Localizing Crosscutting Concerns in Visual Component Based Development

Wim Vanderperren
Vrije Universiteit Brussel
Pleinlaan 2 150 Brussel, Belgium
wvdperre@vub.ac.be

Abstract

This work builds on aspect-oriented development ideas and our previous research where we lift the abstraction level of visual component based development. In component based development, the components are the natural unit of modularization. However, there will always be concerns that cannot be confined to one single component. We introduce composition adapters as a means to localize crosscutting concerns in a separate and reusable entity. A composition adapter describes an adaptation of the interaction protocol between a set of components. An important feature of a composition adapter is that the adaptations are described independent of a concrete API. Using composition adapters, we are able to weave crosscutting aspects in a component based application. The weaving algorithm uses automata theory to allow the state-based insertion of a composition adapter into the interaction protocol. This allows a seamless integration with our component based methodology. We embedded composition adapters and our algorithms into PacoSuite, a visual component composition tool that is used in our lab as a research vehicle. PacoSuite hides the underlying complexity to the component composer, rendering an easy to use visual component based development environment that includes now aspect separation features through composition adapters.

1. Introduction

The software crisis endures already for more than thirty years. Component based development is considered a promising paradigm for curing the so-called software crisis [2]. A component is defined on ECOOP '96 as follows: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [1]. The idea is that applications are created by composing reusable components. Hence both the software quality and the development speed improve substantially. But it's exactly the composition of components that poses a number of problems using current component based approaches. First of all, there is no way to check whether components in a set of components are able to cooperate. Secondly, glue-code that translates syntactical incompatibilities between components has to be written manually most of the time. IBM estimates that 70% of all the code written today consists of such translation of interfaces [2]. Consequently, if we could automate this process in some cases, this would be a big gain. Thirdly, current component composition environments do not allow reusing the collaboration logic between the components. However, the success of design patterns [3] indicates that there are collaboration patterns that are used over and over again. At the System and Software Engineering Lab (SSEL) we propose a solution to cope with these problems. We propose to document components with usage scenarios. A usage scenario specifies how to use the component and is expressed by a Message Sequence Chart (MSC) [4]. We also introduce explicit composition patterns. A composition pattern is an abstract specification of the interaction between a number of roles. Composition patterns are also documented using MSC's. When a component is assigned to a role of a composition pattern, we are able to check whether the component is able to work as the composition pattern prescribes. If all the roles of a composition pattern are filled, glue-code that translates possible syntactical incompatibilities can be generated automatically. These algorithms are based on finite automata theory. This

research has been going on for a couple of years at our lab.

Another research direction that has received lots of attention in the last years is Aspect-Oriented Software Development (AOSD) [5]. Some aspects of an application cannot be cleanly modularized using current software engineering methodologies. Such an aspect is scattered all over the different modules of the system. Similar logic is thus repeated in different modules, with code duplication as a consequence. Due to this code duplication, it becomes very hard to add, edit or remove a crosscutting aspect in the system. The ultimate goal of AOSD is to have a better separation of concerns. Crosscutting aspects have to be separated in a reusable module so that adding, editing and removing an aspect does not affect other modules in the system. The focus of AOSD research has been on separating crosscutting concerns in an object-oriented context. However, the same problem also applies to component based software development. Some aspects cannot be cleanly modularized using our current concepts and are scattered over different components in the system. Typical examples are logging or debugging. More elaborate cases include run-time checking of timing constraints and profile mining. To be able to separate crosscutting concerns in our component based context, we introduce the concept of a composition adapter. A composition adapter describes adaptations of the external behavior of a component independently of a specific API. Because we want to integrate a composition adapter in our current component based methodology, the specification of a composition adapter is also based on MSC's. When a composition adapter is applied on a composition of components, we are able to verify whether this makes sense. Moreover, we are able to automatically insert the adaptations described by the composition adapter into the composition pattern. These algorithms are also based on finite automata theory.

The next part describes our current component based approach. The documentation of components and composition patterns is explained in more detail. Section 3 introduces the composition adapter and section 4 explains the algorithms necessary to automatically insert a composition adapter into a composition pattern. Section 5 presents the tool support that implements these ideas. After a discussion of some related work, we state our conclusions.

2. Research context

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 use the component. A usage scenario is expressed by a special MSC. The main difference with a normal MSC is that the signals are taken from a limited set of pre-defined semantic primitives. Each of these signals is also mapped on the concrete API that performs them. So the documentation of a component is both abstract and concrete. Figure 1 illustrates a usage scenario of a generic TCP/IP network component. One participant of the usage scenario represents the component and the others represent the environment participants the component expects. In this case, there's only one environment participant, namely the NetworkUser participant. This usage scenario documents that the network component either expects data to send over the network or submits events to the NetworkUser environment participant. The component submits an event when he received data, when the connection is established or when he is disconnected.

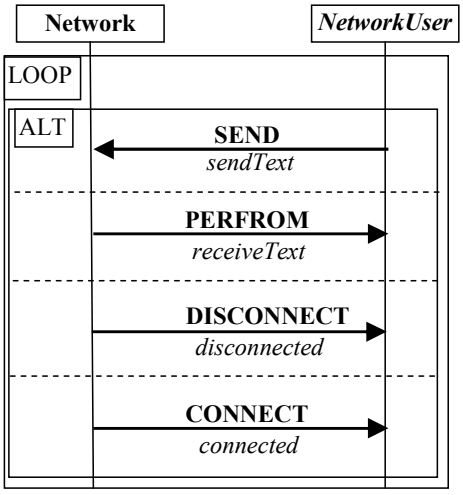

Figure 1: Usage scenario of Network component.

In addition, we introduce explicit composition patterns that are also expressed by MSC's. A composition pattern is an abstract specification of the interaction between a number of roles. The signals between the roles come from the same limited set of semantic primitives. This allows us to compare the signals in a usage scenario of a component with these in a composition pattern. Figure 2 illustrates a generic game composition pattern. This composition pattern specifies the interaction between three roles: the Network, GameGui and Checker roles. One of the applications of this game composition pattern is a distributed scrabble game. The checker role is than filled by a dictionary component that is used to verify the validity of a word. The GameGUI role is filled by a

dedicated Scrabble user interface component. The network role can be filled by the network component of Figure 1.

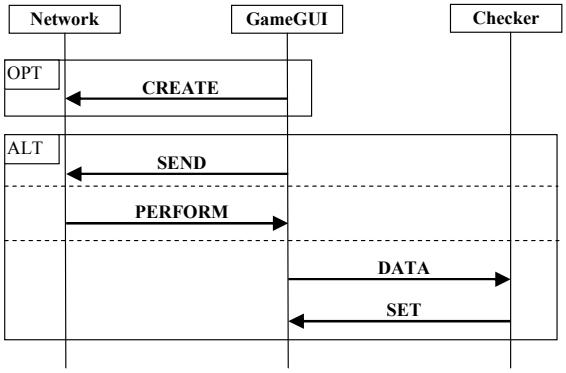

Figure 2: Generic game composition pattern.

The documentation of components and composition patterns allows us to automatically check compatibility of a component with a role. Glue-code that constraints the behavior of the components and that translates syntactical compatibilities is also generated automatically. These algorithms are based on finite automata theory. In this paper, we do not go into the details of these algorithms. The interested reader is referred to [6,7,8].

3. Composition Adapters

3.1. Introduction

Some aspects cannot be cleanly modularized using our current component based approach. Typical examples of such aspects are logging or synchronization. We encountered a more complicated aspect in the SEESCOA[1] research project. In this project we want to verify the quality of service of component based applications. More specifically, we would like to check both statically and dynamically whether a component based application satisfies certain timing constraints. Run-time checking of timing constraints turns out to be a crosscutting concern. If we want to check timing constraints dynamically using our current concepts, we have to alter every composition pattern individually in the same way. Of course, when the application goes into the production phase, we do not want to keep the dynamic timing aspect into the application. Consequently, we have to alter all the involved composition patterns again to remove the timing aspect. To solve this problem, we introduce the concept of a composition adapter.

3.2. Documentation

A composition adapter is able to describe adaptations of the external behavior of a component independently of a specific API. A composition adapter is also documented by a special kind of MSC and consists of two parts: a context part and an adapter part. The composition adapter we use to modularize the timing aspect is depicted in Figure 3. The context part describes the behavior that will be adapted. This can be as simple as one signal send as in Figure 3, but can very well be a full protocol. The adapter part describes the adaptation itself. In the case of the dynamic timing composition adapter every signal between the source and destination role will be re-routed through a Timer role. The Timer role is responsible for taking a timestamp and notifies the ConstraintChecker role. The ConstraintChecker role has a small database of timing constraints and verifies whether every signal it is notified of does not violate a constraint. To minimize the disruption of the system, the component that will be mapped on the ConstraintChecker role could do the verification process offline and/or run on a different CPU.

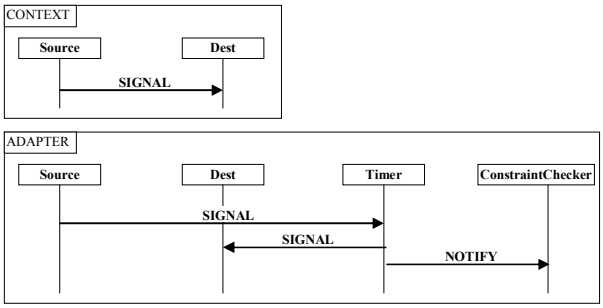

Figure 3: Dynamic timing checker composition adapter.

4. Applying Composition Adapters

4.1. Introduction

When the component composer applies a composition adapter onto an existing composition pattern, the context roles of the composition adapter

[1] The SEESCOA (Software Engineering for Embedded Systems using a Component-Oriented Approach) IWT project is funded by the Flemish government. For more information see: http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/SEESCOA/

have to be mapped onto roles of the composition pattern. For example, suppose we want to time the communication between the GameGUI and Checker roles of the composition pattern in Figure 2. Then we would have to map the Source role of the timing composition adapter of Figure 3 onto the GameGUI role of the composition pattern. Likewise, the Dest role has to be mapped onto the Checker role. The result will be that the DATA signal is not send directly to the Checker/Dest role but is first send to the Timer role. After sending the DATA signal to the Checker/Dest role, the ConstraintChecker role is notified.

Inserting a composition adapter seems obvious from the example explained above. In this example, merely syntactically scanning the affected composition pattern would do the job. However, when the context part of the composition pattern specifies a full protocol, a more involved algorithm is needed. Therefore, we developed an algorithm in three steps based on finite automata theory. The first step is a verification phase. This means searching all paths in the affected composition pattern that correspond to the context part of the composition adapter. If there are no matching paths, the application of this composition adapter makes no sense. In the second step, we insert the adapter part of the composition adapter and in the third phase all paths that match with the context part are removed. The following paragraphs describe these three steps in more detail.

4.2. Step 1: Checking a composition adapter

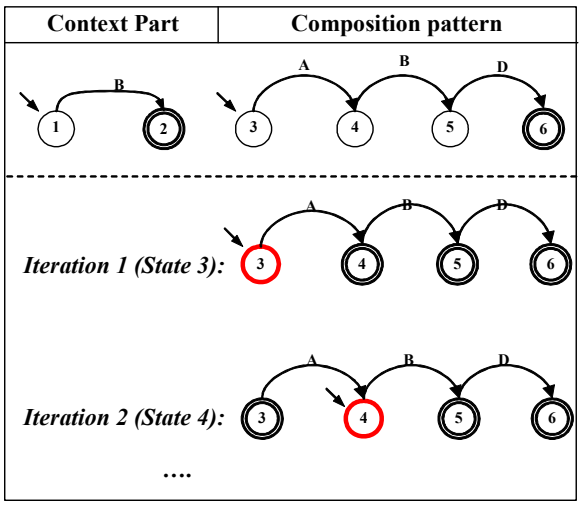

Figure 4: Finding all paths matching with the context part.

The goal of this phase is to search all paths that correspond to the context part of the composition adapter. The algorithm is sketched for a small example in Figure 4. Both the context of the composition adapter and the affected composition pattern are translated to a Deterministic Finite Automaton (DFA). Then, for each state x of the DFA of the composition pattern, we copy this DFA and transform it so that state x becomes the start state and all other states are end states. Subsequently, we calculate the intersection of the transformed DFA with the context DFA. The intersection of DFA's is a standard process and is described in literature [9]. If the intersection is not empty, then we have found a path that matches with the context part of the composition adapter. Notice that to be able to calculate the intersection, we need a mapping from the roles of the composition adapter to the roles of the composition pattern. Recall that the component composer has manually specified this mapping, so there's no problem. After doing this for all states in the composition pattern DFA, we know all paths that correspond to the context part of the composition adapter. In the example of Figure 4, we have one matching path from state 4 to state 5. If there are no such paths, the composition adapter has no effect and a warning is issued.

4.3. Step 2: Inserting the adapter part

In this phase, the adapter part is inserted into the DFA of the composition pattern. Again, we start by translating both the adapter part and the composition pattern to a DFA. Then for each path *p* we have calculated in the previous step (all paths that match with the context part), we insert a copy of the adapter DFA between the start and end state of path *p*. Figure 5 illustrates this algorithm. In this case, we have one path that matches the context part from state 4 to state 5 (denoted by dashed lines). The adapter part is then inserted between state 4 and 5.

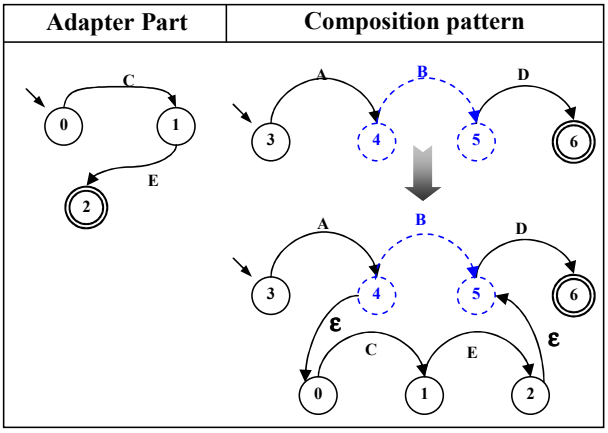

Figure 5: Inserting a composition adapter.

4.4. Step 3: Removing paths that match with the context part

Now, all paths that match with the context part have to be removed. Directly removing the transitions of these paths is not possible, because these transitions could also be part of other paths. Therefore, we invented a more complicated algorithm. This algorithm calculates the difference automaton between the automaton that is generated by the previous phase and a special version of the context DFA. Notice that for calculating the difference automaton, both automata have to be complete. Making an automaton complete is a standard process and is described in literature [9]. But for the context DFA we'll use a different completion algorithm: for every state x which is not an end state of the context DFA, and for every term a in the alphabet of the automaton generated by the previous phase, we add a transition from state x to the start state of the context automaton unless state x already has an outgoing transition with label a. For end states we add a transition for every letter in the alphabet to itself. Figure 6 illustrates the result of this completion algorithm on the context DFA of Figure 4. Notice that due to space constraints the adapted composition pattern automaton in Figure 6 is not made complete. After calculating the difference between the adapted composition automaton and the completed context DFA, the transitions corresponding to the context part are not included in the difference automaton anymore. In the case of Figure 6, the transition B from state 4 to state 5 is deleted.

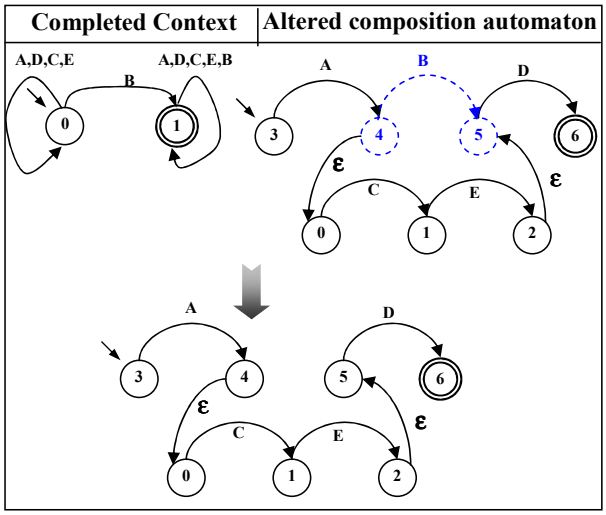

Figure 6: Removing paths that match with the context part.

Notice that these algorithms could render the automaton non-deterministic. In that case we have to make it deterministic again in order to use this automaton. Transforming a non-deterministic automaton to a deterministic automaton is a standard process that is described in literature [9]. Afterwards, this automaton is used to check compatibility with the filled-in components and to generate glue-code using algorithms described in earlier work [6,7,8].

4.5. Composition Adapter Interaction

We allow a component composer to apply multiple composition adapters onto a composition pattern. The composition adapters are inserted in the sequence the component composer specifies. However, a composition adapter that is applied latest could destroy the effect of former composition adapters or vice versa. The "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 problem [16] and some solutions are already emerging.

To cope with the feature interaction problem in our case, we could perform an analysis to verify whether two or more composition adapters possibly conflict. Our solution is based on the observation that two composition adapters can only obstruct each other if their context parts have traces in common. Consequently, we can analyze which composition adapters could possibly obstruct each other. For every pair of composition adapters we calculate the intersection between the context parts. If the intersection is not empty, there is a possible conflict between this pair of composition adapters. The component composition tool then issues a warning. Notice that a more thorough analysis that finds out whether some composition adapters certainly conflict might be desirable. This topic is subject for further research.

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 of component documentation, composition patterns and composition adapters. The PacoWire tool is our actual component composition tool and implements the algorithms we developed in our work [6,7,8]. It uses a pallet of components, composition patterns and composition adapters. The tool allows

dragging a component on a role of a composition pattern. The drag is refused when the component does not match with the selected role and optionally mismatch feedback is given to the user. A composition adapter can be visually applied on a composition pattern. The algorithms described in this paper are then 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. Figure 7 shows some screenshots of our tool.

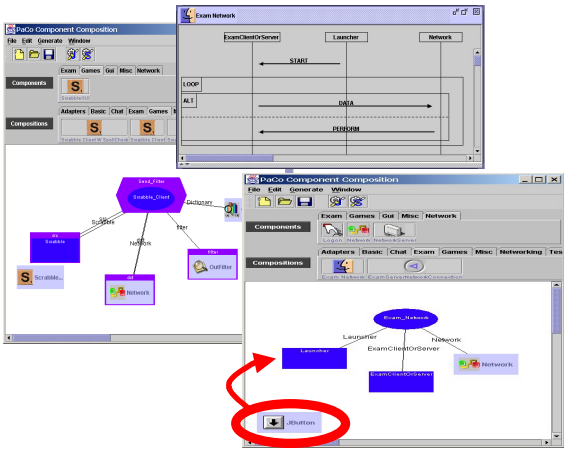

Figure 7: Screenshots of PacoSuite. At the top-right a screenshot of PacoDoc, our documentation tool is shown. At the lower-right, our actual component composition tool called PacoWire is shown. In this screenshot, the component composer is about to map a component on a role of the composition pattern. The leftmost shot shows a composition adapter (in purple) that is applied on a composition pattern.

6. Related Work

There already exists a wealth of generic approaches to separate crosscutting concerns in an object-oriented context. Well known approaches include AspectJ [10], composition filters [11] and HyperJ [12]. However, most of these approaches are not very well suited to be used in a component based context for several reasons: First, components interact in a well-defined manner (e.g. JAVA Beans interact by posting events to interested listeners), so aspects should be able to declare joinpoints specific for the component model. Secondly, components come from different vendors and are not explicitly created to work with each other. In order to make the aspects reusable, the declaration of the aspect behavior has to be separated from the concrete interface of the base component. Finally, source code from third-party components is often not available, therefore source code weaving becomes unfeasible.

Although combining AOSD ideas with component based development is a rather new research direction, some approaches are already emerging. An interesting approach is event based AOSD. Event based AOSD [13] allows specifying crosscuts on events and event patterns using a formal language. Similar to the composition adapter approach, event-based AOSD allows specifying aspects on a full protocol of events instead of a set of methods. Event-based AOSD also uses a wrapping technique to insert the aspects in a given component configuration.

Filman [15] 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 composition adapter idea because both approaches employ a wrapping and filtering technique to insert crosscutting concerns into a composition of components.

The aspectual component approach [14] on the other hand proposes a new component model to be able to specify crosscutting concerns. The declaration of the aspect behavior is independent of the concrete interface of the components. Separate and reusable connectors connect the abstract joinpoints to concrete joinpoints in the components. Eventually, the aspects are weaved into the components using binary code adaptation techniques. The aspectual component approach improves on the composition adapter idea because aspects that alter the internals of a component can be specified. On the other hand, it is impossible to directly recuperate it in our component-based context. Because we do not want to lower the abstraction level, we have to come up with a (preferable graphical) notation of what the consequence of the adaptations on the exterior behavior of the altered components will be. This extra information is needed to allow automatic compatibility checking and glue-code generation.

7. Conclusions

Using composition adapters, we are able to cleanly modularize crosscutting concerns in our component based context. Composition adapters can be verified and inserted automatically in a composition of components. We improve on current aspect-oriented approaches as the joinpoints where the composition adapter will be applied are specified by a full protocol instead of a mere set of methods. Composition adapters

still preserve the high abstraction of our visual component composition methodology. A limitation of this approach is that we can only adapt the exterior behavior of components by re-routing or ignoring their messages.

8. Acknowledgments

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

9. References

1] Szyperski, C., & Pfister, C. (1997). Workshop on Component-Oriented Programming, Summary. In MühlHäuser M. (Ed.) *Special Issues in Object-Oriented Programming – ECOOP96 Workshop Reader*. Dpunkt Verlag, Heidelberg.

[2] Szyperski, C. (1997). *Component Software; beyond Object-Oriented Programming.* Addison-Wesley.

[3] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

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

[5] 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.

[6] Wydaeghe, B. *PACOSUITE: Component Composition Based on Composition Patterns and Usage Scenarios*. PhD Thesis, available at: http://ssel.vub.ac.be/Members/BartWydaeghe/Phd/member_phd.htm

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

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

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

[10] Kiczales G. et al. *An overview of AspectJ.* In Proceedings of the European Conference on Object-Oriented Programming, Budapest, Hungary, 18--22 June 2001.

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

[12] H. Ossher and P. Tarr. *Multi-Dimensional Separation of Concerns and The Hyperspace Approach.* In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000.

[13] R. Douence, O. Motelet, M. Südholt *A formal definition of crosscuts.* Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns, LNCS.

[14] 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.

[15] Filman, R.E. *Applying Aspect-Oriented Programming to Intelligent Synthesis.* Workshop on Aspects and Dimensions of Concerns, 14th European Conference on Object-Oriented Programming, Cannes, France, June 2000.

[16] Workshop on "feature interaction in composed systems" at ECOOP 2001. Program available at: http://www.info.uni-arlsruhe.de/~pulvermu/workshops/ecoop2001/