# Invasive Composition Adapters: an aspect-oriented approach for visual component-based development

Wim Vanderperren
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 62
wvdperre@vub.ac.be

Davy Suvée
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 65
dsuvee@vub.ac.be

Viviane Jonckers
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 67
viviane@info.vub.ac.be

## ABSTRACT

In this paper, we build on previous work that combines ideas from visual component-based software development with aspect-oriented software development. We introduced a composition adapter to modularize crosscutting concerns in our visual component-based methodology developed in earlier work. A composition adapter can be visually applied onto a composition pattern and the changes it describes are automatically inserted using finite automaton theory. The expressive power of a composition adapter is however limited to concerns that alter the exterior behavior of a component. To overcome this limitation, we propose to employ a new aspect-oriented implementation language, called JAsCo, tailored for the component-based context. An invasive composition adapter, which has an implementation in the JAsCo language, is able to express concerns that require more than mere filtering and re-routing. The changes dictated by an invasive composition adapter are automatically inserted into the components and composition patterns.

## 1. INTRODUCTION

Aspect-Oriented Software Development (AOSD) argues that some concerns exist that can not be confined to one single module. Typical examples of such concerns are logging and synchronization. The research to deal with this problem is under constant evolution. Most of this research however is targeted to Object-Oriented Software Development (OOSD). As a consequence, these approaches are not very well suited to be reused in a component-based context. This paper describes our approach to introduce aspect-oriented ideas in Component-Based Software Development (CBSD) from design to implementation.

In previous research [12-15], we developed a component-based approach that lifts the abstraction level for visual component composition. This research resulted in a visual component composition environment called PacoSuite. PacoSuite improves on standard visual composition tools as it allows components to be wired together based on generic interaction protocols, called “composition patterns”, rather than simple event/method pairs. To introduce aspect-oriented ideas into PacoSuite, we proposed a “composition adapter”. A composition adapter transforms the original composition patterns to introduce the specified aspects. Technically, a composition adapter is applied by introducing the aspects in the glue code of a component-based application. As a result, it is impossible to introduce aspects in the components themselves. However, several experiments revealed that it should be possible to adapt the components’ interior to express aspects that require more than mere filtering or rerouting. To solve this problem, we introduce a new aspect-oriented programming language targeted at component-based development, called JAsCo. An “invasive” composition adapter is an enhanced version of a regular composition adapter implemented in the JAsCo language. In this way, concerns that require adaptations to the interior of components can also be expressed.

This paper presents a complete overview of our approach. As a result, technical details of algorithms and formal foundations are not discussed. Section 2 briefly describes our component-based methodology and presents the composition adapter model using run-time checking of timing constraints as a concrete example. Section 3 briefly presents the JAsCo aspect-oriented programming language and the invasive composition adapter model is introduced in section 4. Section 5 presents the tool support we created to support our methodology. Finally, we present some related work and state our conclusions.

## 2. RESEARCH CONTEXT

### 2.1 CBSD in PacoSuite

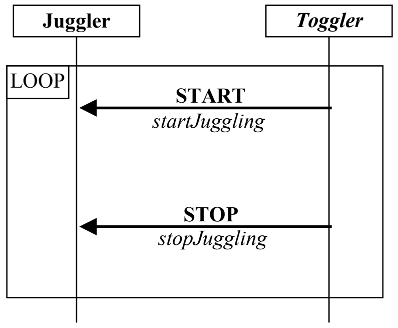

**Figure 1: Usage scenario of a Juggler component.**

We mainly focus our component-based research on lifting the abstraction level for component-based development. We want to realize the plug and play idea of component-based development. Therefore, we propose to document components with usage scenarios that specify how to employ them. A usage scenario is expressed by a special kind of Message Sequence Chart (MSC) [4]. The main difference with a regular MSC is that the signals are taken from a limited set of pre-defined semantic primitives. Each of these signals is mapped on the concrete API that performs them. As a result, the documentation of a component is both abstract and concrete at the same time. Figure 1 illustrates a usage scenario of the well-known *Juggler* bean. One participant of a usage scenario represents the component itself and the other participants represent the environment the component expects. In this case, only one environment participant is specified, namely the *Toggler* participant. This usage scenario documents that the *Juggler* component expects consecutive start and stops. The START primitive is implemented by *startJuggling* and *stopJuggling* implements the STOP primitive.

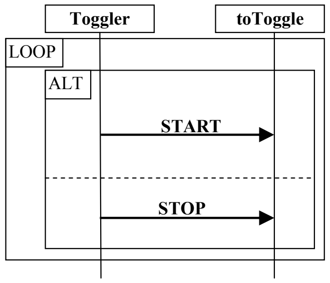

**Figure 2: Toggling composition pattern.**

We introduce explicit and reusable composition patterns that are also expressed using MSC's. A composition pattern is an abstract specification of the interaction between a number of roles. The signals between the roles originate from the same limited set of semantic primitives. This allows comparing the signals in a usage scenario of a component with these in a composition pattern. Figure 2 illustrates a generic toggling composition pattern. This composition pattern specifies that the *Toggler* participant consecutively sends either a START or a STOP to the *toToggle* participant. A possible application of this composition pattern is a simple visual interface that allows toggling the *Juggler* component from a single *JButton* component. To build this application, the *Juggler* component is mapped on the *toToggle* role and the *JButton* component is mapped on the *Toggler* role. Notice that even this simple collaboration can not be wired by most visual composition environments because the collaboration itself requires state.

The documentation of components and composition patterns allows checking the compatibility of a component with a role. The glue-code that constraints the behavior of the components and that translates syntactical compatibilities is generated automatically. Both the algorithms are based on finite automaton theory. In this paper we do not go into the details of these algorithms. The interested reader is referred to [14, 15].

## 2.2 Composition Adapters

Some concerns can not be cleanly modularized using composition patterns and components as are spread into different entities. As a result, editing, adding and removing such a concern becomes a cumbersome and error-prone task. To solve this problem, we propose composition adapters. The next paragraphs present this solution using the run-time checking of timing constraints as a concrete example. If we want to check timing constraints dynamically using our current concepts, every composition pattern needs to be adapted in the same way. Of course, when the application goes into the production phase, the dynamic timing aspect needs to be removed from the application. Consequently, the involved composition patterns need to be altered again to remove the timing aspect.

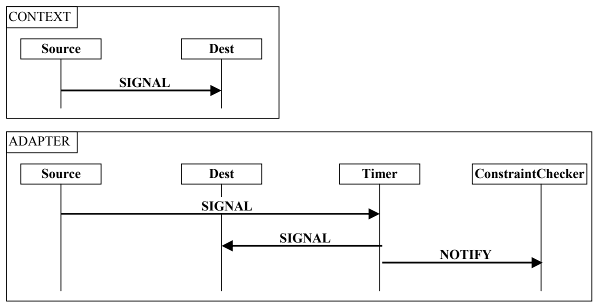

**Figure 3: Dynamic timing verification composition adapter**.

In order to modularize crosscutting concerns in PacoSuite, we introduce a new concept, called a composition adapter. A composition adapter is able to describe adaptations of the external behavior of a component independently of a specific API. A composition adapter is again documented using a special kind of MSC and consists of two parts: a context part and an adapter part. Figure 3 depicts the composition adapter that is used to modularize the timing aspect. The context part of a composition adapter describes the behavior that needs to be adapted. This can be a simple signal send as in Figure 3, but can very well be a full protocol. The adapter part specifies the adaptation itself. In the case of the dynamic timing composition adapter, every signal between the *Source* and *Dest* role will be rerouted through a *Timer* role. The *Timer* role is responsible for taking a timestamp and notifying the *ConstraintChecker* role. The *ConstraintChecker* role is responsible to verify whether every signal it is notified of, does not violate a timing constraint. The component that is mapped on the *ConstraintChecker* role could do the verification process offline and/or run on a different CPU to minimize the disruption of the system.

When a composition adapter is applied onto an existing composition pattern, the context roles of the composition adapter need to be mapped onto roles of the composition pattern. For example, suppose we want to time the communication between the *Toggler* and *toToggle* roles of the composition pattern in figure 2. The *Source* role of the timing composition adapter of Figure 3 has to be mapped onto the *Toggler* role of the composition pattern. Likewise, the *Dest* role has to be mapped onto the *toToggle* role. As a result, the START and STOP signals are not send directly to the *toToggle/Dest* role but are re-routed through the *Timer* role. After sending the START or STOP signal to the *toToggle/Dest* role, the *ConstraintChecker* role is notified.

To automatically apply a composition adapter onto a given composition we developed an algorithm based on finite automata theory. In this paper, we do not discuss this algorithm, a full explanation can be found in [13].

## 2.3 Discussion

The critical reader might have noticed that the composition adapter approach to enable run-time checking of timing constraints is not very accurate. Currently, the timestamp of the event is taken when it arrives at the component mapped on the Timer role. So, there is at least an inaccuracy because of the delay of this message send. If the application works distributed, this delay can not be neglected. Certain sophisticated component systems use a scheduler to pass messages to components. This scheduling process imposes yet another delay, making the timestamp even less accurate. As a result, our composition adapter approach to check timing constraints at run-time is not very well suited if a high precision is desired. The only way to achieve a correct timestamp is to alter the mapped components themselves so that the timestamp is taken before a message is sent or received. However, a composition adapter is only able to alter the exterior behavior of a component by ignoring or re-routing messages. Aspects that require other adaptations can not be described using a composition adapter, which is a major limitation. To solve this problem, we enhance our current model using an implementation in an aspect-oriented programming language. The next section describes the language we designed for allowing a composition adapter to specify invasive changes of a component. Section 4 discusses how this new language is used to realize an *invasive* composition adapter.

## 3. JASCO LANGUAGE

For enhancing the composition adapter model, an implementation in an aspect-oriented programming language is required. Several AOSD-approaches, such as AspectJ [2], composition filters [3] and HyperJ [15], are available. These technologies however, mainly aim at describing crosscutting concerns in an object-oriented context. As a result, they are very well not suitable for being deployed in a component-based context, this because of several restrictions:

- Nearly all AOSD-approaches describe aspects with a specific context in mind, which limits reusability.
- The deployment of an aspect within a software-system is at the moment rather static, as aspects loose their identity when they are integrated within the base-implementation. As a result, aspects are not able to exhibit the same plug-and-play characteristic of components.
- The communication between components depends on the employed component model. Current AOSD-technologies however do not support to specify aspects on these specific kinds of interactions.

For overcoming the problems mentioned above, we propose a new aspect-oriented implementation language called JAsCo. JAsCo has been developed with CBSD, and in particular PacoSuite, in mind. The JAsCo-language stays as close as possible to the regular Java syntax, and introduces two new concepts: *aspect beans* and *connectors*. An aspect bean is a regular Java bean that describes one or more logically related hooks as a special kind of inner classes. A hook is a generic and reusable entity and can be considered as the combination of the AspectJ's pointcut and advice. A connector on the other hand, is used to initialize several logically related hooks with a concrete context. To make the JAsCo language operational, we propose an "aspect-enabled" component model, where components do not require any adaptation whatsoever for aspects to be deployed.

The following two subsections describe the syntax of both the aspect- and connector-language. For more information about JAsCo and the JAsCo Beans component model, we refer to [9].

### 3.1 Aspect Syntax

Aspect beans are used for describing functionality that would normally crosscut several components from which the system is composed. The run-time checking of timing constraints, introduced in section 2, is an example of such a crosscutting concern. Whenever a specific method is executed, a timestamp should be taken such that the defined timing constraints can be checked. Figure 4 illustrates the implementation of this dynamic timer aspect. Aspect beans usually contain one or more hook-definitions (line 17 till 32), and are able to include any number of ordinary Java class-members (line 3 till 15), which are shared amongst all hooks of the aspect. A hook is used for defining **when** the normal execution of a method should be cut, and **what** extra behavior there should be executed at that precise moment in time. For defining when the behavior of hook should be executed, each hook is equipped with at least one constructor (line 21 till 23) that takes one or more *abstract method parameters* as input. These abstract method parameters are used for describing the context of a hook. The *TimeStamp*-hook specifies that it can be deployed on every method that takes zero or more arguments as input. The constructor-body defines how the join points of a hook initialization are computed. In this particular case, the constructor-body (line 22) specifies that the behavior of the *TimeStamp*-hook should be triggered whenever *method* is executed. The behavior methods of a hook are used for specifying the various actions a hook needs to perform whenever one of its calculated join points is encountered. Three kinds of behavior methods are available: *before, after* and *replace*. The *TimeStamp*-hook specifies two behavior methods (line 25 till 31). The before behavior method describes that a timestamp should be taken prior to the execution of *method*. In addition, the after behavior method specifies that all the interested observers should be notified of the timestamp.

```
1  class DynamicTimer {
2
3    private Vector obs = new Vector();
4    void removeTimeListener(TimeListener o) {
5      obs.remove(o);
6    }
7    void addTimeListener(TimeListener o) {
8     obs.add(o);
9   }
10   void notifyListeners(Method m, long t) {
11      for (int i = 0;i < obs.size();i++) {
12        ((TimeListener)obs.elementAt(i)).
13            TimeStampTaken(m,t);
14      }
15   }
16
17   hook TimeStamp {
18
19     private long timestamp;
20                                           When?
21     TimeStamp(method(..args)) {
22       execute(method);
23     }
24                                                  What?
25     before() {
26       timestamp=System.currentTimeMillis();
27     }
28
29     after() {
30       notifyListeners(method,timestamp);
31     }
32   }
33 }
```

**Figure 4: The JAsCo-aspect for dynamic timing**.

## 3.2 Connector Syntax

Connectors are used for initializing a hook with a specific context (methods or events). A hook initialization takes one or more methods or event signatures as input. Figure 5 illustrates the *TimeConnector*. This connector initializes a *TimeStamp*-hook *timer* with the throwing of the *actionPerformed*-event of the *JButton*-component (line 5), and with the *startJuggling* and *stopJuggling*-methods of the *Juggler*-component (line 6 till 7). After initializing this hook, the *TimeConnector* specifies the execution of the before and the after behavior methods. Consequently, the *TimeConnector* has following implication: take a timestamp and notify all observers of the *DynamicTimer* aspect bean whenever the JButton throws an *ActionEvent* and whenever the Juggler starts or stops juggling.

```
1  connector TimeConnector {
2                                                       Where?
3    DynamicTimer.TimeStamp timer =
4      new DynamicTimer.TimeStamp ( { onevent
5        JButton.actionPerformed(ActionEvent),
6        void Juggler.startJuggling(),
7        void Juggler.stopJuggling() } );
8
9    timer.before();
10   timer.after();
11 }
```

**Figure 5: The JAsCo-connector for dynamic timing of the JButton and the Juggler.**

## 4. INVASIVE COMPOSITION ADAPTERS

## 4.1 Documentation

One of the problems encountered with our current composition adapter model is that it is not able to express aspects that require interior adaptations of a component. To solve this problem, we propose to employ the JAsCo language as an implementation for a composition adapter. Hence, the composition adapter model needs to be altered slightly.

Figure 6 illustrates the *invasive* composition adapter that documents the *DynamicTimer* aspect bean of Figure 4. Messages in the context part of an invasive composition adapter can be mapped on a hook. In the case of Figure 6 the SIGNAL message is mapped on the *TimeStamp* hook. As a result, every message between the component that is mapped on the *Source* role and the component that is mapped on the *Dest* role will be given to the *TimeStamp* hook constructor. As a consequence, those messages are changed to take a timestamp and to notify interested observers. The adapter part of an invasive composition adapter includes a new role that represents the aspect bean in the JAsCo language. In the case of Figure 6, the *DynamicTimer* role represents the aspect bean with the same name of Figure 4. The adapter part documents what the effect of the application of the *DynamicTimer* aspect bean will be. In the example of Figure 6, every signal between a certain source and destination component is still sent in the same way. However, the *DynamicTimer* aspect bean declares that a timestamp has to be taken before an adapted method is executed (see Figure 4**,** line 28-30). This behavior is not documented in the composition adapter as it is internal to the aspect bean and no communication with other components is involved. As a consequence, this behavior is not relevant for verifying compatibility and to generate glue-code. After the original method is executed, the *DynamicTimer* aspect bean notifies a *ConstraintChecker* component that verifies whether certain timing constraints are violated (see Figure 4**,** line 32-34). This behavior however, is documented in the composition adapter because it requires communication with other components. Messages that are sent or received by a JAsCo component require an implementation mapping. In Figure 6, the NOTIFY message of the *DynamicTimer* aspect bean is implemented by throwing the *timeStampTaken* event. The implementation mapping is required to be able to generate glue-code that will call the correct method of the component that is mapped on the *ConstraintChecker* role when the *DynamicTimer* throws the *timeStampTaken* event. Notice that the component that will be mapped on the *ConstraintChecker* role does not have to understand the *timeStampTaken* event. Glue-code that translates the *timeStampTaken* event into one or more methods of the mapped component can be automatically generated using the documentation of Figure 6.

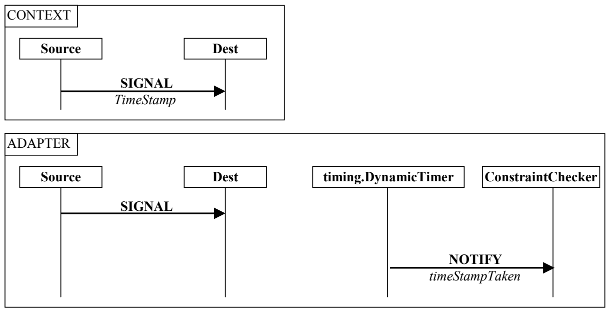

**Figure 6: Invasive Composition Adapter model for the DynamicTimer aspect bean**.

## 4.2 Applying an invasive composition adapter

An invasive composition adapter changes the composition patterns in the same way a regular composition adapter does. As a result, we can still use the same algorithm that was developed for regular composition adapters to determine the effect of an invasive composition adapter on a composition pattern.

An invasive composition adapter however also changes the components themselves through the implementation in the JAsCo language. The adaptations to a component caused by an invasive composition adapter might affect the external behavior of the component. As a consequence, the documentation of a component becomes inconsistent. To be able to still verify the compatibility of an adapted component with a given composition pattern, the documentation of this component needs to be modified. This is easily achieved by a similar algorithm as the one used for adapting composition patterns to the specification of a composition adapter [13]. The specification of an invasive composition adapter is used to alter the documentation of the components that are mapped on the context roles of the composition adapter. In this way, we are still able to check compatibility and automatically generate glue-code. In the case of Figure 6, the documentation does not have to be altered because the original behavior of the components that are mapped on the *Source* and *Dest* roles is not changed.

As a last step, a connector in the JAsCo language is generated to be able to apply the JAsCo implementation of the invasive composition adapter onto the correct components. In order to locate the concrete methods and events the aspect has to be applied to, we have to calculate where the context part of the composition adapter occurs. Luckily, this was already determined in the previous phase. So, only the parts of the documentation of a component where the context part occurs need to be analyzed. In case of Figure 6, this means that all messages that are mapped onto the signal with the *TimeStamp* hook as an implementation, have to be altered by the composition adapter. For instance, if the *Juggler* component of Figure 1 is mapped onto the *Dest* role of the composition adapter of Figure 6, both the *startJuggling* and *stopJuggling* methods would have to be adapted. Figure 5 illustrates the connector generated when the *Juggler* component is mapped onto the *Dest* role and the *JButton* component is mapped onto the *Source* role. The onevent keyword is used because outgoing communication of Java Beans occurs through event posting. When the connector is generated, the JAsCo compiler is executed and the regular glue-code generation process of our visual component composition environment is started. As a result, the *startJuggling* end *stopJuggling* methods and the *actionPerformed* event are timed. Timing constraints that act on these points can be verified at run-time with a more accurate precision than when using a non-invasive composition adapter.

### 4.3 Small Case Study

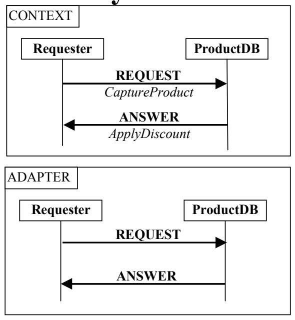

**Figure 7: OldProductDiscount invasive composition adapter.**

It can be argued that using an invasive composition adapter for specifying timing constraints validation is not really necessary. Indeed, a regular composition adapter is also able to describe this concern, only the accuracy of the timestamps differs. Therefore, we shortly present a small case-study that introduces crosscutting concerns that really require an invasive composition adapter. The case study at hand is a digital photo printing laboratory. The system consists of two sub-applications: a client that allows browsing and previewing pictures and a server application that is responsible for printing and calculating the price of an order. We identified four crosscutting concerns and successfully modeled them using an invasive composition adapter. Due to space constraints, only one of them is introduced, namely a business rule that specifies a discount for obsolete products. In this case, the obsolete product is a photo paper format that is no longer in use. To introduce this concern, extra behavior has to be inserted in the product database to be able to persistently store and use the old product information. As a result, the product database returns a discounted price for older products. Figure 7 illustrates the *OldProductDiscount* invasive composition adapter. The context part declares that this invasive composition adapter is applicable on a consecutive REQUEST and ANSWER. Notice that a different hook is mapped on both the primitives of the context part. The *CaptureProduct* hook is responsible for capturing all relevant information of the price request of a certain product. The *ApplyDiscount* acts on the answer of the request and changes the result if the product is considered obsolete. The adapter part of the *OldProductDiscount* invasive composition adapter declares that the request and answer are sent in the same way as before. Notice that the *OldProductDiscount* aspect bean itself is not documented because it does not participate in the interaction. Indeed, this invasive composition adapter only changes the interior behavior of the component that is mapped onto the *ProductDB* role.

## 5. TOOL SUPPORT

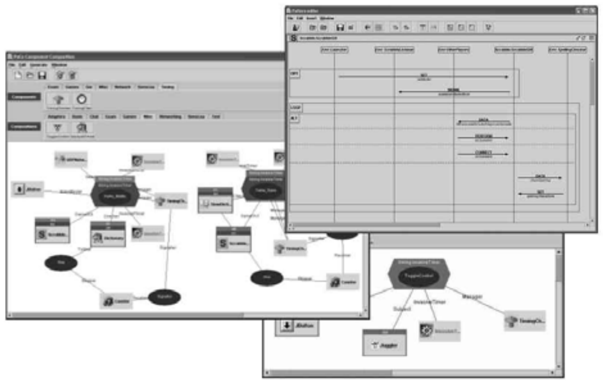

**Figure 8: Screenshots of PacoSuite. The middle left and bottom right screenshots illustrate the visual component composition environment PacoSuite. The rectangles represent components, the ovals stand for composition patterns and the hexagonal shapes symbolize invasive composition adapters. The top-right screenshot shows the documentation of a Scrabble component in the PacoDoc tool.**

The ideas introduced in this paper are implemented in a visual component composition environment called PacoSuite. PacoSuite consists of two visual tools, called PacoDoc and PacoWire, and the command-line tools required by the JAsCo language. PacoDoc is a visual editor for documenting components, composition patterns and composition adapters. PacoWire is our actual component composition environment that allows visually applying a component onto a role of a composition pattern. The drag and drop action is refused when the component is detected to be incompatible with the composition pattern. Composition adapters can also be visually applied on a given composition of components. The changes dictated by a composition adapter are automatically applied using the algorithms mentioned in this paper. In case of an invasive composition adapter, the JAsCo tools are executed transparently to the user. When all the component roles are filled, the composition is checked as a whole and glue-code is generated automatically. Figure 8 illustrates some screenshots of this tool suite.

## 6. RELATED WORK

One of the first approaches to integrate aspect-oriented software development and component-based software development is the aspectual component model of Lieberherr et al [11]. The JAsCo language was partly inspired by this work and quite some similarities exist between both languages. They both employ a separate connector language to deploy an aspect within a specific context. On a technical level, the aspectual components approach uses byte code weaving, while we propose a new component model. Our approach improves on aspectual components by lifting the abstraction for applying aspects from the implementation level to a visual composition environment.

Filman [7] proposes dynamic injectors to introduce aspects into a given component configuration. He incorporates dynamic injectors into OIF (Object Infrastructure Framework), a CORBA centered aspect-oriented system for distributed applications. The dynamic injector approach is very similar to our non-invasive composition adapter idea because both approaches employ a wrapping and filtering technique to insert crosscutting concerns into a composition of components.

Another more recent approach to recuperate aspect-oriented ideas in component-based software development is event based aspect-oriented programming (EAOP). EAOP [4] allows specifying crosscuts on events and event patterns using a formal language. Similar to the composition adapter approach, EAOP allows specifying aspects on a full protocol of events instead of a set of methods. Since EAOP is based on a formal model, EOAP is able to improve on our approach because of the advanced detection and resolution of aspect interactions [5]. Our approach extends EAOP by lifting the abstraction level for aspect application from the implementation level to a visual composition environment.

Duclos et al [6] focus on separating crosscutting concerns in legacy systems built using CCM [3]. Similar to PacoSuite, they specify crosscutting concerns at the architectural level. They also employ two languages, one for declaring an aspect and one for describing how the aspect should be used. Aspects are applied by generating individually tailored CCM containers that include the aspect's logic. In that sense, their approach is similar to wrapping because they do not allow interior changes to the components.

## 7. CONCLUSIONS

In this paper, we introduce an invasive composition adapter in order to specify crosscutting concerns that require interior adaptations of a component on a component-based design level. An invasive composition adapter is an extended version of a regular composition adapter and has an implementation in the JAsCo aspect-oriented language. A component composer is able to visually apply an invasive composition adapter on a given component composition. The invasive composition adapter is verified to be compatible with the composition and is automatically deployed using algorithms based on finite automaton theory. Likewise, an invasive composition adapter can be easily removed from a collaboration when the concern is not desired any longer. The main drawback of this approach is that it is domain dependent. It is possible to agree on a set of semantic primitives to document component interactions for a limited application domain. However, it is unfeasible to come up and agree on a general set of semantic primitives. Another drawback is that this approach is resource intensive. Our current algorithms are of exponential nature and in worst case scenarios this could lead to state explosions. In addition, the glue-code to translate syntactic incompatibilities between components adds an extra level of indirection.

## 8. ACKNOWLEDGMENTS

We owe our gratitude to Prof. Dr. Viviane Jonckers for her invaluable help during our research and for proof reading this paper. Also, we like to thank Dr. Bart Wydaeghe who developed the component based methodology during his PhD research. Since October 2000, Wim Vanderperren is supported by a doctoral scholarship from the Fund for Scientific Research (FWO or in Flemish: "Fonds voor Wetenschappelijk Onderzoek").

## 9. REFERENCES

[1] AspectJ Website. http://www.aspectJ.org.

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

[3] Corba Component Model: see http://www.omg.org.

[4] Douence, R., Motelet, O. and Südholt, M. A formal definition of crosscuts. In Proceedings of the 3rd International Conference on Reflection. (Kyoto Japan, September 2001)

[5] Douence, R., Fradet, P. and Südholt, M. A framework for the detection and resolution of aspect interactions. In Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (Pittsburgh PA, October 2002)

[6] Duclos, F., Estublier, J. and Morat, P. Describing and Using Non Functional Aspects in Component-based Applications. In Proceedings of the 1st international conference on Aspect-oriented software development. (Enschede The Netherlands, April 2002)

[7] Filman, R.E. Applying Aspect-Oriented Programming to Intelligent Synthesis. Workshop on Aspects and Dimensions of Concerns, ECOOP, Cannes, France, June 2000.

[8] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, September 1993.

[9] Suvée, D., Vanderperren, W., and Jonckers, V. JAsCo: an Aspect-Oriented approach tailored for Component-based Software Development .In Proceedings of the second international conference on AOSD, Boston, USA, march 2003.

[10] Lieberherr, K., Lorenz, D. and Mezini, M. Programming with Aspectual Components. Technical Report, NU-CCS-99-01, March 1999. Available at: http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html.

[11] Ossher, H. and Tarr, P. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In Proc. of the Symposium on SACT: The State of the Art in Software Development. Kluwer, 2000.

[12] Vanderperren, W. A pattern based approach to separate tangled concerns in component-based development. ACP4IS workshop at AOSD 2002.

[13] Vanderperren, W. Localizing crosscutting concerns in visual component-based development.In proceedings of Software Engineering Research and Practice (SERP) international conference, Las Vegas, USA, june 2002.

[14] Vanderperren, W. and Wydaeghe, B. Towards a New Component Composition Process. In Proceedings of ECBS 2001, April 2001.

[15] Wydaeghe, B. and Vandeperren, W. Visual Component Composition Using Composition Patterns. In Proceedings of Tools 2001, July 2000.