Towards Reusable Aspects: the Callback Mismatch Problem

Maarten Bynens, Dimitri Van Landuyt,
Eddy Truyen and Wouter Joosen

DistriNet, Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Leuven, Belgium
{maarten.bynens,dimitri.vanlanduyt,
eddy.truyen,wouter.joosen}@cs.kuleuven.be

Abstract

Because software development is increasingly expensive and time-consuming, software reuse gains importance. Aspect-oriented software development modularizes crosscutting concerns which enables their systematic reuse. Literature provides a number of AOP patterns and best practices for developing reusable aspects based on compelling examples for concerns like tracing, transactions and persistence. However, such best practices are lacking for systematically reusing invasive aspects.

In this paper, we present the 'callback mismatch problem'. This problem arises in the context of abstraction mismatch, in which the aspect is required to issue a callback to the base application. As a consequence, the composition of invasive aspects is cumbersome to implement, difficult to maintain and impossible to reuse.

We motivate this problem in a real-world example, show that it persists in the current state-of-the-art, and outline the need for advanced aspectual composition mechanisms to deal with this.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries; D.2.11 [*Software Engineering*]: Software architectures—Information hiding,Languages,Patterns

General Terms Design, Documentation

Keywords reusable aspects, invasive aspects, aspect adapter

1. Introduction

Current AOP languages and approaches often result in aspects that are tightly coupled to the base classes they act upon. For example, it is a common technique to write advice code that involves join point reflection to find out the necessary contextual information [11]. Such advice code typically hard-codes assumptions about the structure and behavior of the base classes. This has a number of negative consequences: the aspect must be maintained together with the base, which makes it difficult to develop aspects and base in parallel, and leads to fragility of aspectual composition (lack of robustness). Additionally, the resulting aspects and their compositions are very specific to the scope of one application, and thus not reusable, for example in an aspect library.

To address these problems, the current state-of-the-art provides a number of techniques and patterns that involve introducing an abstraction layer between the base and the aspect. Examples of this are pointcut interfaces [6], annotations and marker interfaces. Introducing an abstraction layer enables the design of reusable aspects, in the sense that the required interface of the aspect (the elements it needs from the base to perform its function) can be specified uniquely in terms of abstractions that are relevant in the scope of the aspect itself. For example, the required interface of a reusable *authentication* aspect would be defined in terms of aspect-specific abstractions such as the *principal*, *credentials*, etc. To compose this *authentication* aspect to the base application, the developer must implement and provide these abstractions, by mapping elements of the base application (e.g. a *customer* in a web shop) to the aspect abstractions (the principal). Because the aspect is less tightly coupled to the base application, it can be reused more easily across applications.

A common problem in the design of reusable aspects is that of *abstraction mismatch*. This occurs when the elements of the base are not fully compatible with the abstraction required by the aspect. For example, the *credentials* abstraction may consist of a password that is encoded in MD5 —meaning that the *authentication* aspect expects passwords to be provided in MD5—, while the base offers the password in plain text. The solution to this is to introduce an adapter [5] that converts the base abstraction into the aspect abstraction. In the example, the adapter would be responsible for applying the MD5 hash function to the password that is provided by the base and providing the result to the aspect.

These techniques are sufficient to realize a loose coupling between aspect and base for both *spectative* and *regulative* aspects [9]; i.e. aspects that respectively observe the base application without affecting its functionality, or observe the base application and redirect or block the thread of execution in some cases. However, there is a lack of similar patterns or solutions for *invasive aspects* that issue callbacks to the base application to change its state or its behavior.

In this paper, we highlight this problem, which we call the *callback mismatch* problem. This problem arises (i) in the occurrence of abstraction mismatch, and (ii) when the aspect is required to issue a callback to the base application. As a consequence, the composition specification of such aspects becomes cumbersome to implement, difficult to maintain and impossible to reuse.

The structure of this paper is as follows. First, we define and illustrate the *callback mismatch* problem in a case study and we show that this is a realistic problem in the context of parallel development and reuse of aspectual modules. Then, we show that in the current state-of-the-art in aspect-oriented programming (AOP) and related techniques, patterns and notations, this problem persists and there is a need for advanced aspectual composition mechanisms to deal with this issue.

2. The callback mismatch problem

2.1 Problem definition

Pointcuts abstract not only from interesting join points in the base program but also expose relevant context data available at these join points. Abstraction mismatch is the problem where the representation of these abstractions in the base program is not compatible with the representation in the aspect. Dealing with abstraction mismatches is easy by employing a binding aspect that extracts the necessary information from the available base abstractions.

In the presence of callbacks however, specifying such a binding aspect becomes problematic. Callbacks happen when the reusable aspect uses the data and/or the behavior of the base application exposed by a pointcut to intervene in the normal control flow. As presented in Section 1, callbacks are mostly used to realize invasive aspects. To bind the callback to the base program, the binding aspect needs to include adapter functionality that routes the callback to the same base object that triggered the reusable aspect in the first place.

This problem is more complex to overcome than traditional problems with object-oriented libraries and frameworks (e.g. API mismatch). As the reusable aspect is never explicitly called from the base program, the adapter (or in this case the binding aspect) needs to adapt in both directions. It has to make sure that the relevant join points are translated to the aspect abstractions and that callbacks refer back to the original object. As a result, dealing with the callback mismatch problem takes more than solving the mismatch separately in both directions.

In summary, the *callback mismatch problem* leads to the following:

Problem summary. In the current state-of-the art of AOP languages, patterns and best practices, the required composition logic for dealing with both (1) *abstraction mismatch* and (2) *callbacks* is cumbersome to implement, difficult to maintain and impossible to reuse.

2.2 Motivating Example

To illustrate the problem outlined in this paper, we present a simplified example from the car crash management system (CCMS) [10, 16]. This is a large-scale and realistic distributed application that helps the authorities dealing with car crashes more efficiently by (i) centralizing all information related to a car crash, (ii) proposing a suitable crash resolution strategy, (iii) dispatching resource workers (e.g. first-aid workers) to the crash site, and (iv) reassessing the strategy in real-time when new information comes in.

To avoid wasting resources on prank calls and witnesses assuming a false identity, the correct and efficient functioning of the CCMS depends highly on *witness identity validation*, which is implemented in the CCMS as an aspect. More specifically, as long as the system has not successfully validated the identity of the witness, the CCMS will operate in *limited mode*, meaning that only a restricted set of resources can be assigned to that particular car crash.

Figure 1 presents this aspect in detail. The sequence starts when a witness calls the crisis center to report a car crash. The coordinator answering the call enters the name and phone number of the witness into the CCMS.

In this example, `Witness` represents the base abstraction: it provides the information needed by the aspect.

The *witness identity validation* aspect is provided in the form of a reusable identity validation aspect `IdentityValidation`. The required aspect abstractions in this example are `Person` and `ValidationReport`.

This illustrates *abstraction mismatch* in this example: the provided abstraction of the base application is the `Witness` which encapsulates the name, the phone number, and the validity state of the witness. On the other hand, the required interface of the aspect consists of (1) the `Person` abstraction which encapsulates first and last name and phone number, and (2) `ValidationReport` abstraction which encapsulates the validity state.

As pointed out in Section 2.1, this issue can be resolved by specifying a binding aspect with adapter functionality. In this example, we have implemented a *class adapter* which adapts the interface of the `Witness` object to match those of `Person` and `ValidationReport` (message 2).

After this, the aspectual composition with the *witness identity validation* aspect is realized. More specifically, the pointcut for this aspect is specified in terms of the `Person` interface (message 3). Both the `Person` and the `ValidationReport` are exposed through these join points.

Finally, the `IdentityValidation` component contacts a third-party telecom operator to check whether the presented person is indeed listed under the given phone number. The result of this verification activity is set via the `ValidationReport` interface. Because the `Witness` object has previously been adapted to this interface by the adapter, the callback ends up at the witness (message 4).

Section 2.3 illustrates in further detail how the adapter code is affected by the callback mismatch problem.

If the adapter is implemented incorrectly, the CCMS itself will remain in limited mode, and thus addresses the car crash inefficiently, if at all. The fact that the correct functioning of the entire application depends fully on the correct realization of the callback stresses the importance of writing an adapter that realizes the desired behavior in a comprehensible, maintainable and reusable manner.

2.3 Minimal solution in AspectJ

An example implementation of the scenario is included in the appendix. The pointcut `personIdNeedsChecking` in aspect `IdentityValidation` is defined in terms of types `Person` and `ValidationReport`. `Person` contains the data that needs to be checked and `ValidationReport` captures the result of the validation. Since these types are not directly supported by the base code, an adapter needs to be written to bind the aspect to the application. Listing 1 shows the adapter.

```
public aspect Adapter extends IdentityValidation{

  declare parents: Witness implements
      ValidationReport;
  public void Witness.validation(boolean b){
    validate(b);
  }

  declare parents: Witness implements Person;
  public void Witness.setFirstName(String s){}
  public void Witness.setLastName(String s){}
  public String Witness.getFirstName(){
    return getName().split(" ")[0];
  }
  public String Witness.getLastName(){
    return getName().split(" ")[1];
  }
  void around(Person w): execution(*
      Witness.setName(String)) && this(w){
   proceed(w);
   w.setFirstName(w.getFirstName());
   w.setLastName(w.getLastName());
  }

  pointcut report(ValidationReport report):
      this(report);
}
```

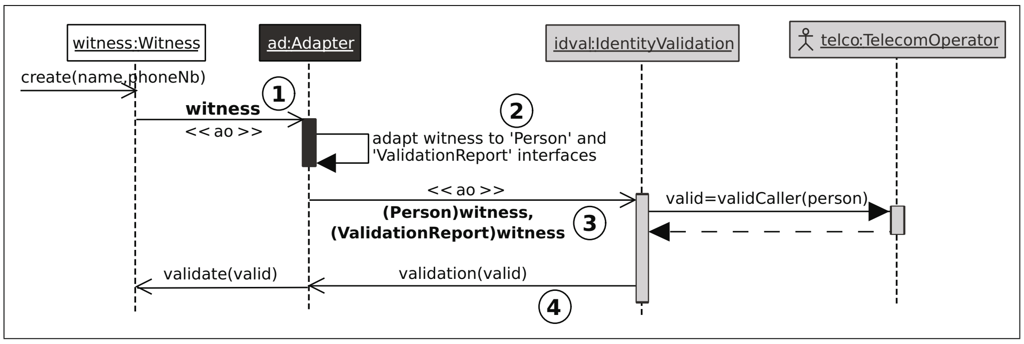

Figure 1. UML sequence diagram to illustrate the role of the adapter (in dark gray), and the *witness identity validation* aspect (in grey).

Listing 1. Example implementation of the adapter

In this scenario, the adapter has two responsibilities. Firstly, it needs to make sure that the callbacks through `ValidationReport` and `Person` are reified in the witness object. Therefore, the class `Witness` is made to implement the types `ValidationReport` (lines 3–6) and `Person` (lines 8–16) by means of *declare parents* and inter-type declarations (methods `setFirstName` and `setLastName` do not need an actual implementation because they are not used as a callback). Secondly, it needs to propagate the relevant join points on `Witness` as required join points on `Person`. This is achieved by around advice that calls *proceed* and additionally calls the appropriate methods (lines 17–21). Because there is a mismatch in the sense that `Person` has separate concepts for first name and last name, extra mapping functionality is required.

This example shows that even in this simple (almost trivial) case, defining the adapter is already a cumbersome task. One that needs to be repeated for every mismatch.

3. Approaches

This section gives an overview of existing AOP languages, techniques and patterns that are related to the problem and briefly argues that none of them sufficiently addresses the callback mismatch problem.

3.1 Explicit Pointcut Interfaces

Approaches like pointcut interface[6], XPI[15] and explicit join points[8] do not help to define bidirectional adapters more easily. The problem is that the aspect will always use a type description to be able to issue callbacks. This type should then be mapped to a concrete type in the base code. The approaches mentioned describe join points and not types and thus cannot be used in this mapping.

In the simplified case, the aspect specifies an abstract pointcut and the callback is issued on one of the exposed parameters. An explicit pointcut interface can help with implementing this abstract pointcut, but the mapping of the callback to the base code still needs to be done. A standard unidirectional adapter is sufficient in this case.

3.2 Type parameters

At first sight, type parameters seem to solve the callback mismatch problem, since an instantiated type parameter will behave as an alias for a concrete type of the base code. Unfortunately, for the aspect to be able to issue callbacks, it needs to refer to an actual type (and e.g. use it as a bound for the type parameter). As a result we end up with the same problems as before.

3.3 Caesar

Caesar supports on-demand remodularization to integrate independent components. Its model is object-based and uses virtual types, mixin composition and the concept of wrapper recycling [12]. As a result, Caesar provides a means to specify expressive, reusable adapters. However, Caesar does not support remodularization of aspect abstractions. In Caesar, the aspect composition is part of the binding and requires manual object wrapping (assisted by dynamic wrapper selection and wrapper recycling) [1, 13]. We can conclude that Caesar doesn't offer a solution to the callback mismatch problem as it not aims to bind abstract aspect compositions.

3.4 Subject-oriented programming

Subject-oriented programming [7] and its descendants Hyper/J and Theme[4] (which all involve Multi-Dimensional Separation of Concerns (MDSOC)) represent a more symmetrical approach to AOSD, meaning that each concern is developed independently. One of the key features of these approaches is *declarative completeness*, meaning that each concern explicitly defines the structure and behavior of the classes it depends on. To assemble an application, these concerns are composed using composition rules. Composition directives includes mechanisms for name-based merging of classes and methods, and support for renaming, overriding, . . .

Because these mechanisms are nondirectional, they are inherently adequate for specifying callbacks. However, the composition mechanisms are not expressive enough to resolve sophisticated abstraction mismatches that can only be resolved with complex adapters involving more than renaming, overriding and merging classes and methods. Therefore these approaches do not solve the abstraction mismatch problem.

4. Conclusion

This paper introduces the *callback mismatch* problem. In essence, this problem is triggered by two key elements: (i) *abstraction mismatch* which is resolved by applying the Adapter design pattern [5], and (ii) *invasive* aspects [9], i.e. aspects that issue a callback to the base application to change its state or behavior. This situation leads to composition logic that is cumbersome to implement, difficult to maintain and impossible to reuse.

We have illustrated the problem in a minimal example from a realistic case study. Additionally, we have presented a number of factors that deteriorate this problem. Finally, we outline a number of related approaches in which this problem persists.

From this, we conclude that the current state-of-the-art is currently is not capable of solving the callback mismatch problem adequately. In our opinion, there are three distinct research directions to be explored for a solution to this problem: (i) next-generation language constructs that allow the described adapters to be defined more elegantly and concisely (e.g. inspired by Caesar and SOP that provide disjoint sets of constructs that solve the problem partly), (ii) middleware-based solutions and framework-specific services that are capable of hiding most of the described adapter complexity, or (iii) AOP design patterns that provide reference solutions to this problem.

Logging, tracing and authentication are aspects addressed pervasively throughout AOSD research. Based on the impact that these aspects have on the base application, they are characterized as either *spectative* or *regulative* [9]. The large body of research into these particular aspects classes suggests that they are well-known, and they can sufficiently be dealt with by current AOP techniques. As AOP matures, it is our opinion that the research focus should shift from *spectative* and *regulative* aspects towards more *invasive* aspects, which represent the most challenging class of crosscutting concerns. We believe that the problem brought to the forefront in this paper is a key hurdle in the road towards advanced AO languages, middleware and patterns that deal with these types of aspects in an efficient, maintainable, and reusable manner.

Acknowledgments

This research is supported by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by the Research Fund K.U.Leuven.

References

[1] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. [14], pages 135–173.

[2] Maarten Bynens and Wouter Joosen. Towards a pattern language for aspect-based design. In *PLATE '09: Proceedings of the 1st workshop on Linking aspect technology and evolution*, pages 13–15, New York, NY, USA, 2009. ACM.

[3] Maarten Bynens, Bert Lagaisse, Eddy Truyen, and Wouter Joosen. The elementary pointcut pattern. In *BPAOSD'07: Proceedings of the 2nd workshop on Best practices in applying aspect-oriented software development*, pages 1–2, 2007.

[4] Siobhán Clarke and Robert J. Walker. Generic aspect-oriented design with Theme/UML. pages 425–458.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, illustrated edition edition, November 1994.

[6] Stephan Gudmundson and Gregor Kiczales. Addressing practical software development issues in aspectj with a pointcut interface. In *Advanced Separation of Concerns*, 2001.

[7] William H. Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA*, pages 411–428, 1993.

[8] Kevin Hoffman and Patrick Eugster. Bridging java and aspectj through explicit join points. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 63–72, New York, NY, USA, 2007. ACM.

[9] Shmuel Katz. Aspect categories and classes of temporal properties. [14], pages 106–134.

[10] Jörg Kienzle, Nicolas Guelfi, and Sadaf Mustafiz. Crisis management systems: A case study for aspect-oriented modeling. Technical Report SOCS-TR-2009-3, School of Computer Science, McGill University, 2009. `http://www.cs.mcgill.ca/research/reports/2009/socs-tr-2009-3.pdf`.

[11] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[12] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 52–67, New York, NY, USA, 2002. ACM.

[13] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM.

[14] Awais Rashid and Mehmet Aksit, editors. *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*. Springer, 2006.

[15] Kevin J. Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/SIGSOFT FSE*, pages 166–175, 2005.

[16] Dimitri Van Landuyt, Eddy Truyen, and Wouter Joosen. Discovery of stable domain abstractions for reusable pointcut interfaces: common case study for ao modeling. Technical report, Department of Computer Science, K.U.Leuven, 2009. `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW560.abs.html`.

A. Entire example

The full source code is available at `http://www.cs.kuleuven.be/~dimitri/callbackmismatch.zip`.

```
public abstract aspect IdentityValidation {

  pointcut personIdNeedsChecking(Person
      person, ValidationReport report):
    (execution(new(..)) || execution(void
        Person.set*(..))) && this(person) &&
        report(report);
  abstract pointcut report(ValidationReport
      report);

  Object around(Person person, ValidationReport
      report):personIdNeedsChecking(person, report){
    Object res = proceed(person, report);
    report.validation(TelecomOperator.validCaller(
        person.getFirstName() + " " +
        person.getLastName(),
        person.getPhone()));
    return res;
  }
}
```

Listing 2. IdentityValidation.aj

```
public interface Person {

  public String getFirstName();
  public String getLastName();
  public String getPhone();
  public void setFirstName(String fname);
  public void setLastName(String lname);
  public void setPhone(String phone);

}
```

Listing 3. Person.java

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.

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.

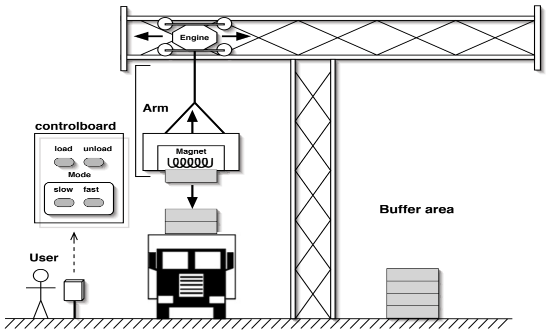

Figure 1. A Schematic Overview of the Crane System

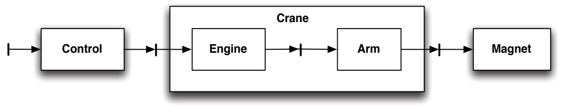

Figure 2. The Crane System Architecture

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

[*] Partially funded by the Miles Project

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.

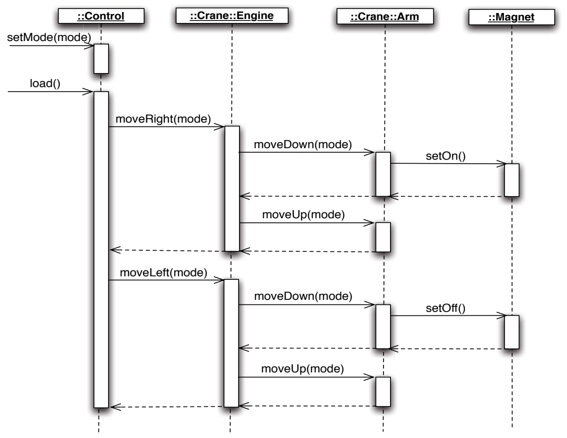

Figure 3. Loading Process for the Crane System

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.

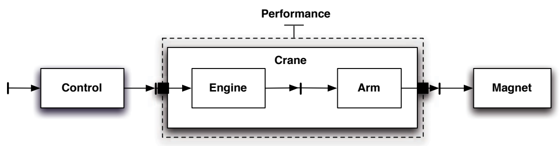

Figure 4. Performance View

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.

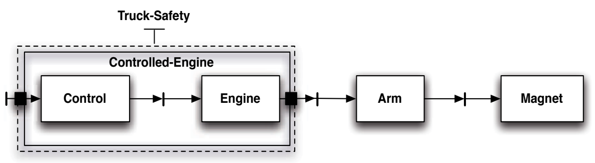

Figure 5. Truck Safety View

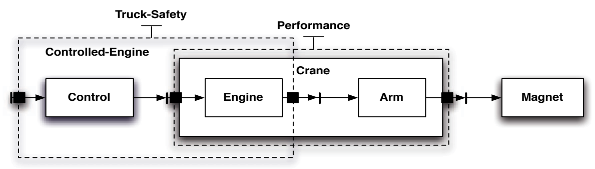

Figure 6. Wrappers Crosscut Phenomenon

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:

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

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 component 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.

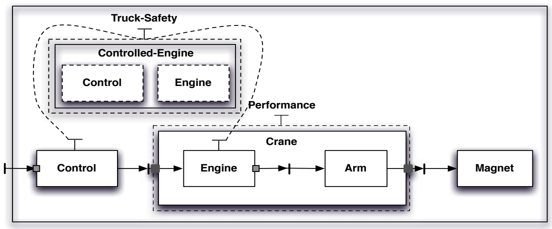

Figure 7. Views in Fractal Component Model

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:

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

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.

```
<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

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.