Adaptive Programming in JAsCo

Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, Viviane Jonckers

Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium

{wvdperre,dsuvee,bverheecke,mcibran}@vub.ac.be , viviane@info.vub.ac.be

ABSTRACT

In this paper we propose an extension to JAsCo for supporting Adaptive Programming in a Component-Based Software Development context. JAsCo is an aspect-oriented programming language targeted at Component-Based Software Development and allows encapsulating crosscutting concerns using highly reusable aspect beans. Adaptive Programming on the other hand, allows capturing crosscutting concerns by structure-shy adaptive visitors. We propose to implement an adaptive visitor as a regular JAsCo aspect bean. As such, the reusability of an adaptive visitor is improved because the same visitor can be reused within different component contexts. We introduce JAsCo traversal connectors to deploy adaptive visitors, implemented as JAsCo aspect beans, upon a concrete component traversal. In addition, these traversal connectors allow to explicitly specify how the behavior of several adaptive visitors, instantiated onto the same component traversal, needs to be combined by making use of the JAsCo precedence and combination strategies. A prototype implementation of the JAsCo Adaptive Programming extension, which employs the DJ library, is available. As a proof of concept, we present an extended case study in the context of the Web Service Management Layer (WSML) project. In this case study, a set of visitors implemented in JAsCo is reused to accomplish multiple tasks.

Keywords

Aspect-Oriented Software Development, Component-Based Software Development, Adaptive Programming, JAsCo.

1. INTRODUCTION

Aspect-Oriented Software Development (AOSD) is a novel software development paradigm that aims at improving the separation of concerns in Object-Oriented Software Development (OOSD) [2]. Typical aspect-oriented approaches, such as AspectJ [5], introduce an explicit aspect construct to capture concerns that crosscut the regular decomposition of the system. Afterwards, the aspect is weaved together with the base application and as such the aspect behavior is triggered at all points the aspect is applicable upon. Adaptive Programming (AP) [14] aims at providing support for a very different kind of crosscutting concerns than the ones tackled by typical aspect-oriented approaches. When an operation involves a set of cooperating classes, one can either localize this operation in one class or split the operation over the set of associated classes. Localizing the operation in one class causes hard-coded information about the structural relationships between these classes and is as such a violation of the well-known Law of Demeter [11]. The other alternative, namely distributing the operation over the set of involved classes, conforms to the Law of Demeter, but causes the logic of the desired behavior to be spread over different classes making evolution very difficult. To capture an operation that involves several cooperating classes, AP introduces adaptive visitors, which allow visiting the objects contained within an application without explicitly describing the structural relationships among these objects. Traversal strategies are responsible for specifying the abstract visiting process for an adaptive visitor. As such, AP allows separating the collaboration concerns (WhenAndWhatToDo), the traversal concerns (WhereToGo) and the object structure concerns (classGraph and objectGraph) [10].

Although AP is originally conceived for OOSD, its ideas are also applicable to Component-Based Software Development (CBSD). The main goal of CBSD is achieving highly reusable, independently deployable components [24]. The current adaptive programming realizations however, such as DJ [18], DemeterJ [13] or DAJ [12,22], do not straightforwardly support the specification of adaptive visitors that are sufficiently reusable to be employed in a component-based context. Recently, the JAsCo aspect-oriented programming language has been proposed to capture crosscutting concerns in a component-based context [23]. JAsCo allows specifying aspect beans that are completely independent of concrete component types and APIs. Aspect beans are deployed onto a concrete context using a separate connector construct. As such, employing a JAsCo aspect bean as a reusable adaptive visitor makes AP more suitable to be employed in a component-based context.

Another drawback of current AP approaches is that there is only limited support for describing combinations of adaptive visitors. The ability to manage combinations of visitors explicitly is however crucial for realizing independent visitors. JAsCo supports an expressive mechanism for combining independent aspects through *combination strategies*. Therefore, introducing a combination strategy concept and as such enhancing the expressivity of the combination mechanism for adaptive visitors, also contributes to AP.

As a proof of concept of JAsCoAP, we present an extended case study in the context of the Web Service Management Layer (WSML) project [26,27]. The WSML allows, among other management actions, the automatic selection and dynamic integration of the most appropriate web services, depending on non-functional properties such as price, reliability and response time. We show how adaptive visitors, implemented as regular JAsCo aspect beans, are ideal to cleanly encapsulate the visiting process of the web service property data-structure in order to facilitate the automatic web service selection process.

In the next section, we show how the ideas of AP and JAsCo can be combined in order to make AP more appropriate to be used in a component-based context. Section 3 introduces the key ideas and concepts of the JAsCo aspect-oriented language. Section 4 illustrates how adaptive visitors can be implemented by means of JAsCo aspect beans in order to improve their reusability. In addition, the traversal connector language is introduced. Section 5 illustrates how the behavior of several adaptive visitors can be combined by employing JAsCo combination strategies. Section 6 elaborates on the implementation of the JAsCoAP extension and section 7 compares JAsCoAP with current state-of-the-art research. Finally, section 8 discusses the WSML case study and afterwards we present our conclusions.

2. PROBLEM STATEMENT/MOTIVATION

One of the main principles of CBSD is to keep one component independent deployable from other concrete components [15]. As such, a component should never explicitly rely onto other specific components in order to execute its behavior [24]. Translating this requirement towards AP means that an adaptive visitor should be completely independent from the components it visits. By the very nature of AP, adaptive visitors are already structure-shy of the application at hand. However, adaptive visitors typically still refer to specific components and APIs, rendering a visitor not as reusable as required by CBSD.

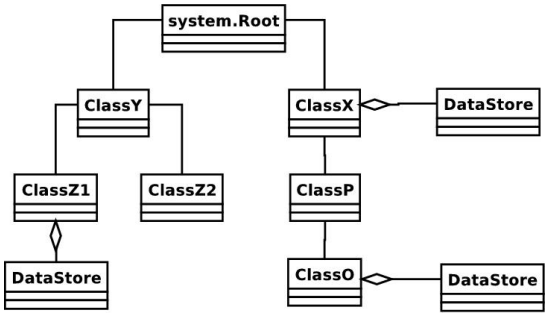

Figure 1: Example class diagram

Consider for instance the class diagram of Figure 1, onto which an incremental backup concern should be deployed that stores the data contained in each `DataStore` object. In order to backup the contents of all `DataStore` objects reachable from an instance of the `System.Root` class, dedicated methods need to be added to almost every class. When the application evolves and new classes are added to the system, these dedicated methods need to be adapted and the backup logic needs to be refactored in order to deal with the new class structure.

When employing AP, this backup concern is encapsulated in a structure-shy manner using an adaptive visitor. Figure 2 illustrates the `DataStorePersistence` adaptive visitor, implemented using DJ [18], which is a Java library that supports AP. Using DJ, adaptive visitors are implemented as plain Java classes. The `DataStorePersistence` visitor allows capturing an incremental backup of the data contained within each `DataStore` object. In this case, the behavior of the visitor is implemented using a before advice (lines 5-14). If the state of the `DataStore` object changed since it was last visited (line 6), the data contained within the `DataStore` object is serialized to file (lines 7-12). When this adaptive visitor is applied onto an application, the entire data structure of the application is traversed. Whenever an object of type `DataStore` is encountered, and its state changed since the last backup, its data is written to file before it is visited. This visitor allows carrying out an effective, incremental backup of all `DataStore` objects contained within an application.

```
1  class DataStorePersistence extends Visitor {
2
3    int i = 0;
4
5    public void before(DataStore store) {
6      if(changedPV(store)) {
7        FileOutputStream fw = new
8            FileOutputStream("state"+i++);
9        ObjectOutStream writer= new
10           ObjectOutputStream(fw);
11       writer.writeObject(store.getData());
12       writer.close();
13     }
14   }
15   public boolean changedPV(DataStore s) {
16     //true if changed since last visit
17   }
18 }
```

Figure 2: DataStorePersistence Adaptive Visitor in DJ that allows serializing each visited data store on file.

Figure 3 demonstrates how this adaptive visitor is deployed in order to backup all `DataStore` objects starting from the root system object of an application (lines 3-4). Notice that, because of the use of an adaptive visitor, the backup method does not need to hard-code the relations among the components it visits. As a result, the `DataStorePersistence` visitor remains applicable even when additional `DataStore` objects are added to the system or when the structural relationships between the system components are changed.

```
1  void backup(system.Root mySystemRoot) {
2    ClassGraph cg = new ClassGraph("system");
3    Strategy sg = new
4      Strategy("from system.Root to *");
5    TraversalGraph tg = new
6      TraversalGraph(sg, cg);
7    tg.traverse(mySystemRoot,
8       new DataStorePersistence());
9  }
```

Figure 3: Instantiating the DataStorePersistence adaptive visitor in order to traverse the system for taking a backup of the state of the application (using DJ).

2.1 Improving the reusability of AP visitors

Although the `DataStorePersistence` visitor of Figure 2 is independent from the structural relationships within the application, it still hard-codes the `DataStore` type and the `getData` method within its implementation. As a result, it is not

possible to reuse the `DataStorePersistence` visitor within a different component context than the one that was foreseen.

By making use of design patterns such as Abstract Factory [3], it is possible to realize more reusable, component-independent adaptive visitors. This solution nevertheless requires writing a lot of infrastructural code and merely provides an ad-hoc solution to prevent context-dependent visitors. The novel Java 1.5 generics feature provides a more straightforward solution to abstract over types and is already extensively used in the implementation of generic containers. Figure 4 illustrates the refactored implementation of the `DataStorePersistence` adaptive visitor by employing Java 1.5 generics.

```
1  class Persistence<T> extends Visitor {
2    public void before(T store) {
3      ...
4        writer.writeObject(store.toString());
5      ...
6    }
7  }
```

Figure 4: Generic adaptive persistence visitor.

The implementation of this generic adaptive is quite similar to the one illustrated in Figure 2, except for the `store.getData()` statement (Figure 2, line 11) that is renamed into the `store.toString()` statement (Figure 4, line 4). This refactoring is mandatory, as the formal type parameter `T` is only able to understand methods declared by type `Object`. As a result, the expressiveness of the internal behavior of the `Persistence` adaptive visitor is quite restricted, as only those methods understood by type `Object` can be called on `store`. To resolve this problem, the Java generics feature introduces *bounded* formal type parameters. Figure 5 illustrates this concept by specifying that the concrete parameter type, which replaces the formal parameter type `T`, needs to implement the `IDataStore` interface or one of its declared subinterfaces. As such, it remains possible to call the `getData` method, specified by the `IDataStore` interface, on object `store`. Although the use of bounded parameters improves the expressiveness of a visitor, it still requires the objects to be visited to implement a dedicated interface. As a result, it is not possible to deploy and reuse this adaptive visitor upon off-the-shelf third-party components. Notice that this limitation is not a deficiency in the concrete realization of generics in Java 1.5, but is rather a more profound problem with generics in object-oriented languages.

```
1  class Persistence<T extends IDataStore>
2    extends Visitor {
3      public void before(T store) {
4        ...
5          writer.writeObject(store.getData());
6        ...
7      }
8 }
```

Figure 5: Generic adaptive persistence visitor using bounded formal type parameters.

2.2 Combining JAsCo and AP

JAsCo aspect beans are abstract and reusable entities which do not rely onto specific component types and APIs. Therefore, in order to increase the reusability and achieve context-independent and expressive adaptive visitors, we propose to implement adaptive visitors as JAsCo aspect beans. As such, integrating this aspect independence idea into AP contributes to achieving a higher reusability and flexibility for adaptive visitors in a component-based context.

Another area where the JAsCo ideas are able to contribute to the AP research, consists of describing expressive combinations among aspects in order to provide a solution to the feature interaction phenomenon [19]. Suppose for instance that in addition to the persistence behavior, a log needs to be maintained for every object that is saved to file. Using the current AP implementations, this can only be achieved by appending some logging code to the `DataStorePersistence` adaptive visitor itself. However, doing so causes the logging concern to be tangled with the backup concern. A tangled "backup-logging" concern can nevertheless be avoided if a strong combination mechanism is provided that allows specifying that the `TraceVisitor` (standard logging visitor in DJ library) and the `DataStorePersistence` visitor visit the same traversal simultaneously in such a way that the behavior of `TraceVisitor` is only triggered whenever the `DataStorePersistence` visitor saves a visited object to file. As such, the JAsCo ideas concerning expressive combinations of aspects using precedence and combination strategies also contribute to AP.

3. INTRODUCTION TO JASCO

JAsCo is a dynamic AOP approach originally aiming at combining the ideas of Aspect-Oriented and Component-Based Software Development. The JAsCo language is an aspect-oriented extension for Java that intends to stay as close as possible to the original Java syntax and concepts and introduces only two additional entities: *aspect beans* and *connectors*. An aspect bean is responsible for capturing crosscutting behavior in a context-independent manner and a connector allows instantiating aspect beans onto a concrete component context. The next paragraphs shortly present the JAsCo aspect bean and connector language. For more detailed information about JAsCo, the interested reader is referred to [23].

```
1  class DataStorePersistence {
2
3    hook Backup {
4
5      int i = 0;
6
7      Backup(triggeringmethod(..args)) {
8        execute(triggeringmethod);
9      }
10
11     isApplicable() {
12       //true when changed since last visit
13     }
14
15     before() {
16       FileOutputStream fw =
17         new FileOutputStream ("state"+i++);
18       ObjectOutputStream writer = new
19         ObjectOutputStream(fw);
20       writer.writeObject(getDataMethod());
21       writer.close();
22     }
23
24     refinable Object getDataMethod();
25   }
26 }
```

Figure 6: JAsCo DataStorePersistence aspect bean that implements a reusable backup concern.

An aspect bean is an extended version of a regular Java bean [21] and is specified independent of concrete components and APIs, making it a highly reusable entity. An aspect bean usually contains one or more logically related hooks that describe the crosscutting behavior itself. Figure 6 illustrates the implementation of the `DataStorePersistence` aspect bean. The behavior of this aspect bean is similar to the behavior of the `DataStorePersistence` visitor of Figure 2. The aspect bean contains one hook, the `Backup` hook (lines 3-26), which implements the crosscutting backup behavior. The constructor of a hook specifies a kind of abstract pointcut and takes one or more abstract method parameters as input. These abstract method parameters are bound to concrete methods in a connector. In this case, the constructor (lines 7-9) specifies that the behavior of the hook is only triggered when the concrete method(s) bound to the `triggeringmethod` abstract method parameter are executed (line 8).

The `isApplicable` method is able to specify an additional triggering condition for the hook and employs the expressiveness of full Java. In this case the `isApplicable` method (lines 11-13) returns true if the state of the object, on which `triggingmethod` is executed, changed since it was last visited. This can either be checked by using the *equals* method, supported by all Java objects, if the previous backup is still available or by employing a *refinable* method.

JAsCo supports *before*, *around* and *after* advices for specifying the behavior that needs to be performed when the hook triggers. The semantics of these advices are identical to the AspectJ counterparts. The `DataStorePersistence` aspect bean implements one before advice (lines 15-22) that serializes the visited object to file.

Hooks are able to postpone the implementation of certain behavior in order to remain type independent. For example, in order to fetch the concrete data from the executing object, the *refinable* method `getDataMethod` is used (line 24). As such, the hook does not hard-code how the data is fetched and remains reusable. On the contrary, the visitor of Figure 2 relies on the `getData` method of the `DataStore` type, making it less reusable. Refinable methods are implemented using type-specific refinements for the hooks. Figure 7 illustrates an example refinement of the `Backup` hook for objects of class `DataStore`. Here, the implementation of the `getDataMethod` returns the object that is retrieved by executing the `getData` method of class `DataStore` (lines 3-6). When a refinable method is executed at run-time, late binding is applied, i.e. the most specific refinement for the current object is searched for and executed. If no appropriate refinement has been defined, the refinement needs to be implemented inline when instantiating an aspect bean in the connector. When both an inline refinement and an external refinement are available, the inline refinement has precedence.

```
1  refining DataStorePersistence.Backup for
2    DataStore {
3      public Object getDataMethod() {
4        DataStore store = thisJoinPointObject;
5        return store.getData();
6      }
7  }
```

Figure 7: Specific refinement for objects of class DataStore.

In order to deploy the `DataStorePersistence` aspect bean within an application, a JAsCo connector is employed. Figure 8 illustrates the deployment of this aspect bean upon each *setter* method defined within the `DataStore` class. This is achieved by instantiating the `Backup` hook (lines 2-4) on each method defined within the `DataStore` class whose name starts with `set`. This binds the `triggeringmethod` abstract method parameter from the `Backup` hook constructor to each *setter* method of the `DataStore` class. Hence, whenever the state of a `DataStore` object is altered using a method whose name starts with `set`, the `before` behavior of the `Backup` hook is triggered and the backup action is performed.

```
1  connector PersistenceConnector {
2    DataStorePersistence.Backup hook = new
3      DataStorePersistence.Backup(
4        * DataStore.set*(*));
5  }
```

Figure 8: JAsCo Backup Connector.

4. ASPECT BEANS AS AP VISITORS

An adaptive visitor is similar to a set of related advices as it groups several before, after and around methods together that need to be executed whenever a corresponding component type is visited. Therefore, it seems natural to employ a regular JAsCo aspect bean as a reusable and loosely coupled adaptive visitor. As such, aspect beans are able to be deployed in both a traditional and a traversal oriented context, increasing their reusability even more. The first part of this section illustrates how aspect beans can be deployed as being adaptive visitors. The second part discusses how the aspect bean model is mapped upon the adaptive visitor model.

4.1 Deploy Aspect beans as adaptive visitors

In order to deploy an aspect bean as an adaptive visitor, a new kind of connector is introduced, namely a *traversal connector*. A traversal connector instantiates one or more hooks as adaptive visitors onto a specified traversal strategy.

```
1  traversalconnector BackupTraversal
2    ("from system.Root to *") {
3
4    DataStorePersistence.Backup hook = new
5      DataStorePersistence.Backup(
6          visiting DataStore);
7
8    hook.before();
9
10 }
```

Figure 9: JAsCo BackupTraversal traversal connector.

Figure 9 illustrates an example traversal connector that instantiates the `DataStorePersistence` aspect bean (lines 4-6) upon the "`from system.Root to *`" traversal strategy (lines 1-2). The `visiting` keyword allows declaring on which specific type of objects, encountered during the traversal, the behavior of the hook needs to be triggered. In this particular case, the behavior of the `Backup` hook is triggered whenever `DataStore` objects are encountered. The traversal connector also specifies that whenever the hook is triggered, the before advice has to be executed (line 8). To sum up, the `BackupTraversal` traversal connector has the following semantics: the object structure of an application is traversed as specified by the traversal strategy "`from system.Root to *`" and the `before` advice of the

Backup hook is triggered each time a DataStore object is encountered.

By declaring which advices to execute, it is possible to explicitly specify precedence of the advices in case multiple hooks are instantiated in the same traversal connector. These precedence strategies are instance-based, which allows changing the precedence over different instantiations of the same hook.

The major difference between mainstream aspect-oriented approaches, such as AspectJ, and AP is the way their crosscutting behavior is invoked. The behavior of traditional aspects is invoked implicitly whenever the current joinpoint matches the pointcut specification of the aspects. Traversal strategies however, need to be invoked explicitly in order to start the traversal. Figure 10 illustrates how the traversal specified in the BackupTraversal connector is explicitly invoked (line 5). In this particular case, the starting point for the traversal is the object SystemRoot, which is an instance of the system.Root class.

```
1  public void backup(system.Root systemRoot) {
2
3    BackupTraversal myBackup =
4      BackupTraversal.getInstance();
5    myBackup.traverse(systemRoot);
6 }
```

Figure 10: Invoking the JAsCo BackupTraversal connector.

4.2 Deployment discussion

One might wonder which specific methods are bound to an abstract method parameter of a hook when an aspect bean is deployed as an adaptive visitor. In case of the BackupTraversal connector illustrated in Figure 9, the triggeringmethod parameter of the backup hook is bound to the *"*visiting DataStore*"* declaration, which is not a concrete method signature. The resulting effect of this *"*visiting DataStore*"* declaration is that all DataStore objects encountered during the traversal are visited by the DataStorePersistence aspect bean. It is possible to perceive this visiting behavior as a method execution and it is even implemented as such when the DJ library is employed. As a result, the method bound to the triggeringmethod parameter corresponds to the implicit method that implements the visiting behavior. This allows an aspect bean to effectively manage its visiting process in a hook. If a hook implements an around method, the hook itself might decide to continue or stop the visiting process by invoking or not invoking the proceed method. This concept is illustrated in the SearchBean aspect bean, from which the implementation is shown in Figure 11.

The SearchBean aspect bean is able to search for a specific object within an application and builds a path of all objects that are visited while the application structure is traversed (Note that this path is not necessarily the shortest path). The BuildPath hook implements a constructor (lines 12-14), which takes one abstract method parameter, visitingmethod as input. The hook implements an around advice (lines 16-21) that is responsible for building up a list of visited nodes. Whenever the object to search for has been reached, the traversal stops, as the method bound to the abstract method parameter visitingmethod is not invoked any longer. In the other case, the current visited object is added to the list of visited nodes (line 18) and the traversal continues by explicitly invoking the proceed method (lines 19). Notice that thisJoinPointObject, which in a traditional aspect bean context refers to the object on which the method was called, now refers to the object that is currently being visited. This mapping can easily be understood: if the traversal behavior itself is perceived as a method, the visiting of an object can be perceived as the execution of that method on that object.

Figure 12 illustrates a traversal connector that instantiates the BuildPath hook on all classes in the system using a wildcard (lines 4-5). The resulting traversal starts at an instance of the class system.Root and visits all reachable objects from that instance onwards (lines 1-2). When the object that needs to be found is located, the traversal is halted.

```
1  class SearchBean {
2
3    Object target;
4    List visitednodes = new List();
5
6    List getResultingPath() {
7      return visitednodes;
8    }
9
10   hook BuildPath {
11
12     BuildPath(visitingmethod(..args)) {
13       execute(visitingmethod);
14     }
15
16     around() {
17       if(!thisJoinPointObject.equals(target)) {
18         visitednodes.add(thisJoinPointObject);
19         proceed();
20       }
21     }
22   }
23 }
```

Figure 11: Search aspect bean that allows building a path of objects visited in order to reach a specific object.

Note that this point of view is compatible with the JoinPoint Model (JPM) for adaptive programming that Wu et al. [30] describe and identify as parallel to the JPM of traditional aspect-oriented approaches such as AspectJ. Additionally, they propose employing around advices with non-void return types as a means to express functional visitors. These visitors provide a natural and convenient way of expressing computation along the traversal over recursive object structures, similar to traditional recursive functions. While we expect that their extension to DJ can also be applied to JAsCoAP in a straightforward manner, the feasibility of advices with non-void return types that apply in both traditional and traversal-oriented contexts remains to be investigated.

```
1  traversalconnector SearchTraversal(
2    "from system.Root to *") {
3
4    SearchBean.BuildPath builder = new
5      SearchBean.BuildPath(visiting *);
6
7    builder.around();
8 }
```

Figure 12: Instantiating the BuildPath hook on all the classes within the system in a JAsCo traversal connector.

4.3 Traversal language

The traversal strategies employed in JAsCo traversal connectors are an extension of the traversal language supported by the DJ library [18]. Figure 13 provides a set of examples which

illustrates the use of more advanced keywords that allow specifying sophisticated traversal strategies.

The *bypassing* keyword allows to denote classes that may not be visited during the traversal. For example, in the traversal strategy at line 1, an instance of the class C can not be visited in order to traverse starting from an instance of class A and reaching an instance of class B. In other words, when considering the class hierarchy as a class graph, all paths from class A to B that contain the class C are not visited. An exception is thrown if such a traversal is not possible.

The *via* keyword denotes the opposite of bypassing, namely that a certain type must be visited during the traversal. The traversal strategy at line 2 specifies that at least one object of type C or a subtype of C has to be visited when traversing from A to B. Notice the + operator to specify all subtypes of a type.

It is also possible to specify a concrete edge in the class graph (denoting a method or field) that has to be followed or bypassed. The traversal strategy at line 3 denotes a traversal starting at an object of class A that has to pass through this field z of class B. The field z can be of any type as specified by the wildcard.

```
1  from A bypassing C to B
2  from A via C+ to B
3  from A via ->B,z,* to C
```

Figure 13: More sophisticated JAsCo Traversal Strategies.

4.4 Pointcut language

In addition to the `execute` pointcut designator, the JAsCo aspect bean language also supports other frequently employed pointcut designators, such as `cflow` and `withincode`. Similar to `execute`, these designators also have a semantics in the adaptive programming context. The `cflow` keyword allows making sure that the advices of the hook are only executed when an object of a certain type has been visited before reaching the current object. The `cflow` construct captures the concrete object of the declared type and thus allows for reflection about the history of the visiting process. The `withincode` keyword makes sure that the advices of the hook are only executed when an object of a specific type has been encountered just before reaching the current object. In other words, when considering the class hierarchy as a class graph, the visiting process has followed an edge from an object of the type denoted by the `withincode` keyword until the current object. Similar to `cflow`, the concrete object of `withincode` is captured and can be used for reflection.

Notice that the traversal strategy keyword `via` has a different semantics than the `cflow` keyword in a hook's constructor. The `via` keyword allows selecting specific paths that the adaptive visitors have to follow while the `cflow` keyword delimits specific nodes in the traversal graph where the hook is triggered. Furthermore, the `cflow` and `withincode` keywords allows capturing the relevant nodes for reflection.

5. COMBINING ADAPTIVE VISITORS

As already mentioned in the problem statement of section 2, current implementations of AP only offer limited support for combining several adaptive visitors in a single traversal simultaneously. These approaches are only able to statically define precedence of a combination of visitors. Aspect beans are also able to statically specify precedence on a per advice type basis. In a JAsCo traversal connector, precedence strategies allow to specify the sequence in which the advices of the instantiated hooks need to be executed. In order to support more expressive combinations of visitors, JAsCoAP employs the concept of *combination strategies*.

5.1 JAsCo Combination Strategies

A combination strategy is able to influence the aspectual behavior at a certain traversal joinpoint by filtering the list of applicable hooks at that traversal joinpoint. An applicable hook at a traversal joinpoint is a hook that triggers on that specific type of node in the classgraph associated with the traversal joinpoint and where the `isApplicable` condition evaluates to true. Per default, all applicable hooks defined in a traversal connector are triggered. A combination strategy however, is capable to influence this set of hooks and as such able to limit the advices that are executed.

Combination strategies are regular Java classes that implement the `CombinationStrategy` interface. This interface, illustrated in Figure 14, specifies one method named `validateCombinations` (line 2) that takes as input a list of applicable hooks at the current traversal joinpoint. This method filters this list of applicable hooks in order to influence the aspectual behavior of the traversal joinpoint at hand. A combination strategy can be observed as a *function* that takes as input a set of hooks and which has as result another set of hooks. As such, a combination strategy always returns the same set of hooks given a particular input set. This allows optimizing combination strategies performance wise by caching their result and as such avoiding the execution of a combination strategy for every encountered traversal joinpoint. This optimization is valuable, as it is very likely that the same set of hooks is encountered frequently during a traversal because of the crosscutting nature of the modularized concern.

```
1  interface CombinationStrategy {
2    HookList validateCombinations(HookList l);
3  }
```

Figure 14: Combination Strategy Interface.

It is also possible to implement *reflective combination strategies* which are not functions. The main difference with a regular combination strategy is twofold: 1) they are always executed for each encountered traversal joinpoint and 2) they are able to access reflective information about the current encountered traversal joinpoint. Figure 15 illustrates the reflective combination strategy interface. Notice the additional `TraversalJoinPoint` argument (line 3) for the `validateCombinations` method, which can be used for performing reflection about the current traversal joinpoint at hand (node in the classgraph). The `TraversalJoinPoint` class provides an extensive reflective API for obtaining the currently visited object, the type of this object, the employed traversal strategy, …

```
1  interface ReflectiveCombinationStrategy {
2    HookList validateCombinations(Hooklist l,
3      TraversalJoinPoint jp);
4  }
```

Figure 15: Reflective Combination Strategy Interface.

Figure 16 illustrates an example combination strategy, named `Twins`. The strategy's constructor takes two hooks as input (lines 5-8). The `validateCombinations` method (lines 10-14) specifies that the behavior of the first hook needs to be triggered in order to trigger the behavior of the second hook. This is

implemented by checking whether the list of applicable hooks contains `hook1` (line 11). When this is not the case, `hook2` is removed from the list of applicable hooks as well (line 12). Notice that the relationship between both input hooks is asymmetric: when the behavior of the second hook is not triggered, it is still possible to trigger the behavior of the first hook.

```
1  class Twins implements CombinationStrategy {
2
3    Hook hook1, hook2;
4
5    Twins(Hook hook1,Hook hook2) {
6        this.hook1 = hook1;
7        this.hook2 = hook2;
8    }
9
10   HookList validateCombinations(HookList l) {
11     if(!l.contains(hook1))
12        l.remove(hook2);
13     return l;
14   }
15 }
```

Figure 16: Twins combination strategy ensuring that hook2 is only triggered when hook1 is triggered.

The `Twin` combination strategy can now be employed to implement the desired combined logging-backup behavior as outlined in section 2. This combined behavior has as purpose creating a log that contains the list of all objects that are effectively stored. Remember that the `Backup` hook of Figure 6 takes an incremental backup. Its behavior is only triggered when the visited object is altered since it was last visited. As such, the behavior of the logging concern can only be triggered if the behavior of the backup hook is triggered as well. This behavior is realized by employing the traversal connector illustrated in Figure 17.

The `Backup` hook is instantiated as before (line 4). Imagine that also an `OutputLogging` hook is available, which is instantiated on visiting all types (line 5-6). The `Twins` combination strategy is instantiated using the `backup` and `logging` hook instances as arguments. Afterwards, the `Twins` combination strategy is added to the traversal connector by employing the `addCombinationStrategy` keyword (line 8-9). The resulting visiting behavior is the following: both hooks visit objects along the described traversal strategy "`from System.Root to *`" and for objects of

- types different from `DataStore`: the behavior of the backup hook is not triggered and as such the combination strategy makes sure that the behavior of the logging hook is not triggered as well.
- type `DataStore` with changed state: the behavior of the backup hook is triggered. The logging hook is kept in the list of applicable hooks and its behavior is triggered as well.
- type `DataStore` with same state: the behavior of the backup hook is not triggered. The combination strategy removes the logging hook from the list of applicable hooks and as such the logging behavior is not triggered.

It is also possible to define multiple combination strategies in the same traversal connector. All defined combination strategies are merged using an approach similar to UNIX pipes. The sequence in which they are specified corresponds to the order in which they are employed in the pipeline. The first combination strategy receives the list of applicable hooks and filters them. The second combination strategy then receives this filtered list of hooks as input, performs its own filtering logic and passes the result on to the next combination strategy and so on. The hook list returned by the last combination strategy is then the list of hooks that have to be triggered at the current traversal joinpoint.

```
1  traversalconnector BackupLDetect(
2    "from System.Root to *") {
3
4      DataStorePersistence.BackupHook backup = …
5      Logging.OutputLogging logging = new
6        Logging.ReportLogging(visiting *);
7
8      addCombinationStrategy(new
9        Twins(backup,logging));
10 }
```

Figure 17: Traversal connector that instantiates the Twins combination strategy on the backup and logging hooks.

5.2 Reusing JAsCo Combination Strategies

A common issue in AP applications is avoiding visiting the same object instance more than once. In current AP realizations, it is the responsibility of the adaptive visitor itself to verify whether a certain node in the classgraph has already been visited and to stop the visiting process accordingly. As a consequence, this detection code is scattered over all adaptive visitors that require it.

JAsCoAP provides a more elegant solution by reusing the `Twins` combination strategy and a generic loop detection aspect bean. The `LoopDetect` aspect bean, illustrated in Figure 18, makes sure that it is only applicable when the current node in the classgraph was not visited before. To this end, the `LoopDetection` hook implements an `isApplicable` method (line 11-18), which checks whether the currently visited object has already been encountered (line 12). If this is the case, the `isApplicable` method returns false (line 13). When the currently visited object has not been encountered before, it is added to the set of visited objects (line 15) and returns true (line 16). As such, this hook only triggers when the currently visited object was not encountered before.

```
1  class LoopDetect {
2
3    hook LoopDetection {
4
5      Set visited = new TreeSet();
6
7      LoopDetection(visitingmethod(..args)) {
8        execute(visitingmethod);
9      }
10
11     isApplicable() {
12       if(visited.contains(thisJoinPointObject))
13         return false; //loop detected!
14       else {
15         visited.add(thisJoinPointObject);
16         return true;
17       }
18     }
19   }
20 }
```

Figure 18: Loop Detection aspect bean.

In order to make sure that the persistence aspect bean avoids visiting the same objects more than once, the `LoopDetect` aspect bean is combined with the `DataPersistence` aspect bean in such a way that the behavior of the `DataPersistence` aspect bean is only triggered if the behavior of the `LoopDetect` aspect bean is triggered. This is achieved by reusing the `Twins` combination strategy of Figure 16. Figure 19 illustrates the instantiation of this `Twins` combination strategy with an instance of the loop detection and persistence hook (lines 4-5). As a result, the persistence hook does not visit the same object more than once. The JAsCoAP solution allows for a better separation of concerns as other aspect beans remain completely oblivious of whether this loop detection mechanism is deployed or not.

```
1  traversalconnector BackupLDetect(
2    "from System.Root to *") {
3
4      DataStorePersistence.Backup backup = …
5      LoopDetect.LoopDetection detect = …
6
7      addCombinationStrategy(new
8        Twins(detect,backup));
9  }
```

Figure 19: Traversal connector that instantiates the Twins combination strategy for avoiding visiting the same object more than once.

5.3 Reflective Combination Strategies

The above example shows how a combination of several hooks can be described using a combination strategy. Combination strategies are similar to *combinators* employed in Strategic Programming (SP) [29]. The main difference is that SP combinators are specified declaratively, which has several advantages regarding understandability, automatic optimizations and analysis. In JAsCo however, we explicitly choose for an imperative approach as this allows employing the full expressiveness of Java. Reflective combination strategies for instance, are able to alter the properties of the encountered aspects depending on the concrete combination of hooks. Properties can be changed by invoking the appropriate method on the hooks themselves. It is also possible to change properties depending on the traversal joinpoint at hand. In a reflective combination strategy, the traversal joinpoint information is available through the `TraversalJoinPoint` argument of the `validateCombinations` method. Because a combination strategy has full control over the list of applicable hooks, it is even possible to change the precedence of the hooks dynamically.

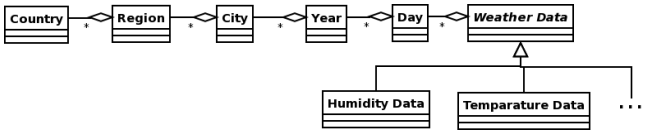

Figure 20: Class Graph of a weather information system.

Consider the class graph of Figure 20 that is part of a weather information system. The application contains various weather related information for several cities throughout the world categorized by dates. In order to generate a report containing useful statistical information, such as average temperature, a visitor is employed. Depending on the city or region, different weather related information can be more important. For example, for cities in the Sahara region, rainfall information is essential while for cities in a touristic coastal region in Spain, average water temperature is more interesting. It would be appealing to have the most interesting data for each region at the top of the report. As such, the sequence of visitors that log weather related data to file needs to be altered dynamically depending on the interest of the region, which is in this case related to the type of visited data.

This kind of visiting behavior is cleanly captured using a reflective combination strategy, as it has full control over the returned list of hooks and is also able to access the currently executed traversal joinpoint. Figure 21 illustrates this `DynamicPrecedence` combination strategy. The constructor of this hook (lines 8-12) receives one hook as input. This hook is an instance of the `ContextFetcher` hook that is able to retain selected objects during the traversal. The combination strategy contains a field defining comparators for specific context objects (line 4). These comparators are used to reorder the list of applicable hooks depending on the object stored by the context fetcher hook (lines 21-29).

```
1  class DynamicPrecedence implements
2    ReflectiveCombinationStrategy {
3
4      Map<Object,Comparator> comparators;
5      ContextFetcher context;
6
7      DynamicPrecedence(ContextFetcher aContext){
8         context = aContext;
9      }
10
11     HookList validateCombinations(HookList
12       List, TraversalJoinPoint jp) {
13         Object context = context.getContext();
14         list = reorder(context,list);
15         return list;
16     }
17
18     HookList reorder(Object context,
19       HookList list) {
20       Comparator c = comparators.get(context);
21       if(c==null)
22         throw new IllegalArgumentException(
23            "illegal context object "+context);
24       Collections.sort(list,c);
25       return list;
26     }
27 }
```

Figure 21: DynamicPrecedence Combination Strategy.

The traversal connector shown in Figure 22 illustrates how to apply this dynamic precedence combination strategy. The context fetcher hook is instantiated on objects of type `Region` (lines 4-5). As such, the `ContextFetcher` hook stores the last `Region` instance encountered during the traversal. Two report generating visitors are instantiated, one for generating water temperature reports (line 7) and one for generating rainfall reports (line 8). The context fetcher hook is passed as input for the dynamic precedence combination strategy (lines 10-11). Furthermore, the map of comparators of the dynamic precedence combination strategy is initialized (line 12). As such, the context fetcher hook remembers the last visited region and depending on this region, the hooks are dynamically reordered.

The dynamic precedence combination strategy illustrated in Figure 21 does not rely on the specific component types encountered during the traversal, nor does it rely on specific types

of hooks apart from the necessary context fetcher hook. As such, it is reusable for all class graphs where a set of visiting hooks needs to be dynamically reordered depending on the specific objects encountered during the traversal. A similar strategy could also be implemented for dynamically reordering hooks depending on for example the encountered traversal joinpoint.

The solution for fetching previously encountered objects during the traversal is currently somewhat ad-hoc as a new aspect bean has to be implemented for retaining a specific type of object during the traversal. When a more complicated history needs to be maintained, another aspect bean should be manually implemented. The DJ library supports a cleaner solution for capturing context information through a special kind of visitor, called `ContextVisitor`. Subclasses of a `ContextVisitor` automatically gather the visited objects and support an extensive API for querying the visiting context. Integrating a similar feature in JAsCo is anticipated in the future.

```
1  traversalconnector GenWeatherReport (
2    "from Country via Region to WeatherData+) {
3
4      Context.ContextFetcher context = new
5       Context.ContextFetcher(visiting Region);
6
7      ReportGen.Temperature watertemp = …
8      ReportGen.Humidity rainfall = …
9
10     DynamicPrecedence prec = new
11         DynamicPrecedence(context);
12     prec.comparators = …
13
14     addCombinationStrategy(prec);
15 }
```

Figure 22: traversal connector for generating reports using a dynamic precedence combination strategy.

6. IMPLEMENTATION

In order to implement JAsCo traversal connectors, a proof-of-concept implementation is provided which is made available through the regular JAsCo distribution [4]. The *compileTraversal* tool is supplied which allows compiling a traversal connector into its binary Java class representation. Each traversal connector is first translated into a Java class which implements the traversal connector logic. Afterwards it is compiled using the standard Java compiler. Compiled traversal connectors employ the DJ library in order to perform the traversals themselves. Each generated Java traversal connector class contains an inner class that implements the DJ visitor interface. This inner class takes care of executing the correct hooks whenever an object instance is visited. The applicable hooks are computed by dynamically invoking the `isApplicable` method for each hook. Afterwards, all applicable hooks are filtered by applying each combination strategy instantiated in the traversal connector. Finally, the advices of the hooks that are left are triggered using the precedence sequence defined in the traversal connector. When invoking the traversal connector, the DJ adaptive visitor is instantiated onto the traversal strategy specified in the traversal connector.

In addition to the command-line *compileTraversal* tool, an IDE for JAsCoAP is provided as plugin of the Eclipse framework. The IDE provides a dedicated editor with syntax coloring for both aspect beans and traversal connectors and also includes support for creating aspect beans and traversal connectors by employing intuitive visual wizards. A traversal connector can be automatically generated using the visual wizard. The only exception is the implementation of the refinable methods that has to be provided later on. The IDE employs the built-in eclipse Java compiler for compiling the generated Java files which has several advantages regarding compilation speed and consistency of errors.

7. DISCUSSION OF RELATED WORK

DAJ is an extension of AspectJ that aims at providing Adaptive Programming support for Java using an extension of the AspectJ language [12,22]. As such, DAJ pursues the same goal as JAsCo, namely realizing a unified language for both Adaptive and Aspect-Oriented programming. In DAJ, traversal strategies are declared in an AspectJ aspect. In addition, adaptive visitors are applied to a traversal strategy in an aspect declaration. Adaptive Visitors are however still implemented as a regular java class. As such, it is not possible to reuse an aspect as both a traditional AspectJ aspect and an adaptive visitor. DAJ also offers weaker support for specifying aspect combinations in comparison to the expressive combination strategies offered by JAsCo. On the other hand, because DAJ is a statically weaved approach, it is possible to analyze whether certain traversals are possible or not. In addition, the concrete traversal graph can already be computed beforehand. The JAsCo implementation employs run-time reflection to compute and execute the traversals, which induces a significantly larger run-time overhead.

XAspects is an original plugin based language for supporting domain specific concern languages [20]. The XAspects system delegates the concrete compilation of the domain specific aspects to the correct plugins. XAspects is built on top of AspectJ. As such, domain specific languages are translated into AspectJ code during compilation. XAspects already contains a plugin to allow Adaptive Programming and aspect-oriented programming using AspectJ. As such, XAspects also combines regular aspect-oriented programming with Adaptive Programming. In comparison to JAsCo however, XAspects suffers from the same limitations as DAJ, namely adaptive visitors are not reusable as regular aspects and it provides a weaker aspect combination mechanism. However, introducing more powerful aspect compositions is possible using the extensible plugin mechanism.

Strategic programming (SP) [7] is a generic programming idiom for processing compound data, such as parse trees of programming languages. It was initiated in the context of term rewriting (using Stratego [28]), but has been transposed to other programming paradigms such as functional programming (based on Strafunski [9]) and object-oriented programming [29] (based on JJTraveler/JJForester[6]). SP allows programmer-definable generic traversal schemes (strategies) that, unlike AP, offer full control over the traversal. The definition of these strategies relies on traversal primitives and combinators that take other strategies as arguments. As such, building on a small suite of combinators, a wide and expressive variety of traversals can be defined in a declarative way. In [8] for example, an AP domain specific language is built on top of SP. An actual traversal is then synthesized by passing problem-specific basic computations as arguments to the appropriate traversal scheme. This corresponds to the separation of the traversal specifications and code behavior in AP, although SP employs the same type of entity for both of these functions. This specific property of SP additionally allows

employing strategy combinators on problem-specific basic computations. As such, these strategy combinators serve as reusable combination operators that, in the case of object-oriented incarnations of SP [29], bear a large resemblance to the combination strategies of JAsCoAP. For example, [8] features a visitor for cycle detection very similar to the Loop Detection aspect bean of Figure 18, which can also be generally used to prevent node processors from visiting the same node twice, without scattering the code of the processor with this concern.

Note however that JAsCoAP uses a very different approach than unifying these functions in one type of entity. In SP, both the navigation specification (traversal strategy in AP) and the visitor composition are combined, which makes it difficult to evolve them separately. On the contrary, in JAsCoAP, the visitor behavior, the visitor composition and the navigation logic are independent, reusable and first-class entities, which allows for a cleaner separation of concerns. Additionally, a remaining weakness of SP compared to approaches tailored for object structures such as AP, is that it does not employ static meta-information on the class graph to limit the traversal to those branches in the given object graph that can possibly lead to a target node [8]. Furthermore, SP does not allow to check compatibility of a traversal strategy with a given class graph, as AP does allow.

Adaptive Plug-and-Play Components (APPC) [16] aims at making AP not only structure-shy, but also independent from the concrete class graph it visits. To this end, every APPC describes its visiting behavior in terms of an interface class graph. This interface defines a set of participants and their mutual relationships. In addition, for each participant an explicit set of expected operations is provided. An APPC is instantiated by mapping each abstract participant upon a concrete class. As such, it is possible to deploy the behavior of an APPC upon various component contexts. This solution however, still expects the concrete components to match a particular operation interface, and is as such subjected to the same limitations as adaptive visitors implemented using Java 1.5 generics (section 2.1).

Caesar [17] describes aspects in terms of Aspect Collaboration Interfaces (ACI). Each concrete aspect implements the required methods specified by a corresponding ACI. Aspect bindings are responsible for connecting these aspect implementations to different concrete component contexts. One of the major contributions of the Caesar approach is the introduction of *aspectual polymorphism*, a notion similar to the late binding concept found in object-oriented languages. Aspectual polymorphism allows deferring the decision of which concrete binding to employ until run-time. In this perspective, it is similar to JAsCo refinable methods and their corresponding refinements. In JAsCo, the concrete refinements to use are automatically resolved at run-time while in Caesar, it is the responsibility of the programmer to manually select the concrete binding between the available alternatives.

8. CASE STUDY: THE WEBSERVICE MANAGEMENT LAYER (WSML)

We present an extensive case study where JAsCoAP is employed in the context of the Web Service Management Layer (WSML). The first part of this section shortly introduces the architecture and functionalities offered by the WSML. The second part of this section illustrates how adaptive visitors, implemented as regular JAsCo aspect beans, can be employed to facilitate the automatic web service selection process.

8.1 Overview of the WSML

Web Services are a new and increasingly popular standard of the World Wide Web Consortium. The promise of Web Services is to enable a distributed environment in which any number of applications, or application components, can interoperate seamlessly among and between organizations in a platform-neutral, language-neutral fashion [1]. Current approaches for the integration of web services [25] however, hard-wire web services into the client applications themselves, affecting adaptability and reusability. To enable the development of more flexible and robust applications, the Web Services Management Layer (WSML) is proposed [26,27], which is placed in between the application and the web services as a transparent layer. The WSML is developed in the context of the MOSAIC[1] project and a prototype implementation is already available. The current implementation of the WSML consists of more than 400 Java classes and about 22000 LOC. Figure 23 shows a schematic overview of the architecture and the functionalities offered by the WSML.

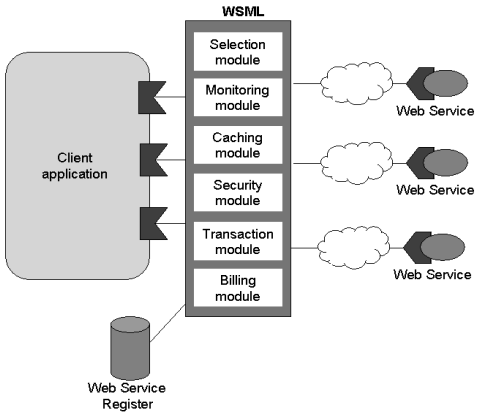

Figure 23: General architecture of the WSML.

8.2 Automatic selection of Web Services by employing JAsCoAP

The WSML offers, among other functionalities, support for the automatic selection, integration and composition of web services. Depending on the specific requirements described by an application manager at run-time, the WSML is able to automatically switch to the most appropriate web services or web service compositions. To this end, a set of non-functional properties, such as price, reliability and response time, are available for each web service separately. This information is stored into an extensive data-structure, from which the architecture is illustrated in Figure 24. Semantically equivalent web services are referenced by employing a *ServiceType*, which offers a unified interface of the service for the application in which it is integrated. Each *ServiceType* is realized by one or

[1] MOSAIC is partly funded by the IWT, Flanders (Belgium), partners are the University of Brussels (VUB) and Alcatel Belgium.

more *ServiceTypeImplementations*. A *ServiceTypeImplementation* provides a *WebServiceComposition* which encapsulates one or more concrete *WebService* representations that implement the functionality offered by their corresponding *ServiceType*. Each *WebService* separately specifies a dedicated set of properties, which are presented using primitive type encapsulations, such as *WebServiceIntProperty* and *WebServiceStringProperty*.

Depending on the requirements specified by an application manager, the most appropriate *ServiceTypeImplementation* is chosen for a particular kind of *ServiceType.* If an application manager is for instance interested in the cheapest solution, all possible *ServiceTypeImplementations* for a particular *ServiceType* are visited and for each *ServiceTypeImplementation* separately the total price of their incorporating web services is calculated and compared. Depending on the results of these calculations, the cheapest service type implementation is chosen and deployed within the application.

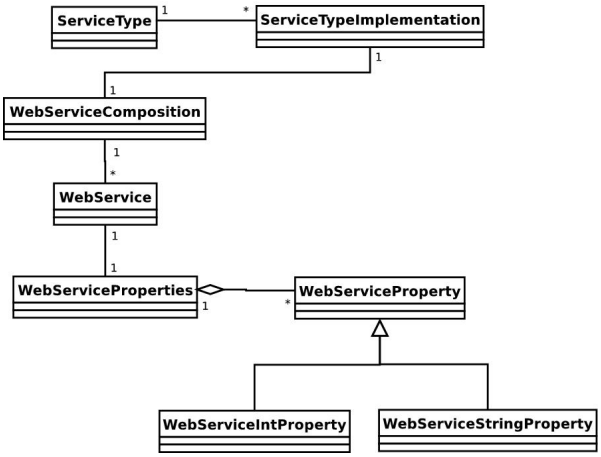

Figure 24: Architecture of the web service property data-structure of the WSML.

Prior to the use of JasCoAP, the implementation of this visiting and comparing process was hard-wired within the implementation of the WSML. For each non-functional requirement defined by a web service, a separate visitor was required to for instance calculate the sum, average, minimum or maximum value for a particular web service type implementation. This resulted in a huge amount of visitor classes, which in many cases contained duplicated processing logic. In addition, the implementation of these visitors has to be manually adapted whenever the property data structure is altered.

In order to keep the implementation of the visiting and comparing process reusable and structure-shy, adaptive visitors, implemented as JAsCo aspect beans, are employed. JAsCoAP allows reusing a very small set of adaptive visitors for automatically selecting the most appropriate web service, as these generic visitors can easily be tailored towards the functional properties they need to visit. Currently, a set of reusable adaptive visitors are implemented which allow calculating the sum, average, minimum and maximum value of all non-functional properties employed within the WSML. These generic aspect beans are resistant to structural changes in the property data structure of the WSML. It is for instance envisioned to also provide *ServiceTypeCompositions* next to concrete *WebServiceCompositions* in order to describe web service compositions that rely on service types. Regular AP could also have been employed in this case study in order to separate the visiting process from the concrete web service property data structure. JAsCoAP has the advantage over regular AP as it is able to reuse only one visitor to for instance calculate the sum of int, float, and double web service properties, while with regular AP, dedicated visitors are required for each kind of property type. Furthermore, the combination strategy solution for avoiding visiting the same object more than once outlined in section 5 is also employed.

A disadvantage of the current application of JAsCoAP within the WSML is the performance penalty at run-time. The current implementation incorporates the DJ library which employs run-time reflection to compute and perform traversals. As a result, selecting the most appropriate web service at run-time using JAsCoAP is about four times slower than the original WSML implementation where the visiting behaviour of the web service property data-structure is hard coded. However, this performance hit will be reduced in the near the future as it is envisioned to either incorporate a custom-made traversal compiler for JAsCoAP or to employ one of the available adaptive programming approaches.

9. CONCLUSIONS

This paper illustrates how the ideas behind Adaptive Programming and JAsCo can be combined to make Adaptive Programming fit into the component-based software development world. Adaptive visitors are described by means of traditional, context-independent JAsCo aspect beans which are deployed making use of JAsCo traversal connectors. Adaptive visitors implemented as aspect beans are now truly reusable as no context information is hard coded. This opens the possibility of reusing an aspect bean both as an aspect instantiated in a regular JAsCo connector and as an adaptive visitor instantiated in a traversal connector. By having the same language syntax, it is also easier to write AOP aspects once AP aspects are learned and vice versa. This makes the JAsCo language very suitable for teaching.

Adaptive visitors implemented as aspect beans can easily be combined in traversal connectors in order to visit the same traversal as specified by the common traversal strategy. JAsCo precedence and combination strategies can be employed in order to describe complex interactions between several adaptive visitors applied upon the same traversal strategy. Furthermore, a case study is presented where JAsCoAP is applied to a real-life and non-trivial application in order to validate the proposed approach.

A drawback of utilizing the same syntax for AP aspects and AOP aspects is that the semantics is different depending on the deployment context. Although the semantics of JAsCo specific keywords is not altered drastically, for example the *thisJoinPointObject* keyword refers either to the object where a method is invoked upon or the object that is currently visited, this could cause confusion and make aspect beans somewhat less readable and maintainable. Another drawback of the approach is that combination strategies are not specified declaratively. While this allows very expressive combination strategies, the basic strategies are not that easy to understand because of the imperative implementation.

10. ACKNOWLEDGEMENTS

First of all, we would like to thank the anonymous reviewers for reviewing or paper in depth and providing us with interesting feedback. We also would like to thank Bruno De Fraine and Therapon Skotiniotis for proof reading the paper. Special thanks to Karl Lieberherr for his feedback on this work. Davy Suvée is supported by a doctoral scholarship from the Flemish Institute for the Improvement of the Scientifical-Technological Research in the Industry (IWT).

11. REFERENCES

[1] Chappell, D. and Jewell, T. *Using Java in Service-Oriented Architectures: Java Web Services.* O'Reilly, 2002.

[2] Elrad, T., Filman, R. and Bader, A. (Eds). *Aspect-Oriented Programming.* Communications of the ACM, 44(10):28-97, 2001.

[3] Gamma, E., Helm, R., Johnson, R. and Vlisside, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] JAsCo including JAsCoAP extension available at: http://ssel.vub.ac.be/jasco

[5] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W., G. *An overview of AspectJ.* In Proceedings of ECOOP'2000, SpringerVerlag, 2000.

[6] Kuipers, T. and Visser, J. *Object-oriented tree traversal with JJForester.* In ENTCS, volume 44, Elsevier Science, 2001.

[7] Lämmel, R. *Typed Generic Traversal With Term Rewriting Strategies*. Journal of Logic and Algebraic Programming, 54:1.64, September 2002.

[8] Lämmel, R., Visser, E. and Visser, J. *Strategic Programming Meets Adaptive Programming*. In Proceedings of the second International Conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[9] Lämmel, R. and Visser, J. *Typed Combinators for Generic Traversal*. In Proc.of Practical Aspects of Declarative Programming (PADL), LNCS 2257, January 2002.

[10] Lieberherr, K. *Towards a Theory of Design.* In proceedings of ICSE 2004 (keynote paper), Edinburgh, UK, May 2004.

[11] Lieberherr, K. and Holland, I. *Assuring Good Style for Object-Oriented Programs.* IEEE Software, pages 38-48., September 1989.

[12] Lieberherr, K. and Lorenz, D. *Coupling Aspect-Oriented and Adaptive Programming.* Aspect-Oriented Software Development. Addison Wesley, 2004.

[13] Lieberherr, K and Orleans, D. *Preventive Program Maintenance in Demeter/Java*. In Proceedings of International Conference of Software Engineering (ICSE), pp. 604-605, 1997.

[14] Lieberherr, K., Orleans, D. and Ovlinger, J. *Aspect-Oriented Programming with Adaptive Methods.* Communications of the ACM, Vol. 44, No. 10, October 2001.

[15] Meyer, B. *Onto components*. In IEEE Computer, Volume 32, January 1999.

[16] Mezini, M. and Lieberherr. *Adaptive Plug-and-Play Components for Evolutionary Software Development*. In Proceedings of OOPSLA'98. Vancouver, Canada, October 1998.

[17] Mezini, M. and Ostermann, K. *Conquering Aspects with Caesar*. In Proceedings of the second international conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[18] Orleans, D. and Lieberherr, K. DJ: *Dynamic Adaptive Programming in Java.* In Proceedings of Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, September 2001.

[19] Pulvermüller, E., Speck, A., Coplien, J.O., D'Hondt, M. and De Meuter, W. Proceedings of Workshop on "feature interaction in composed systems" at ECOOP 2001. Available at: http://www.info.uni-karlsruhe.de/~pulvermu/workshops/ecoop2001/

[20] Shonle, M., Lieberherr, K. and Shah, A. *XAspects: An Extensible System for Domain Specific Aspect Languages.* In Proceedings of OOPSLA international conference (ACM), Anaheim, USA, October 2003.

[21] Sun, JavaBeans(TM) Specification 1.01. Available at: http://java.sun.com/products/javabeans/docs/spec.html

[22] Sung J. and Lieberherr, K. DAJ: *A Case Study of Extending AspectJ.* Northeastern University Technical Report NU-CCS-02-16, 2002. Available at: http://www.ccs.neu.edu/research/demeter/biblio/DAJ1.html

[23] Suvée, D., Vanderperren, W. and Jonckers, V. *JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development.* In Proceedings of the second International Conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[24] Szyperski, C. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, ISBN 0-201-17888-5, 1998.

[25] Szyperski, C. *Components and Web Services.* Beyond Objects column, Software Development, Vol. 9, No. 8, August 2001.

[26] Verheecke, B., Cibran, M. A. and Jonckers, V. *AOP for Dynamic Configuration and Management of Web services in Client-Applications*. In Proceedings of 2003 International Conference on Web Services. Erfurt, Germany, September 2003.

[27] Verheecke, B., Cibran, M. A. and Jonckers, V. *WSML*. available at: http://ssel.vub.ac.be/wsml

[28] Visser, E., Benaissa, Z.-e.-A. and Tolmach, A. *Building program optimizers with rewriting strategies*. In Proceedings of the 3rd International Conference on Functional Programming. Baltimore, USA, September 1998.

[29] Visser, J. *Visitor combination and traversal control*. In Proceedings of the OOPSLA 2001 International Conference. Tampa Bay, USA, November 2001.

[30] Wu, P., Krishnamurthi, S. and Lieberherr, K. *Traversing Recursive Object Structures: The Functional Visitor in Demeter.* In Proc. of Software Engineering Properties of Languages for Aspect Technologies (SPLAT) Workshop of AOSD'03, Boston, USA, March 2003.