# A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development

Davy Suvée, Bruno De Fraine, and Wim Vanderperren

System and Software Engineering Lab (SSEL)
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
{dsuvee,bdefrain,wvdperre}@vub.ac.be

**Abstract.** In this paper, we propose a novel approach towards integrating the ideas behind Aspect-Oriented and Component-Based Software Development. Our approach aims at achieving a symmetric, unified component architecture that treats aspects and components as uniform entities. To this end, a novel component model is introduced that does not employ specialized aspect constructs for modularizing crosscutting concerns. Instead, an expressive configuration language is provided that allows to describe both regular and aspect-oriented interactions amongst components. This paper presents the ongoing FuseJ research, a first experiment for realizing this symmetric and unified aspect/component architecture.

## 1 Introduction

Aspect-Oriented Software Development (AOSD) [11] is a recent software engineering paradigm that aims at improving the separation of concerns offered by present-day software engineering methodologies. A proper separation of concerns is crucial for implementing comprehensible, reusable and maintainable software applications [15]. AOSD research argues that by employing classic software engineering approaches, including Component-Based Software Development (CBSD) [5], the implementation of certain concerns, such as logging, security and caching, cannot be confined into a single logical module. These concerns are called *crosscutting* as their implementation virtually crosscuts the traditional decomposition of an software application. AOSD provides a solution for modularizing these crosscutting concerns by introducing a new modularization entity, called an *aspect*.

Currently, a wealth of technologies are available that all aim at integrating the ideas of both AOSD and CBSD. Examples of such technologies include JAC [16], JAsCo [20], Caesar [14], CAM/DAOP [18], JBoss/AOP [4], AspectWerkz [3] and Spring/AOP [8]. Some AOSD technologies introduce an *asymmetric*, AspectJ-like [10] approach, where crosscutting concerns are implemented through means

of a dedicated aspect language. Other, framework-based AOSD technologies implement aspects through the base programming language. Although framework-based approaches allow for a more straightforward integration of aspects within the standard software development process, they still enforce aspects to implement a set of so-called *aspect* interfaces. Hence, similar to asymmetric AOSD approaches, aspects are still considered, treated and implemented as different kinds of entities within the application. This explicit distinction between aspects and components however induces several disadvantages. Inherently, the behavior provided by aspects is not that different from regular component behavior. Both implement some functionality required within the application and it is only the way in which they interact with the rest of the software system that differs. The crosscutting composition mechanism of current aspect modules however, resides itself tangled with the behavior of the concern, explicitly ruling out other ways of integrating its behavior within the application. In addition, the reusability and applicability of existing software components is constrained. Nowadays, several mature, feature-rich components are available that for instance allow managing the security issues within an application. At the moment however, there is no elegant and straightforward solution available for integrating the behavior of existing components in an aspect-oriented fashion.

The research presented in this paper aims at exploring the possibilities and advantages of introducing a symmetric, unified approach towards combining the ideas and concepts of AOSD and CBSD. Instead of introducing and considering aspects as specialized entities, we propose to apply aspect-oriented composition mechanisms upon the existing component constructs. On the one hand, this allows aspects to straightforwardly adopt the same characteristics of components, namely being reusable and independently deployable while at the same time exposing and adhering to a contractually specified interface [21]. On the other hand, the decision whether components should be integrated in a regular or an aspect-oriented manner can be postponed until component composition time and can easily be changed afterwards.

The remainder of this paper presents the ongoing FuseJ research [19], a first experiment for achieving a symmetric and unified aspect/component architecture. The next section introduces the FuseJ component model and its configuration language by presenting a small case study situated in a *Peer-To-Peer* (P2P) file sharing environment. Section 3 discusses related work. Finally, we present our conclusions and future work.

## 2 The FuseJ Approach

In order to achieve a seamless unification between aspects and components, FuseJ mingles ideas from the AOSD and CBSD world in a simple, expressive component model and introduces a novel configuration language for describing the aspect/component composition. As a small case study, we employ a simplified and partial implementation of a P2P file sharing application. The *download controller subsystem* is responsible for managing the retrieval of shared file

```
1  interface TransferI {
2     byte[] getFileFragment(String aFileName)
3     FileFragementInfo findFileFragment(String aFileName);
4  }
5
6  interface NetworkI {
7     void send(String host, String info);
8     byte[] get();
9  }
10
11 service TransferS {
12    provides TransferI;
13    expects NetworkI;
14 }
```

**Listing 1.** The TransferS service specification

fragments from remote hosts. It features four components, namely `Transfer`, `Network`, `Optimizer` and `Logger`. The `Transfer` component retrieves file fragments and employs the functionalities offered by the `Network` component to communicate with remote hosts. The `Optimizer` component is responsible for optimizing the file fragment transfer strategy depending on several user criteria: one user could be interested in first downloading file fragments that are not very well spread, while other users could be interested in first downloading file fragments from hosts that have a broadband connection. Instead of hard-coding and tangling the logic of these various transfer strategies within the implementation of the `Transfer` component itself, one can better opt for modularizing these strategies as aspects. The next subsections illustrate how FuseJ implements both regular and crosscutting concerns as components and elucidates how the FuseJ configuration language helps at integrating and composing them in the P2P download controller subsystem.

### 2.1 FuseJ Component Model

FuseJ employs a simple, straightforward Java-based component model, built upon the well-known concept of *provided-expected* interfaces. Its main objective is to keep coupling amongst components as low as possible, hence achieving maximum reusability. To this end, FuseJ proposes the concept of a *service specification*. A service specification defines the set of operations implementing components should *provide* to and can *expect* to be offered by the environment in which they are eventually deployed. The *provided* and *expected* operations of a service specification are described in terms of regular Java interfaces.

Listing 1 illustrates the `TransferS` service specification. Components that implement this service specification are required to provide an implementation for operations that are part of the `TransferI` interface, while at the same time they can employ operations that are part of the `NetworkI` interface within their internal implementation. Hence, the set of provided interfaces make up the publicly accessible interface of the component, while the expected interfaces describe the set of interaction points with operations offered by other components.

```
1  class TransferC implements TransferS {
2
3     public byte[] getFileFragment(String aFileName) {
4        FileFragementInfo info = findFileFragment(aFileName);
5        send(info.host(), "get|" + aFileName + "|" + info.filefragement());
6        return get(); }
7
8     public FileFragementInfo findFileFragment(String aFileName) {
9        /* Code for sequential retrieval of file fragments */ }
10
11 }
```

**Listing 2.** The TransferC component implementation

Listing 2 illustrates the simplified implementation of the `TransferC` component that implements the `TransferS` service specification. This component is required to provide an implementation for all operations defined within the `TransferI` interface. Whenever the `TransferC` component is ordered to retrieve a shared file fragment, it employs the `findFileFragment` operation. The default implementation of the `findFileFragment` operation employs a non-optimized download strategy, namely a sequential retrieval of file fragments. When a specific file fragment to download is found, the *expected* operations `send` and `get` are employed in order to retrieve the file fragment from a remote host. All operations that are part of the expected interfaces of a component (e.g. the `send/get` methods) can be transparently invoked from within the component 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 service specifications and components.

The FuseJ component model does not support the language level specification of non-functional properties typically encountered in CBSD systems, such as quality of service, security and life-cycle management. As these kind of non-functional properties have already been identified as being crosscutting [7], FuseJ provides and models these properties as regular components, which are later on composed with specific application concerns in an aspect-oriented fashion. The next section describes how components are composed/integrated into a single application by making use of the FuseJ configuration language.

### 2.2 FuseJ Configuration Language

For describing the component composition process, the FuseJ configuration language makes use of an explicit *configuration* construct, a concept borrowed from architecture systems [6]. A configuration acts as a kind of mediator, which prescribes how two or more components should interact by linking *provided*/*expected* operations. Listing 3 illustrates the structure of a FuseJ configuration entity. Each configuration configures two or more components and the resulting composition again complies with a particular service specification. Each configuration is built up out of one or more *linklets*. Each linklet *links* the operations defined in one or more components and is generally built up out of four individual parts:

```
configuration <name> configures (<comp>|<serv>)+ as <serv> {
    (linklet <linkname> {
        execute|expose : (<compop>|<servop>)+
        for|before|after|around|as : (<compop>|<servop>)+
        (where: (<parameter_mapping>)+)?
        (when: (<compop>|<servop>)+)?
    })+
}
```

**Listing 3.** General structure of a FuseJ configuration entity

- A **target role** that enumerates the set of operations to execute (line 3).
- A **source role** that enumerates the set of operations that act as trigger (line 4).
- An optional **property mapping** that enumerates the set of property mappings, described in terms of source, target or external operations (line 5).
- An optional **condition specification** that enumerates the set of preconditions, described in terms of source, target or external operations (line 6).

As FuseJ implements both regular and crosscutting concerns as basic components in order to achieve unification, the distinction between both, namely the way in which their interaction takes place, emerges at the configuration level. In its most basic form, a linklet links up two operations, either defined at the component or the service level.

```
configuration TransferNetC configures
  TransferC, NetworkC as TransferNetS {

  linklet send {
    execute:
      NetworkC.sendData(Ip ip, String st);
    for:
      TransferC.send(String ho, String st);
    where:
      ip = IpConvertC.convert(ho);
  }

  linklet get { ... }
}
```

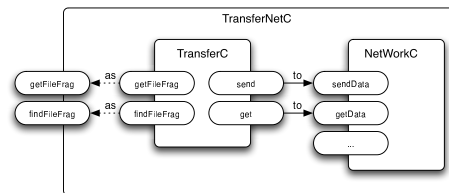



**Fig. 1.** A component-based interaction between the TransferC - NetworkC components

Figure 1 illustrates a configuration that specifies two regular, component-based interactions. It *configures* the `TransferC` and `NetworkC` components as the new `TransferNetC` component that complies with the `TransferNetS` service specification. Two separate *linklets* are employed. The `send` linklet interconnects the `send` and `sendData` operations of respectively the `TransferC` and `NetworkC` components. Hence, whenever the `TransferC` component employs the *expected* `send` operation, the *provided* `sendData` operation of the `NetworkC` component is executed. A linklet also prescribes how operation properties (i.e. input and output parameter) are matched. Properties employed within the source and target

roles of a linklet are specified through a unique identifier. When these specified identifiers match in both a source and target role (e.g. the `st` parameter), they are automatically reified. When this is not possible (because of distinct parameter types), the *where*-clause declares how the mapping takes place (e.g. the `ho` String parameter that gets converted to a parameter of type `Ip`).

In order to comply with the `TransferNetS` service specification, the configuration implicitly exposes the `getFileFragment` and `findFileFragment` operations of the `TransferC` component, although a separate *expose-as* linklet can be employed if required. The newly configured `TransferNetC` component can be employed within other configurations, hence supporting the hierarchical construction of applications.

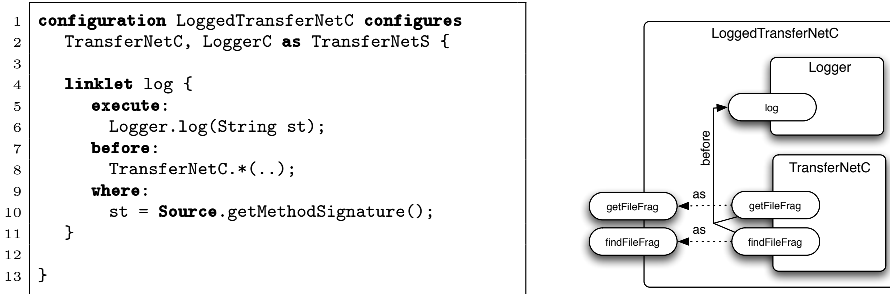
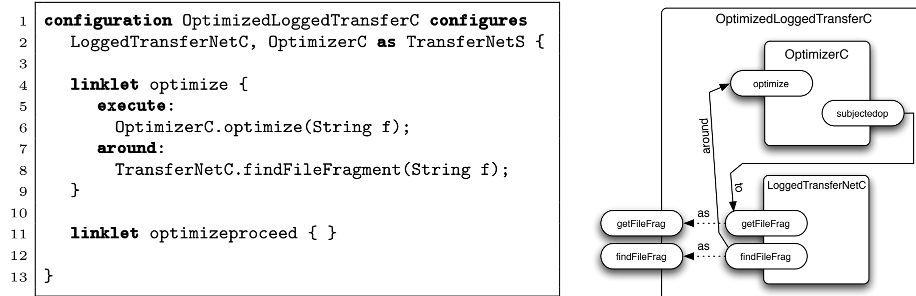



**Fig. 2.** An aspect-oriented before interaction between the TransferNetC - LoggerC components

Next to regular, component-based interactions, a configuration can also describe aspect-oriented interactions, this by declaring the source role as being advised. At the moment, three kinds of crosscutting interactions are supported, namely *before*, *after* and *around*. The *before* and *after* interactions trigger the behavior of additional operations, which act as *advice*, before or after an advised operation. The configuration illustrated in Figure 2 for instance, makes sure that each execution of an operation that is part of the `TransferNetC` component is logged for future reference. For this, *quantification* is employed in order to select the appropriate methods that should be advised by the `Log` operation of the `Logger` component. The *where* clause inits the *st* parameter with the method signature of the triggering operation. For this, it accesses the `Source` object, a component that is the run-time reification of the operation that triggered the interaction (i.e. *join point*). In a similar fashion, `Target` allows to access the run-time reification of the operation that is executed by the interaction.

An *around* interaction wraps and possibly replaces the original behavior of an operation. FuseJ models the continuation of an around advice, which corresponds with the *proceed* concept in asymmetric AOSD approaches, through means of an explicit proceed operation, specified as an *expected* operation. Figure 3 illustrates a configuration that specifies a crosscutting around interaction



```
configuration OptimizedLoggedTransferC configures
  LoggedTransferNetC, OptimizerC as TransferNetS {

  linklet optimize {
    execute:
      OptimizerC.optimize(String f);
    around:
      TransferNetC.findFileFragment(String f);
  }

  linklet optimizeproceed { }

}
```



**Fig. 3.** An aspect-oriented around interaction between the LoggedTransferNetC - OptimizerC components

through its *optimize* linklet. It recuperates the `LoggedTransferNetC` component and wraps the behavior of its `findFileFragement` operation with the `optimize` operation declared by the `OptimizerC` component. Depending on whether the request can be optimized, the original file fragment retrieval behavior of the `TransferNetC` component is either executed or not. For this, the `subjectedop` expected operation of the `OptimizerC` component is back-linked to the advised operation through the `optimizeproceed` linklet.

## 3 Related Work

Several aspect-oriented technologies have been introduced that also aim at avoiding a specialized aspect module. Multi-Dimensional Separation Of Concerns is one of the first approaches that promotes the simultaneous modularization of multiple concerns, without one dominating the other [13]. HyperJ, its practical realization, captures concerns in so called *hyperslices*. *Hypermodules* are used to compose a set of hyperslices in order to build up the application. One of the main differences between HyperJ and FuseJ however, is that FuseJ concentrates on describing interactions between components, while HyperJ focuses on describing mappings. In many cases, the HyperJ approach requires components to share common method names and arguments, which easily gives raise to problems when combining independently specified third-party components.

Invasive Software Composition is a component-based approach that unifies several software engineering techniques, such as architecture systems and generic and aspect-oriented programming [2]. 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. Hooks are similar to the *provided/expected* operations of FuseJ components. FuseJ component operations however, only expose the component's public interface, while hooks can be attached at any programming construct. Hence, hooks support 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 not contractually specified, the composition could easily break later on when the component implementation evolves.

More recently, two approaches, namely FAC [17] and DyMac [12], have emerged that, similar to FuseJ, specifically aim at eliminating the dissimilarities between aspects and components. When FAC and DyMac are employed, software applications are decomposed into regular components and *aspect components*, where an aspect component is a regular component that modularizes the behavior of a crosscutting concern. Similar to FuseJ, dedicated binding constructs are introduced that specify the (crosscutting) interactions amongst individual components. In contrast with FuseJ however, FAC and DyMac do not strive for a full unification between aspect and components. Component methods that are employed as advices still need to comply to a particular set of requirements (for instance method names and argument types), which obstructs a full symmetric model for aspects and components.

## 4 Conclusions and Future Work

In this paper we present the ongoing FuseJ research, a symmetric and unified approach towards combining the ideas and concepts of aspects and components. To this end, the FuseJ research introduces a novel component model that does not employ specialized aspect constructs for modularizing crosscutting concerns. Instead, aspect-oriented composition mechanisms are provided through means of an expressive component configuration language that allows to describe both regular and aspect-oriented interactions amongst components. Next to the features described in this paper, the FuseJ configuration language also provides support for more advanced aspect-oriented mechanisms including more involved *pointcut* designators such as *cflow*, *dynamic triggering conditions* and *aspectual polymorphism*. A first prototype implementation of the FuseJ component architecture is available.

Although the FuseJ unified aspect/component architecture yields several advantages, some aspect-oriented encapsulation and composition techniques still need to be integrated in order to achieve full AOSD expressiveness. For instance, the integration of *aspect precedence/combinations* still needs to be examined. In addition, experiments will be conducted that investigate the applicability of aspects at the architectural level itself.

## References

1. M. Akşit, editor. *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*. ACM Press, Mar. 2003.
2. U. Aßmann. *Invasive Software Composition*. Springer, 1st edition, 2003.
3. J. Bonér and A. Vasseur. AspectWerkz: simple, high-performant, dynamic, lightweight and powerful AOP for Java. Home page at `http://aspectwerkz.codehaus.org/`, 2004.

4. B. Burke et al. JBoss Aspect-Oriented Programming. Home page at `http://www.jboss.org/products/aop`, 2004.
5. F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In Kiczales [9], pages 65–75.
6. D. Garlan and M. Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, 1:1–40, 1994.
7. S. Göbel, C. Pohl, S. Röttger, and S. Zschaler. The COMQUAD component model: enabling dynamic selection of implementations by weaving non-functional aspects. In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 74–82. ACM Press, Mar. 2004.
8. R. Johnson et al. *Spring Java/J2EE Application Framework, Reference Documentation*, 2004. Available at http://www.springframework.org/docs/spring-reference.pdf.
9. G. Kiczales, editor. *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*. ACM Press, Apr. 2002.
10. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.
11. 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.
12. B. Lagaisse and W. Joosen. Component-based open middleware supporting aspect-oriented software composition. In *Proceedings of CBSE 2005*, pages 139–254, St. Louis, USA, May 2005.
13. 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.
14. K. Ostermann and M. Mezini. Conquering aspects with Caesar. In Akşit [1], pages 90–99.
15. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, Dec. 1972.
16. R. Pawlak, L. Seinturier, L. Duchien, L. Martelli, F. Legond-Aubry, and G. Florin. Aspect-oriented software development with Java Aspect Components. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 343–369. Addison-Wesley, Boston, 2005.
17. N. Pessemier, L. Seinturier, T. Coupaye, and L. Duchien. A model for developing component-based and aspect-oriented systems. In *Proceedings of the 5th International Symposium on Software Composition*, Vienna, Austria, 2006.
18. M. Pinto, L. Fuentes, M. Fayad, and J. M. Troya. Separation of coordination in a dynamic aspect oriented framework. In Kiczales [9], pages 134–140.
19. 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.
20. D. Suvée and W. Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In Akşit [1], pages 21–29.
21. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* 1st edition, 1998.