Evaluating FuseJ as a Web Service Composition Language

Davy Suvée, Bruno De Fraine, María Agustina Cibrán,
Bart Verheecke, Niels Joncheere, and Wim Vanderperren
System and Software Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{dsuvee,bdefrain,mcibran,tw56218,njonchee,wvdperre}@vub.ac.be

Abstract

With the increasing popularity of web services, a number of technologies have emerged that target the integration and composition of web services as lightweight components. However, a number of problems have been identified in these approaches, for example regarding an overly static integration and lacking support for the modularization of crosscutting concerns. In this paper, we evaluate FuseJ, an architectural description language for unifying aspects and components, as an approach for the composition of web services. We outline how FuseJ can be used to this end and present an evaluation that compares FuseJ to four other web service composition approaches according to criteria such as the organization and flexibility of the composition and the support for Aspect-Oriented Software Development (AOSD). Although FuseJ does not allow describing complete business processes, we find that it excels at selective and dynamic composition and that it supports advanced separation of concerns without the need to introduce additional constructs.

1. Introduction

Web services [1] are a recent application technology based on open standards defined by the World Wide Web Consortium. They enable a number of application components to interoperate in a platform- and language-neutral fashion. Contrarily to what their name suggests, web services are not limited to services offered via the World Wide Web. They are also regarded as a new light-weight component model suited for Component-Based Software Development (CBSD) [18, 19]. The web services technology even improves on current component models: as its communication protocol is entirely standardized, it allows for the seamless integration of components running on middleware platforms of different vendors.

Although web services are a relatively new technology, they are increasingly popular and a large number of supporting technologies exists that enable the development of service-oriented applications. Some shortcomings can however be identified in the approaches that are typically used to compose web services. As references to the interfaces of the used web services are hard-coded in the client applications, the integration does not explicitly support the dynamic reconfiguration of which specific web services are used. This significantly complicates adapting the application to changes in the set of available services in the business environment. As such, the maintainability of applications using web services is compromised [3, 13].

Another shortcoming is that some approaches do not accommodate for the specific needs of web services in terms of management (e.g. services can be asynchronous or latent, can become unavailable due to unpredictable network conditions, etc). To address this issue for several services, code has to be repeated and scattered throughout the application, which renders the development and maintenance of the application more difficult. More generally, it can be said that there is little or no support for the modularization of crosscutting concerns such as management. Aspect-Oriented Software Development (AOSD) [12] appears as a solution for the modularization of such crosscutting concerns. Previous research has shown that the combination of AOSD and web services has significant benefits [3, 20, 4, 21].

In this paper, we aim at resolving these problems by offering a complementary solution from a different context. We evaluate FuseJ, an architectural description language that aims at unifying aspects and components, as an approach for the composition of web services. Although we note that FuseJ was not conceived with web services in mind, we claim that it has a number of properties that make it well suited for this task. More specifically, FuseJ features decentralized and dynamic connections that can flexibly model web service compositions. These connections seamlessly combine regular and aspect-oriented inter-

actions and offer advanced selection mechanisms.

The outline of this paper is as follows. The next section presents the general FuseJ approach for unifying aspects and components by introducing a detailed case study. Section 3 then evaluates FuseJ as an approach for the composition of web services and compares it to four other web service composition approaches. Finally, section 4 wraps up the paper by presenting conclusions and discussing future work.

2. The FuseJ Approach

Nowadays, a wide range of technologies is available that aim at integrating aspect-oriented concepts into a component-based context [15, 9, 11, 8, 17]. All approaches introduce 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 aspect-interfaces. Hence, aspects are considered, treated and implemented as a different kind of entity. However, the behavior implemented by aspects is not that different from ordinary component behavior. Both implement some functionality required within the application and only the way in which they interact with other software entities differs. The introduction of a separate aspect construct however, has several disadvantages. Firstly, the crosscutting composition mechanism of an aspect module is tangled with the behavior of the crosscutting concern, inherently ruling out other ways of integrating its behavior within the application. Secondly, it limits the reusability and applicability of existing components. Instead of introducing a specialized aspect module, we propose to apply aspect-oriented composition mechanisms upon existing module constructs [16]. As such, independently specified components can be deployed in both a regular and an aspect-oriented fashion, achieving a seamless integration between aspects and components.

As a concrete case study to introduce and validate this symmetric AOP approach, called FuseJ, we present an online hotel/apartment booking system, which incorporates four basic components: a `booking` component, a `payment` component, an `SMS` component and a `discount` component. The booking component offers the required functionalities to reserve hotel rooms and apartments. When a customer confirms a reservation, he/she is billed accordingly. Two additional functional requirements are specified: (1) when a customer confirms a reservation, he/she is notified by an SMS-message and (2) when a hotel room is booked during the Christmas holidays, a discount of 10 percent is attributed on the specified room/apartment rate.

2.1. FuseJ Component Model

In order to achieve a seamless integration between aspects and components, FuseJ employs a component model based on *provided-expected* interfaces. Its main objective is to keep coupling amongst components as low as possible, hence achieving maximum reusability and reconfigurability. To this end, FuseJ proposes the concept of a *service specification*. A service specification defines the set of operations (i.e. methods) components should provide and enumerates the set of operations that implementing components *expect* to be offered by the environment. The *provided* and *expected* operations of a service specification are described in terms of regular Java interfaces, making it straightforward to integrate and reuse existing component interfaces within the FuseJ approach.

Listing 1 illustrates the `BookingService` specification (lines 15-18). Components that implement this service specification are required to provide an implementation for both the `BookHotel` (lines 1-4) and `BookApartment` (lines 6-9) interfaces, while at the same time these components can employ the operations described by the `ChargeCustomer` (lines 11-13) interface within their internal implementation. Together, the provided interfaces make up the publicly accessible interface of the component, providing operations that can be employed by other components within the application. The expected interfaces on the other hand, describe the set of interaction points that need to be connected with provided operations offered by other components. As the interfaces themselves are not predefined to be either provided or expected, they can be employed and reused interchangeably: an interface can play the provided role in one service specification, while at the same time play the expected role in another one.

Listing 2 illustrates a component that implements this `BookingService` specification. The `BookingComponent` provides an implementation for all operations described in both the `BookHotel` (lines 6-17) and `BookApartment` (lines 19-30) interfaces. When a customer confirms a reservation, he/she should be billed. The *charging of the customer* is performed through calling the `chargeAmount` operation (line 10, line 23) defined within the `ChargeCustomer` interface. All operations, which are part of the expected interfaces of a component, can be transparently invoked from within the implementation. Hence, the entire implementation of a concrete component is implemented in terms of its own service specification, this way minimizing coupling with other concrete interfaces and components. The FuseJ execution environment ensures that at run-time, the appropriate component behavior is executed for expected operations, depending on the concrete component operations they are connected to. Remember that two

```
interface BookHotel {
  boolean bookHotel(String hotelName, int numberOfNights);
  int computeHotelPrice(String hotelName, int numberOfNights);
}

interface BookApartment {
  boolean bookApartment(String apartmentName, int numberOfDays);
  int computApartmentPrice(String apartmentName, int numberOfDays);
}

interface ChargeCustomer {
  void chargeAmount(int amount);
}

service BookingService {
  provides BookHotel, BookApartment;
  expects ChargeCustomer;
}
```

Listing 1. The BookingService specification

```
class BookingComponent implements BookingService {

  HotelPriceDb hotel_database = new HotelPriceDb();
  ApartmentPriceDb apartment_database = new ApartmentPriceDb();

  public boolean bookHotel(String hotelName, int numberOfNights) {
    // Some management code should come here ...
    // Compute the price for this hotel
    int price = computeHotelPrice(hotelName, numberOfNights);
    chargeAmount(price);
    return true;
  }

  public int computeHotelPrice(String hotelName, int numberOfNights) {
    int rate = hotel_database.getPrice(hotelName);
    return rate * numberOfNights;
  }

  public boolean bookApartment(String apartmentName, int numberOfDays) {
    // Some management code should come here ...
    // Compute the price for this apartment
    int price = computeApartmentPrice(apartmentName, numberOfDays);
    chargeAmount(price);
    return true;
  }

  public int computeApartmentPrice(String apartmentName, int numberOfDays) {
    int rate = apartment_database.getPrice(apartmentName);
    return rate * numberOfDays;
  }

}
```

Listing 2. BookingComponent implementing the BookingService specification

other functional requirements are defined: customers are notified through an SMS message when they confirm their booking and customers are attributed a discount when they book a hotel room during the Christmas holidays. These functionalities are not implemented at the level of the `BookingComponent`, as their implementation would crosscut its base functionality. These so-called *aspects*, or more concretely, these aspect-oriented interactions, are modeled at the FuseJ component composition level, described in detail in the next section, which provides support for both regular as well as aspect-oriented interactions amongst components.

Listing 3 provides an overview of the other set of interfaces and components employed throughout the rest of the case study. The `DiscountComponent` (lines 13-19) illustrates an example of a component that does not employ expected interfaces, as it is able to execute its tasks all by itself, this way reaching maximum cohesion. In that case, a component can implement its providing interfaces directly, similar to a Java class implementing a set of regular interfaces, omitting the description of a service specification. Its implemented interfaces are then implicitly combined into a service specification, which does not contain any expected interface declarations.

The next section describes how independently specified components are composed into a single application by making use of the FuseJ component composition language.

2.2. FuseJ Component Composition Language

Once service specifications and components have been described and implemented, their operations are combined in order to build up an application. For describing this component composition process, FuseJ makes use of an explicit *connector* construct, a concept borrowed from architecture systems [10]. A connector acts as a kind of mediator, which prescribes how two or more components should interact by linking their *provided* and *expected* interfaces. In general, a FuseJ connector is built up out of four individual parts:

- A **target role**, which enumerates the set of provided operations that should be executed, augmented with component selectors.
- A **source role**, which enumerates the set of provided/-expected operations that trigger the interaction, augmented with component selectors.
- An optional **property mapping**, which enumerates the set of property mappings, described in terms of source, target and external operations.
- An optional **condition specification**, which enumerates the set of preconditions, described in terms of source, target and external operations.

As described early on in this paper, the goal of FuseJ is to achieve a seamless integration between AOSD and CBSD. As FuseJ implements both aspects and components as basic components, the distinction between both, namely the way in which their interaction takes place, emerges at the connector level. In its most basic form, a FuseJ connector interconnects two operations. This connection can take place on three levels: the component level, the service specification level or the interface level. Depending on the level that is employed within the connector, a greater degree of flexibility and reuse is achieved.

Listing 4 illustrates a connector, which enables a regular, component-based interaction that connects expected with provided operations. The purpose of this `BookingPayment` connector is to charge a customer's credit card when he/she gets billed for booking a hotel room or apartment. For this, the `chargeCreditCard` operation acts as target role (lines 3-4) and is connected with the `chargeAmount` operation that acts as source role (lines 5-6). The `chargeAmount` operation is declared at the component level. As a result, this interaction takes place when the `chargeAmount` operation is called within the `BookingComponent` implementation. The `chargeCreditCard` operation is declared at the service specification level, namely `CreditCardPayment`. Hence, this operation does not refer to a concrete component type and the FuseJ run-time execution environment is responsible for picking the most appropriate component, which implements this service specification, to handle this operation request. A connector also describes how operation properties (i.e. input parameters and output values) are matched. This mapping of properties takes place by means of a *where*-clause. All operation properties employed within the source and target roles are attributed a unique identifier. When these specified identifiers match in both source and target role, they are automatically reified. When this is not the case, the *where*-clause is able to declare how a mapping takes place. In this particular case, the `amount` property of both the `chargeCreditCard` and `chargeAmount` operation is automatically reified: the value that is given as input for the `chargeAmount` operation, is given as input for the `chargeCreditCard` operation. The `cardnumber` property has no equivalent within other employed operations and is therefore set on *012-34567-89*[1]. Hence, whenever the customer books a hotel through the `BookingComponent`, he/she is automatically billed by the most appropriate component that implements the `CreditCardPayment` service specification.

Next to describing component-based interactions, the FuseJ connector language is also able to specify crosscut-

[1]The number of the customer's credit card could be retrieved by executing some external operation. This is omitted in the case study to keep the example easy and straightforward to understand.

```
interface CreditCardPayement {
  int chargeCreditCard(int amount, String cardNumber);
}

interface Sms {
   void sendSms(String number, String message);
}

interface Discount {
   int applyDiscount(int price, int percent);
}

class DiscountComponent implements Discount {

   int applyDiscount(int price, int percent) {
      return (price - ((price*percent)/100));
   }

}
```

Listing 3. Remaining set of interfaces and components employed within the case-study

```
connector BookingPayment {

   connect:
      CreditCardPayment.chargeCreditCard(int amount, String cardNumber);
   for:
      BookingComponent.BookingService.ChargeCustomer.chargeAmount(int amount);
   where:
      cardnumber = "012-34567-89";

}
```

Listing 4. Connector for billing a customer when he/she books a hotel room or apartment

```
connector BookingSms {

   connect:
      SmsService.sendSms(string number, string message) &&
        host("www.sms-webservice.com") && select("price", "MIN");
   after:
      BookingService.Book*.book*(*);
   where:
      message = "Booked";
      number = "0123456789";

}
```

Listing 5. Connector for sending the customer an SMS-message when he/she confirms a booking

ting interactions, this by declaring the connection type as being crosscutting. At the moment, three kinds of crosscutting interactions are supported, namely *before*, *after* and *around returning*. The *before* and *after* interactions allow to trigger the behavior of an additional operation before or after the execution of some other operation. The *around returning* interaction allows to change the return value of an operation after it has been executed.

Listing 5 illustrates a connector that specifies a crosscutting *after* interaction. This connector makes sure that an SMS-message is sent to the customer when he/she confirms a booking. Similar to a component-based interaction, two operations are again connected, this time through the *after* interconnection type. The `sendSMS` operation is declared at the interface level, while the source role operation is declared at the service specification level. Hence, all concrete components that implement this interface and service specification are taken into consideration by this connector. The *after*-clause (lines 6-7) makes use of quantification: it specifies that those operations declared within the `BookingService` specifications are considered whose name starts with `book` and which are part of an interface whose name starts with `Book`. This kind of quantification allows describing a precise set of operations, without having to enumerate those operations one by one. The target role (lines 3-5) illustrates how the context of an operation is described and/or delimited. For this, *selectors* are employed that declaratively specify the conditions a component should meet in order to be considered. The built-in `host` selector employs the distributed nature of the FuseJ run-time execution environment by defining a remote location for components. In this case, all components that implement the `SmsService` specification and which are on the remote location *www.sms-webservice.com*, are taken into consideration. Another selector declares that the cheapest component is preferred. This kind of requirements are specified by making use of the `select` keyword, which handles custom non-functional properties. For each such property, a dedicated handler is implemented as a FuseJ language plug-in, which is then employed for evaluating the corresponding *select* expression ("price" in this case). Hence, this allows to easily extend the FuseJ language to take into account custom selection criteria.

Listing 6 finally, illustrates a connector that enables an aspect-oriented interaction of type *around returning*. This connector makes sure that the `applyDiscount` operation (lines 3-4) of the `DiscountComponent` is wrapped around the execution of the `computeHotelPrice` operation (lines 5-7), hence returning a new value, namely the *discounted* price. For this, the original return value of the `computeHotelPrice` operation is captured using the `returns` selector. This connector also illustrates how additional preconditions can be specified by using a *when*-clause. Here, one or more operation executions can be combined using boolean operators. Only when this expression evaluates to *true*, the specified interaction is triggered.

This section only introduces a subset of the various expressive composition language features FuseJ has to offer. Amongst others, FuseJ also provides support for the typical aspect-oriented pointcuts such as *cflow*, which are again specified as built-in *selectors*.

2.3. FuseJ Run-Time Execution Environment

In order to deploy applications that have been developed employing the FuseJ component model and composition language, a container-based run-time execution environment is proposed, that enables both regular and aspect-oriented interactions amongst components. Each component is deployed in a dedicated container that automatically exposes the provided and expected component operations. This generated container acts as a wrapper, inherently ruling out unsolicited interactions. The exposed operations contain a run-time mechanism for the dynamic attachment and detachment of interactions. When a connector is deployed within the execution environment, a FuseJ component that manages the specified interaction and selection process is automatically generated and attached at the containers of the involved components, even if these containers are situated on a remote FuseJ-enabled location. Connectors can be dynamically added, changed and removed and are as such able to influence the interactional behavior between the involved components at run-time. For a more elaborate explanation about the run-time execution environment, the interested reader is referred to [16].

3. Evaluation

The FuseJ language was originally not conceived as a web service composition language, but in our opinion it has some features that contribute to web service composition as well. Therefore, we evaluate the FuseJ language as a web service composition language and compare it to the following existing composition approaches: WS-BPEL [2] as the established approach and AO4BPEL [4], CASS [7] and WSML [20] as advanced, AOP-enabled, but less mature approaches. The following paragraphs shortly introduce these approaches, and the evaluation will be carried out by discussing a number of criteria in the next sections. Table 1 provides a summarizing overview of the results of the evaluation.

WS-BPEL (Web Services Business Process Execution Language) is a service choreography and orchestration language that allows the definition of sophisticated web service compositions. It is a process-based language as it describes business processes as interactions between web ser-

```
connector BookingDiscount {

  connect:
    DiscountComponent.Discount.applyDiscount(int price, int percent);
  around returning:
    BookingComponent.BookingService.BookHotel.computeHotelPrice(String, int) &&
      returns(int price);
  where:
    percent = 10;
  when:
    TimeComponent.Timing.isChristmas();

}
```

Listing 6. Connector that attributes a discount when a hotel room is booked during the Christmas holiday

vices. AO4BPEL is an aspect-oriented extension to WS-BPEL that allows for more modular and dynamically adaptable web service compositions. AO4BPEL extends WS-BPEL with the *aspect*, *pointcut* and *advice* concepts.

Contextual Aspect-Sensitive Services (CASS) is a distributed aspect platform that targets the encapsulation of coordination, activity life cycle and context propagation concerns in service-oriented environments. CASS advocates the decomposition of applications into a set of collaboration layers, next to the well-known class, component or service-based decomposition. In a CASS-enabled service oriented architecture, a collaboration layer captures the protocols all services should implement to fulfill an interaction. CASS aspects factor out the crosscutting concerns that arise when services are combined into distinct collaboration layers.

The Web Services Management Layer (WSML) is an AOP-enabled framework for the client-side integration, composition, selection and management of web services. The WSML acts as a mediator between the client application and the world of web services. Its primary goal is to separate any service-related code from the client application to achieve a better separation of concerns. Its secondary goal is to offer dynamic adaptability for the constantly changing service- and network environment. The WSML is implemented on top of the JAsCo [17] dynamic AOP language.

3.1. Seamless AOP

In FuseJ, a web service can be composed with another web service in an aspectual way, i.e. triggered as an advice, whereas the same functionality can also be integrated in a third web service in a non-aspectual way. We say that FuseJ supports *seamless AOP*, as services do not need to explicitly declare at service development time whether they are meant to be used as aspects. As such, every web service can eventually be used in an aspectual way, if they are composed accordingly. FuseJ uses the concepts of connectors and components, which are well known in the world of component-based software development, and does not need to introduce a different aspect concept to deal with crosscutting concerns.

WS-BPEL on the contrary, does not tackle the description of crosscutting concerns in a modularized way. As a consequence, these concerns appear tangled with the specification of the main composition. AO4BPEL improves on WS-BPEL and adds explicit and general support for the modularization of crosscutting concerns, which are encapsulated and defined in terms of typical aspect constructs. Contrary to FuseJ, web services cannot be seamlessly used as advices and an additional aspect construct is required. The price paid by FuseJ is a more limited AOP expressiveness: it cannot support a general *around* advice concept as AO4BPEL does, because this implies that the service is aware that it is used as advice.

The WSML relies on the lower level aspect language JAsCo, which does introduce an explicit aspect construct. However, in the ideal case, the user of the WSML is not exposed to the complexity of AOP as built-in domain specific languages, based on XML, are available for specific concerns. CASS does not offer seamless AOP as the concept of aspects is explicitly present in the programming language.

3.2. Explicit Support for Process Descriptions

Composing web services has two main viewpoints: orchestration and choreography. The choreography viewpoint covers the perspective of a collaboration between several services that realizes a value chain. It describes the interactions between service providers. The orchestration viewpoint describes the behavior that a service provider performs internally within a collaboration.

	FuseJ	WS-BPEL	AO4BPEL	CASS	WSML
Seamless AOP	+	-	+-	+-	+-
Explicit process description support	-	+	+	+	-
Reusable composition	+-	+-	+-	n/a	+-
Selective composition	+	-	+-	+	+
Dynamic composition	+	-	+-	+	+
Automatic discovery	+	-	-	-	+
Compatibility requirements	+-	+	+	+-	+-

Table 1. Overview of the comparison of the evaluated web service composition approaches

FuseJ is not a dedicated process description language. FuseJ connectors are used to interconnect a set of web services. The specification of the process is spread over these connectors and possibly higher-level components. This way, a decentralized composition is deployed. Therefore, FuseJ is proposed as a lower-level composition approach. It does not offer explicit support to maintain state across components or to manage activity contexts, exceptions, transactional integrity, load balancing, etc. WS-BPEL on the other hand is a dedicated language for specifying business process behavior based on web services and is suited for both orchestration and choreography.

WSML is, similar to FuseJ, not targeted at describing process descriptions. Compositions can be specified as a series of higher-level statements or in Java/JAsCo code and are deployed as first-class entities. In the WSML, a hierarchical composition, possibly specified using several layers of abstraction, is deployed in a centralized fashion.

CASS is suited for decentralized choreography and targets specifically the encapsulation of coordination, activity life cycle and context propagation concerns for web services. Similar to FuseJ, CASS does not use a centralized engine to coordinate message exchanges. Instead, coordination logic is integrated directly at the level of the message handlers of each individual web service.

3.3. Reusable Composition

FuseJ connectors refer to interfaces instead of concrete services, which allows reusing them. In addition, a connector encapsulates the deployment information of the employed services, this way avoiding to spread this information all over the composition.

WS-BPEL and AO4BPEL are process-based languages, and thus compositions are written as processes. In both approaches, compositions can refer to partner roles that are to be deployed on concrete services. Thus, compositions can be reused at deployment time for different services that comply with the partner roles. Glue code for data adaptability (adaptation of the interfaces of the composed web services), on the contrary, is not localized but scattered all over the composition, with negative effects on reusability. In the WSML, the compositions are reusable since they can be specified without referencing to concrete service interfaces. CASS aims at providing reusable compositions. However, it is not clear how this goal is currently achieved.

3.4. Selective Composition

An important property in the volatile web services world is whether the selection of web services is not hard coded, but might change depending on business-specific properties. FuseJ supports *selective composition* on two different levels:

1. The target and source roles are able to contain selectors that limit the number of matching services. For example, a service might not be chosen if it is located on a different continent than the source service. These selectors have a declarative nature, can be user-defined and are completely dynamic, i.e. they are re-evaluated for every request[2].

2. The conditional specification allows to describe an additional triggering condition using a Java expression or by invoking another service. As such, in cases where the declarative nature of the selectors is not very well suited, the full expressiveness of Java can be exploited.

WS-BPEL does not explicitly support selective composition, although it can be modeled as part of the process description. However, these kinds of business rules that guide the decision of which service to invoke (e.g. depending on their average speed) have been identified as being crosscutting [6]. As such, the selection logic appears tangled and/or scattered among the process specification and thus seriously hampers maintainability of both the basic process and the service selection logic. In FuseJ, the selection engine takes care of the concrete decision process based on the declarative selector specification. When using BPEL, an instantiation of such an engine (i.e. the selection decision making

[2]This might sound very inefficient, but we expect that smart caching strategies are able to reduce the overhead significantly.

process) has to be included as part of the process description leading to a tangled and thus less readable and maintainable process description.

AO4BPEL allows tackling this by encapsulating these business rules as modularized aspects. However, in AO4BPEL the rules have to be programmed at a lower level in comparison to FuseJ. The declarative nature of the FuseJ selectors allows for a more concise selection description and enables possible optimizations by the FuseJ run-time system by e.g. caching resulting requests for given input properties.

WSML offers support for selection policies that govern the web service selection process. Selection policies specifying selection criteria on non-functional properties can be described declaratively in a domain-specific language and are automatically translated to JAsCo code. FuseJ however integrates the complete selective composition logic in one uniform connector language. Similar to WSML, CASS also supports a system based on selection policies. However, these policies can only rely on anticipated data available in the local message context.

3.5. Dynamic Composition

Selective composition allows choosing between several alternatives for a certain request based on predefined conditions. In the context of Web Services, this might not be enough, as not all possible conditions can be foreseen at deployment time. FuseJ is fully dynamic: it is possible to add, alter and remove connectors while the system is running. Connectors themselves are also highly dynamic since the *selectors* and *condition specification* are re-evaluated for each request.

In WS-BPEL, complex dynamic modifications that might involve semantic changes are not possible. When a WS-BPEL process is deployed, the WSDL descriptions of all the services participating in the composition must be known and once a process has been deployed, there is no way to change it dynamically. Furthermore, because of the centralized setup, it would be very difficult to alter specific parts of a WS-BPEL specification dynamically without affecting the complete composition. FuseJ is however organized decentrally and therefore other interactions are not affected when a connector specification is altered or removed.

AO4BPEL allows attaching and removing aspects dynamically, so the aspect logic itself might be altered during run-time. The basic WS-BPEL workflow description cannot be straightforwardly influenced during run-time. However, technically AO4BPEL is able to alter the basic WS-BPEL workflow description dynamically by patching it using aspects. However, this abuses the aspect expressiveness for concerns that are not necessarily crosscutting. As such, an aspect is not used for capturing a specific crosscutting concern, but merely to patch certain places in the process description to do something else than originally intended. When applying this technique over and over again during the lifetime of the product, the control flow and modularization of the system will be completely cluttered with *patching aspects*. The end result will be a system with a poor modularization and thus less understandable and maintainable. This is in clear contrast with the original motivation for AOSD, namely better separation of concerns.

Both CASS and WSML offer support for dynamic integration and composition of web services through redirection aspects. WSML is able to deploy compositions with a variable set of partners that can be swapped at runtime. CASS proposes to use redirection aspects to implement load balancing and fault tolerance algorithms, although the mechanism can also be used to dynamically change a selected service. While WSML and CASS are framework-based approaches, FuseJ on the contrary introduces explicit declarative language support for capturing dynamism of the selective composition.

3.6. Automatic Discovery

Web service composition approaches in which web services are hard-wired in the clients can easily lead to unmanageable applications that cannot adapt to changes in the business environment. Therefore, one of the goals of web service technology is the *automatic discovery* of web services [5]. FuseJ aims to support such automatic discovery by referring to interfaces instead of concrete web services in a connector specification. At run-time, the concrete web services that match these interfaces can be looked up automatically by the FuseJ run-time environment, thus achieving a high decoupling with concrete web services. Such a look-up can be achieved by using UDDI as a registry for concrete web services.

WS-BPEL, AO4BPEL and CASS do not explicitly support the automatic discovery of web services, although an extended engine could support it. Hence, they could in practice offer more or less the same functionality as FuseJ regarding automatic discovery. The WSML uses the OWL-S language [14] to define a domain ontology for web services and as such allows to verify compatibility of web services on a semantic level [5]. As such, the WSML improves on FuseJ in the sense that it not only considers the concrete interfaces and non-functional properties, but also the semantic description of web services, which is required for full-fledged automatic discovery. As a requirement however, web services have to be documented in OWL-S in order to be employed in such a scheme.

3.7. Compatibility Requirements

Some approaches for web service composition impose specific requirements on the web services they can handle, for example by requiring these services to implement a special model or to be implemented in a dedicated language. Web services that do not satisfy these composition approaches' compatibility requirements cannot be composed using these approaches. Other approaches do not impose such requirements: there, web services can be used as is. In FuseJ, services need to be FuseJ-aware (i.e. deployed on a FuseJ platform and documented using service interfaces) in order to exploit the full power of FuseJ. When a service does not specify a service interface, it is seen as a service with an implicit provided interface, and can thus only be invoked. It is also impossible to apply AO interactions (e.g. to intercept join points) on web services that are not deployed on a FuseJ platform.

In WS-BPEL and AO4BPEL, services do not need to be aware of the fact that they are being used in a composition. Consequently, they do not impose any compatibility requirements on the web services they can handle. In order to allow automatic service discovery, the WSML requires documenting every web service using OWL-S. Similar to FuseJ, CASS requires services to be deployed on a CASS-aware platform in order to allow AO interactions.

4. Conclusions and Future Work

In this paper, we present the FuseJ architectural description language and evaluate its applicability as a web service composition language by comparing it to existing approaches. We identify that FuseJ excels at supporting selective and dynamic composition because of the declarative nature of its language. Furthermore, FuseJ does not need to introduce additional constructs (apart from the existing component and connector Architectural Description Language constructs) for supporting advanced separation of concerns through AOSD. However, FuseJ requires web services to be FuseJ-aware to exploit its full expressive power. In addition, the FuseJ composition language is only able to provide connectivity (although with advanced features), while approaches like BPEL allow describing a full centralized view of the business process.

In order to solve the latter limitation, we plan to investigate the combination of BPEL and FuseJ. Here, BPEL can be used for describing the processes in an abstract, reusable manner while the FuseJ composition language is employed to connect abstract roles with concrete components. As FuseJ was not developed with web services in mind, we did not take the existing web services standards, such as WSDL, into account. In order to make FuseJ more appropriate for the web services world, we plan for instance to incorporate the WSDL standard into FuseJ, either by extending the WSDL language with FuseJ concepts or by providing an automatic translation process for FuseJ service specifications towards WSDL descriptions.

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 in the Industry (IWT). Wim Vanderperren is supported by a post-doctoral fellowship from the Flemish Funds for Scientific Research (FWO).

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer Verslag, 2004.

[2] T. Andrews et al. Business process execution language for web services specification, version 1.1, May 2003. http://www.ibm.com/developerworks/library/ws-bpel/.

[3] A. Arsanjani, B. Hailpern, J. Martin, and P. Tarr. Web services: Promises and compromises. *Queue*, 1(1):48–58, 2003.

[4] A. Charfi and M. Mezini. Aspect-oriented web service composition with AO4BPEL. In *In Proc. of the European Conference on Web Services ECOWS 2004, LNCS 3250*, Erfurt, Germany, Sept. 2004.

[5] M. Cibrán, B. Verheecke, D. Suvée, W. Vanderperren, and V. Jonckers. Automatic service discovery and integration using semantic descriptions in the web services management layer. In *Proceedings of the 3rd Nordic Conference on Web Services*, Växjö, Sweden, Nov. 2004.

[6] M. A. Cibrán and B. Verheecke. Aspect-oriented programming for dynamic web service monitoring and selection. In *In Proc. of the European Conference on Web Services ECOWS 2004, LNCS 3250*, Erfurt, Germany, Sept. 2004.

[7] T. Cottenier and T. Elrad. Dynamic and decentralized service composition with contextual aspect-sensitive services. In *In the proceedings of the First International Conference on Web Information Systems and Technologies*, Miami, USA, May 2005.

[8] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In Yonezawa and Matsuoka [22], pages 170–186.

[9] R. E. Filman. Applying aspect-oriented programming to intelligent synthesis. In C. Lopes, L. Bergmans, M. D'Hondt, and P. Tarr, editors, *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.

[10] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, 1:1–40, 1994.

[11] JBOSS Group. *JBoss/AOP website*, 2005. http://www.jboss.org.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[13] J. Malhotra. Challenges in developing web services-based e-business applications. Whitepaper, interKeel Inc., 2001.

[14] The OWL Services Coalition. *OWL-S 1.0 Release*, 2003. http://www.daml.org/services/owl-s/1.0/.

[15] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In Yonezawa and Matsuoka [22], pages 1–24.

[16] D. Suvée, B. De Fraine, and W. Vanderperren. FuseJ: An architectural description language for unifying aspects and components. In L. Bergmans, K. Gybels, P. Tarr, and E. Ernst, editors, *Software Engineering Properties of Languages and Aspect Technologies*, Mar. 2005.

[17] D. Suvée and W. Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 21–29. ACM Press, Mar. 2003.

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

[19] C. Szyperski. Components and web services. *Software Development*, Aug. 2001.

[20] B. Verheecke, M. A. Cibrán, W. Vanderperren, D. Suvé, and V. Jonckers. AOP for dynamic configuration and management of web services in client-applications. *International Journal of Web Services Research*, 1(3):25–41, 2004.

[21] D. Verspecht, W. Vanderperren, D. Suvée, and V. Jonckers. Jasco.net: Unravelling crosscutting concerns in .net web services. In *Proceedings of the 2nd Nordic Conference on Web Services*, Växjö, Sweden, Nov. 2003.

[22] A. Yonezawa and S. Matsuoka, editors. *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001), LNCS 2192*. Springer-Verlag, Sept. 2001.