Separating concerns in a high-level component-based context

Wim Vanderperren[1] Bart Wydaeghe[2]
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 {2962|2975}
{ wvdperre| bwydaegh }@info.vub.ac.be

Abstract

Building on the work of architectural description languages and aspect-oriented programming, we try to improve current visual component composition environments. In our research, we introduced the concept of a composition pattern to lift the abstraction level of current visual wiring to a protocol rather than event/action pairs. This work is summarised briefly in this paper before we present the main topic: composition adapters. In component-based development, the components are the natural unit of modularisation. 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 entity. We use state information deduced from the composition pattern to weave composition adapters into the component-based application. In this paper, we explain how composition adapters are checked on their compatibility with the composition pattern and how this state based insertion of composition adapters is done.

1. Introduction

Component based development is considered a promising paradigm to cure the software crisis [4]. A classic metaphor often used to describe component-based development is the children's building system called Lego. Creating an application becomes as easy as selecting suited building blocks and assembling them together. Obviously, this vision is too naïve. Current visual component composition tools are still far away from reaching this ideal. Current tools do not have support for notifying the component composer of incompatibilities between components. In addition, the wiring between components cannot be reused. However, the success of design patterns proves that the same interaction protocols are used in many different applications. In general, we feel that the abstraction level of current component composition tools is still too low. Therefore, we introduce a specification of the protocol of a component. In addition, we propose to explicitly document generic interactions between components using composition patterns. Using this documentation, we are able to automatically check compatibility of a component with a role in a composition pattern. Moreover, glue-code that translates syntactical incompatibilities between the components is automatically generated. This research has been going on at our lab for a couple of years [9,10] and has been finalized in Bart Wydaeghe's PhD thesis [8].
Building on this research, we investigate how to separate crosscutting concerns in a distinct entity. In an object-oriented context, aspect-oriented programming (AOP) is introduced to modularise crosscutting concerns [5]. However, due to the black box nature of components, the problem of crosscutting concerns proves to be even more difficult in a component-based context. A typical example is accounting behaviour. Every component has to be created with accounting functionality in mind and the same behaviour is scattered over all the components. Consequently, altering the accounting behaviour becomes very difficult. We want to be able to separate crosscutting concerns in a distinct entity. In an object-oriented context, a couple of aspect -oriented programming languages have been proposed, where AspectJ [6] is a well-known example. However, AspectJ is not very well suited to be used in a

[1] Supported by the FWO

[2] This research was partly conducted with the support of the Flemish Government Project: Adapsis (IWT)

component-based context. This is because the point(s) where the aspect will be applied has to be hard coded into the aspect. Therefore, there is no separate connecter to compose an aspect with the other classes. Secondly, AspectJ uses source code adaptation, which is unfeasible in a component-based context. We propose composition adapters to be able to modularise crosscutting concerns in our component-based context. Composition adapters can be automatically checked for validity and automatically inserted into a composition of components.
The next section introduces our component-based approach and explains our documentation of components and composition patterns in more detail. In section 3, our notion of compatibility is elucidated. In addition, we briefly discuss our algorithms to check compatibility and to generate glue-code. Section 4 explains the composition adapter idea in more detail. The algorithms and ideas elucidated in this paper are implemented in a prototype tool. Section 5 presents this tool. Finally, the last section states our conclusions.

2. Documentation

The idea is to document how components and composition patterns should be used. We propose to use a special kind of Message Sequence Charts (MSC's) [1] to document these scenarios. Figure 1 summarizes our scenario syntax. This syntax is mainly the MSC syntax. It contains a set of participants, a set of signal sends between these participants and a set of control blocks and structuring mechanisms. We use the OPT, ALT and LOOP control blocks from the MSC syntax. The OPT keyword means an optional block and the ALT keyword indicates alternatives. The LOOP keyword indicates iteration over a part of the scenario (i.e. zero or more times).

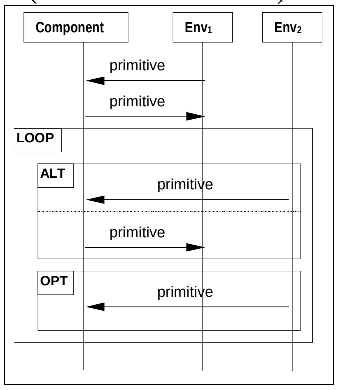

Figure 1: Summary of the scenario syntax

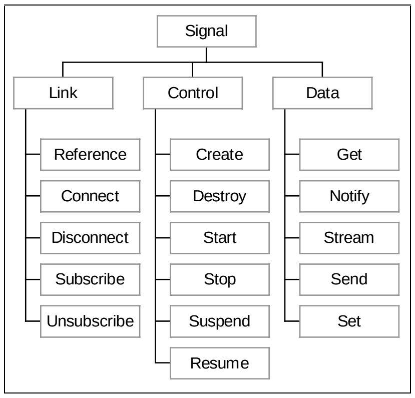

Figure 2: Set of Primitives

Our documentation uses the standard MSC graphical symbols, but the signal sends are taken from a compact set of terms with a known meaning. This stands in contrast with standard MSC messages that are expressed directly in terms of API calls. Building automatic tool support based on concrete API calls is very difficult. The "update" API call in a GUI component for example has not the same meaning as the "update" API call found in a database component. It takes a human and a lot of documentation to distinguish the two. This ambiguity not only burdens the construction of automatic tool support it eventually forces the developers to experiment with the component to see what happens. Figure 2 shows the set of primitives we use in our experiments. These primitives are classified in a simple hierarchy where the *signal* primitive is the most general one. We recognized the need for this kind of hierarchy while modelling output events. This hierarchy is used during the matching process described further in this text in the sense that we allow more specific primitives to map on more generic primitives and vice versa.
From our experience in building a set of primitives for our experiments, we learned that it is very hard to come up with a general set that is usable for all kinds of domains. One should rather construct a set of primitives for a specific application domain. Therefore, we state that this approach is especially useful to build "construction kits". It gives developers the opportunity to build a set of components and to document for that set how they should be used and combined. Part of this research is done for the Advanced Internet Access (AIA) project where we try to build construction kits for Internet services. For this project we built a construction kit that allows us to build all kinds of distributed exams for the Internet (real time, offline, multiple choice, open questions, authorized, non authorized, with or without multimedia, etc.) using this approach. The set we present in Figure 2 proved to be sufficient to document

all components and compositions in this set. This set was constructed during an iterative process of several months. We started with a basic set of primitives that simply seemed to be reasonable and adapted it based on the feedback from people documenting the exam components and compositions.
It is important to note that this set of primitives is just a proof of concept. We do not claim that this is the only set of primitives or even that it is a good set of primitives. We use this set for our experiments only. However, it gives indications on how such a set should look like and how it can be constructed.

2.1.1. Component documentation

We propose to document a component with a number of usage scenarios using the sequence diagrams introduced above. The usage scenario describes the interaction of the component with its environments. Therefore, we introduce the "environment" participant. An environment participant stands for any other cooperating component the component expects. In addition to the environments, a usage scenario for a component contains also one “main” participant that represents the component.
Recall that the signal sends between participants are documented using higher-level primitives. In addition to this abstract documentation, every signal send is mapped on one or more API calls that actually perform the primitive. Figure 3 illustrates a usage scenario of the JButton bean.

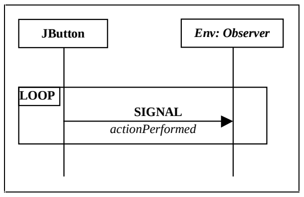

Figure 3: Usage Scenario of JButton bean.

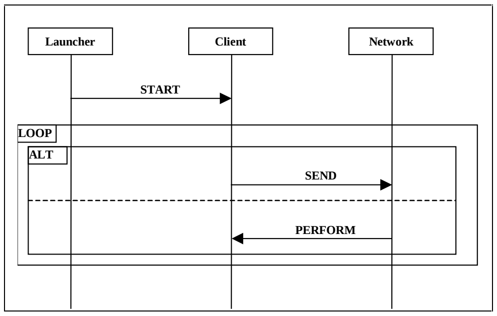

Figure 4: Example of a composition pattern from the exam service construction kit.

2.1.2. Composition documentation

Compositions are documented in a very similar way. I.e. a composition is also documented using a scenario that uses the fixed set of primitives we introduced. A composition scenario describes the interaction between a set of roles and can thus be viewed as a kind of use case for (a part of an) application. As a composition describes an interaction between roles, it does not contain environment participants or implementation mappings. A composition pattern is a high level description of the cooperation between several roles without any indication on how this cooperation will be implemented. Figure 4 shows a composition pattern from the exam service construction kit.

3. Matching

In the previous section, we introduced the documentation for components and composition patterns. The goal of this documentation is to allow automatic compatibility checks and code generation. The following sections describe our notion of compatibility and briefly discuss our algorithms to verify compatibility of a component with a role. For more information about our algorithms and our component-based approach in general, we refer to [8,9,10].

3.1. Compatibility

We distinguish two different kinds of compatibility. A component needs to be compatible with the role it plays in the composition pattern and the combination of a set of components should be compatible with the composition pattern that connects them.

3.1.1. Local Compatibility

We consider composition patterns as reusable entities. This means that a generic composition pattern often provides several alternatives. Figure 5 shows a typical composition expressing an interpretation of observer behaviour. This scenario contains one optional part. After the observer role receives a notification it can refresh its own data by getting the new data or it can ignore the new value (this is the case for example for notifications of a pressed button).

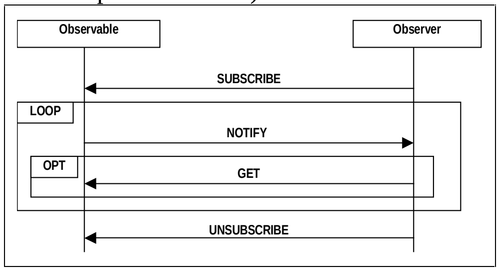

Figure 5: Observable behaviour with an optional block

It is clear that we do not want every component that is used in this composition to implement the optional block. The same goes for several alternatives. Suppose that the composition pattern supports two different kinds of observer connections. One based on notification and another one based on polling. Again, we want components to implement only one of these alternatives.
The situation where the component offers more options and alternatives than what the composition asks for is even more likely. Our exam construction kit for example contains a generic network component that can be used as server or as client. A given application will never use both functionalities at once. Therefore, we can only check if there exist at least one compatible trace through both the component and the corresponding role of the composition pattern. I.e. we check if the component and the role of the composition pattern have common behaviour.

3.1.2. Global Compatibility

The local compatibility as defined above does not guarantee that a composition of components has a *common* compatible trace. Every component in the composition could have another trace in common with its role rendering a composition that deadlocks immediately. Global compatibility means that there exists at least one trace that is common for all the participating components and the composition pattern.
These definitions of compatibility only guarantee that there exists at least one trace as specified by the composition pattern that is supported by all the components. We have no guarantee that one of the components will not follow a different trace than the common trace. Therefore, we generate glue-code that constrains this unwanted behaviour. See section 3.3 for details.

3.2. Matching

Here we explain briefly the global checking algorithm used in our approach. The general idea is to combine the behaviour of all selected components first and to take the intersection with the composition pattern afterwards. The algorithm is now shortly sketched.
The first step is to convert the usage scenario of the components to deterministic finite automata. Using the standard translation algorithm [2], the direction of messages between participants is lost. Because we cannot ignore direction, we had to adapt the translation algorithm slightly to consider direction. Each message or primitive in our case is augmented with a direction tag indicating if a message is send form the component or received by the component. We also add the component to the labels. I.e. we obtain a state machine where the transitions are labelled as in: “ComponentX MessageY out/in”.
Next, we calculate the shuffle automata of these automata to obtain all possible interactions between the components we want to combine. Calculating the shuffle automata itself is a well known process. See [7] for details. As the composition pattern only describes interactions betwene components we are only interested in possible synchronization points between components. Such points are easily found as they comply with the template as shown in Figure 6. I.e. we look for states where one component sends a messages and another component is ready to receive this message. Thus, we contract these “Out/In” couples in the shuffle automata to one transition and prune all other traces. During this step, we also combine the component mappings. For example a transition labelled “C1 A out” followed by a transition labelled “C2 A In” becomes one transition “A (C1,C2) ”. This step is needed to calculate the intersection with the composition automaton.

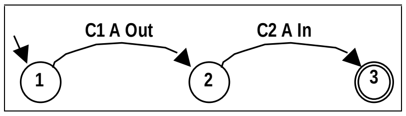

Figure 6: Template for a "valid" trace in the shuffle automaton

The composition automaton contains transitions of the form "A (Role1, Role2)", while the post-processed shuffle automaton contains transitions of the form "A (Component1, Component2)". To calculate the intersection automaton between the composition automaton and the shuffle automaton we need to map roles on components in the composition pattern. The application builder normally provides this mapping by dragging the right component on the role it has to play in the composition pattern. We also developed an algorithm that searches for all possible mappings of roles on components that render a working application.
Finally, we calculate the intersection. If there exists a start-stop path in the resulting automaton, we know that there is at least one possible trace through the selected set of components that complies with a trace in the composition pattern. Our glue code generation forces the resulting application to follow this trace.

3.3. Code generation

We use the Java Beans component model. In this model, component communication is based on events and method calls. More precisely a component sends events to any subscribed listener and any piece of code can call its API. Glue code typical connects output events with a call on another component.
The resulting automaton of the global checking process contains the common behaviour of all components that match with the composition pattern. This automaton will be used as glue code between the components. We generate code that simulates this automaton. This code then translates outgoing events of one component to incoming calls on another component based on the current state. At the mean time, it restricts incompatible traces of components by ignoring illegal events for a given state.
The result is then stripped from all paths that are not members of a start-stop path and a glue code class is generated that implements this result. A main class is also generated were the glue code class and all cooperating components are instantiated. This class also subscribes the glue code class to receive the events of every component that is a member of this composition. If an application contains multiple compositions, a glue code class is generated for every composition. All these classes are then started in their own thread. This allows a component to be part of multiple compositions at the same time.

4. Composition Adapters

4.1. Introduction

After doing some case studies we felt that some concerns cannot be cleanly modularised in our component-based context. For example, to add tracing behaviour, all composition patterns have to be manually altered in the same way. Because we have no way to describe these adaptations in a separate module, new composition patterns that include both the original and the tracing behaviour have to be created.
We see two different possibilities to modularise crosscutting concerns in our component-based context. The first solution consists of using a new component model that allows a component to describe adaptations in other components. Prof Lieberherr and others present a concrete proposal for such a component [3]. They call these components aspectual components. They propose to have a new type of interface that allows components to describe adaptations independent of the concrete components that will be adapted. At composition time, special compositions connect the adaptations with the concrete components. The adaptations are then weaved into the components using binary code adaptation. This approach is very powerful, because the adaptations are described by a programming language (in fact a special version of JAVA). Although this is an interesting approach, 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 behaviour of the altered components will be. This extra information is needed to allow automatic compatibility checking and glue-code generation.
Therefore, we propose to use another alternative, namely having special compositions that could adapt other compositions. Composition adapters are only able to alter the exterior behaviour of components by re-routing or ignoring their messages. However, the code for the compositions is not yet generated, so

adapting these compositions requires no code adaptation whatsoever. This approach is clearly less powerful, but by far a more easier and flexible solution.

4.2. Documentation

We propose to document composition adapters by MSC's similar to regular composition patterns. Composition adapters consist of two parts, a context and an adapter part. The context part describes the behaviour that will be adapted. The adapter part describes the adaptation itself. Figure 7 illustrates an example of a composition adapter. In this example, the composition adapter will re-route every occurrence of a SEND from role *Source* to role *Dest* through a *Filter* role. Suppose we apply this composition adapter to the composition pattern of Figure 7. Then we manually map the *Source* role of the composition adapter onto the C*lient* role of the composition pattern in Figure 8. Likewise, the *Dest* role is mapped onto the *Network* role. The result of applying the composition adapter is that every SEND from *Client* to *Network* will be sent through the *Filter* role (see Figure 8). The *Filter* role and the combined *Source*/*Client* and *Dest*/*Network* roles are afterwards filled in by concrete components. In the aspectual component approach, the Filter component would be an aspectual component that adds Filtering logic either to the component mapped on the *Source* role or the component mapped on the *Dest* role. Notice that from this example it seems useful to be able to express wildcard roles in composition adapters. Wildcard roles would be automatically mapped onto roles of the affected composition. This would free the component composer of manually mapping composition adapter roles.

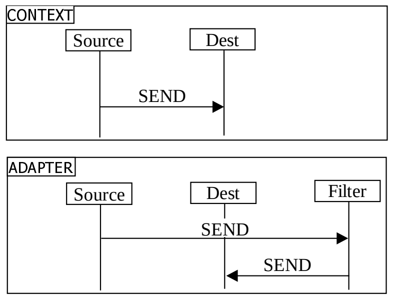

Figure 7: Filtering composition adapter.

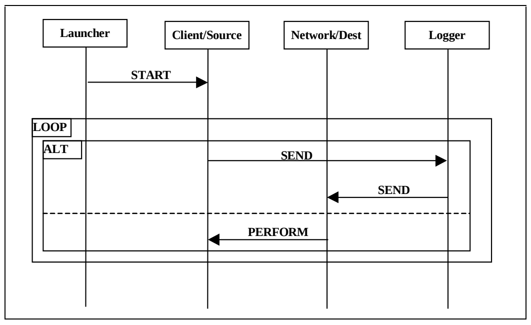

Figure 8: Logging composition adapter applied to the composition pattern of Figure 4.

4.3. Applying a Composition Adapter

Automatically applying a composition adapter requires two steps. In the first step we check whether the adaptation makes sense, this means checking if the context of the composition adapter appears in the composition pattern the composition adapter is applied upon. In a next step, all paths that match with the context part are replaced by the adapter part of the composition adapter.

4.3.1. Checking a Composition Adapter

The goal of this phase is to search all paths that correspond to the context part of the composition adapter. Although this seems obvious from the example in Figure 4, where we just have to search for a SEND in the composition pattern of Figure 2, in most cases syntactically scanning the affected composition won't work. If the context is described by loops and/or other control blocks, a more evolved algorithm that matches the MSC's on a semantic level is needed. The algorithm is sketched for a small example in Figure 9. Both the context of the composition adapter as 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 others states are end states. Subsequently, we calculate the intersection of the transformed DFA with the context DFA. 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 9, 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.

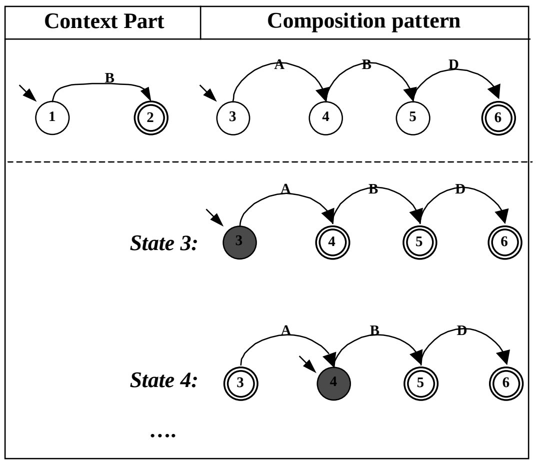

Figure 9: Finding all paths matching with the context part.

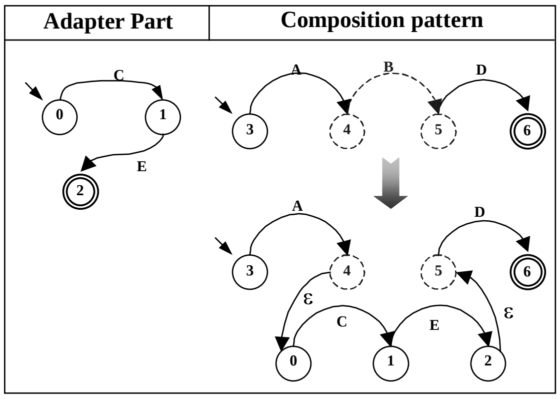

Figure 10: Inserting a composition adapter.

4.3.2. Inserting a composition adapter into the composition

To insert the composition adapter into the composition, we have to replace all paths that correspond to the context part of the composition adapter with the adapter part. Again, we start by translating both the adapter part as the composition pattern to a DFA. Then for each path we have calculated in the previous step (all paths that match with the context part), we remove that path and replace it by a copy of the adapter DFA. Figure 10 illustrates this algorithm. We have one path that matches the context part from state 4 to state 5 (denoted by dashed lines). We remove that path and insert the adapter part between state 4 and 5. Notice that this renders the automaton non-deterministic, so before we are able to use this automaton it has to be made deterministic again. Afterwards, this automaton is used to check compatibility with the filled-in components and to generate glue-code using the algorithms described in section 3 and 4.

4.4. Composition Adapter Interaction

We allow a component composer to apply multiple composition adapters onto a composition. 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. Notice that two composition adapters can only obstruct each other if their context parts have traces in common. Consequently, we can analyse 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 composition tool and implements the algorithms described in this paper. It uses a pallet of components, composition patterns and composition adapters. This 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. If a composition adapter is applied to a composition pattern, the components are checked to be compatible with the adapted composition pattern. When all the component roles are filled, the composition is checked as a whole and glue-code is generated. Figure 11 shows some screenshots of our tool.

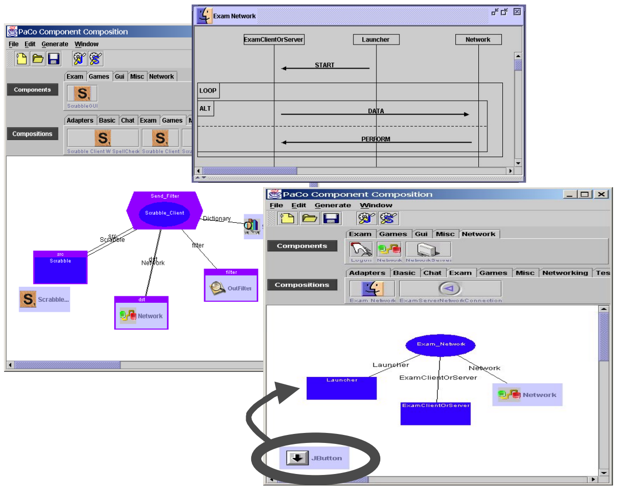

Figure 11: Screenshots of PacoSuite.

6. Conclusions

Using composition adapters, we are able to cleanly modularise 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 join points 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. Additionally, we propose an analysis to warn for possible conflicts between interacting composition adapters. A drawback of this approach is that we can only adapt the exterior behaviour of components by re-routing or ignoring their messages. Composition adapters that are also able to adapt the internals of a component are a topic for further research.

7. References

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

[2] Aho, A.V., Sethi, R. & Ullman, J. D. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1985.

[3] 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.

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

[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] AspectJ home page: http://www.aspectj.org.

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

[8] 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

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

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