FuseJ: An architectural description language for unifying aspects and components

Davy Suvée
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
dsuvee@vub.ac.be

Bruno De Fraine
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
bdefrain@vub.ac.be

Wim Vanderperren
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
wvdperre@vub.ac.be

ABSTRACT

In this paper, we propose a novel component architecture and language that aims at achieving a natural unification between aspects and components. In order to modularize crosscutting concerns, this approach does not introduce a specialized aspect construct. Instead, an expressive aspect-oriented composition mechanism is proposed that is applied upon existing component modules. When employing this unified component architecture, regular components are accessed through the use of gates. The behaviour behind these homogeneous gates is combined using expressive connectors that are able to describe both regular and aspect-oriented interactions. We present the FuseJ architectural description language, which realizes the unified component architecture in the setting of the JavaBeans component model. A first proof-of-concept implementation of the FuseJ language is available. In this paper, we evaluate the impact of the unified component architecture and language onto several software engineering properties, in particular comprehensibility, evolvability and predictability.

Keywords

Aspect-Oriented Software Development, Component-Based Software Development, Unification.

1. INTRODUCTION

Component-Based Software Development (CBSD) is a software engineering paradigm that aims at improving the reusability of individual components and component compositions. Ideally, when building component-based applications, a number of off-the-shelf, third-party components are brought together into a single application. In order to keep such third-party components independently deployable, CBSD demands that components never explicitly refer to nor rely on other specific components [18]. With the advent of Aspect-Oriented Software Development (AOSD) on the other hand, a software engineer is able to cleanly modularize crosscutting concerns such logging [11], contract verification [19] and security policies [20]. Without AOSD, the implementation of these concerns remains scattered over and tangled with the different modules of the system, making it hard to add, edit and remove them afterwards.

Currently, a wealth of technologies are available that aim at integrating aspect-oriented ideas into a component-based context. Examples of such technologies are JAC [14], OIF [9], JBoss/AOP [10], EAOP [6], and JAsCo [17]. All of these approaches however focus at introducing new programming languages or frameworks for modularizing crosscutting concerns: aspects are either specified in a dedicated language or are required to implement a particular set of interfaces. Hence, an aspect is considered, treated and implemented as a different kind of entity within the application. But can this differentiation between aspects and components really be justified? Inherently, the behaviour provided by aspects is not that different from component behaviour. Both implement some functionality required within the application, and only the way in which they interact with the rest of the software system differs. The introduction of a separate aspect construct however has several disadvantages. Firstly, the implied crosscutting composition mechanism of an aspect module resides itself tangled with the behaviour of the concern, inherently ruling out other ways of integrating its behaviour within the application. Secondly, the reusability and applicability of existing components is constrained. Nowadays, several mature, feature-rich components are available which for instance allow managing the security issues within an application. At the moment however, there is no straightforward solution available for integrating these existing components in an aspect-oriented fashion. One can either wrap these components in an aspect module or refactor their entire implementation towards an aspect. Both tasks however are cumbersome and error-prone and sometimes even impossible as components are often only available as black-box entities.

Instead of introducing a specialized aspect module, we propose to apply aspect-oriented composition mechanisms upon existing module constructs. As such, independently specified components can be deployed in both a regular and aspect-oriented fashion. An analog concept is found in the context of the preparation of a dinner. First, one goes to a food store and buys the required ingredients such as potatoes and eggs. Depending on the recipe at hand, one only

Submitted to the AOSD 2005 Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT '05).

decides at the moment of preparation whether these potatoes and eggs should be integrated into the dish either boiled or fried. It is evident that these basic ingredients can be bought without predefined stipulations about their use: the customer is able to employ them in the way he/she desires. Obtaining a similar concept for aspect-orientation, would allow one to buy existing components and only decide at application composition time whether a component should be integrated into the software application in a regular or aspect-oriented fashion.

In this paper, we present the first steps towards a new component architecture, which aims at achieving a natural unification between aspects and components. Within this unified architecture, all functionalities required by an application are implemented as regular software components. The core idea of this unified architecture is to augment each of these components with a dedicated interface that exposes their services through the use of gates. Gates provide centralized access to the internal functionality offered by a component and allow their behaviour to be composed in both a regular and aspect-oriented fashion by making use of expressive, declaratively specified connectors. The next section of this paper sketches an overview of the main ideas and concepts of the unified component architecture. Section 3 presents the FuseJ architectural description language, which is a practical realization of the unified component architecture for the JavaBeans component model. Section 4 describes the internals of the container-based execution model that empowers the FuseJ language. Section 5 presents the impact of the unified component architecture, and in particular the FuseJ language, upon several software engineering properties, such as comprehensibility, predictability and evolvability. Finally, we end up by giving an overview of some related work and present our conclusions.

2. TOWARDS THE UNIFIED COMPONENT ARCHITECTURE

In order to achieve a seamless integration between aspects and components, a novel unified component architecture, as illustrated in figure 1, is proposed. When adopting this architecture, each concern required within the software system is mapped upon a regular component. The behaviour of these components is implemented by employing some base component language and no additional language constructs are provided for implementing possible aspect-oriented interactions. Hence, concerns that are typically employed in an aspect-oriented context, such as logging or security, no longer require a dedicated aspect module, but can have one multi-purpose implementation. As already mentioned in the introduction, we consider each component to be a black-box entity that is specified and deployed independently from other specific components [18].

In general, components offer one or more functionalities, which we call *services*. In order for other components to employ these services, each component is provided with a *gate* interface. The intention of a gate is twofold:

1. Delimit and expose the internal implementation of the component.
2. Offer a homogeneous access-point for enabling both regular and aspect-oriented interactions with other components.

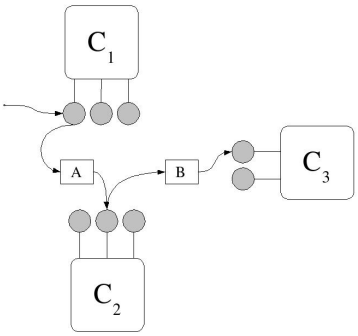

Figure 1: Unified component architecture

Each gate provides access to a specific service by binding it to one or more concrete methods implemented by the component. Hence, the gate interface of a component describes the set of possible joinpoints, which can be employed in both a regular and aspect-oriented fashion. For this, a gate acts as a two-way communication channel. Incoming gate communication has following semantics: *execute the service of the component the gate provides access to.* As such, when triggering the behaviour of a gate, the internal methods to which this gate is bound, are executed. Outgoing gate communication on the other hand has following semantics: *whenever the service of the component the gate provides access to is executed, trigger some additional behaviour.* This *additional behaviour* depends on the services of the other gate(s) the outgoing communication is referring to. Hence, whenever one of the services of a component is executed through its corresponding gate, the gate itself is able to trigger some additional behaviour. The component composition illustrated in figure 1 features three components, namely C_1, C_2 and C_3. Components C_1 and C_2 each expose 3 gates, while component C_3 exposes 2 gates to access its provided services. The specification of how these gates interact (in a regular or an aspect-oriented way) with the gates offered by other components is not specified in the gate itself, but is deferred until the component composition process.

The interaction between gates is described by making use of expressive *connectors*. A connector is responsible for describing the regular and aspect-oriented interactions between one (or more) gate(s), as this interactional description is omitted in both the component and gate specification. Connectors that specify regular component interactions are quite similar to the connectors found in most component models. They are mainly employed for gluing together the independently specified behaviour of two or more gates by resolving possible syntactic incompatibilities concerning method names and argument types. Connectors that specify aspect-oriented interactions are an extension of connectors that describe regular component interactions: additionally they allow specifying how the behaviour of one gate crosscuts the behaviour of another gate. The component composition illustrated in figure 1 employs two connectors in order to specify the interaction between the involved gates. When the behaviour of the first gate of component

C_1 is executed, it automatically triggers an additional interaction through connector A which, depending on the type of this connector, is regular or aspect-oriented.

Deferring the aspect-oriented interaction specification to the component composition mechanism itself (i.e. gates and connectors) has multiple advantages. The reusability of components is increased, as a developer does not need to decide at implementation time whether a component is supposed to interact in a regular or an aspect-oriented fashion. All concerns are implemented as plain components and a connector is responsible for specifying how the interaction between the involved components takes place. As a result, the services offered by a component can be employed in both regular and aspect-oriented fashion at the same time and additionally this allows reusing existing components in an aspect-oriented setting. The next section illustrates the FuseJ language, which is a practical realization of the unified component architecture.

3. FUSEJ LANGUAGE

In this section, the FuseJ architectural description language is introduced. The FuseJ language illustrates a practical realization of the unified component architecture, introduced in section 2, by mapping its ideas and concepts onto a real-world component model. Our first proof-of-concept language targets the *JavaBeans* component model [16]. Although JavaBeans does not provide a full-fledged component infrastructure, it is an ideal testing framework for our first language experiments, as we are not exposed to all the bells and whistles provided by mature component models such as Enterprise JavaBeans [5] and Microsoft COM/DCOM [8]. In this section, a first prototype language for gates as well as for connectors is presented. Keep in mind that at the moment the FuseJ language does not support all possible aspect-oriented interactions among components. Identifying and representing this set of required interactions is subject to future research.

Consider a hotel booking case study which employs three components: the `BookingService`, the `PaymentService` and the `DiscountService` components. The first two components provide a user with regular component features, such as *book a hotel* or *bill a particular amount of money onto a customers credit card*. The latter component provides a number of business specific services such as *calculate a discounted price of n percent*. Business-rules have already been identified as typical examples of crosscutting concerns [4], as their implementation is almost always scattered and tangled with the base implementation of the software system. In addition, these features are volatile to change as their behaviour needs to be added, changed and removed quite frequently in order to adhere to the new business requirements of a company. Using FuseJ, all three presented components are implemented as regular JavaBeans, independent of whether they implement crosscutting or non-crosscutting concerns. In the next two subsections, we illustrate how these components are specified and composed using the FuseJ architectural description language.

3.1 FuseJ Gate Language

The FuseJ gate language provides a set of declarative language constructs that allow augmenting each component

```
1  interface BookingService for BookingServiceComponent {
2
3    gate BookHotel {
4      binds:
5        Float bookHotel(String hotelname);
6      exposes:
7        String inputHotelName = hotelname;
8        Float outputPrice = returnvalue;
9    }
10
11   outputgate ChargeForHotel {
12     binds:
13       void fireChargeRequest(ChargeEvent event);
14     exposes:
15       ChargeEvent chargeEvent = event;
16   }
17
18 }
```

Figure 2: Gate interface for the booking service component

```
1  interface DiscountService for DiscountServiceComponent {
2
3    inputgate Discount {
4      binds:
5        Float getDiscountPrice(Float price, Float percent);
6      exposes:
7        Float inputPrice = price;
9        Float discountPercentage = percent;
10       Float outputPrice = returnvalue;
11   }
12
13 }
```

Figure 3: Gate interface for the discount service component

with a dedicated interface that describes an overview of the functional services it provides. Each gate description is bound to one or more methods provided by a component and explicitly delimits and exposes their internal implementation. Figure 2 illustrates the gate interface description of the `BookingService` component.

The `BookingService` interface groups together two gates, namely `BookHotel` and `ChargeForHotel`. Each gate itself is responsible for specifying whether it can receive incoming communication, trigger outgoing communication or both. The `BookHotel` gate (lines 3-9) allows initiating the execution of its internal functionality from outside the component and is itself able to trigger additional aspect-oriented or component-based interactions. The `ChargeForHotel` gate (lines 11-16) on the other hand, declares itself as an output gate. As such, it is impossible to execute the behaviour of this gate from outside the component. In fact, this gate merely throws an event to charge a customer when he/she books a particular hotel. The `Discount` gate (lines 3-11) of the `DiscountService` gate interface illustrated in figure 3 declares itself as being an input gate. Hence, it remains possible to execute its internal functionality, but the gate itself will not trigger interactions with other gates, for instance to apply additional discounts on the already discounted price.

A gate specification typically consists out of two parts: a *binding* and an *exposition* description. The binding part maps a gate onto one or more methods that are part of the internal implementation of the component. The `BookHotel` gate for instance, is mapped onto the `bookHotel` method. As such, when triggering the behaviour of this gate, the

```
1  interface PaymentService for PaymentServiceComponent {
2
3    gate ChargeAmount {
4      binds:
5        void chargeAmount(String ccnumber, Float amount);
6      exposes:
7        String inputCCNumber = ccnumber;
8        Float inputAmount = amount;
9    }
10
11   gate ReserveAmount {
12     binds:
13       void reserveAmount(String ccnumber, Float amount);
14     exposes:
15       String inputCCNumber = ccnumber;
16       Float inputAmount = amount;
17   }
18
19   outputgate BillingActions {
20     binds:
21       void *Amount(String ccnumber, Float amount);
22     exposes:
23       String inputCCNumber = ccnumber;
24       Float inputAmount = amount;
25   }
26
27 }
```

Figure 4: Gate interface for the payment service component

`bookHotel` method is in fact executed. When a gate is mapped upon multiple methods implemented within the same component, this is either specified by enumerating the involved methods one by one or by making use of a regular expression. This concept is illustrated by the `BillingActions` gate (lines 19-25) of the `PaymentService` component interface illustrated in figure 4. Although a dedicated gate is available for both the `chargeAmount` and `reserveAmount` methods, the `BillingActions` gate is mapped on both. As such, when acting as output gate, this allows to attach some additional gate interactions (for instance logging) whenever a customer is being billed, either by charging or reserving a particular amount of money onto his/her credit card. When acting as an input gate, this allows for multicast abilities, i.e. all methods specified by the gate are executed one by one. The `BillingActions` gate however declares itself as output gate, as generally both charging and reserving a particular amount of money at the same time does not make much sense. The gate *exposition* part describes the properties of a gate, such as the input arguments and return value. This allows to selectively expose those properties that can employed and/or altered when this gate is involved in one or more interactions with other gates. The `BookHotel` gate of the `BookingService` component exposes two properties: the `inputHotelName` property that refers to the name of the hotel that a customer wants to book and the `outputPrice` property that refers to the total price that needs to be charged for booking a particular hotel. The latter property is the return value of the method on which this gate has been mapped and is denoted using the `returnvalue` keyword.

Providing each component with a dedicated gate interface could seem overkill in some cases. This declarative specification however allows one to explicitly expose the services provided by a component and delimits the way other components are able to interact and influence their internal behaviour. As such, the comprehensibility, predictability and analyzability of single components and component compositions is improved. For combining several independently specified components into a working software system, FuseJ connectors are employed.

3.2 FuseJ Connector Language

FuseJ connectors are responsible for declaratively specifying the overall architecture of the application. For this, each connector combines one (or more) gate(s) and describes how these gates should interact (i.e. regular or aspect-oriented interaction). The hotel booking case study requires two connectors in order to fulfil its functionality. The first connector is responsible for charging a particular amount of money onto a customer's credit card when he/she books a hotel. The task of the second connector is to assign a particular discount, depending on business-specific properties. The first connector is a typical example of a regular, component-based interaction. The `BookingService` component throws a charge event and the `PaymentService` component is responsible to take the appropriate actions in order to bill the customers credit card. The specification of this component-based interaction is illustrated in the `BookingPayment` connector shown in figure 5.

```
1  connector BookingPayment {
2
3    execute:
4      PaymentService.ChargeAmount;
5    for:
6      BookingService.ChargeForHotel;
7    where:
8      PaymentService.ChargeAmount.inputCCNumber =
9        BookingService.ChargeForHotel.chargeEvent.visaNumber;
10     PaymentService.ChargeAmount.inputAmount =
11       BookingService.ChargeForHotel.chargeEvent.amount;
12
13 }
```

Figure 5: BookingPayment connector describing a regular component interaction

Each FuseJ connector specifies one (or more) gate interactions(s), which are typically built up out of two parts, namely a *connection* part and a *mapping* part. The connection part (lines 3-6) describes the interaction between the two involved gates. In this case, the `BookingPayment` connects the `ChargeForHotel` gate of the `BookingService` component with the `ChargeAmount` gate of the `PaymentService` component. The interaction specifies that the behaviour of the `ChargeAmount` gate should be executed for each triggering (i.e. throwing of an event) of the `ChargeForHotel` gate. As such, whenever a customer books a hotel through the `BookingService` component, he/she gets charged by employing the `PaymentService` component. The mapping part of the connector (lines 7-11) is used for translating the involved properties of the gates and possibly resolving syntactic incompatibilities, i.e. distinct method name and argument types. In this case, the `chargeEvent` property of the `ChargeForHotel` gate is mapped upon the `inputCCNumber` property and `inputAmount` property of the `ChargeAmount` gate.

The `BookingDiscount` connector, which is an example of an aspect-oriented interaction, is illustrated in figure 6. The purpose of this connector is to assign a discount of 15 percent whenever a customer books a hotel during the Christmas holidays. To enable this crosscutting interaction, the

BookingDiscount connector connects the BookHotel gate of the BookingService component with the Discount gate of the DiscountService component. The interaction specifies that the behaviour of the Discount gate should be triggered around the behaviour of the BookHotel gate. Other aspect-oriented interactions, such as before and after can also be employed. The mapping part of this connector (lines 7-10) illustrates that a gate property should not necessarily be mapped onto another gate property. In this case, the discount percentage property is set on the constant value 15 (line 10). An additional triggering condition is set to this interaction by employing a when clause (lines 11-12): the 15 percent discount should only be attributed when the customer books a hotel during the Christmas holidays. This when clause always expects a Boolean value and in this case employs the ChristmasHolidayDate gate of a DateService component to retrieve the required information. As such, the reusable DiscountService component can be employed in order to realize different kinds of discounts by fine-tuning it in a corresponding connector.

```
1  connector BookingDiscount {
2
3    execute:
4      DiscountService.Discount;
5    around:
6      BookingService.BookHotel;
7    where:
8      DiscountService.Discount.inputPrice =
9        BookingService.BookHotel.outputPrice;
10     DiscountService.Discount.discountPercentage = 15;
11   when:
12     DateService.ChristmasHolidayDate.outputValue;
13
14 }
```

Figure 6: BookingDiscount connector describing an aspect-oriented component interaction

4. FUSEJ EXECUTION MODEL

In order to make the FuseJ language operational, we propose a novel container-based execution model which is compliant with the JavaBeans standard. The core idea behind this model is to enable both regular and aspect-oriented interactions between JavaBeans without having to fall back on specialized weaving mechanisms such as running the application in debug mode, using dedicated class loaders or employing advanced instrumentation APIs such as JPLIS (Java Programming Language Instrumentation Services). At the same time, we still aim at providing a dynamic weaving mechanism which allows attaching and detaching interactions between components at run-time of the application. At the moment, the FuseJ implementation is built up out two main components: the *gate preprocessor* and the *FuseJ runtime execution environment*, whose functionalities are shortly sketched in the following two subsections.

4.1 Gate Interface Preprocessor

Remember that a software engineer is able to augment each component with an interface that describes its provided services through the use of gates. This interface is employed by the FuseJ runtime execution environment to generate a container that hosts a particular instance of that component and enables it to participate in both regular and aspect-oriented interactions. The task of this preprocessor is twofold:

1. Translate each gate specified within the interface towards a dedicated Java 1.5 annotation instance. These annotation instances, which mainly package a set of meta-data, contain all information related to a particular gate, such as its name, type and the values it exposes.
2. Resolve the bindings of gates upon their described component. As each gate can be bound to several methods or events specified using regular expressions, these bindings are statically resolved in order to optimize the run-time performance of a gate.

The generated meta-data is integrated within the component by making use of the Javassist [3] byte-code manipulation tool: gate annotations are attached at the level of the component type and their corresponding bind annotations are attached at the level of methods and events. Notice that this integration process does not alter the semantics of the component itself. The employed process merely integrates meta-data and, unlike weaved-based AOP approaches, does not invasively change the functionality offered by the component. As such, each component becomes a self-contained unit that can be immediately deployed within the FuseJ run-time execution model.

The gate interface preprocessor can either be employed by the component vendor in order to make its component compatible with the FuseJ execution environment or it can be used by a component deployer to document and restrict access to the involved components.

4.2 FuseJ Execution Environment

The FuseJ execution environment is responsible for managing the deployment of both components and connectors. When a component is deployed for the very first time, a dedicated container for this component is automatically generated. For this, the execution environment inspects the annotations that are provided by the self-describing component. This generated container wraps the deployed component, inherently ruling out unsolicited interaction. The container automatically installs its corresponding gates, which act as central access-points to the functionality of the component. The implementation of these gates internally contains their exposed values and provides a mechanism for the attachment and detachment of regular and aspect-oriented interactions. As such, each gate itself is aware of its interactions with other components. After this generation process, the component is activated within the execution environment and ready for communication with other components. Notice that a component, which does not contain gate annotations, can still be automatically deployed within the FuseJ execution environment. The JavaBeans standard describes the way a component should provide its internal functionality. These coding conventions for instance specify the availability of a dedicated getter and setter method for each property of a JavaBean. As such, when no gate interface is retrieved from a component, all methods (including getters-setters) and events, are automatically exposed as gates for this component.

When a connector is deployed within the execution environment, a JavaBean that manages the interaction between the

involved gates is automatically generated. This connector allows to automatically attach and detach itself from a gate and is as such able to influence the interactional behaviour between the involved components at run-time. Note that this connector component also gets deployed within the execution environment, similar to a regular component. Hence, this connector component is automatically wrapped within a container, such that it can again be employed to initiate specific interactions with other components, for instance when it gets attached/detached.

4.3 Prototype Implementation

Currently, a first prototype of FuseJ execution model has been developed. At the moment, tool support includes the gate interface preprocessor, the FuseJ runtime execution environment and an administration GUI window (illustrated in figure 7) that allows introspecting, adapting and deploying components and connectors within the system. In the future, the development of a visual plugin for Eclipse [7] is envisioned. This should further ease the development of component-based applications using FuseJ.

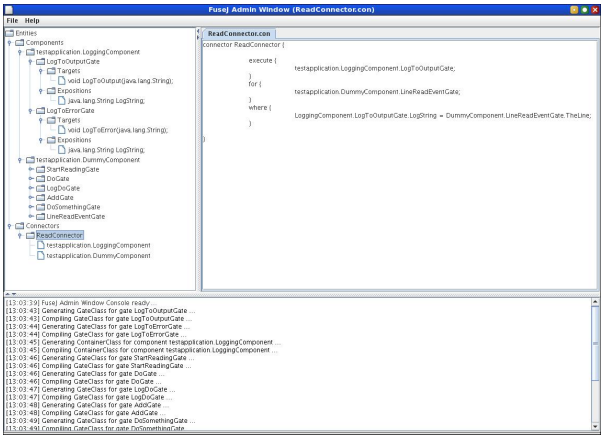

Figure 7: The FuseJ administration window

5. IMPACT ON SOFTWARE ENGINEERING PROPERTIES

This section sketches the impact of the unified component architecture, and in particular the FuseJ architectural description language, on software engineering properties, such as comprehensibility, predictability, evolvability and semantic interactions.

5.1 Comprehensibility

One of the common critiques on AOP is that it makes it more difficult to grasp a total overview of the control-flow within the application as one needs to consider two programming dimensions at the same time. FuseJ aims at easing this development and management process. In the first place, comprehensibility is enhanced by modularizing both crosscutting and non-crosscutting concerns as independently deployable components. Hence, this allows to analyze, comprehend and evolve (non-)crosscutting concerns in total isolation. In addition, components need to explicitly declare their set of possible interaction points (joinpoints) which simplifies the analysis of possible interactions with other components. The declarative FuseJ connector language in turn allows describing the interactions between independent components in a modular fashion. As both regular and aspect-oriented interactions are specified within the same dimension, it becomes easier to comprehend the total control-flow of the application. Each connector describes one specific interaction, which makes it possible to observe this entity in total isolation. The declarative nature of these connectors should also be able to contribute to the semi-automatic detection of possible conflicting semantic interactions. Finally, the FuseJ execution environment explicitly supports the incremental development and integration of aspects within a component-based application: one first builds up a subsystem of the application, and interconnects these separate parts later on in order to compose the total application in an iterative fashion.

5.2 Predictability

In traditional AOP approaches, an aspect itself is responsible for defining the set of joinpoints it wants to advise. As such, there is no implied limit on the expressive power of traditional aspects: as long as one is able to describe the joinpoint, one is able to advise it. In many cases however, this sometimes leads to unforeseen and unpredictable behaviour. When the FuseJ language is however employed, each component itself describes the set of joinpoints that can possibly be advised through the use of gates. As such, other components are only able to initiate communication (both regular and aspect-oriented) on those specific interaction points. Although this limits the expressivity of the FuseJ approach in a way, it makes it easier to predict and analyze the resulting outcome of composition between the involved components.

5.3 Evolvability

FuseJ supports evolvability at the level of the component composition itself, as well as at the level of a single component. The proposed execution model inherently supports the incremental development of component-based applications: by attaching new or detaching existing connectors, possibly at run-time, one is able to add and remove functionalities to the built-up application in a modular fashion. FuseJ also supports the evolution of single components. All concerns, both crosscutting and non-crosscutting, can evolve in total isolation, as no hard links between them are maintained. In general, as long as the interface of the component remains constant, its corresponding gates do not require any adaptation and the component composition should remain valid. In case the original interface of the component is changed, the intermediate gate layer is able to handle these possible syntactic or semantic differences by rebinding gates onto the newly refactored component interface. As such, the component composition remains valid and does not require any adaptation whatsoever.

Also, the use of gates makes it impossible to advise the internal behaviour of the component itself. Although each gate is bound to a particular method implemented by a component, internal calls to these bound methods will not induce the triggering of possible regular and aspect-oriented interactions. As such, even when the internal implementation of a component evolves without changing the components interface, the resulting behaviour of the component compo-

sition remains intact and will not trigger unforeseen component behaviour.

5.4 Semantic Interactions

At the moment, the FuseJ language does not provide explicit language support to detect and resolve possible conflicting semantic interactions between components. When a gate is involved in two or more regular or aspect-oriented interactions, they are handled sequentially, depending on their type and the order in which their corresponding connectors were loaded in the execution environment. Although this mechanism allows a user to define precedence between multiple interactions, it merely provides an ad-hoc solution which should be made more explicit at the language level. Also, the current solution does not provide the expressive power required to describe that some interactions should be ignored when other specific interactions are also applicable. Although still an open issue that needs to be resolved, we are considering an approach that employs a set of dedicated *combinator* components which are, on their turn, able to describe the relationships between the corresponding connector components.

6. RELATED WORK

Several aspect-oriented technologies have been introduced which also aim at not introducing a specialized aspect module. Multi-Dimensional Separation Of Concerns (MDSOC), is one of the first approaches which support the modularization of multiple concerns simultaneously, without one dominating the other [12]. The practical realization of MDSOC is HyperJ for Java [13]. HyperJ captures each concern (crosscutting or not) in a so called *hyperslice*. Similar to FuseJ, HyperJ employs pure Java for describing hyperslices, allowing easier integration of existing components. *Hypermodules* are used to compose a set of hyperslices in order to build up the application. One of the main difference between HyperJ and FuseJ however is that FuseJ concentrates on describing interactions between components, while HyperJ focuses on describing mappings. In many cases, their approach requires components to share common method names and arguments, which easily gives raise to problems when combining independently specified third-party components. At the implementation level, HyperJ merges all mapped hyperslices using byte-code transformations. As such, hyperslices lose their identity at run-time. FuseJ on the other hand, allows components to remain first-class entities, even at run-time. As such, replacing or deleting a component dynamically is still possible.

Composition Filters [2] is an aspect-oriented approach that allows expressing crosscutting concerns by attaching filters to existing classes. ConcernJ [15] is one of the practical realizations of the Composition Filters approach that modularizes all concerns of an application into the *concern* construct. As such, no specific aspect construct exists. The ConcernJ language does however induce a major refactoring of existing code in order to be able to modularize regular classes as concerns. FuseJ however, is backward compatible with regular JavaBeans which can be immediately incorporated into the approach. Furthermore, likewise to HyperJ, ConcernJ invasively alters the concerns in order to insert crosscutting behaviour. This property renders the ConcernJ approach less suitable to be employed in a component-based context.

Invasive Software Composition is a component-based approach that unifies several software engineering techniques, such as architecture systems and generic and aspect-oriented programming [1]. Invasive Software Composition aims at improving the reusability of software components. To this end, software components are equipped with both explicit and implicit hooks. These hooks are composed using a separate composition mechanism. Hence, hooks are similar to the gate concept of FuseJ. FuseJ gates are however only able to depend on the component's public interface, while hooks can be attached at any programming construct. As such, hooks are able to describe a finer level of granularity and the resulting composition has more expressive power. The downside however is that, as the internals of a component are subject to evolution, this could easily break the composition later on. Similar to HyperJ, Invasive Software Composition merges the components into one application at run-time. Undoubtedly, merging components renders a very efficient output but makes components lose their identity.

7. CONCLUSIONS

In this paper, we present the first steps towards a natural unification between aspects and components. In our opinion, a specialized aspect construct is not necessarily required in order to modularize crosscutting concerns. Inherently, the behaviour provided by aspects is not that different from component behaviour: both implement some functionality required within the application and it is only the way in which they interact that differs. Therefore, we propose to employ an expressive aspect-oriented composition mechanism which can be applied upon existing components. In order to support this idea, a unified component architecture is proposed where the internals of regular components are accessed through the use of gates. Homogeneous gates are combined by making use of expressive connectors that describe both regular and aspect-oriented interactions. The FuseJ architectural description language is proposed that realizes the unified component architecture for the JavaBeans component model. A first proof-of-concept implementation of the FuseJ language is available. In our opinion, achieving a unified architecture for aspects and components eases the development and management of component-based applications and has a positive impact onto software engineering properties such as comprehensibility, evolvability and predictability.

The proposed FuseJ gate and connector language is only a first prototype of our ideas. At the moment, the expressiveness of the FuseJ connectors do not cover the complete aspect-oriented composition possibilities. Investigating on how to integrate more advanced joinpoint designators, such as `cflow`, is subject to future research. However, we encountered more involved problems when aspects are modularized as regular components. The `proceed` concept for instance, allows an aspect to specify whether the original method it wraps should still be executed. When implementing aspects as regular components however, a proceed concept is not available. As such, advanced language mechanisms for the connector will need to be developed that enable this kind of specification. However, we always keep mind that our introduced language mechanisms should keep the composition easy to understand, and not make things more complicated. In order to validate our approach, we are considering real-

life experiments, to check whether an unified approach for both aspects and component is really feasible.

8. ACKNOWLEDGMENTS

Davy Suvée and Bruno De Fraine are supported by a doctoral scholarship from the *Institute for the promotion of Innovation by Science and Technology in Flanders* (IWT or in Dutch: *Instituut voor de aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen*).

9. REFERENCES

[1] U. Aßmann. *Invasive Software Composition*. Springer, 1st edition, 2003.

[2] L. Bergmans, M. Akşit, and B. Tekinerdoğan. Aspect composition using composition filters. In Kluwer, editor, *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2001.

[3] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the second International Conference on Generative Programming and Component Engineering*, Erfurt, Germany, Sept. 2003.

[4] M. Cibran, M. D'Hondt, and V. Jonckers. Aspect-oriented programming for connecting business rules. In *Proceedings of the 6th International Conference on Business Information Systems*, pages 1–6, Colorado Springs, USA, June 2003.

[5] S. Denninger and I. Peters. *Enterprise JavaBeans*. Addison Wesley, 1st edition, 2000.

[6] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Reflection*, pages 170–186, Kyoto, Japan, Sept. 2001.

[7] Eclipse Consortium. *Eclipse IDE Framework*. http://www.eclipse.org/.

[8] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1st edition, 1998.

[9] R. Filman. Applying aspect-oriented programming to intelligent systems. In *Proceedings of the ECOOP 2000 workshop on Aspects and Dimensions of Concerns*, Cannes, France, June 2000.

[10] JBOSS Group. *JBoss/AOP website*. http://www.jboss.org.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, pages 220–242, Atlanta, USA, Oct. 1997.

[12] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In Kluwer, editor, *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000.

[13] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.

[14] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac: A flexible solution for aspect-oriented programming in java. In *Proceedings of the 3rd International Conference on Reflection*, pages 1–24, Kyoto, Japan, Sept. 2001.

[15] P. Salinas. *Adding Systemic Crosscutting and Super-Imposition to Composition Filters*. MSc. Thesis, Vrije Universiteit Brussel, Belgium, 2001.

[16] Sun Microsystems, Inc. *JavaBeans(TM) Specification 1.01*. http://java.sun.com/products/javabeans/docs/spec.html.

[17] D. Suvée, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 21–29, Boston, USA, Mar. 2003.

[18] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, Reading, Massachusetts, USA, 1st edition, 1998.

[19] W. Vanderperren, D. Suvée, and V. Jonckers. Combining aosd and cbsd in pacosuite through invasive composition adapters and jasco. In *Proceedings of Node 2003 International Conference*, pages 36–50, Erfurt, Germany, Sept. 2003.

[20] B. Vanhaute, B. D. Win, and B. D. Decker. Building frameworks in aspectj. In *Proceedings of the ECOOP 2001 workshop on Advanced Separation of Concerns*, Budapest, Hungary, June 2001.