Automatic Feature Interaction Analysis in PacoSuite

Wim Vanderperren
Davy Suvée
Bart Verheecke
María Agustina Cibrán
Viviane Jonckers
System and Software Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
{wvdperre,dsuvee,bverheeck,mcibran,vejoncke}@vub.ac.be

ABSTRACT

In this paper, we build upon previous work that aims at recuperating aspect oriented ideas into component based software development. In that research, a composition adapter was proposed in order to capture crosscutting concerns in the PacoSuite component based methodology. A composition adapter is visually applied onto a given component composition and the changes it describes are automatically applied. Stacking multiple composition adapters onto the same component composition can however lead to unpredictable and undesired side-effects. In this paper, we propose a solution for this issue, widely known as the feature interaction problem. We present a classification of different interaction levels among composition adapters and the algorithms required to verify them. The proposed algorithms are however of exponential nature and depend on both the composition adapters and the component composition as a whole. In order to enhance the performance of our feature interaction analysis, we present a set of theorems that define the interaction levels solely in terms of the properties of the composition adapters themselves.

Keywords: Aspect Oriented Software Development – Component Based Software Development – Visual Component Composition - Feature Interaction

1. INTRODUCTION

Both component based software development (CBSD) and more recently, aspect oriented software development (AOSD) have been proposed to tackle problems experienced during the software engineering process. CBSD enables to develop a full-fledged software-system by assembling a set of premanufactured components. Each component is a black-box entity, which can be deployed independently and is able to deliver one or more specific services [11]. The deployment of this paradigm drastically improves the speed of development and the quality of the produced software [6]. In previous research, we developed the PacoSuite development environment that lifts the abstraction level for visual component based software development [14,16]. PacoSuite allows automatic verification of the compatibility between a set of components. Glue-code that translates the syntactical incompatibilities between these components is automatically generated afterwards.

AOSD [1,8] on the other hand, aims at improving the separation of concerns [10] in current software engineering methodologies. When a software system is developed, the different concerns of the application should ideally be described and contained in separate modules. This separation of concerns makes it possible to independently analyze, reuse, change and extend the features provided by a system. Some properties of a software system however cannot be cleanly modularized into one single module as their implementation crosscuts several modules of the system. Typical examples of such crosscutting concerns are synchronization, access control and logging. To solve this issue, AOSD proposes to describe these crosscutting concerns as separate entities, called *aspects*, which are woven into the base implementation of the system later on. This way, other parts of the system are not affected when aspects are added, edited or removed.

Originally, aspect-oriented research and practice focused on modularizing crosscutting concerns in an object-oriented context. However, also component based software development suffers from the problems that arise with the tyranny of the dominant decomposition [10] as crosscutting concerns and tangled code are easily introduced in order to keep the coupling between components as low as possible. To cope with the issue of crosscutting concerns in PacoSuite component-based environment, we proposed the notion of a *composition adapter* [15,16]. A composition adapter describes crosscutting concerns, by specifying a transformation which is able to adapt the original composition pattern. Composition adapters can be visually applied onto a component composition and the described changes are automatically inserted into the component composition by using finite automata theory.

One of the main problems of performing aspect oriented software development using composition adapters in PacoSuite, consists of unpredictable and often undesired side-effects when multiple composition adapters are applied onto the same component composition. This problem is not unique to our approach, but is common to all AOSD approaches and is referred to as the "feature interaction" problem [12]. In this

paper, we propose an approach to detect the possible conflicting application of multiple composition adapters onto the same component composition. The next section, describes the PacoSuite methodology in more detail. Section 3 introduces the composition adapter model and section 4 explains our feature interaction analysis approach. In section 5, we shortly present the tools we developed to support the PacoSuite methodology. Finally, we discuss some related work and state our conclusions.

2. PACOSUITE

The PacoSuite research has been going on for a couple of years at our lab and resulted in the PhD of Bart Wydaeghe [18]. In this paper we do not go into the details of the PacoSuite component based approach, nor do we discuss how this approach relates to other approaches. Instead, the approach is shortly sketched as it is employed as a basis for the remainder of this paper.

The PacoSuite component based research mainly focuses on lifting the abstraction level for component based software development. The goal is to achieve the plug and play concept of component based software development. PacoSuite allows to do automatically check the validity of a component composition. Furthermore, PacoSuite allows to automatically generate glue-code from a visually wired application to translate syntactical incompatibilities between the deployed components. In order to provide these functionalities, components are documented with usage scenarios that specify how to employ them. A usage scenario is expressed by using a special kind of Message Sequence Chart (MSC) [8]. The main difference with a regular MSC is that the signals are taken from a limited set of predefined semantic primitives. In addition, each of these signals contains an implementation mapping on the concrete methods that implement the signal.

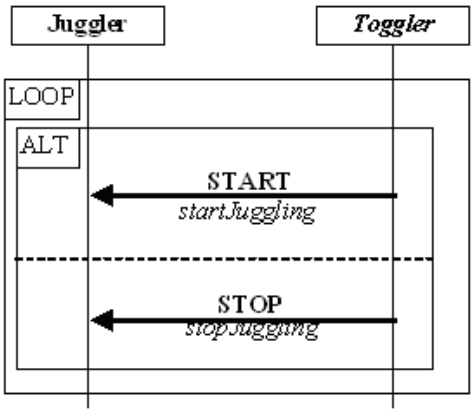

Figure 1: Usage scenario of the Juggler component.

Currently, the PacoSuite methodology is realized for the Java Beans component model. Figure 1 illustrates a usage scenario of the well-known *Juggler* bean from Sun's BeanBox. 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. The usage scenario documents that the *Juggler* component expects consecutive start and stop signals. The START primitive is implemented by the *startJuggling* method and the STOP primitive is implemented by the *stopJuggling* method of the *Juggler* component.

Explicit and reusable composition patterns are introduced as higher-level connectors. A composition pattern is an abstract specification of the interaction between a number of roles and is also expressed by making use of an MSC. The signals between the roles originate from the same limited set of semantic primitives as employed for documenting components. This allows comparing the signals in a usage scenario of a component with these in a composition pattern. Figure 3 illustrates a generic toggling composition pattern. This composition pattern specifies that the *Control* participant consecutively sends either a START or a STOP to the *Subject* participant. A possible application of this composition pattern is a simple visual interface that allows toggling the Juggler component from a single JButton component (see Figure 2). To build this application, the *Juggler* component is mapped on the *Subject* role and the *JButton* component is mapped on the *Control* role.

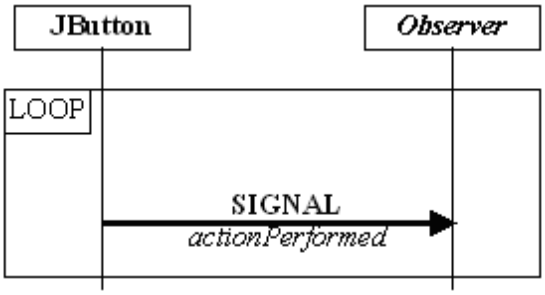

Figure 2: Usage Scenario of the JButton Component.

Notice that even this simple *ToggleControl* collaboration can not be wired by most visual composition environments because the collaboration itself requires state. The *JButton* always fires the same *actionPerformed* event and depending on the state of the interaction *startJuggling* or *stopJuggling* should be called onto the Juggler component.

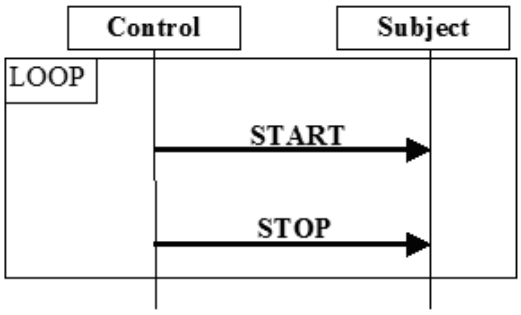

Figure 3: ToggleControl Composition Pattern.

The documentation of both the components and the composition patterns allows checking the protocol compatibility of a component with a role in a composition pattern. For example, the *Juggler* component is clearly protocol incompatible with the *Control* role of the ToggleControl composition pattern because that role sends messages while the *Juggler* is only able to receive messages. We developed an algorithm based on finite state automata to automatically validate the compatibility of a component with a role in a composition pattern. When all component roles are filled, and their compatibility is verified, glue-code is generated that

realizes the composition. Furthermore, the glue-code translates possible syntactical incompatibilities between the collaborating components. In the *ToggleControl* example for instance, the Juggler components expects *startJuggling* and *stopJuggling* messages while the JButton only fires *actionPerformed* events. The translation between both is performed by employing glue-code. PacoSuite also includes an algorithm to automatically generate glue-code from a given component composition. As such, the visual plug-and-play component composition idea is realized. For an in depth explanation of the PacoSuite approach, we refer to [18,19,20].

3. COMPOSITION ADAPTERS

The running example used in this paper consists of dynamically validating timing contracts. This means that the application validates predefined timing contracts at run-time, such as: *"the time between event A and event B has to be shorter than 100 ms"*. In order to achieve this, timestamps have to be taken of application events A and B. In PacoSuite, two different solutions are possible to achieve this dynamic checking of timing contracts. Either all involved components or all involved composition patterns are adapted in order to incorporate this timing contract checking logic. Both solutions however spread and duplicate the concern among several components, which seriously hampers future evolution of both this concern and other concerns required within the application.

In order to modularize crosscutting concerns in PacoSuite, the composition adapter model is proposed [15,16]. A composition adapter is able to specify transformations of a composition pattern, which describe crosscutting concerns in a modular and reusable way. The changes described by a composition adapter are independent of a specific API, as similar to usage scenarios and composition patterns, a composition adapter employs the same set of PacoSuite semantic primitives.

A composition adapter contains two parts, a *context part* and an *adapter part*. A composition adapter specifies that every occurrence of the context part in the target composition pattern needs to be replaced by the adapter part. In other words, the context part describes what needs to be altered and the adapter part describes the transformations themselves.

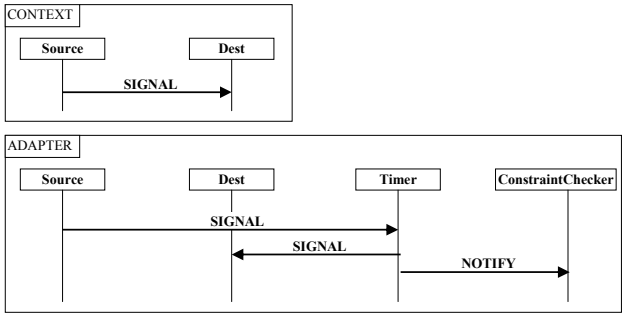

Figure 4: Dynamic timing checker composition adapter.

Figure 4 illustrates a composition adapter for dynamically checking timing contracts. This composition adapter specifies in its context part that it is applicable to every signal sent between certain *Source* and *Dest* role. The adapter part specifies that this signal should be re-routed through a *Timer* role and that a *ConstraintChecker* role should be notified accordingly. 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 one of the imposed timing contracts. The component that is mapped on the *ConstraintChecker* role could do the timing contract verification process offline and/or run on a different CPU in order to minimize the disruption of the software system.

When applying a composition adapter onto a composition pattern, the roles in a composition adapter context part need to be mapped onto the roles of the composition pattern in order to *pattern match* the context part. Roles occurring only in the adapter part operate as newly introduced roles for the target composition pattern. For example, in order to apply the composition adapter depicted in Figure 4 onto the composition pattern of Figure 3, the *Source* role has to be mapped onto the *Control* role and the *Dest* role onto the *Subject* role. The result of applying the dynamic timing checker composition adapter onto the *ToggleControl* composition pattern is depicted in Figure 5. Take in mind that the the PacoSuite primitives are organized into a primitive hierarchy. The SIGNAL primitive is the top-most primitive in the hierarchy and matches with all the lower primitives. As a result, the SIGNAL primitive matches with both the START and STOP primitives. Therefore, the composition adapter is applicable at both these protocol fragments and as a result these are transformed as specified by the adapter part. As such, the START and STOP primitives are not sent directly to the *Subject/Dest* role anymore, but rerouted through the *Timer* role. The component mapped onto the *Timer* role is able to take a timestamp of the corresponding event and notifies the *ConstraintChecker* role.

Notice that mapping the *Source* role onto the *Subject* role and the *Dest* role onto the *Control* role results in an invalid application of the composition adapter because the context part does not occur in the composition pattern. In that case, we declare that this application of the composition adapter does not match with the composition pattern.

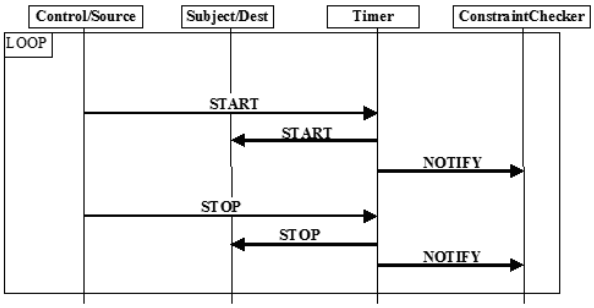

Figure 5: Result of applying the composition adapter of Figure 4 onto the composition pattern of Figure 3.

By employing the composition adapter, the time stamping concern can be inserted in a composition pattern while it is effectively modularized. Removing the time stamping concern from the composition pattern is as simple as deleting the composition adapter. The composition adapter of Figure 4 is also reusable as it specifies an abstract protocol in both its context and adapter parts. As such, the composition adapter is applicable on all composition patterns that contain a protocol fragment that matches with the context part of the composition adapter.

Matching and inserting a composition adapter into a composition pattern seems obvious from the example explained above. In this example, merely syntactically scanning the affected composition pattern would do the job. In case the context part specifies a full protocol however, a more involved algorithm is required. We developed such an algorithm based on finite automata theory in order to automatically match and apply the transformations specified by a composition adapter onto a given composition pattern. In this paper, the algorithm is only shortly sketched. A more elaborate explanation of the algorithm can be found in [11].

The algorithm does not work directly on MSC's but on Deterministic Finite Automata (DFA). The transformation of an MSC to a DFA is a standard process and described in literature [6]. The first step is a verification phase. Here, all paths in the affected composition pattern that correspond to the context part of the composition adapter are looked up. In fact, this amounts to finding all *induced subgraphs* [5] of the composition pattern DFA which are isomorphic to the context part DFA. If there is at least one matching path, the application of this composition adapter is valid. Otherwise, the application of this composition adapter is declared invalid. In the second step, the adapter part of the composition adapter is inserted in the composition pattern at the paths that match with the context part. The last step consists of removing all matching paths.

Although a composition adapter is able to cleanly encapsulate crosscutting concerns, it is quite limited in its expressiveness. As a composition pattern only captures protocol, it is only able to describe protocol transformations. As a result, it is impossible to describe aspects which are able to influence the internal behaviour of the components themselves. Recently, we have introduced an extended version of a composition adapter, namely an *invasive composition adapter*, which is be able to influence the interior behavior of the components' themselves. An invasive composition adapter contains an implementation in the JAsCo [9] aspect oriented language in order to describe these invasive adaptations. The invasive composition adapter model is however out of the scope of this paper. For more information, we refer to [12].

4. COMPOSITION ADAPTER INTERACTION

4.1 Composition Adapter Interaction Analysis

Using PacoSuite, a component composer is able to apply multiple composition adapters onto a single composition pattern. In that particular case, the composition adapters are deployed in the sequence the component composer specifies. However, a composition adapter that is applied as last one could destroy the effect of former applied composition adapters. This "feature interaction" problem is not unique to our approach but is common to many aspect oriented approaches. Several workshops at ECOOP and other conferences have focused on this issue [14] and some solutions are already emerging [2,3,4,8].

In order to cope with the feature interaction problem in the case of composition adapters, we propose to categorize different levels of interference. A severe case of interference consists of a composition adapter that causes another composition adapter to become invalid. In other words, after the application of the first composition adapter, the context part of the second composition adapter does no longer occur. In the following sections, three levels of interaction are presented. To be able to describe these cases correctly, we define the following:

- F and G are composition adapters.
- X is a composition pattern.
- F_C is the context part of F translated to a DFA.
- F_A is the adapter part of F translated to a DFA.
- $A \cong B \Leftrightarrow$ A and B are isomorphic.
- isValid(F,X): $\exists Z$: Z is an induced subgraph of X and $Z \cong F_C$
- **F(X)** is the application of composition adapter F onto the composition pattern X. Returns the resulting composition pattern. This operation is only defined if isValid(F,X).
- χ**(F,G)** = $\{X|\ isValid(F,X) \wedge isValid(G,X)\}$

Definition 1: F and G are *composable* $\Leftrightarrow \forall\ X \in \chi(F,G)$: $isValid(F,G(X)) \wedge isValid(G,F(X))$.

Definition 2: F and G are *orthogonal* $\Leftrightarrow$ F and G are *composable* $\wedge \forall\ X \in \chi(F,G)$: $F(G(X)) = G(F(X))$

Definition 3: F and G are *interfering* $\Leftrightarrow$ F and G are *composable* $\wedge \exists\ X \in \chi(F,G)$: $F(G(X)) \neq G(F(X))$

Definition 4: F and G are *in conflict* $\Leftrightarrow \exists\ X \in \chi\ (F,G)$:

$isValid(F,G(X)) \wedge \neg\ isValid(G,F(X))$

Definition 5: F and G are *mutually exclusive* $\Leftrightarrow$

$\exists\ X \in \chi\ (F,G)$: $\neg isValid(F,G(X)) \wedge \neg\ isValid(G,F(X))$

Orthogonal means that two composition adapters can be safely applied together and no unintended side-effects are able to occur. When applying two composition adapters that are either in conflict or interfering with each other, precedence has to be taken into account. Mutually exclusive composition adapters can never coexist and always result in an invalid composition of both adapters.

4.2 Composition Adapter Classification

The definitions introduced in the previous section are not practical to check whether possible malicious interactions among composition adapters take place because they are described in terms of all possible composition patterns. These definitions could however be reformulated in such a way that they are described solely in terms of a specific composition patterns. As a result, it would be possible to for example verify that composition adapter F and G are orthogonal for a specific composition pattern X. However, the algorithm to apply a composition adapter is of exponential nature. As a consequence, the interaction analysis becomes very resource intensive and could easily lead to state explosions. To overcome this limitation, we propose a classification of composition adapters, based on only the type of a composition adapter itself, which

can be determined beforehand. In the next section a couple of theorems are proposed that allow determining different interaction levels between composition adapters.

The categorization of composition adapters is dependent on the level of changes the composition adapter describes. The level of change is able to range from a composition adapter that adapts nothing to a composition adapter that completely deletes the context part. In the following paragraphs, these different cases are presented and formally defined using automata theory. To be able to describe the different cases correctly, the following notation is introduced:

- **ContainCount(DFA_1,DFA_2)** = #{Z| Z is an induced subgraph of $DFA_2 \wedge Z \cong DFA_1$}

Definition 6: F is *externally fixed* $\Leftrightarrow F_C \cong F_A$

Informally, a composition adapter is called *externally fixed*, if and only if the context part equals the adapter part. In other words, the composition adapter does not change anything at all, at least not externally. An externally fixed composition adapter is not very useful in case of a regular composition adapter as it does not imply any changes. However, an externally fixed composition adapter is able to occur in case an invasive composition adapter is implemented in the JAsCo aspect oriented language. The invasive composition adapter of Figure 6 for instance does not change the context part at all. As a result, it is externally fixed. Figure 6 illustrates an invasive composition adapter which implements a discount business rule for old products in an e-commerce environment. The external protocol of the affected components is not altered. The invasive composition adapter however changes the interior behavior of the product database in order to be able to persistently store and use old product information. Similar to the composition adapter of Figure 4, this invasive composition adapter is externally fixed.

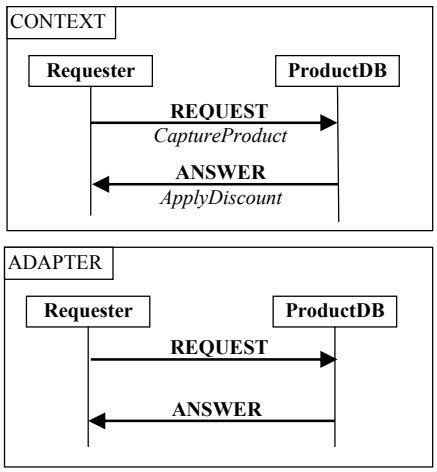

Figure 6: OldProductCount Invasive Composition Adapter

Definition 7: F is *conservative* $\Leftrightarrow$ ContainCount(F_C, F_A) = 1

Informally, a composition adapter is *conservative* if and only if the adapter part only adds extra behavior that doesn't match the context part. In other words, the context part occurs only once in the adapter part. Figure 7 illustrates an example of a conservative composition adapter. The *SecureLogin* composition adapter contains the context part exactly once in its adapter part and adds some extra behavior, namely an optional NOTIFY signal.

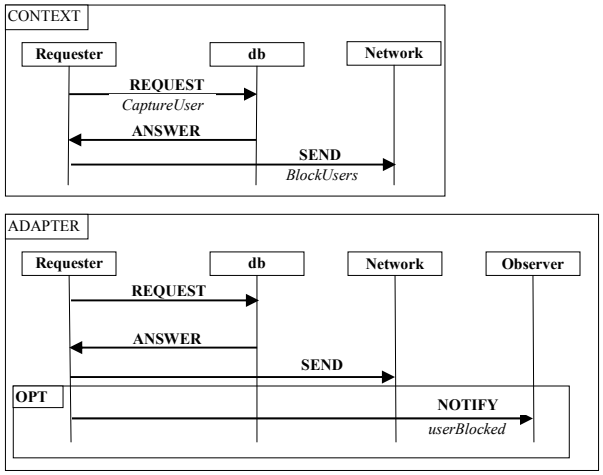

Figure 7: SecureLogin invasive composition adapter.

Definition 8: F is *context preserving* $\Leftrightarrow$

ContainCount(F_C, F_A) > 0

Informally, a composition adapter is called *context preserving* if and only if the context part still occurs in the adapter part. For example, if the composition adapter of Figure 7 would repeat the REQUEST-ANSWER protocol three times, it would still be context preserving, but not conservative anymore.

Definition 9: F is *destructive* $\Leftrightarrow$ ContainCount(F_C, F_A) = 0

Informally, a composition adapter is *destructive* if and only if the context part does not occur in the adapter part. As a consequence, the context part is partially removed.

It is quite easy to prove that the definitions 6 to 8 are ordered from most specific to least specific. In other words, if a composition adapter is externally fixed, it is also context preserving.

4.3. CA Interaction Analysis Revisited

Using the classification of composition adapters introduced in the previous section, it is possible to deduct some theorems that define the different interaction levels in terms of solely the composition adapters themselves.

Theorem 1a: isExternallyFixed(F) $\wedge$

isExternallyFixed (G) $\Rightarrow$ F and G are orthogonal.

If a composition adapter is externally fixed, it does not change anything. As a consequence, an externally fixed composition adapter can be considered as the identity function with respect to the composition adapter application operator.

Theorem 1b: $F_C \neq G_C \wedge \text{containCount}(F_C, G_A) = 0 \wedge \text{containCount}(G_C, F_A) = 0 \Rightarrow$ F and G are orthogonal.

If two composition adapters have different context parts and they do not include the context part of one other in the adaptation they describe, they are never going to interfere with each other and as a result they are *orthogonal*.

Theorem 2: $F_C \neq G_C \wedge \text{containCount}(F_C, G_A) > 0 \wedge \text{containCount}(G_C, F_A) = 0 \Rightarrow$ F and G are interfering.

If a composition adapter adds behavior that matches the context part of another composition adapter and not vice versa, they are *interfering*.

Theorem 3: $F_C = G_C \wedge$ (isDestructive (F) $\wedge$ $\neg$isDestructive (G)) $\Rightarrow$ F and G are in conflict.

If the context parts of two composition adapters are equal and exactly one of them is destructive, then applying the non-destructive composition adapter first amounts to a valid combination. However, applying the destructive composition adapter first, causes the application of the other composition adapter to become invalid.

Theorem 4: $F_C = G_C \wedge$ isDestructive(F) $\wedge$ isDestructive(G) $\Rightarrow$ F and G are mutually exclusive.

If the context parts of two composition adapters are equal and both of them are destructive, then both composition adapters always make each other invalid. As a consequence, they can never be applied together.

5. TOOL SUPPORT

The work described in this paper has been implemented in a prototype tool called PacoSuite. PacoSuite is entirely written in JAVA and consists of two applications, PacoDoc and PacoWire. PacoDoc is a graphical editor that allows drawing, loading and saving component documentations, composition patterns and composition adapters. The PacoWire tool is our actual component composition tool and implements the algorithms we developed in our work [13,15]. It uses a pallet of components, composition patterns and composition adapters. The tool allows dragging a component on a role of a composition pattern. This action is refused when the component does not match the selected role and optionally mismatch feedback is given to the user. A composition adapter can be visually applied onto a composition pattern. The algorithms mentioned in this paper are used to automatically insert the composition adapter into the composition. When all the component roles are filled, the composition is checked as a whole and glue-code is generated automatically. Figure 8 and Figure 9 illustrate some screenshots of the PacoSuite tool suite.

Currently, preliminary support for detecting possible feature interaction problems is provided. A complete implementation of the algorithms described in this paper is subject to future work.

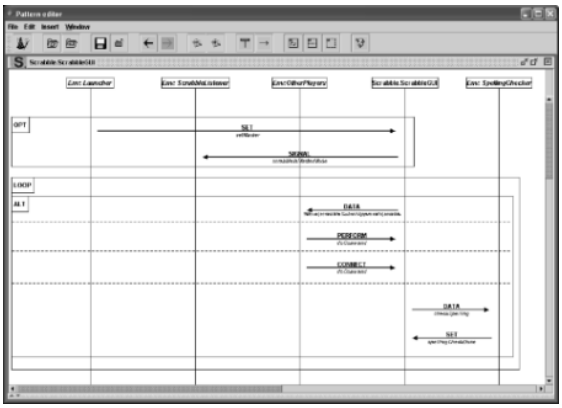

Figure 8: Screenshot of PacoDoc tool that shows the documentation of a Scrabble component in the PacoDoc tool.

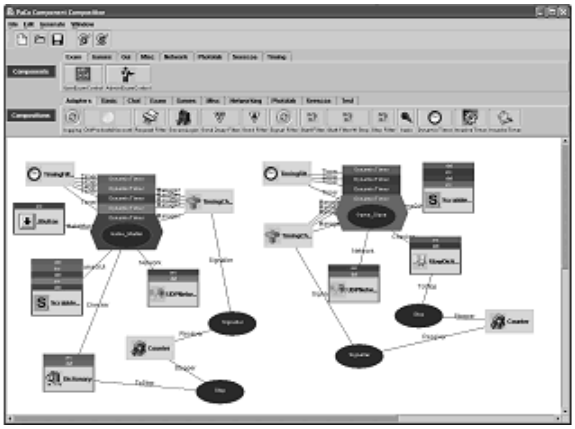

Figure 9: Screenshot that illustrates the visual component composition environment PacoWire. The rectangles represent components, the ovals stand for composition patterns and the hexagonal shapes symbolize invasive composition adapters.

6. RELATED WORK

The AOSD research is under constant evolution and the feature interaction problems encountered are not unique to our approach. Quite some research has been devoted into finding a solution for this problem. They can be divided into three groups: *manual conflict prevention*, *semi-automatic conflict detection* and *automatic conflict detection*.

Some feature interaction problems can be *manually prevented* by describing how a set of aspects should be composed. Brichau et al. [2] modularize aspects as logic metaprograms. For combining aspects, an aspect-combination module is employed. This logic module is parameterized with one or more aspects and contains rules that describe how the functionality of these aspects should be combined. A similar approach is employed in the JAsCo-language [9]. JAsCo provides a mechanism of precedence and user-implemented combination strategies, which specify how a set of aspects should work together. In [3], a number of features interaction problems are described and a set of solutions are proposed. These solutions however also aim at manually describing how aspects should cooperate. As a result no automatic feature interaction conflict resolution is possible.

In [8], a *semi-automatic* conflict detection approach is proposed. Here, the interaction between aspects is described as a set of invariants and post-conditions, which can be checked both at compile-time and run-time. Although this system promotes automatic conflict detection, it is still the responsibility of the software developer to describe which aspects are able to cooperate and which aspects not.

A last approach consists of *automatically checking* whether the deployment of a set of aspects is a valid combination or not. In [4], a formal model is presented which is used to describe the join points and the behavior of aspects. As aspects are described formally, it is possible to detect if a combination of aspect will induce conflicting behavior or not. However, this approach should be mapped on a real implementation language to make it practically useful. The approach described in this paper can be also be categorized in this last set as we achieve automatic conflict detection without any user input.

7. CONCLUSIONS

Using composition adapters, we are able to cleanly modularize crosscutting concerns in the PacoSuite component based methodology. Stacking multiple composition adapters onto the same composition can however lead to unpredictable and undesired side-effects. In this paper, we propose a classification of different levels of side-effects, ranging from totally none to making the resulting composition invalid. Using this classification, it is possible to automatically check whether a certain composition might possibly conflict. The algorithms are however of exponential nature and depend on both the composition adapters and the composition as a whole. Therefore, such a validation becomes unacceptable performancewise. To cope with this problem, we propose a couple of theorems that define the interaction levels based on only properties of the composition adapters. Because the interaction analysis detection depends solely on the composition adapters themselves, it can be calculated beforehand. As a consequence, checking possible malicious interactions becomes easily acceptable performancewise, as it only requires querying a database of cached results.

A critical remark is that the theorems presented in this paper are not complete in the sense that they are only able to proof interaction levels in certain cases. A complete set of theorems is subject for further research.

8. ACKNOWLEDGEMENTS

We owe our gratitude to Ragnhild Van Der Straeten for helping us out with the formal specifications. 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”). Davy Suvée is funded by a doctoral scholarship from the Institute for the Innovation of Science and Technology in Flanders (IWT).

REFERENCES

[1] AOSD Website: http://www.aosd.net.

[2] Brichau, J., Mens, K. and De Volder, K. **Building Composable Aspect-specific Languages with Logic Metaprogramming.** In Proceedings of GPCE 2002. Pittsburgh, USA, October 2002.

[3] Bussard, L., Carver, L., Ernst, E., Jung, M., Robillard, M. and Speck, A. **Safe Aspect Composition.** In Workshop Reader of ECOOP2000. Cannes, France, June 2000.

[4] Douence, R., Fradet, P. and Sudholt, M. **Detection and resolution of aspect interactions.** Rapport de recherché Nr 4435, April 2002.

[5] Grimaldi, R.P. **Discrete and Combinational Mathematics.** Addison-Wesley, Third Edition. 1994.

[6] Heineman, G. T. and Councill, W. T. **Component-Based Software Engineering**. Addison-Wesley. 2001.

[7] Hopcroft, J. E., Motwani, R., and Ullman, J. D. **Introduction to Automata Theory, Languages and Computation.** Addison-Wesley, Second Edition. 2001.

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

[9] Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A. and Murphy, A. **Aspect-Oriented Programming.** In proceedings of the 19th International Conference on Software Engineering (ICSE), Boston, USA. ACM Press. May 1997.

[10] Klaeren, H., Pulvermuller, E., Rashid, A. and Speck, A. **Aspect Composition applying the Design by Contract Principle.** In Proceedings of the GCSE 2000, Second International Symposium on Generative and Component-Based Software Engineering. Erfurt, Germany, 2000.

[11] H. Ossher and P. Tarr. **Using multidimensional separation of concerns to (re)shape evolving software.** Communications of the ACM, 44(10):43-50, 2001.

[12] Parnas, D. L. **On the Criteria to be Used in Decomposing Systems into Modules.** In Communications of the ACM. Vol. 15. No. 12. Pages 1053-1058. December 1972.

[13] Pulvermüller, E., Speck, A., Coplien, J.O., D'Hondt, M. and De Meuter, W. **Proceedings of Workshop on “feature interaction in composed systems” at ECOOP 2001.** Available at: http://www.info.uni-karlsruhe.de/~pulvermu/workshops/ecoop2001.

[14] Suvée, D., Vanderperren, W. and Jonckers, V. **JAsCo: an Aspect-Oriented approach tailored for CBSD.** In Proceedings of AOSD International Conference. Boston, USA, March 2003.

[15] Szyperski, C. **Component software: Beyond Object-oriented programming.** Addison-Wesley, 1998.

[16] 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.

[17] Vanderperren, W., Suvée, D. and Jonckers, V. **Combining AOSD and CBSD in PacoSuite through Invasive Composition Adapters and JAsCo.** Submitted to Node 2003 international conference. Erfurt, Germany, September 2003.

[18] Vanderperren, W. and Wydaeghe, B. **Towards a New Component Composition Process.** In Proceedings of the International Conference on the Engineering of Computer-Based Systems (ECBS). Washington, USA, April 2001.

[19] B. Wydaeghe. **PACOSUITE: Component Composition Based on Composition Patterns and Usage Scenarios.** November 2001. PhD. Dissertation, Vrije Universiteit Brussel. http://ssel.vub.ac.be/pacosuite.

[20] Wydaeghe, B. and Vandeperren, W. **Visual Component Composition Using Composition Patterns.** In Proceedings of the Tools 2001 International Conference. Santa Barbara, USA, July 2001.