Views for Aspectualizing Component Models

Abdelhakim Hannousse [*]
Ascola Team, Ecole des Mines de Nantes, Inria, Lina
abdel-hakim.hannousse@emn.fr

Gilles Ardourel
Coloss Team, Université de Nantes, Lina CNRS UMR62441
Gilles.Ardourel@univ-nantes.fr

Rémi Douence
Ascola Team, Ecole des Mines de Nantes, Inria, Lina
Remi.Douence@emn.fr

Abstract

Component based software development (CBSD) and aspect-oriented software development (AOSD) are two complementary approaches. However, existing proposals for integrating aspects into component models are direct transposition of object-oriented AOSD techniques to components. In this article, we propose a new approach based on views. Our proposal introduces crosscutting components quite naturally and can be integrated into different component models.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures–*Languages*

General Terms Aspect-Oriented Software Development, Component Based Software Development.

Keywords Aspectualization, VIL, Views, Crosscutting wrappers

1. Introduction

Component based software development (CBSD) and aspect-oriented software development (AOSD) are two complementary approaches: while CBSD focuses on the modularity and the reusability of software systems by assembling components [10], AOSD focuses on the modularity of crosscutting concerns [7]. However, existing proposals for integrating aspects into component models are direct transposition of object-oriented AOSD techniques to components. Moreover, current proposals consider only specific component models and do not address the issue on its general form. Furthermore, most of them are unable to handle both integration and interaction of aspects. In this article, we contribute by proposing a new approach based on views. A view is defined as a reconfigured component architecture by introducing new composites encapsulating some of its original components. These new composites can then be wrapped to alter the behavior of their inner components. Views can be integrated into different component models. In this paper we show how views can be used for Fractal component model [3]. We also introduce a language for views, we call VIL, that makes integrating views and wrappers into a component architecture more expressive. However, integrating aspects following views consideration introduces crosscutting wrappers (i.e. crosscutting aspects). In this paper we highlight crosscutting wrappers issue and discuss the need of a formal specification of both components and wrappers behavior in order to detect and tackle their interaction issue.

The rest of this paper is organized as follows: section 2 describes a motivating example that we use to demonstrate how views are powerful enough to describe aspects. Section 3 introduces our language for views VIL. Section 4 shows how VIL can be integrated into Fractal component model. Section 5 discusses wrappers interactions and how VIL could contribute for detecting conflicting views. Section 6 reviews related work and section 7 concludes and discusses our key perspectives.

[*] Partially funded by the Miles Project

2. Motivating Example

In this section, we show with an example how views enable the integration of aspects into component architectures. Our example is a revised version of the one given in [2]. It describes a software controller of a crane that can lift and carry containers from arriving trucks to a buffer area or vice versa. The crane system is composed of an engine that moves the crane left to the truck and right to the buffer area, a mechanical arm that moves up and down and a magnet for latching and releasing containers by activating and deactivating its magnetic field. The engine and the arm may run in two different modes: *slow* and *fast*. Users interact with the crane using a control board. The control board allows users to choose a running mode for the crane and start crane loading or unloading containers. Figure 1 and figure 2 depict, respectively, a schematic overview and a possible component architecture of the crane system.

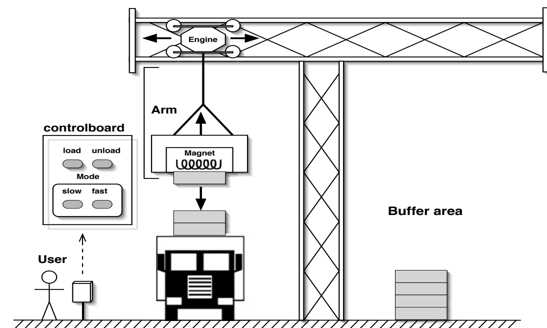

Figure 1. A Schematic Overview of the Crane System

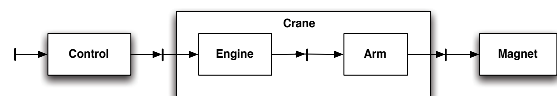

Figure 2. The Crane System Architecture

In Figure 2, components are depicted by rectangles and provided and required interfaces are represented by input and output arrows, respectively. Figure 2 models the crane system as a component architecture with three main components: *controller, crane* and *magnet*. The controller component provides an interface that permits to

set the running mode of the crane and start loading and unloading containers. Upon receipt of user commands, the control component transforms those commands into signals and requires the crane to act following those signals through its required interface. The crane component is a composite of the *engine* and the *arm* components. The engine component provides an interface that permits to move the crane left and right following a running mode and requires an interface to call the arm to move up and down. The arm, in turn, provides an interface for moving up and down following a running mode and requires an interface to ask the magnet to latch or release a container. Finally, the magnet component provides merely an interface for latching and releasing containers.

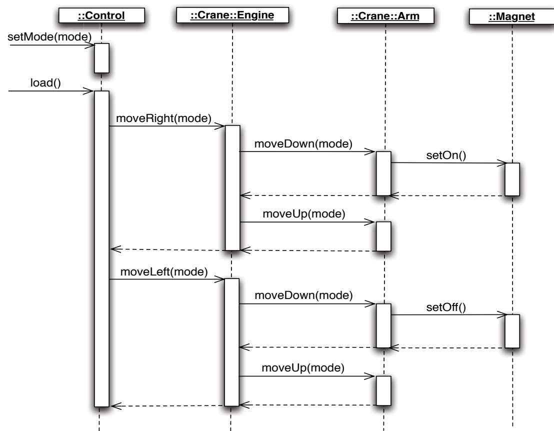

Figure 3. Loading Process for the Crane System

Figure 3 shows the UML sequence diagram of loading a single container. The process of loading a container starts when the user sets the running mode for the crane and presses the load button on the control board. These two actions are transformed into calling *setMode* and *load* operations, respectively, on the provided interface of the control component. When the control component receives a load call, it requires the engine component to move right by calling *moveRight* operation on its required interface. Upon receipt of *moveRight* call, the engine does the action and requires the arm to move down by calling *moveDown* operation. The arm accepts the call, moves down and asks the magnet to latch a container from the buffer area by calling *setOn* action on the arm. When the container is latched, the engine calls the arm to move up throwing a *moveUp* call. When all this done, the control requires the engine to move left to the truck by calling *moveLeft* operation. The engine receives the call, asks the arm to move down which in turn asks the magnet to release the latched container by calling *setOff* action.

2.1 An Optimized Crane System

Now we want to enhance the functionality of the crane system by forcing it to fulfill the following constraints:

C1 *When the arm is not carrying a container, the crane should run in fast mode.*

C2 *When the crane is loading a container on the truck, the arm should move down slowly.*

It is obvious that running the crane in fast mode when the arm is not carrying a container enhances the performance of the crane. Moreover, moving the arm slowly when it is carrying a container to be released on the truck ensures the safety of the truck. We call the above constraints *performance* and *safety* constraints, respectively.

In the following, we show how views can be used in order to force the crane system to fulfill the above constraints.

In this article, we use the term *view* to refer to a component architecture with additional composites encapsulating some its original components. We also use the term *wrapper* to refer to each entity that surrounds a component, intercepts calls on its provided and required interfaces and may alter its behavior.

Views implementation differs from one component model to another. As an example, a view in Fractal component model can be implemented as a controller associated to a composite that acts when calls are intercepted on its interfaces.

2.1.1 Fulfilling Performance Constraint

The crane system can be forced to fulfill the performance constraint by adding a wrapper around the engine and the arm components. The added wrapper intercepts calls on the provided interfaces of the engine and required interfaces of the arm. The wrapper stores and updates the state of the magnet whenever *setOn* and *setOff* operations are called. Thus, whenever the wrapper intercepts *moveLeft* and *moveRight* calls, it first checks the stored state of the magnet; if the state of the magnet is off it forces the engine to run in fast mode by proceeding the intercepted call with fast as a value of its parameter.

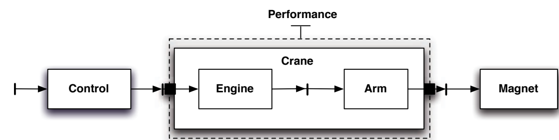

Figure 4. Performance View

Since the engine and the arm already belong to the same composite, the performance wrapper can be integrated at the crane composite level, which gives the first view of the crane system as shown in figure 4. This view is equivalent to the basic architecture with the exception of adding a wrapper to a composite level. The wrapper in figure 4 is presented as dashed border rectangle around the crane component. The small dark squares in the figure indicate the intercepted interfaces. We use the same notation for all the wrappers described in this paper.

2.1.2 Fulfilling Truck Safety Constraint

Considering truck safety in the crane system can be made by integrating a wrapper around the control and the engine components. This way, the integrated wrapper will intercept calls on provided interfaces of the control and required interfaces of the engine. The wrapper stores and updates the state on which the control is under loading or unloading a container. So that, whenever the second call of *moveDown* is intercepted, on the required interface of the engine, and the control is being loading a container it proceeds the *moveDown* call in slow mode.

In this case, we need another view of the component architecture of the crane where the control and the engine are encapsulated in the same composite. Figure 5 shows this required view.

Views make it simple to fulfill either constraints **C1** and **C2** shown above. However, when we consider both constraints, wrappers crosscut each other as shown in figure 6. It is obvious that the structure of the component system must be transformed in order to enable both wrappers at the same time. In the following, we introduce a specialized language for views definitions and show how it can be integrated with fractal in order to weave crosscutting wrappers.

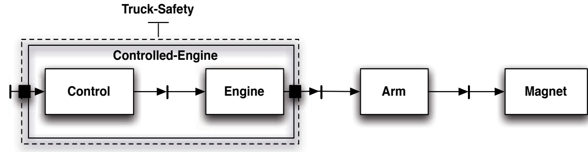

Figure 5. Truck Safety View

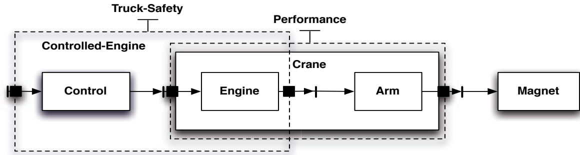

Figure 6. Wrappers Crosscut Phenomenon

3. VIL: Views Language

In this section we introduce a specialized language we call VIL for managing views in component models. Views can be specified using VIL to deal with the integration of wrappers into component architectures. We start by reviewing FPath language [5], used in VIL to access the required components which are going to be integrated into the same view.

3.1 FPath Query Language

FPath is a query language developed to deal with the introspection of the Fractal component architectures [5]. FPath uses declarative path expressions to introspect Fractal elements: components, interfaces and attributes.

$$engPath = \$root/child :: *[name(.) = crane]/child :: *[name(.) = engine]$$

For example, $engPath$ is an FPath expression that provides an access to the engine component in the architecture given by figure 2. This expression is divided into three steps separated by "/". The first step "$\$root$" indicates a value of an FPath variable to denote the component representing the root of the crane system. This later is considered as an input to the next step. The second step "$child :: *[name(.) = crane]$" takes the root component, denoted by the previous step, checks all its inner components "$child :: *$" and selects the one who has the name crane "$[name(.) = crane]$". The third step "$child :: *[name(.) = engine]$", which is similar to the second step, starts from the crane, denoted by the previous step, and provides an access to the engine component by checking all its inner components and selects the one who has the name engine. Similarly, $crnPath$ and $ctrPath$ provide accesses to the crane and the control components in the crane system architecture, respectively.

$$\begin{aligned} crnPath &= \$root/child :: *[name(.) = crane] \\ ctrPath &= \$root/child :: *[name(.) = control] \end{aligned}$$

3.2 VIL Language

Now we describe the views introduced in section 2.1 using VIL. As described in section 2.1.1, performance view wraps the crane component, intercepts all its provided and required interfaces. This can be expressed in VIL as follows:

$$V_1 = \mathbf{view}\ crnPath$$

In VIL, the $\mathbf{view}$ keyword defines a view for a component architecture by wrapping the component described by $crnPath$ expression and intercepts all its provided and required interfaces.

Besides $\mathbf{view}$ keyword, $\mathbf{req}$ and $\mathbf{prov}$ keywords are used to define views by wrapping a component and intercept all its required and provided interfaces, respectively. Moreover, a wrapper may be interested to intercept calls on only some interfaces of a component, in this case, we use the "c $\mathbf{except}$ s" expression to indicate that the corresponding wrapper intercepts all the interfaces of the component c except those defined in s where s is a set of interface names.

In the case where the components that are going to be wrapped do not belong originally to the same composite, different sub-views should be defined each of which wraps one component and intercepts only its concerned interfaces. For example, in the truck safety case, the control and the engine components do not belong to the same composite; so, we need to define two sub-views, one to wrap the control and intercept all its provided interfaces and a second to wrap the engine and intercept all its required interfaces. These two sub-views can be defined in VIL as "$\mathbf{prov}$ $ctrPath$" and "$\mathbf{req}$ $engPath$" respectively. The complete view can be defined by composing sub-views using predefined views composition operators. For truck safety case, the two above sub-views can be composed using the "$\sqcup$" (i.e. *union*) operator. The result view describes the act of introducing a composite that wraps all the components defined by all its sub-views and intercepts all the interfaces intercepted by all its sub-views. The following is the complete VIL expression describing the truck safety view:

$$V_2 = \mathbf{prov}\ ctrPath \sqcup \mathbf{req}\ engPath$$

Besides "$\sqcup$" operator, "$\sqcap$" and "$-$" operators are used to describe intersection and difference operations on views. These three operators are used to extend the scope of wrappers, to determine conflicts on wrappers and to separate the scope of one wrapper from another in views, respectively. These operators are inspired by those defined in set theory. The following is the complete syntax we propose for the VIL language:

$$\begin{aligned} v \in View \quad ::= & \quad \mathbf{view}\ e \mid \mathbf{req}\ e \mid \mathbf{prov}\ e \mid v_1\ \mathbf{except}\ s \\ \mid & \quad v_1 \sqcup v_2 \mid v_1 \sqcap v_2 \mid v_1 - v_2 \end{aligned}$$

VIL is portable, declarative and robust language. VIL is portable because it does not depend on a specific component model, it is an independent language which can be integrated into different component models. We will show later in the next section how VIL can be integrated into Fractal component model. VIL inherits its declarative property from the FPath language [5]. Moreover, views can be composed using a set of declarative operators which enable programmers to define new abstractions (such as *controlled-engine*) on component architectures. Finally, when a component architecture is reconfigured, some views definitions may remain valid. For example, adding a new component between the engine and the arm components on the architecture depicted in figure 6 does not alter neither the performance nor the truck-safety views. Of course, arbitrary modifications of component architectures may also break views.

4. VIL Mapping to Fractal

In this section, we show how VIL can be integrated into Fractal Component Model [3]. We suppose here that the reader is familiar with Julia implementation of Fractal and Fractal-ADL. Fractal uses an Architecture Description Language (ADL) to describe component architectures. It supports hierarchies, introspection and com-

ponent sharing. We distinguish two cases for views mapping: the first case is when the components to be wrapped are directly related to each other and already belong to the same composite. Here we need just to associate a controller to that composite in order to intercept its interfaces and implement the wrapper behavior.

The second and more interesting case is when the components to be wrapped do not belong to the same composite or their are not directly related to each other. In this case view mapping is divided into two steps. The first step consists in finding the closest common parent of the components to be wrapped. This can be done using FPath language: Consider c_1 and c_2 two different components, the following FPath expression provides a set of all their common parents including the root component:

$$e = c_1/ancestor :: *[in(c_2/ancestor :: *)]$$

The "$c_1/ancestor :: *$" sub-expression returns the set of all the ancestors of c_1 including the root component. With the predicate "in" presented between square brackets, only the ancestors of c_1 that belong to the set of ancestors of c_2 will be returned. The closest parent c belongs to that set and has the following particularity: $descendant(c) \cap e = \phi$ which means that the descendants of the closest parent do not belong to the set returned by e.

The second step consists in adding a new composite as an inner component of the common parent of c_1 and c_2 found by the previous step. The new composite declares c_1 and c_2 as its inner components sharing them with their common parent. This way, the original architecture is not affected by views integration. Integrating a view means associating a controller to each shared component. The added controller intercepts calls and route them to the nesting composite. Figure 7 shows how the performance and truck safety views are integrated into the Fractal component architecture of the crane system. In this figure, the performance view is integrated following the first case and truck safety view is integrated following the second case. The component architecture transformation becomes a tedious and error prone task when the architecture grows. Our approach makes it possible to automatize this task.

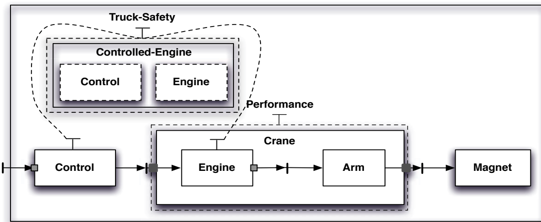

Figure 7. Views in Fractal Component Model

Figure 8 shows the equivalent Fractal-ADL code of the architecture given in Figure 7. The underlined lines of code is the ones that can be generated automatically as a result of analyzing the following VIL expression that describes truck safety view:

$$V_3 = \mathbf{prov}\ (\$root/control) \sqcup \mathbf{req}\ (\$root/child :: crane/child :: engine)$$

```
<component name=root>
  <component name=control>
    ...........
    <controller name=prov>
  </component>
  <component name=crane>
    <component name=engine>
      ...........
    <controller name=req>
    </component>
    ...........
    <component name=arm>
      ...........
    </component>
    <binding ...........>
    <controller name = performance>
  </component>
  <component name=controlled-engine>
    <component name=control definition=/control>
    <component name=engine definition=/crane/engine>
    <controller name=truck-safety>
  </component>
  <binding ......>
</component>
```

Figure 8. Views Integration into Fractal-ADL

5. Wrappers Interactions

We have shown how C1 and C2 constraints can be satisfied by introducing wrappers. We have also shown how both wrappers implementing C1 and C2 can be introduced at the same time in an automatically transformed architecture. In this case, the intercepted interfaces by both wrappers are disjoint and they are not in conflict with each other. However, this is not a general rule. So, we cannot consider that two wrappers are not in conflict just because they do not intercept common interfaces. As counterexample, let us consider the following *saving energy* constraint:

C3 *After carrying a thousand of containers in a day the arm should run in slow mode.*

Saving energy view requires the control and the engine components to be in the same composite. This time, the wrapper intercepts the provided interfaces of the control and the required interfaces of the engine. When load and unload calls are intercepted, the wrapper updates the number of carried containers. When the threshold is reached, the wrapper forces all the subsequent calls of moving the arm up and down to be in slow mode.

Consider the intercepted calls by the wrappers implementing C2 and C3, respectively. They are not disjoint, but when the wrappers intercept common calls (i.e. *moveUp* and *moveDown*) they agree to run them in slow mode. Indeed, C2 forces the arm to move down slowly in some cases (i.e. when it is loading the truck) and C3 forces the arm to move up and down slowly in some cases (i.e. when the threshold is reached). So, when the wrappers implementing C2 and C3 are applied at the same time, both constraints are once again satisfied.

Now consider the case of C1 and C3. The intercepted calls by their wrappers are disjoint. However, when both wrappers are applied at the same time, both constraints are not satisfied. In fact, both constraints can not be satisfied. Indeed, while the performance view forces the crane (and the arm) to run in fast mode when the arm is empty, the saving energy view forces the arm to run in slow mode once the threshold number of carried containers is reached. The exact behavior at run time depends on the implementation. Possible outcomes are:

- only one constraint is satisfied, because the first wrapper to be applied overrides the second one
- only one constraint is satisfied, because the second wrapper to be applied overrides the first one
- or worse, none of the two constraints is satisfied, because the implementation interleaves wrappers code.

Unsurprisingly, these conflicts are similar to aspect interactions. We believe that a support for conflicts detection and resolution is mandatory for aspectualizing component models. It is simple in VIL to detect views intersections. But as we have seen, this information is not sufficient in general to detect conflicts. Related work on aspect interactions [12] is a good starting point for future study. We also believe that component models offer properties such as protocols or contracts that could help in conflict detection. Finally, the notion of views could also help to specify what a conflict is and how it can be solved. For instance if a wrapper introduces transactions, we could specify that nested wrappers (*i.e.* nested transactions) are not allowed, or we could also declare that it is allowed to automatically extend the scope of a wrapper (*i.e.* it wraps more components) in order to expand the corresponding transaction.

6. Related Work

Many works are dedicated to aspectualize component models. However, most of them are interested in a specific component model and all of those works have failed to satisfy the two following requirements: (1) integrate aspects into component models in a natural way and (2) handle aspects interactions. In our opinion, their failure is due their lack of expressiveness as well as their lack of a formal model to analyze and verify properties on the result aspectualized architectures.

Some of the proposals to aspectualize component models (*e.g.*, FAC [8], FRACTAL-AOP [6], SAFRAN [4]) propose to extend component models with aspect-oriented concepts. Others (*e.g.*, FuseJ [9] and CaesarJ [1]) introduce new component models. To the best of our knowledge, all of them directly transpose object-oriented AOP concepts into existing CBSE. In particular, they rely on AspectJ-like pointcut expressions to define where aspects weave components. Our approach relies on alternative views to get rid of the tyranny of the primary decomposition and naturally introduces crosscutting at the level of components.

In all models but JAsCo, aspects are components. Currently in our proposal a wrapper is not always a component. When an aspect is a component, this promotes aspects reuse and enable to consider aspects of aspects. It should be studied how our approach can be extended in order to consider aspects of aspects. In the other hand, no aspectualized component model but JAsCo, proposes conflict detection support (beyond AspectJ-like detection of overlapping crosscut). JAsCo offers an API dedicated to compose aspects in a programmatic way. Our approach introduces crosscutting at the component level and could help to study interaction (*e.g.*; detect when two wrappers intersect, or when a wrapper is nested into another).

Unlike AspectJ-like pointcut expressions [7], VIL expressions are declarative and AspectJ pointcuts are imperative. This can be shown through the ability of VIL expressions to specify a pointcut for different joinpoints without so much care about the actions to be executed for each joinpoint. In the case of AspectJ, pointcuts and advices are strongly related. Moreover, VIL expressions are not used only to specify joinpoints but also to reconfigure component architectures in a way that wrappers can be integrated at the right positions. Our proposal can also be compared with Composition Filter model (CF) [2, 11] in the sense that each wrapper can be shown as an interface layer with input and output filters surrounding a component. However, views address more general concerns than those specified as filters. Moreover, according to the CF model presented in [11], filters can only be associated to only one component where a wrapper may alter more than one component. Furthermore, even if filters can be generalized to wrap many components it will be difficult to those filters to wrap components at different levels of hierarchies and share states on those components.

7. Conclusion and Future Work

In this paper we proposed VIL. A specialized language for aspectualizing component models. It relies on the concept of views that alter the basic component architecture by introducing new composite components. These extra composites can then be wrapped in order to intercept their interfaces and alter their basic behaviors for satisfy extra constraints. We have proposed a declarative language to define views. Our language do not rely on a specific component model. We have shown how to implement VIL in Fractal component model. Finally, we have discussed views interactions. Indeed, several views may share components and interact at common intercepted interfaces. This may lead to a conflict between views and violation of their satisfied constraints. However, views that do not share components may also interact. As future work, we are interested in providing a mechanism for conflicts detection and resolution. For conflict detection, both components and views behaviors should be considered. Each view should be associated with one or more constrains, then the compatibility of constraints associated to each pair of views should be checked to see whether or not they are in conflict with each other. For conflict resolution many strategies can be considered. We can mention as examples: associate priorities to views and define rules for views applications (e.g. when *v1* is applied *v2* cannot be applied).

References

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I (TAOSD I)*, vol. 3880 of LNCS, pages 135-173. Springer, 2006.

[2] L. Bergmans and M. Akşit. Composing synchronization and real-time constraints. *Journal of Parallel and Distributed Computing*, 36(1): 32-52, 1996.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software- Practice and Experience*, 36(11-12):1257-1284, 2006.

[4] P. C. David and T. Ledoux. Towards a Framework for self-adaptive component-based applications. In *Distributed Applications and Interoperable Systems*, vol. 2893 of LNCS, pages 1-14. Springer, 2003.

[5] P. C. David, T. Ledoux, M. Léger, and T. Coupaye. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annales des Télécommunications*, 64(1-2):45-63, 2009.

[6] H. Fakih, N. Bouraqadi, and L. Duchien. Aspects and Software Components: A case study of the Fractal Component Model. In *Proceedings of the International Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, 2004.

[7] G. Kiczales and M. Mezini. Aspect-Oriented Programming and Modular Reasoning. In *Proceedings of the 27th international conference on Software engineering (ICSE'05)*, pages 49-58. ACM, 2005.

[8] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye. A Component-based and Aspect-oriented model for software evolution. *International Journal of Computer Applications in Technology*, 31(1/2):94-105, 2008.

[9] D. Suvée, B. D. Fraine, and W. Vanderperren. A symmetric and unified approach towards combining aspect-oriented and component-based software development. In *Component-Based Software Engineering (CBSE)*, vol. 4063 of LNCS, pages 114-122. Springer, 2006.

[10] C. Szyperski, D. Gruntz, and S. Murer. Component Software: Beyond Object-Oriented Programming. *Component Software Series*. ACM Press and Addison-Wesley, 2nd edition, 2002.

[11] L. Bergmans and M. Akşit. Composing Crosscutting Concerns Using Composition Filters. *Communications of the ACM*, Vol. 44, No. 10, pp. 51-57, October 2001.

[12] R. Douence, P. Fradet, and M. Sudhot. A framework for the detection and the resolution of aspect interaction. In *GPCE'06: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative programming and component engineering*, pages 173-188, Springer-Verlag, 2002.