JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development

Davy Suvée
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 65
dsuvee@vub.ac.be

Wim Vanderperren
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 62
wvdperre@vub.ac.be

Viviane Jonckers
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 67
viviane@info.vub.ac.be

ABSTRACT

In this paper we introduce a novel aspect oriented implementation language, called JAsCo. JAsCo is tailored for component based development and the Java Beans component model in particular. The JAsCo language introduces two concepts: aspect beans and connectors. An aspect bean describes behavior that interferes with the execution of a component by using a special kind of inner class, called a hook. The specification of a hook is context independent and therefore reusable. A connector on the other hand, is used for deploying one or more hooks within a specific context. To implement the JAsCo language, we propose a new "aspect-enabled' component model, which contains build-in traps that enable to interfere with the normal execution of a component. The JAsCo component model is backward-compatible with the Java Beans component model. Furthermore, the JAsCo component model allows very flexible aspect application, adaptation and removal at run-time. The necessary tool support for the JAsCo approach has been implemented. In addition, we present a performance assessment of our current implementation.

1. INTRODUCTION

Component based software development (CBSD) and more recently, aspect-oriented software development (AOSD) have been proposed to tackle problems experienced during the software engineering process. When applying CBSD, a full-fledged software-system is developed by assembling a set of pre-manufactured components. Each component is a black-box entity, which can be deployed independently and is able to deliver specific services [18]. The deployment of this paradigm drastically improves the speed of development and the quality of the produced software. AOSD on the other hand, tries to improve the separation of concerns [14] in current software engineering methodologies, by providing an extra separation dimension along which the properties of a software-system can be described.

Currently available AOSD-research mainly focuses on object-oriented software development (OOSD). CBSD however, also suffers from the problems that arise with the tyranny of the dominant decomposition [13]. Similar to OOSD, aspects such as synchronization and logging are encountered, which crosscut several components from which the system is composed. Consequently, the ideas behind AOSD should also be integrated into CBSD. The other way around, namely the integration of CBSD within AOSD, is a valuable concept as well. CBSD puts a lot of stress on the plug-and-play characteristic of components; for example, it should be possible to extract a component from a particular composition and replace it with another one. Introducing a similar plug-and-play concept in AOSD, would make aspects reusable and their deployment easy and flexible.

Combining the AOSD and CBSD principles is a valuable contribution to both paradigms. However, currently available AOSD and CBSD research cannot be straightforwardly integrated, this because of several restrictions which are imposed by the existing approaches:

- The deployment of an aspect within a software-system is at this moment rather static. In AspectJ for example, an aspect looses its identity when it is integrated within the base-implementation of a software system. This makes it very difficult to extract an aspect from a particular composition and to replace it afterwards with a totally different aspect. This plug-and-play property is vital in some environments where the dynamic characteristic of components is considered an essential requirement.
- Most AOSD-approaches describe their aspects with a specific context in mind. Therefore, it is impossible to reuse aspects. This is not acceptable within CBSD, since every component of a software-system should be independently deployable.
- The communication between the various components from which an application is composed, is in most cases specific to the employed component model. Java Beans for instance, makes use of an event-model. Currently available AOSD-technologies however, are not suited to deal with these specific kinds of interactions.

To integrate the ideas of AOSD into CBSD, we introduce a new aspect-oriented implementation language, named JAsCo, which is designed especially for CBSD. This language enables the development of software along another separation dimension, on

AOSD 2003 Boston, MA USA

top of the Java class hierarchy. JAsCo stays as close as possible to the regular Java syntax and introduces two concepts: aspect beans and connectors. An aspect bean is a regular Java bean that is able to declare one or more logically related hooks, as a special kind of inner classes. Hooks are generic and reusable entities and can be considered as a combination of the AspectJ's pointcut and advice. Since aspect beans are described independent from a specific context, they can be reused and applied upon a variety of components. The initialization of a hook with a specific context is done by making use of connectors.

To make the JAsCo language operational, we introduce a new "aspect-enabled" component model. The JAsCo Beans component model is a backward compatible extension of the Java Beans component model where the traps are already built-in. These traps are used to attach and detach aspects. As a result, JAsCo beans do not require any adaptation whatsoever to be subject to aspect application. The JAsCo component model enables run-time aspect application and removal. In addition, aspects remain first class entities at run-time as they are not weaved and spread into the target components.

The next section gives an explanation about the different features the JAsCo-language has to offer. The syntax of aspects and connectors is discussed by making use of some small examples. Section three introduces the JAsCo component model in more detail. Afterwards, research that is related to this work is summarized and the integration of the JAsCo language in a visual component composition environment is shortly sketched. Finally, we describe our future research and state our conclusions.

2. THE JASCO LANGUAGE

The intention of JAsCo is to provide an aspect-oriented extension to Java. The principal aim is to keep our language as close as possible to the regular Java syntax and concepts, this by introducing a minimal number of new keywords and constructs. The JAsCo-language is primarily based upon two existing AOSD approaches: AspectJ [1] and Aspectual Components [12]. AspectJ's main advantage is the expressiveness of its "join point"-language. However, aspects are not reusable, since the context on which an aspect needs to be deployed is specified directly in the aspect-definition. To overcome this problem, Karl Lieberherr et al introduce the concept of Aspectual Components. They claim that doing aspect-oriented programming means being able to express each aspect separately, in terms of its own modular structure. Using this model, an aspect is described as a set of abstract join points which are resolved when an aspect is combined with the base-modules of a software system. This way, the aspect-behavior is kept separate from the base components, even at run-time. JAsCo combines the expressive power of AspectJ with the aspect independency idea of Aspectual Components.

This section introduces the various features JAsCo has to offer and the syntax of both the aspect and the connector language. This explanation is given by presenting a basic aspect, which enables to control the access to components.

2.1 Aspect syntax

The JAsCo-language introduces two constructs: *aspects beans* and *connectors*. Aspects beans are used for describing some functionality that would normally crosscut several components from which the system is composed. An example of such crosscutting concerns is *access-control*. Database-systems for instance, need some control-mechanism to manage the user-access to the objects they hold. A similar concern could be stipulated in an ordinary application. Imagine a piece of software that runs on an operating system that allows only one user at the same time to be logged in. If users don't have the required permission, the access to some services of this system should be denied. Figure 1 shows an access-control aspect, specified using JAsCo.

```
1  class AccessManager {
2
3    PermissionDb p_db = new PermissionDb();
4    User currentuser = null;
5
6    void login(User user, String pass)
7      { //login code }
8    void logout()
9      { //logout code }
10
11   void addAccessManagerListener(AML listener)
12     { //adding code }
13   void removeAccesManagerListener(AML listener)
14     { //remove code }
15
16   hook AccessControl {
17                                              When?
18     AccessControl(method(..args)) {
19       execute(method); }
20                                                    What?
21     replace() {
22       if(p_db.check(currentuser,cobject) {
23         return method(args); }
24       else {
25         throw new AccessException(); }
26     }
27
28   }
29 }
```

Figure 1: The JAsCo-aspect for access-control.

An aspect bean usually holds one or more hook-definitions (line 16 till 26), and is able to contain any number of ordinary Java class-members (line 3 till 14), which are shared amongst all hooks. Each hook is a participant of an aspect, and is used for specifying:

- **when** the normal execution of a method of a component should be "cut".
- **what** extra behavior there should be executed at that precise moment in time.

A hook specifies at least one *constructor* (line 18 till 19) and one *behavior method* (line 21 till 26), and is able to contain any number of ordinary Java class-members. A hook-constructor is similar to a regular Java constructor. It is identified with the name of the hook it belongs to, followed by one or more *abstract method parameters*. Abstract method parameters outline the input of a hook and are used for defining the context of a hook initialization. They are substituted for the concrete method signatures at aspect application time. The constructor (line 18) of the *AccessControl*-hook specifies that this hook can be deployed on every method which takes zero or more arguments as input. The constructor-body (line 19) describes how the join points of a hook initialization should be computed. In this particular case, the behavior of the *AccessControl*-hook is performed whenever a method, which has been taken as input of the hook, is executed.

Another consruct, *cflow,* can be used to delimit the context of a hook to the control-flow of another method. Both *cflow* and *execute* constructs can be combined using logical operators. This makes it possible to describe more advanced join point calculations. Similar to regular Java classes, hooks are able to contain additional constructors. This makes it possible to initialize an aspect in more than one way.

The behavior methods of a hook are used to specify the various actions a hook needs to execute when one of its calculated join points is encountered. Three kinds of behavior methods are available: *before, after* and *replace*. The *AccessControl*-hook specifies only a *replace*-method (line 21 till 26), which substitutes the normal execution of the method which initialized the hook. The *replace*-method checks if the currently logged-in user has the proper access-permissions to the component that was called. This is done, by employing the *cobject*-keyword, which refers to the object that contains the point where the normal execution has been interrupted. The permissions-database, which contains the various logins and permissions for each user of the system, verifies if the logged-in user has the proper permissions for accessing the *cobject*. When no problems are encountered, *method* is executed. Otherwise, an *AccessException* is thrown.

2.2 Connector syntax

Deploying an aspect within an application is done by making use of connectors. Imagine that our application contains a printer component. Only people who have the appropriate print permissions, may access this component. Figure 2 shows the connector that is used for deploying the *AccessControl*-aspect upon the Printer component:

```
1  connector PrintAccessControl {
2
3    AccessManager.AccessControl control =
4      new AccessManager.AccessControl(
5        * Printer.*(*));
6
7    control.replace();
8
9  }
```

Where?

Figure 2: The JAsCo-connector for print-access-control.

A connector contains three kinds of constructs: one or more hook-initializations (line 3 till 5), zero or more behavior method executions (line 7), and finally any number of regular Java constructs. A hook-initialization is identical to a Java class instantiation, and takes one or more method signatures (dependant on the number of abstract method parameters specified in the hook-constructor) as input. The *PrintAccessControl* connector contains one hook-initialization *control*, which is deployed upon all methods defined in the Printer component interface. Afterwards, the execution of the replace behavior method on the *control* hook is specified. To sum up, the connector of Figure 2 indicates that the *replace* method of the *AccessControl* hook should be executed, whenever one of the methods of the Printer component is called. After compiling the *PrintAccessControl* connector, a unique instance of this connector will exist at run-time. Imagine the application also includes a fax component, which should use the same permissions-database as the one of the printer. Figure 3 illustrates a connector where the *AccessControl* hook is applied on both the printer and the fax component. By inserting multiple method signatures between braces, the same hook-initialization can be applied onto different components.

```
1    AccessManager.AccessControl control = new
2      AccessManager.AccessControl(
3        { * Printer.*(*) , * Fax.*(*) } );
```

Figure 3: JAsCo-connector for multiple-hook-initialization.

In the Java Beans component model, the outgoing communication is done by posting "events". JAsCo-aspects are able to intercept these events, by initializing a hook with the *onevent* keyword. If the Printer component throws an event when it finishes printing a document, this event can be intercepted by a hook in order to execute some appropriate action. Figure 4 shows the initialization of a logging-aspect which writes some statistics to file, when a printing-job has finished.

```
1    Logging.FileLogger logger = new
2      Logging.FileLogger(
3        onevent Printer.jobFinished(PrintEvent));
```

Figure 4: Hook-initialization on events.

Calling aspect behavior methods in the connector (figure 2, line 7) is not really necessary. When no behavior methods are called on a hook initialization, the *default* behavior of the hook is assumed. This *default*-method is automatically generated when an aspect is compiled. It specifies the execution of all behavior methods that were specified in the aspect, applying the following order: *before – replace – after*.

A double motivation exists for permitting the calling of behavior methods in the connector. The first advantage is that it enables advanced users of an aspect to tightly control the execution of the aspect-behavior. The *default*-method on the other hand provides an easy way for deploying an aspect within an application, without needing any knowledge about how the aspect-behavior is executed. The second advantage of this approach is that it provides a solution for *the feature interaction* problem [23]. When multiple aspects are applied upon the same join points of an application, some way is needed to order the execution of their behaviors. JAsCo partly addresses this open issue in AOSD by the specification of the behavior executions in the connector. Imagine that only one user at the same time may address the *Printer*-component, and the access to the printer still needs to be managed. Figure 5 illustrates the simultaneous deployment of a *Lock*-aspect and an *AccessManager*-aspect upon the Printer component.

JAsCo allows arranging the execution of the aspect behaviors by specifying the order in the connector body. Whenever mutual join points of both the *acontrol* and the *lcontrol* hooks are found, the order in which the behavior execution is specified in the connector is applied. In this particular case, the access to the Printer component is locked, by calling the *before* behavior method on the *lcontrol* hook (line 11), so that no other user can access it. Next, the *AccessManager* checks if the user has the correct permissions to use the printer (line 12). Afterwards, the lock is freed (line 13), by calling the *after* behavior method, such that other users can access the *Printer* component again.

```
1  connector PrintLockAccessControl {
2
3    AccessManager.AccessControl acontrol =
4      new AccessManager.AccessControl(
5        * Printer.*(..));
6
7    LockingManager.LockControl lcontrol =
8      new LockingManager.LockControl(
9        * Printer.*(..));
10
11   lcontrol.before();
12   acontrol.replace();
13   lcontrol.after();
14
15 }
```

Figure 5: Connector that controls the precedence of hooks.

In comparison to AspectJ, JAsCo allows a more fine-grained control on the order in which aspects should be executed. In addition, JAsCo allows the precedence of aspects to vary over different applications as this is not hard coded into the aspect.

2.3 Advanced aspect combinations

The primitive support for arranging the execution of aspect-behavior presented above is not sufficient for specifying more complex aspect relationships. For example, one might have to specify that when aspect *A* is applied aspect *B* can not be applied. This problem could be solved by introducing an extra connector-keyword *excludes* which specifies that aspect *A* excludes aspect *B*. However, other aspect combinations require additional keywords and it seems impossible to be able to define all possible combinations in advance. That's why we propose a more flexible and extensible system that allows to define a combination strategy using regular Java. A *CombinationStrategy* interface is introduced (see Figure 6) that needs to be implemented by each concrete combination strategy. A JAsCo *CombinationStrategy* works like a filter on the list of hooks that are applicable at a certain point in the execution.

```
1  interface CombinationStrategy {
2
3    public HookList verifyCombinations(Hooklist);
4
5  }
```

Figure 6: The CombinationStrategy-interface.

To clarify how JAsCo combination strategies work we propose a slightly adapted version of the *AccessControl*-hook, presented in Figure 7. The *ExtAccessControl*-hook specifies that when the administrator has logged in, no access-control checks have to be performed because the administrator has access to all parts of the system. For implementing this functionality, the *isApplicable* method is introduced. Often, the execution of a hook depends on more than the programmatic conditions that are defined when a hook is instantiated. The *isApplicable* method allows to specify whether the hook has to be executed depending on external conditions that are checked at run-time. In absence of such a construct, this condition has to be tested in all the aspect behavior methods that are implemented. Making this condition explicit by introducing a new keyword has the advantage that it allows more elaborated aspect combination strategies as the hook is only executed when it is applicable on both programmatic and external conditions. In addition, introducing a new keyword allows optimizing this condition performance-wise in comparison to having it implicitly in each hook behavior method.

```
1  class ExtAccessManager extends AccessManager {
2
3    hook ExtAccessControl extends AccessControl {
4
5      isApplicable() {
6       return !p_db.isAdmin(currentuser);
10    }
11   }
12 }
```

Figure 7: AccessControl with admin-check.

Now, we want to able to apply both the *ExtAccessControl*-hook and the *FileLogger*-hook, however we only want to log actions whenever the *ExtAccesControl*-hook is also applied. This behavior is accomplished by implementing the *CombinationStrategy*-interface of Figure 6. Each combination-strategy needs to implement the *verifyCombinations*-method, which filters the list of applicable hooks and possible modifies the behavior of individual hooks.

The relationship between the *AccessManager*-aspect and the *Logging*-aspect is defined as a *twin*-combination, since the behavior of the *Logging*-aspect should only be executed when the behavior of the *AccessManager*-aspect has been performed. Figure 8 shows the implementation of a reusable *TwinCombinationStrategy*. This combination-strategy specifies that *hookB* should be removed whenever *hookA* is not found (line 11-16). This way, the behavior of *hookB* is never executed, if the behavior of *hookA* is not performed.

```
1  class TwinCombinationStrategy
2                implements CombinationStrategy {
3
4    private Object hookA, HookB;
5
6    TwinCombinationStrategy(Object a,Object b) {
7      hookA = a;
8      hookB = b;
9    }
10
11   HookList verifyCombinations(Hooklist hlist) {
12     if (!hlist.contains(hookA)) {
13       hlist.remove(hookB);
14     }
15     return hlist;
16   }
17
18 }
```

Figure 8: The twin combination-strategy.

The combination-strategy of Figure 8 can now be added to a connector where both the *ExtAccesControl*-hook and the *FileLogger*-hook are instantiated. Figure 9 illustrates such a connector. Both the *ExtAccessControl*-hook and the *FileLogger*-hook are applied upon the same context (line 3 till 9), and are used as input of the twin-combination-strategy (line 11-12). This strategy is added to the list of combination-strategies of the connector (line 14). To add a combination strategy to a connector, the *addCombinationStrategy*-method has to be called. Of course, it is possible to add multiple combination strategies to the same connector. In that case, the result of the first combination strategy is passed on to the second and so on.

```
1  connector LoggingAccessControl {
2
3    ExtAccessManager.ExtAccessControl control =
4      new ExtAccessManager.ExtAccessControl(
5        * System.*(..));
6
7    Logging.FileLogger logger =
8      new Logging.FileLogger(
9        * System.*(..));
10
11   TwinCombinationStrategy twin = new
12     TwinCombinationStrategy(control,logger);
13
14   addCombinationStrategy(twin);
15
16   logger.before();
17   control.replace();
18   logger.after();
19
20 }
```

Figure 9: Connector using a combination-strategy.

3. JASCO COMPONENT MODEL

3.1 Introduction

To make the JAsCo language operational for CBSD, "normal" weaving is not an option. First of all, source code weaving is not possible because third party components are often only available in binary form. Byte code weaving on the other hand is technically quite complex in comparison to source code weaving. In addition, byte code weaving leads to serious problems when considering quality of service guarantees for third party components. Third party components often ship with several quality of service (QOS) guarantees, like for example memory usage in certain conditions. If one weaves aspects into a component, obviously all the guarantees become void. It is even possible that a component is encrypted or digitally signed so that it becomes impossible to modify the component at all. Another severe disadvantage of normal weaving is that it is too static. In most weaving approaches aspects can't be dynamically loaded and unloaded at run-time. When considering CBSD, and in particular the world of web services where flexibility and dynamicity are of great importance, static weaving becomes unfeasible.

We considered two options to implement the JAsCo language for component based development: modify the virtual machine or introduce a new component model. The first option consists of modifying the virtual machine so that aspects can intercept method calls and execute their own behavior instead. It is clear, that developing a new virtual machine should suffer less performance penalties. However, the main drawback of this approach is that it is not very flexible, since a specialized virtual machine has to be used. The solution we eventually choose for implementing JAsCo consists of introducing a new "aspect enabled" component model. The JAsCo Beans component model is a backward compatible extension of the Java Beans component model where the traps are already built-in. The idea is that regular Java Beans are transformed to JAsCo components at component development time. A component developer can sell "aspect-enabled" JAsCo Beans that do not require any adaptation whatsoever to apply aspects. In this way, the component developer can guarantee QOS for the components. In addition, our approach allows the component developer to shield some crucial parts of the component from aspect interaction. Our approach also allows aspects to keep their identity at run-time, as they are not woven and spread out into the base components, but are still separate entities at run-time. In addition, dynamic connector loading and unloading becomes possible.

3.2 Our approach

Figure 10 illustrates schematically how JAsCo is implemented. The central connector registry serves as the main addressing point for all JAsCo entities. The connector registry is notified when a trap has been reached or when a connector has been loaded. The left-hand side of Figure 10 shows the JAsCo bean *comp1*. All methods of *comp1* are equipped with traps, so that when a method is called, the execution is deferred to the connector registry. The main method of communication of Java Beans is event posting, so throwing an event also reschedules execution to the connector registry. The connector registry contains a database of connectors. When a trap is reached, the connector registry looks up all connectors that registered for that particular method or event. The connector on its turn dispatches to the hooks that have been instantiated with the corresponding method or event.

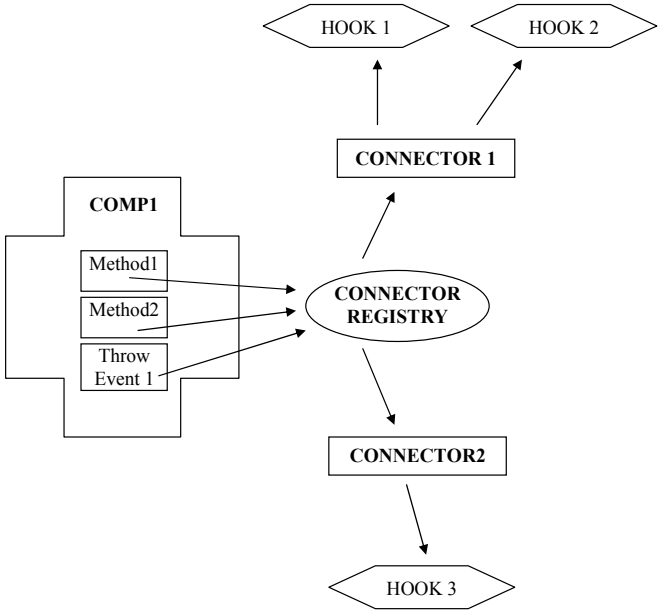

Figure 10: JAsCo architecture.

While the advantages of having explicit connectors at compile-time are obvious, one might question why we keep this extra level of indirection at run-time. The idea is that connectors serve as collections of related aspects. We want to add, remove or edit the behavior of these aspects at the same time. For example, if we want to alter a connector to only trigger on a certain instance of a JAsCo bean and not on other instances, we can now do this easily by accessing the connector. Otherwise, we have to manually notify all the affected aspects of this change. In addition, this would require that the aspects have public methods for adapting their behavior other than those defined in the aspect's source code, what could be confusing. In short, it is possible to get rid of an explicit connector at run-time to gain efficiency, but we would

end up with a polluted model where the connector's logic is spread over different entities.

Our approach is very flexible to support unanticipated run-time changes. Connectors can be easily loaded and un-loaded at run-time. The connector registry detects whether connectors are removed or added to the system and takes appropriate actions. We also support the instantiation of a hook on expressions that contain wildcards. These limited regular expressions are matched at run-time. Consequently, when a new component is added to an application, it is automatically affected by all aspects that were declared using wildcards. In addition, run-time wildcard matching makes the compilation of a connector that instantiates a wildcard aspect very fast, this in comparison to approaches that resolve these wildcards at compile-time. On the other hand, matching the wildcard expressions at run-time degrades the run-time performance.

3.3 Implementation of tool support

In this section, we introduce the different tools developed to realize the JAsCo language without going into too much technical detail. The most important tool is the bean transformation tool that transforms a regular Java bean into a JAsCo bean. This tool takes as input a Java bean in binary format and inserts traps at every method the bean implements. At run-time, the traps inquire the connector registry for hooks that are registered on this method. If no hooks are registered, the normal execution of the method continues, otherwise the hooks' behavior is executed at the appropriate moment (i.e. before, after, replace). Applying hooks on the throwing of events is somewhat trickier. The Java Beans convention doesn't standardize the naming of the method that causes the event to fire. Although *fire* followed by the event name is used for Java swing components, in general we can't even count on the existence of a method that throws the event. Therefore, we employ the following strategy: we use the fire method if it exists; otherwise we patch the standardized methods for adding and removing listeners, so that we are in control of the firing of the corresponding event.

Our approach doesn't allow calling the original method implementation from outside, so bypassing the execution of aspects is not possible. Also, the resulting JAsCo bean has the same interface as the original bean, because we do not add any public members.

Technically, the transformation process employs byte-code manipulation techniques. We use a custom-made byte code adaptation library [11] for adapting methods and adding new behavior. This library allows, in contrast to most other adaptation libraries, to inject plain Java code in a class file. The library takes care of compiling the inserted Java source code to Java byte code. This simplifies implementing the transformation process greatly. Our library uses BCEL [4] to transform Java byte codes to human readable jasmin assembler code and Jasmin [10] is used to do the transformation the other way around, i.e. from jasmin to a Java class file.

Beside the transformation tool, we have four other tools: *CompileConnector*, *RemoveConnector*, *CompileAspect* and *Introspect*. The first tool allows compiling a connector to its Java class representation. To enable the connector's logic, the connector has to be in the classpath of the target application. *RemoveConnector* is a tool that allows to remove a certain connector. The removal is detected by the connector registries of applications that have that connector in their classpath. *CompileAspect* compiles a JAsCo aspect to a normal Java bean. The generated bean is also equipped with traps so that it is possible to apply other aspects on a certain aspect. *Introspect* is a GUI tool that allows introspecting what connectors are loaded for a given classpath. The tool also shows the various hooks that are instantiated by the connectors and the targets on which these hooks are applied.

3.4 Performance assessment

Table 1: Three performance experiments using a tracing aspect. The time values are in milliseconds.

Experiment	Aspect hard coded	Aspect applied direct	Wildcard aspect application
1000x short method	501	1332	1822
10x long method	10015	10155	10246
Event throwing	117	160	231

When designing and implementing our approach, we systematically choose for maximum flexibility and having a high-quality model. That these choices have a negative effect on performance is no surprise. To prove that our approach is still functional in practice we perform three small performance experiments (see Table 1). The experiments are all fine grained in the sense that we only benchmark one method instead of the operation of a whole application. This because we clearly want to show the effect of applying an aspect on a method. If we take a whole system as benchmarking artifact, a bunch of factors have to be taken into account for explaining performance differences. For all three experiments, we use the same simple tracing aspect. This is because only the location where the aspect is applied to matters for checking the performance of our system. The first experiment consists of executing a short method thousand times. The overhead of using JAsCo to separate the tracing logic is quite large here and becomes even more than 300% when aspects are applied using wildcards. In the second experiment, we apply our tracing aspect to one long method that is executed 10 times. As one might expect, the overhead is much less here. But the wildcard aspect application still poses an overhead of nearly 2,5%. The last experiment applies the tracing aspect to the throwing of the *actionPerformed* event in the JButton bean and shows similar results. In general, our approach causes a fixed amount of overhead per method call or event throwing. On our test system[1] this overhead ranges from 0.8ms to 50ms per call, depending on the number of times the method is executed. At first sight this overhead might seem quite huge; however some considerations have to be taken into account. First of all, as we're dealing with black-box components, only the interface (i.e. public methods) of a component is equipped with traps and is subject to a performance overhead. Aspects should not be dependent on the interior of a component, because the implementation of a component might evolve or change completely when new

[1] Pentium III 933Mhz, 192MB ram, Windows XP, JDK 1.4

versions of a component are released. The larger overhead for wildcard application of aspects might be explained by the fact that we use a full regular expression matching library. However, we only support wildcards for now, so a custom made wildcard matching library should decrease this overhead significantly. We also want to stress that the tool support is in an early prototype phase, so there's still room for improvement. On the other hand, these benchmarks already suggest that our approach is not applicable in domains with limited resources, as for example the very popular embedded systems market.

4. RELATED WORK

One of the first approaches to integrate aspect oriented software development and component based software development is the aspectual component model of Lieberherr et al [12]. The JAsCo language was partly inspired by this work and quite some similarities exist between both languages. They both employ a separate connector language to deploy an aspect within a specific context. On a technical level however, aspectual components uses byte code weaving, while we propose a new component model.

Another, more recent approach to recuperate aspect oriented ideas in component based software development is event based aspect oriented programming (EAOP). EAOP [6] allows specifying crosscuts on events and event patterns using a formal language. Since EAOP is based on a formal model, EOAP is able to improve on JAsCo because of the advanced detection and resolution of aspect interactions [7]. On a technical level, EAOP uses a similar model to make the language operational. A central event monitor, similar to our connector registry, serves as the main addressing point of all EAOP artifacts.

Invasive composition [2] proposes an original approach to assemble a set of reusable components. Typical component composition approaches generate glue-code for enabling cooperation between components. Invasive composition on the other hand adapts the components themselves so that they are able to interact with the other components in the composition. The invasive composition model introduces a box as a generic and programming language independent component. Similar to JAsCo, a box contains a set of hooks where the normal execution can be altered. Unlike JAsCo however, these hooks are implicit. Moreover, behavior is inserted at a given hook by making use of program transformations. As a consequence, invasive composition is less flexible than JAsCo regarding unanticipated run-time changes.

Filman [9] proposes dynamic injectors to introduce aspects into a given component configuration. He incorporates dynamic injectors into OIF (Object Infrastructure Framework), a CORBA centered aspect-oriented system for distributed applications. A dynamic injector is a first class object and can be added or adapted at run-time. Unlike JAsCo, Filman uses a wrapping technique to inject the aspect's logic into the application which has the advantage that no new component model is needed.

Duclos et al [8] focus on separating crosscutting concerns in legacy systems built using CCM[5]. Similar to JAsCo they employ two languages, one for declaring an aspect and one for describing how the aspect should be used. They improve on JAsCo by lifting the abstraction level for aspect declaration from the implementation level to the architecture level. They apply aspects by generating individually tailored CCM containers that include the aspect's logic. In that sense, their approach is similar to wrapping because they do not allow interior changes to the components. Unlike JAsCo, their approach doesn't allow flexible run-time aspect application and removal. The Dynamic Aspect-Oriented Platform (DAOP) [16] is another approach that targets legacy component based systems. Opposite to [8], it allows flexible application of aspects at run-time. DAOP is a distributed platform, where the middleware layer stores the composition information. This idea is similar to the JAsCo connector registry. In addition, DAOP does not require any component adaptation and allows aspects to remain first-class entities at run-time.

PROSE[17] is an aspect oriented library for Java that is not really designed for component based development. Similar to JAsCo, it allows very flexible run-time aspect weaving and unweaving. Their approach is based on using the Java Virtual Machine Debugger Interface (JVMDI) for intercepting events where an aspect is interested in. As a consequence, they are able to apply aspects on a much wider range of execution points than JAsCo, such as the loading of a certain class. However, their approach doesn't outperform JAsCo, because the JVMDI requires the JVM to run in debug-mode. This imposes serious performance restrictions on the entire application, whereas in JAsCo only a fraction of the system is subject to a performance overhead.

Another interesting approach that allows dynamic aspect weaving is Handiwrap [3]. Handiwrap is designed as an extension of the Java language. Similar to JAsCo, they also insert traps that allow dynamical wrapping using byte code adaptation techniques. Unlike JAsCo, handiwrap doesn't employ a central registry but directly inserts the wrappers into the base classes. As a consequence, the handiwrap approach is less flexible than JAsCo. For example, adapting or removing a wrapper at run-time is not possible using handiwrap. Also, applying aspects on wildcards can not be achieved. In addition, the external interface of wrapped base classes is changed, what can cause some confusing and unexpected side effects. On the other hand, the handiwrap implementation clearly outperforms the implementation of JAsCo.

Another similar approach that allows run-time aspect addition and removal is JAC [15]. JAC is an aspect-oriented framework which doesn't introduce a new language for describing a crosscutting concern. Therefore programming in JAC is situated on a lower abstraction level than JAsCo, since JAsCo introduces extra language constructs for AOSD. On the other hand, JAC is more flexible than JAsCo because run-time changes to where and when aspects need to be applied are easily achieved through some calls to the framework. In JAsCo, most changes require to write and compile a new connector, which is a lot more cumbersome and error-prone. Similar to JAsCo, JAC introduces traps in base components to be able to interfere with their execution. Unlike JAsCo however, these traps are installed at load-time.

In addition, in the world of Meta Object Protocols (MOP), some approaches exist which allow run-time adaptation of a class. Kava [22] in particular uses an approach very similar to ours. Kava also inserts traps that refer to the meta-level, by byte-code transformations. This way, they also achieve strong non-bypassability [22] and JVM independence. However, the overhead of a full meta-object protocol might not always be desirable. In addition, MOPs are not specifically designed for

AOSD and are thus not as efficient in expressing a crosscutting concern.

5. INTEGRATION INTO PACOSUITE

JAsCo is integrated in a visual component composition environment, called PacoSuite [21,24]. PacoSuite allows component based development on a high abstraction level without in-depth technical knowledge of the components. Recently, a new concept called a composition adapter [19,20] is added to enable the modularization of concerns that do not fit in the normal PacoSuite entities. Composition adapters can be applied on a given component composition in a visual way. However, a limitation of this approach is that only the exterior behavior of components can be adapted by re-routing or ignoring their messages. To cope with this limitation, a revised version of the model, called an invasive composition adapter has been proposed. An invasive composition adapter has an implementation in the JAsCo language to make the model operational. The tool automatically generates one or more JAsCo connectors from a visually wired model. Figure 11 shows two screenshot of PacoSuite. The bottom-right screenshot illustrates the application of an invasive composition adapter (the hexagonal shape) on a given component composition. In the top-left screenshot an invasive composition adapter is applied onto a more complex component collaboration. Notice that here several invasive composition adapters are stacked onto the same collaboration.

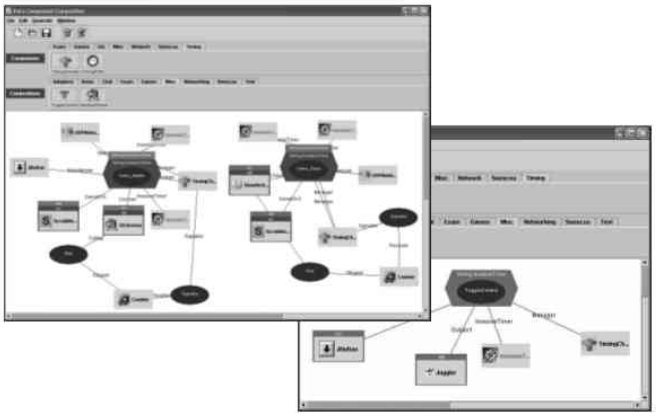

Figure 11: Screenshots of PacoSuite. The hexagonal shapes represent an invasive composition adapter that is implemented in JAsCo.

6. FUTURE WORK

We are currently planning to port JAsCo to several other platforms beside Java Beans. Two of the most important platforms are J2EE and Microsoft .NET. Microsoft's .NET framework has promising features for AOSD. The .NET framework allows integration of programs written in various languages. Currently more than 20 languages are supported, including C++, COBOL and Visual Basic. Similar to Java, .NET compiles all these languages to a common intermediate language, called MSIL. If our component transformation process is able to work on MSIL, we get all the languages supported by .NET for free! J2EE and EJB are Sun's component based solution for legacy systems. It would be interesting to examine how our language has to be extended to cope with this new environment.

On a more conceptual level we want to investigate how we could achieve a symbiosis between aspects and components. Indeed, one might question why aspects and components are considered different entities since our aspects are regular JAsCo Beans equipped with the same traps. Therefore, we plan to develop a powerful connector language that is able to wire aspects and components written in the same base language.

7. CONCLUSIONS

JAsCo is a new aspect oriented implementation language tailored for component based software development and the Java Beans component model in particular. Aspects described using JAsCo are context-independent and first-class entities. A separate connector language is used to apply an aspect onto target components. JAsCo partly addresses the infamous feature interaction problem by allowing to order conflicting aspect behaviors and by introducing explicit and reusable combination strategies. In addition, JAsCo allows the precedence of aspects to vary over different applications as this is not hard coded into the aspect.

To make the JAsCo language operational, we propose a new component model that already incorporates the necessary traps to enable dynamic aspect application and removal. Another advantage of this new component model is that component developers are still able to guarantee QOS for their components. However, the dynamicity and flexibility gained by using this new component model comes with a price. Our approach imposes a rather large performance overhead compared to static languages, like for example AspectJ. As a consequence, our approach is unsuitable in environments where resources are limited.

8. ACKNOWLEDGMENTS

We owe our gratitude to Prof. Dr. Viviane Jonckers for her invaluable help during our research and for proof reading this paper. Since October 2000, Wim Vanderperren is supported by a doctoral scholarship from the Fund for Scientific Research (FWO or in Flemish: "Fonds voor Wetenschappelijk Onderzoek").

9. REFERENCES

[1] AspectJ Website. http://www.aspectJ.org.

[2] Assman, U. A Component Model for Invasive Composition. Position paper at the ECOOP 2000 workshop on Aspects and Dimensions of Concerns. (Cannes France, June 2000)

[3] Baker, J. and Hsieh, W. Runtime aspect weaving through metaprogramming. In Proceedings of the 1st international conference on Aspect-oriented software development. (Enschede The Netherlands, April 2002)

[4] Byte Code Engineering Library http://bcel.sourceforge.net.

[5] Corba Component Model http://www.omg.org.

[6] Douence, R., Motelet, O. and Südholt, M. A formal definition of crosscuts. In Proceedings of the 3rd International Conference on Reflection. (Kyoto Japan, September 2001)

[7] Douence, R., Fradet, P. and Südholt, M. A framework for the detection and resolution of aspect interactions. In Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (Pittsburgh PA, October 2002)

[8] Duclos, F., Estublier, J. and Morat, P. Describing and Using Non Functional Aspects in Component Based Applications. In Proceedings of the 1st international conference on Aspect-oriented software development. (Enschede The Netherlands, April 2002)

[9] Filman, R.E. Applying aspect-oriented programming to intelligent systems. Position paper at the ECOOP 2000 workshop on Aspects and Dimensions of Concerns. (Cannes France, June 2000)

[10] Jasmin Library http://mrl.nyu.edu/~meyer/jvm/jasmin.html.

[11] Java Byte code editor and library http://ssel.vub.ac.be/Members/dsuvee/jbe/index.htm.

[12] Lieberherr, K., Lorenz, D. And Mezini, M. Programming with Aspectual Components. Technical Report, NU-CSS-99-01, March 1999. Available at: http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html.

[13] Ossher, H., and Tarr, S. Multi-Dimensional Separation of Concerns in Hyperspace. Position paper at the ECOOP '99 Workshop on Aspect-Oriented Programming (Lisbon Portugal, June 1999)

[14] Parnas D.L. On the Criteria to be Used in Decomposing Systems into Modules. In Communications of the ACM. Vol.15. No. 12. Pages 1053-1058.

[15] Pawlak, R., Seinturier, L., Duchien, L. and Florin, G. JAC: A flexible solution for aspect-oriented programming in Java. In Proceedings of the 3rd International Conference on Reflection. (Kyoto Japan, September 2001)

[16] Pinto, M., Fuentes, L., Fayad, M.E. and Troya, J.M. Separation of Coordination in a Dynamic Aspect Oriented Framework. In Proceedings of the 1st international conference on Aspect-oriented software development. (Enschede The Netherlands, April 2002)

[17] Popovici, A., Gross, T. and Alonso, G. Dynamic Weaving for Aspect-Oriented Programming. In Proceedings of the 1st international conference on Aspect-oriented software development. (Enschede The Netherlands, April 2002)

[18] Szyperski, C. Component software: Beyond Object-oriented programming. Addison-Wesley, 1998.

[19] Vanderperren, W. A pattern based approach to separate tangled concerns in component based development. ACP4IS workshop at AOSD 2002. (Enschede The Netherlands, April 2002)

[20] Vanderperren, W. Localizing crosscutting concerns in visual component based development. In proceedings of Software Engineering Research and Practice (SERP) international conference. (Las Vegas NV, june 2002)

[21] Vanderperren, W. and Wydaeghe, B. Towards a New Component Composition Process. In Proceedings of ECBS 2001. (Washington DC, April 2001)

[22] Welch, I. and Stroud, R. Kava - A Reflective Java based on Bytecode Rewriting. Lecture Notes in Computer Science 1826 from Springer-Verlag (2000).

[23] Workshop on "feature interaction in composed systems" at ECOOP 2001. Program available at http://www.info.uni-karlsruhe.de/pulvermu~/workshops/ecoop2001.

[24] Wydaeghe, B. and Vanderperren, W. Visual Component Composition Using Composition Patterns. In Proceedings of Tools 2001. (Santa Barbara CA, July 2001)