents at production stage. It is possible, for example, to embed, without modifying the engine implementation, a planner on the top of the latter. From events triggered by a monitor, this planner can take actions to avoid any disruption and to adjust the process. Such a tool can be useful particularly for long running processes.

Selecting Web Services is another example of a useful adaptation. Instead of choosing at design or deployment time which Web Service to use, the engine can choose one at runtime, in accordance with specified criteria and constraints, on the first occasion the service is invoked. There is also a need to be able to replace, at runtime, a Web Service that is slow, unresponsive, or no longer useful for the current iteration. In this way, the workflow can be adapted to improve performance or QoS (*Quality of Service*), or to avoid termination because there is no answer from one partner, or to use another similar service in a loop or on user demand. The substitution can only occur if the new Web Service is service-signature compliant (same WSDL description as there is no service adaptor) and if the service to be replaced is in a stable state (not in a transaction, and without an initialisation or one that does not impact on other partners).

A further useful adaptation is to execute local code for integration purposes between two service invocations, as proposed in BPELJ [4]. Converting message data into another format to prepare messages can be an example of the use of this capability. In this way, creating new Web Services to deal with business internal operations can be avoided.

2.3 BPEL processes

The existing service description languages and Web Service flow languages address business process dynamics and non-functional properties poorly. For example, in the current BPEL version, it is not possible, at runtime, to add on demand an unforeseen Web Service into the process, to replace Web Services, or to hot-fix processes. The process needs to be stopped to be extended. For long running processes, adapting a workflow by stopping it is not acceptable. An important example of such hot-fixes is the composition, on demand, of a new Web Service and thus the addition of its choreography interface into the process. We have been investigating a specific case in which services are used to support a large Grid-based computational chemistry application [13]. In this application, there is a need for *steering*, in other words, changing the end of the process dependent upon results identified in earlier stages.

BPEL processes need to have the ability to be extended to meet unforeseen post-deployment requirements and user needs. Hot-deploying Web Services requires the addition of new computational instructions into the process, whereas hot-fixing implies the replacement or deletion of some instructions.

3. ASPECT WEAVING

Objects have proven to be too small as units of reuse and inadequate to capture crosscutting concerns. New programming paradigms such as components and AOP have emerged to complement object-oriented programming. With aspects, crosscutting concerns can be cleanly modularized, making development, maintenance, and reuse easier. An aspect may contain structure refinements (for example, extending a class), different code fragments, and location descriptions to identify where to plug the code fragments in. In AspectJ [22], the most well-known AOP implementation for the Java programming language, these different pieces are respectively called inter-type declarations, advice, and pointcuts; the well-defined points in the program flow that can be selected by the pointcuts are called join points. The ultimate aim of AOP is to replicate the same execution as when the code of the business logic and aspects was tangled. This composition is performed by an aspect weaver, either by source code transformation or by use of hooks, at compilation or at execution time.

In this paper, we show that applications with aspect-weaving capabilities can be quickly adapted to meet new requirements. Aspects that can be plugged in on the fly (dynamic aspects) can be one way of solving the application adaptability issue. The first step to adding such capabilities is to identify the points in the application flow (the join points) where possible adaptations or extensions may be required in the future. Then, depending on the AOP implementations, the application architecture may need to be slightly modified to integrate hooks at these points and some logic to manage the plugging in and removal of aspects. The application itself is the aspect weaver. We now detail what the join points are for our three examples.

3.1 Semantic analysers described in Smart-Tools

The possible points to adapt semantic analysers are at each instruction treatment. With wrappers around these treatments, the base code can be easily adapted in different ways. At execution time, aspects can be manually plugged in or removed, before or after these join points, on a particular language instruction (operator, for example the `while` operator), on a specific set of operators (DSL type), or on all the operators. Composing different aspects at a same join point is done in a first-plugged-in first-executed ordering policy. A priority ordering policy might be introduced in the weaver in case of the around weaving use (that indicates the substitution of the advice execution for the base code one). In order to have powerful adaptations, the execution context (that is the current node and the treatment arguments) is provided by the weaver to the advice.

Analysers, as explained in the previous section, also need to be extended as a new instruction can be added in the DSL or their semantic refined. We have chosen to encapsulate each analyser in an independent module with a method per DSL operator. With this design, they are easy to extend, in a glass-box manner, by inheritance and method overriding.

3.2 BPEL engine

As the BPEL engine is an interpreter based on a SmartTools semantic analyser, it has the same join points and extensibility mechanisms as those described in the previous sub-section. The difference lies in the fact that these points need to be identified in a finer grain manner. Not only BPEL operators or type names can be used in this order but also attributes, or location in the document (such as the second `invoke` in a loop).

As we wanted not only a BPEL engine that is standard compliant, but one that also contains extra features such as a BPEL aspect weaver, we have chosen to implement the latter as semantic analyser adaptations, that is aspects plugged onto the engine. In this way, the extra features are transparent to the engine and can be plugged in or out at any time during the execution.

3.3 BPEL processes

The pointcuts are the same as those of the BPEL engine aspects but the weaving is different as these aspects have as a goal to extend (to hot-fix) BPEL documents. Weaving such an aspect implies transforming the document and the BPEL engine environment (for example, to add a new variable), at execution time, in a multi-thread context as each activity in a `flow` is interpreted in a different thread. These transformations can only be applied to the processes at precise points, under certain conditions and when all the threads are suspended, to ensure the stability of the system. For example, replacing a BPEL sequence can only occur if the engine has not started interpreting it.

The dynamic aspect technology is our solution to address dynamic composition of Web Services: the choreography interface can be seen as pieces of advice and where to weave them as pointcuts. Composition is an important example of such hot-fixes.

4. DOMAIN-SPECIFIC ASPECT LANGUAGES

The most popular aspect language is AspectJ [22]. It is a general-purpose aspect language for Java. Because of this generality, the language is low-level [25]. Its pointcut language is rich as there exists many join points to identify, making its use more difficult, especially for non-programming experts. Typically, the full range of pointcuts and inter-type declarations is not used when aspects are specified for an application.

We believe that each class of applications should have its own Domain-Specific Language (DSL) [6] to specify its aspects. In this way, the aspect language is tailored to the application, gaining in conciseness and easiness of use. Developing a DSL [11], particularly when you have the domain knowledge, is not that complicated. Tools or formalisms to support DSL development exist, most notably XML Schema to describe the structure of the language, XML parsers, and the DOM API to manipulate the documents.

The main questions to focus on when designing a new Domain-Specific Aspect Language (DSAL) are the following:

- How the join points should be identified and with what granularity ?
- What structural introductions into the base level should be enabled (for example, adding new fields to a class) ?
- What information should be available in the advice to enable powerful adaptations, recognizing that there is a trade-off in which such adaptations can destabilize the system ?

The issue of how inter-aspect dependencies should be handled when different aspects are composed (is an ordering needed?) is related to the weaver design. To illustrate, we now present the DSALs of our three examples.

4.1 Semantic analysers described in SmartTools

As described above, we wanted to be able to plug in Java code fragments just before or after semantic treatments and to identify these points either with a DSL operator name or a DSL type name. With AspectJ, the pointcuts would be too low level as the identification of the semantic treatments would be based on the method names and parameter type names. In case where the analyser method names are changed or a new operator is added into a type, these pointcuts would need to be updated. An independance (abstraction) from the analyser implementation was therefore required. Enabling the introduction of new members (fields, methods, or constructors) into the analysers has proven not to be useful.

We have chosen to design an aspect as a Java class that implements an interface with two methods, `before` and `after` (see Figure 1) as it can be easier to understand. These methods are two pieces of advice that can be executed respectively before and after the semantic treatment of an operator. Thereby, aspects can be extended by inheritance and benefited from any object-oriented programming advantage. When an aspect is deployed, information on which operator or type it should be plugged into is provided. The three methods to deploy it are given in Figure 2 as well as the equivalent AspectJ pointcuts[1] as comparison. The first-plugged-in first-executed ordering policy is applied if more than one aspect is plugged in at the same join point.

4.2 BPEL engine

We have built a BPEL engine as an interpretor based on a SmartTools semantic analyser. Therefore, it has the same join points but the requirements for its DSAL are slightly different: the pointcut language should be finer grain as well as independant from the implementation, and aspects may need to share information, though this means that they are interdependent. Only BPEL documents should be the focus of the users of this DSAL not the implementation of the engine which should remain a black box. To have more

[1]We are not AspectJ experts. Any suggestion to improve the pointcuts is welcome.

```
package uk.ac.ucl.cs.bpel.aspect.engineAspect;

import java.util.ArrayList;
import fr.smarttools.core.tree.UntypedNode;
import fr.smarttools.core.tree.visitor.aspect.Aspect;
import uk.ac.ucl.cs.bpel.aspect.engineAspect.AspectRef;

public class EngineAspectManager implements Aspect {
 private UntypedNode curNode = null;

 public void before(Type context, Object[] vParams)
       throws VisitorException {
  curNode = (UntypedNode) vParams[0];
  executeAdviceIfAny("getAspectsBefore", curNode);
 }

 public void after(Type context, Object[] vParams)
       throws VisitorException {
  curNode = (UntypedNode) vParams[0];
  executeAdviceIfAny("getAspectsAfter", curNode);
 }

 private void executeAdviceIfAny(String methN,
                                   UntypedNode node) {
  ArrayList aspectRefs = executeMethod(methN, node);
  if (aspectRefs != null) {
   if (aspectRefs.size() > 0) { // advice to execute
    AspectRef aspRef;
    String aspN;
    String adviceN;
    for (int i=0; i<aspectRefs.size(); i++) {
     aspRef =((AspectRef)aspectRefs.get(i));
     aspN = aspRef.getAspectName();
     if (aspectDB.isPresent(aspN)) {
      adviceN = aspRef.getMethodName();
      executeMethod(adviceN, aspectDB.getAspect(aspN));
     } else {
      // do nothing. The aspect has been unplugged.
     }
 }}}} ...
}
```

Figure 1: Example of a SmartTools aspect: the BPEL aspect weaver aspect

```
addAspect(aspectObject);
pointcut p1(fr.smarttools.core.tree.ast.UntypedNode n,
           uk.ac.ucl.cs.bpel.engine.BPELEnv env) :
 target(BPELEngine) &&
 args(n, env) &&
 execution(* execute(uk.ac.ucl.cs.bpel.ast.*Node,
                     uk.ac.ucl.cs.bpel.engine.BPELEnv);

addAspectOnOperator("invoke", aspectObject);
pointcut p2(uk.ac.ucl.cs.bpel.ast.InvokeNode n,
           uk.ac.ucl.cs.bpel.engine.BPELEnv env) :
 execution(* BPELEngine.execute(n, env));

addAspectOnType("Activity", aspectObject);
pointcut p3(uk.ac.ucl.cs.bpel.ast.ActivityType n,
           uk.ac.ucl.cs.bpel.engine.BPELEnv env) :
 target(BPELEngine) &&
 args(n, env) &&
( execution(* execute(uk.ac.ucl.cs.bpel.ast.InvokeNode,
                     uk.ac.ucl.cs.bpel.engine.BPELEnv))
|| execution(* execute(uk.ac.ucl.cs.bpel.ast.WhileNode,
                     uk.ac.ucl.cs.bpel.engine.BPELEnv))
|| similar pointcut for all the operators that
     belong to this type (14 operators)
);
```

Figure 2: The three SmartTools methods to deploy an aspect (in bold) and their equivalent AspectJ pointcuts underneath

powerful adaptations, direct access to the BPEL engine environment as well as the current node should also be enabled from the advice.

We have decomposed a BPEL engine aspect into two parts: the specifications of the pointcuts and a Java class that contains the pieces of advice. To avoid learning a new language for the pointcuts, we have chosen XPath, a language specialized for addressing parts of an XML document (a BPEL process is an XML document). Therefore only static values of the documents can be used to identify the join points; the dynamic ones can filter the code execution in the advice bodies. XPath is a well-known powerful standard for which widely available tools exist that our AOP implementation benefits from. The only restriction we have imposed for implementation purposes is to only address BPEL nodes not attributes (though the filtering on attributes is enabled). The first part of an aspect (Figure 3) contains its name (a fully qualified Java class name), and at least one pointcut to specify the nodes to select (an XPath expression) and the advice name (a Java method) to execute before or after their interpretation. For example, the last pointcut of Figure 3 indicates that the method `shipping` (the advice) of the `uk.ac.ucl.cs.test.EngineAspectExample` aspect should be executed just before the interpretation of each `invoke` instruction that has `shippingPT` with a `lns` namespace prefix as value for its `portType` attribute. The second part (Figure 4) is the corresponding Java class that encapsulates the advice code. From any advice method, it is possible to call methods to get the node that is currently interpreted (`getCurrentNode`), the BPEL environment, and any other aspect plugged in that is identified by its name.

```
<aspect name="uk.ac.ucl.cs.test.EngineAspectExample">
 <after  where="//:process" methodName="process"/>
 <before where="//:partnerLink" methodName="partnerL"/>
 <after  where="//:variable" methodName="variable"/>
 <before where="//:invoke[@portType='lns:shippingPT']"
         methodName="shipping"/>
</aspect>
```

Figure 3: Pointcut specifications of an engine aspect

```
package uk.ac.ucl.cs.test;
import uk.ac.ucl.cs.bpel.ast.VariableNode;

public class EngineAspectExample extends
 uk.ac.ucl.cs.bpel.aspect.engine.EngineAspect {

 public void variable() {
  VariableNode node = (VariableNode) getCurrentNode();
  System.out.println("Var decl=" + node.getNameAttr());
 }
 public void process() {}
 public void partnerL() {}
 public void shipping() {}
}
```

Figure 4: Advice code of an engine aspect

4.3 BPEL processes

For this DSAL, the pointcuts are the same as those in the BPEL engine, but there is a requirement to introduce or remove members (such as variables, partners, catch exception handlers, etc.) in/from the process. The advice bodies are also different as they are BPEL-based. As the process fixes (aspects) can be plugged in on the fly, the execution environment needs to be updated with the new members that are involved in the process. The possible modifications that can be performed on a process are inserting, replacing, or removing BPEL instructions. By contrast to the other DSALs, these aspects do not need to be removable.

A process aspect (Figure 5) is decomposed into three parts: *i)* the declarations of member addition or removal (variables, catches, catch all, compensate handlers, event handlers, partners, partner links, correlation sets), *ii)* the actions to perform on the process (insert, replace, or delete) and at which locations (also XPath expressions), and finally *iii)* the BPEL instruction fragments (advice bodies) to add into the process. For example, when the aspect of Figure 5 is plugged into an existing business process, a new variable, `var1`, is added to the scope `scope1` and the instructions (the `while` statement) of the advice `advice1` are appended just after the second `invoke` instruction of the process if it is possible.

```
Workflow aspect ProcessAspectExample
Members {
   add <variable name="var1" messageType="orderType"/>
       in scope1
}
Pointcuts {
   after "//:invoke[2]" insert advice1
}

Advices {
  advice1
  <while condition="bpws:getVariableData(order) > 100">
  ...
  </while>
}
```

Figure 5: Example of a process aspect

5. ARCHITECTURES

This section details the architecture and implementation of each example.

5.1 Semantic analysers described in SmartTools

Each SmartTools semantic analyser is based on the visitor design pattern [7, 14]. It contains one `visit` method per language operator and traverses, from top to bottom, the ASTs (*Abstract Syntax Trees*) that represent the programs or documents that are being analysed. Because of the operator class implementation, these trees are not only strictly typed to meet the pattern requirements but are also based on the DOM API, enabling XPath selections of their nodes, which is useful for the implementation of our aspect languages.

These classes are generated by the toolkit from the language structure definition (for example, an XML Schema) as well as a default tree visitor that can be extended by inheritance and `visit` method overriding to develop a specific semantic analyser. To ease the writing and the maintenance of the analysers, SmartTools offers additional features: *i)* the `visit` methods and traversal can be configured, *ii)* the redirection methods usually named *accept* are hidden, and finally *iii)* the analysers can also be adapted dynamically through aspects. By these means, only appropriate nodes are visited (this ability can be useful on large trees), type castings on the parameters as well as on the return values are avoided, and the given method names are more meaningful. Each concern can be modularized instead of being scattered in the different methods of the analyser. All these benefits are derived from the intensive use of generative programming.

Typically, redirection methods are useful in the Java programming language as it does not support the late-binding on dynamic types of method parameters. With these methods, the right method according to the dynamic types of the parameters is executed. SmartTools handles the redirection methods internally and hides the underlying complexity from developers. For example (Figure 6), the `execute` method on the `Activity` son is directly called instead of `node.getActivityNode().accept(env)`. There is no `accept` method in each operator class. As the method signatures can be configured, these classes would have been modified each time a new analyser would have been defined to integrate the corresponding `accept` methods. Instead each recursive call is forwarded to a central method that dispatches according to the operator id of the current node to the appropriate analyser method. To avoid using reflection, a method is generated for each analyser that sorts all the "`visit`" method calls in a switch in function of the operator ids.

```
package uk.ac.ucl.cs.bpel.engine;

import uk.ac.ucl.cs.bpel.BPELEnv;
import uk.ac.ucl.cs.bpel.visitors.TravBPELEngVisitor;
import uk.ac.ucl.cs.bpel.ast.*;
import fr.smarttools.core.tree.visitor.VisException;

public class BPELEngine extends TravBPELEngVisitor {
 public Object execute(WhileNode node, BPELEnv env)
         throws VisException {
  executeStandardElements(node, env);
  String condition = node.getConditionAttr();
  if (testCondition(condition, env) == true) { // true
   execute(node.getActivityNode(), env); // activity
   execute(node, env); // recursive while execution
  }
  return null;
 }
 ...
}
```

Figure 6: Code snippet of a visitor: the BPEL engine

Because of this central method, it was relatively straightforward to enhance the semantic analysers with a specific AOP. By integrating weaving mechanisms before and after the dispatch method, the semantic analysers have been made adaptable.

5.2 The BPEL engine

The BPEL engine is an example of SmartTools semantic analysers. Glass-box extension of the default visitor, `TravBPELEngVisitor`, generated from the BPEL structure definition and the visitor configuration is shown in Figure 6. The core business logic of the engine is contained in this analyser, except for the management of the namespaces which is implemented as an aspect. Namespaces can be defined anywhere in documents (attribute `xmlns`), however they can only be used inside the node that defined them and its nested nodes. Instead of scattering code in each visit method, to collect the namespaces and to remove them from the environment at the end of the operator interpretation, we have chosen to modularize it in an aspect, plugged on all the operators. The management of source and target links that drive the process execution can be implemented as an aspect that can also be plugged on all the operators that belong to the `Activity` type.

All extra features, not compliant with the standards, are adaptations (aspects), transparent to the engine core. The environment visualisation as well as the engine aspect manager are examples of such adaptations. Having a specific AOP on the top of the engine enables other adaptations such as monitoring the processes, preparing the Web Service messages, or checking constraints (Figure 7).

The engine aspect weaver is itself an aspect (Figure 1). When an engine aspect is plugged, it is registered on the system and the different nodes of the AST selected by the XPath expressions of its pointcuts are annotated with the aspect name and the name of the advice to execute. Before and after interpreting any instruction, the weaver checks if there is any annotation. If one exists and if the aspect is still registered (plugged), a call by reflection is performed to execute the advice. Unweaving an aspect only means removing it from the registry.

5.3 BPEL processes

The workflow aspect manager is also an adaptation, independent from the engine. When a process aspect is plugged in, it suspends the engine at some stable points and performs the transformations at the nodes identified by the XPath expressions if possible. The workflow aspect manager also needs to get access to the data environment to add or remove members. By propagation, the engine aspects already plugged in are applied to any new BPEL instruction added by insertion or replacement.

6. RELATED WORK

Constructing applications that are easy to adapt and extend is a long-standing goal in software engineering. Solutions to offer greater modularisation, variation, and configurability to the code have been proposed such as generative programming, meta-programming and reflection, aspect-oriented software development, and model-driven architecture. Given the scope of this work, it is impossible to review everything. We have therefore chosen to make reference to only the most immediate relevant works.

Using aspects for dynamic adaptations has been proposed in [26]. In this work, a non-adaptive distributed conferencing application is transformed into one that is adapt-ready. The adaptation infrastructure - wrappers and dynamic management code of the adaptations (conditions and actions) - is added to the application core, using AspectJ. At runtime, adaptations can be loaded or removed, and the application behaves as a code weaver with the actions to execute when the corresponding conditions are satisfied. Embedding code weaving mechanisms inside the application core brings flexibility. We believe that with the new capability of annotating Java programs (see JSR 175 [18]), more and more applications will be code weavers.

Many projects such as [1, 16, 21] are interested in apply-

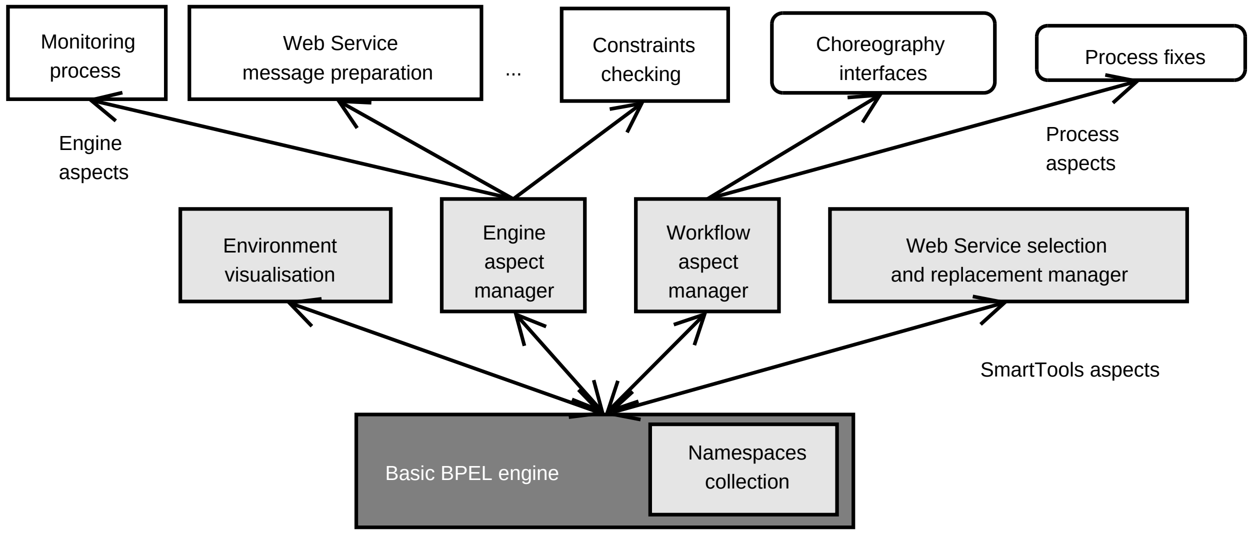

Figure 7: The basic BPEL engine with its different aspects

ing dynamic aspects into component architectures to have more flexible and adaptable applications. For example, the JAsCo infrastructure [21] enables the execution of component based applications, which can be dynamically adapted by aspects. The components (Java Beans) when loaded into the infrastructure are modified to insert traps at every public method which invoke the aspect weaver when they are called. A management layer based on the JAsCo aspects [24] was developed to monitor and adapt Web Service applications.

The JAsCo infrastructure has also its own aspect language, a Java language extension, specific for the domain: the components. Krzysztof Czarnecki and Ulrich Eisenecker advocate [5] the use of a separate language or a language extension specialized to the problem instead of a conventional library approach. The earliest aspect languages, COOL and RIDL, were domain-specific, respectively for synchronisation and distribution. The semantic of domain-specific aspect languages is clear as they are restricted and designed for the problem. Other interesting work that advocates the use of Domain-Specific Aspect Languages includes XAspects [19].

A toolkit for making aspect languages for object-oriented legacy software is proposed in [8], based on a language-independent weaver as the back-end. This generic weaver also uses AST transformations to plug the advice code into the legacy software.

The main problem with weaving new code at runtime is to ensure the stability of the system. Despite the dynamic changes, code consistency and structural correctness should be maintained. Using a restricted set of join points where the changes can occur may help to check if the changes (aspect) can be performed safely. Formal models may be used such as ADEPT [17] in case of dynamic structural changes in workflows.

7. CONCLUSION

Adaptability and extensibility are requirements that should be taken into consideration during the application architecture design. The approach outlined in this paper is to prepare applications to cope with dynamic adaptations by inserting aspect weaving mechanisms. This approach has been shown to work. For example, the core of our BPEL engine, a SmartTools semantic analyser, has been enhanced with unforeseen adaptations - the BPEL engine aspect weaver and the BPEL process aspect weaver - in a transparent manner. Aspect tools should themselves use aspects for their own development. With these adaptations, the engine itself can be adapted and the processes extended without stopping their interpretation. By these means, Web Service hot-deployments and workflow hot-fixes are possible. The benefits of these adaptation mechanisms outweigh, we believe, the potential performance impact. Furthermore, in our case of applications based on Web Services, this potential performance impact associated with the weaving mechanisms is not comparable with that due to remote service invocations.

Designing aspect languages specific for the applications simplifies the use of aspects, especially for non-programming experts, as these languages are tailored specifically for a specific domain and are higher level (hide the complexity) than general-purpose ones. For instance, our aspects (pointcuts) are independent from the implementation and therefore do not need to be updated if the BPEL engine implementation is changed.

Additional adaptations and extensions to the BPEL engine may be required as its implementation is not finished. We plan, for example, to be able to select Web Services at runtime instead of at deployment time and even replace them if they are no longer compliant with the selection policy. We want to refine the way process transformations are performed to ensure the stability of the system. Locations where these transformations can occur need to be identified and consistency tests performed.

8. ACKNOWLEDGEMENTS

This work is supported by the Generative Software Development project funded by BT Group. Anthony Finkelstein is grateful for support from an IBM faculty partnership award. The authors want to thank Didier Parigot from INRIA Sophia-Antipolis for his work on SmartTools and

Patrik Mihailescu from BT for his comments about this article as well as Anne Liret.

9. REFERENCES

[1] ObAsCo (Objects, Aspects, and Components) Research Group. http://www.emn.fr/x-info/obasco/.

[2] E. E. Allen. Designing extensible applications. Technical report, IBM developerWorks, September 2001. http://www-106.ibm.com/developerworks/java/library/j-diag0925/.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, and I. Trickovic. Business Process Execution Language for Web Services version 1.1. Technical report, BEA, IBM, Microsoft, SAP, Siebel Systems, May 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[4] M. Blow, Y. Goland, M. Kloppmann, F. Leymann, G. Pfau, D. Roller, and M. Rowley. *BPELJ: BPEL for Java*. BEA and IBM, March 2004. white paper, http://www-106.ibm.com/developerworks/java/library/j-diag0925/.

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programming*, chapter 8. Addison-Wesley, 2000. ISBN 0-210-30977-7.

[6] A. V. Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–35, June 2000. http://www.cwi.nl/~arie/papers/dslbib.pdf.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Pub Co, January 1995. ISBN 0201633612.

[8] J. Gray and S. Roychoudhury. A Tecgnique for Constructing Aspect Weavers Using a Program Transformation Engine. In K. Lieberherr, editor, *the Third International Conference on Aspect-Oriented Software Development*, pages 36–45, Lancaster, UK, March 2004. http://www.cis.uab.edu/gray/Pubs/aosd-2004.pdf.

[9] J. Greenfield and K. Short. Software Factories. Assembling Applications with Patterns, Models, Frameworks and Tools. In *the second OOPSLA workshop on Generative Techniques in the context of Model-Driven Architecture*, Anaheim, USA, October 2003. http://www.softmetaware.com/oopsla2003/greenfield.pdf.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220-242, Jyväskylä, Finland, June 1997. http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf.

[11] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *Submitted to ACM Computing Surveys*, 2004. http://marcel.uni-mb.si/marjan/ACM-ComputingSurveys.pdf.

[12] N. Mukhi. Reference guide for creating BPEL4WS documents. Technical report, IBM, November 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-bpws4jed/.

[13] H. Nowell, B. Butchart, D. S. Coombes, S. L. Price, W. Emmerich, and C. R. A. Catlow. Increasing the Scope for Polymorph Prediction using e-Science. In *the 2004 UK E-Science All Hands Meeting (AHM)*, pages 967–971, Nottingham, UK, September 2004. UK Engineering and Physical Science Research Council. http://www.cs.ucl.ac.uk/staff/w.emmerich/publications/AHM04/final_nowell.pdf.

[14] J. Palsberg and C. B. Jay. The Essence of the Visitor Pattern. In *Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998. http://www.cs.ucla.edu/~palsberg/paper/compsac98.pdf.

[15] D. Parigot, C. Courbis, P. Degenne, A. Fau, C. Pasquier, J. Fillon, C. Held, and I. Attali. Aspect and XML-oriented Semantic Framework Generator: SmartTools. In M. van den Brand and R. Lämmel, editors, *ETAPS'2002, LDTA workshop*, volume 65 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, Grenoble, France, April 2002. Elsevier Science. http://www.elsevier.nl/gej-ng/31/29/23/117/52/33/65.3.009.pdf.

[16] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns: Third International Conference, Reflection'01*, volume 2192 of *LNCS*, pages 1–24, Kyoto, Japan, September 2001. http://jac.aopsys.com/papers/reflection.ps.

[17] M. Reichert and P. Dadam. ADEPTflex - Supporting Dynamic Changes of Workflow without Loosing Control. *Intelligent Information Systems special issue one Workflow Management Systems*, 10(2):93–129, March/April 1998. http://www.informatik.uni-ulm.de/dbis/01/staff/reichert/papers/journals/reda98c.pdf.

[18] Sun Microsystems. JSR 175: A Metadata Facility for the Java Programming Language, 2004. http://www.jcp.org/en/jsr/detail?id=175.

[19] M. Shonle, K. Lieberherr, and A. Shah. XAspects: an Extensible System for Domain-Specific Aspect Languages. In *OOPSLA 2003 Domain-Driven Development Track*, Anaheim, USA, October 2003. http://www.ccs.neu.edu/research/demeter/biblio/XAspects.html.

[20] N. Subramanian and L. Chung. Architecture - Driven Embedded Systems Adaptation for Supporting Vocabulary Evolution. In *Int. Symposium on Principles of Software Evolution (ISPSE2000)*, pages 144–153, Kanazawa, Japon, November 2000. http://www.utdallas.edu/~chung/ftp/ISPSE00.doc.

[21] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, Boston, USA, March 2003. http://ssel.vub.ac.be/jasco/papers/aosd2003.pdf.

[22] The AspectJ Team. *The AspectJ Programming Guide*, AspectJ 1.2 edition. http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html.

[23] A. Tsalgatidou and T. Pilioura. An Overview of Standards and Related Technology in Web Services. *Distributed and Parallel Databases, special issue on e-services*, 12:135–162, 2002. Kluwer Academic Publishers, http://www.di.uoa.gr/~afrodite/PADP2002.pdf.

[24] B. Verheecke and M. Cibrán. Dynamic Aspects for Web Service Management. In *Dynamic Aspects Workshop of the AOSD Conference*, pages 146–152, Lancaster, UK, March 2004. RIACS Technical report 04.01. http://aosd.net/2004/workshops/daw/Proc-2004-Dynamic-Aspects.pdf.

[25] K. D. Volder, J. Brichau, K. Mens, and T. D'Hondt. Logic Meta Programming, a Framework for Domain-Specific Aspect Languages. http://www.cs.ubc.ca/~kdvolder/binaries/cacm-aop-paper.pdf, 2001.

[26] Z. Yang, K. S. Betty Cheng, J. Sowell, M. Sadjadi, and P. McKinley. An Aspect-Oriented Approach to Dynamic Adaptation. In D. Garlan, J. Kramer, and A. Wolf, editors, *Proceedings of the ACM SIGSOFT Workshop On Self-healing Systems (WOSS02)*, Charleston, USA, November 2002. http://www.cse.msu.edu/~yangzhe1/research/pub/woss-02.pdf.

[27] M. Zenger. *Programming language abstractions for extensible software components*. PhD thesis, Ecole polytechnique fédérale de Lausanne, Lausanne, Switzerland, 2004. http://ahdoc.epfl.ch/EPFL/theses/2004/2930/EPFL_TH2930.pdf.