Towards a New Component Composition Process

Wim Vanderperren[1] Bart Wydaeghe [2]
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 {2969|2975}
{ wvdperre | bwydaegh }@info.vub.ac.be

Abstract

Component Based Development is considered to be a promising technology to cure the software crisis. However, until now designing and developing component-based applications turns out to be very hard. Current component based development not only involves a component configuration phase, but also writing a lot of glue-code. Much of the existing glue-code in current systems is written to "hack" components together instead of following a careful design. In this paper we introduce a component composition methodology based on the concept of composition patterns. A composition pattern describes an interaction between a set of roles using an extended sequence chart. It serves as a bridge between the design and the implementation. We further propose a component documentation and a set of algorithms based on finite automata theory to perform an automatic compatibility check and glue-code generation to support this methodology.

1. Introduction

Components come in a variety of formats, designs and implementations. Components can be designed to work together or they can be obtained from very different sources. This influences greatly the amount and kind of composition work that is to be done. Two different approaches can be taken in this context. One view is that component composition should not be attempted when components are not specifically designed to work together. The other view - also our opinion - is that components should be glued together until fitting. The first view reduces component reusability and implicitly implies that the way components work together is fixed. The developer is thus not only forced to choose from the available set of components that is designed to work together, moreover he is forced to do it in the prescribed way. Discussions with industrial partners confirm that this is no real option; especially huge components are reused no matter their design.

This means that components should be combined even when there is no direct match (both on a syntactic and semantic level). Depending on the mismatch between the components we want to compose, more or less glue-code has to be written. Component composition thus ranges from plugging together over wrappers and adapters to writing extensive glue-code [1]. The existing component documentation does not offer support to the developer to write this glue-code or even to distinguish a perfect match from a complete mismatch of a set of components.

It is widely believed that component based development follows all major software engineering principles regarding project management and methodology. However, in the current state of component-based technology it is rather pointless to do a thorough design of the project at hand. Indeed, once the design finished, the implementation phase starts with a search for suitable components. Once these are found these components are "hacked" together, ignoring all the beautiful design principles proposed by the design team. For example, in the framework approach, the framework implementation reflects the design and individual applications are all based on this common design. This is clearly a much smoother transition from design to implementation than the component based development process.

[1] Supported by the FWO.

[2] This research was partly conducted with the support of the Flemish Government Project: Advanced Internet Access (ITA II)

In this paper we propose a new methodology for component-based development that cures this problem. This is done by introducing composition patterns. These can be considered as a kind of micro-architectures for the application at hand. We also discuss how the compatibility of a set of components with such an architecture can be checked. In addition, we describe how we automatically generate glue-code between the selected components based on the wanted architecture. Finally, we present a prototype tool that implements these algorithms.

2. Overview of the Solution

To identify components in a design document we propose to look at the roles. A role is typically "filled" by one component. The reason is that a key property of a component is its "independent nature". A component should be independently deployable [2]. This means that its behavior should be self-contained as long as it is not composed with other components. A role has exactly the same property.

To check the compatibility between a role in the design specification and a given component we describe typical component interactions with its environment in a similar way as a design diagram describes role interactions. More specifically we propose to use a special kind of Message Sequence Charts [3] to model composition patterns (role interactions) at the design phase and to use the same diagrams to model typical component usage scenarios (i.e. typical interactions of a component with the environment). Based on this documentation we perform an automated compatibility check using finite automata theory.

Once we identified a set of components that are compatible with the design we need to generate glue-code is generated in accordance with the design.

3. Documentation

The compatibility check between components and a design specification is performed based on sequence chart documentation.

3.1. Syntax

The idea is to document how components should be used. We propose to use a special kind of Message Sequence Charts (MSC's). Each component is documented with a set of MSC's. Each MSC describes a scenario for one of the functionalities supported by this component. The main difference with standard MSC's lies in the kind of signals sent. We developed a compact set of primitives with a predefined meaning.

Instead of using API calls we use these primitives to model the components behavior thus avoiding the confusion that stems from the use of API calls for the signal labels. 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. This section describes these syntax elements and their meaning.

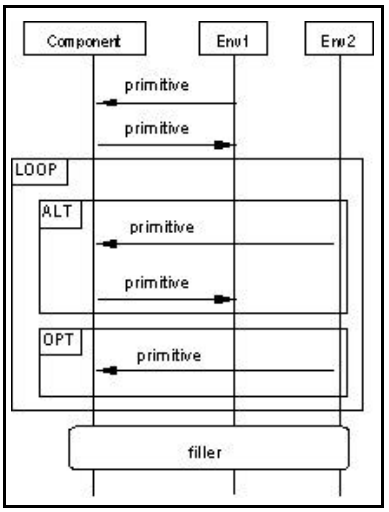

Figure 1: Component Documentation Syntax

3.1.1. Participants

Message Sequence Charts describe interactions between a number of participants. We want to use this documentation to document components. To do this we introduce the "environment" participant. An environment participant stands for any other cooperating component or glue-code. A sequence diagram specifies a contract for any component or glue-code that plays the role of this participant. It specifies what kind of messages the component expects from it and what kind of information or services are sent to or called on it.

As can be seen, a scenario contains exactly one "component" participant. All other participants are "environment" participants. An "environment" participant is labeled "ENV_i". A "component" participant is labeled with the component name.

3.1.2. Messages

Our documentation uses the standard MSC graphical symbols, but the signal sends are taken from a compact set of terms with a known meaning. Those terms are then mapped on the API of the component. This in contrast with standard MSC's 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 also forces the developers to experiment with the component to see what happens. The primitives we propose are used to map API calls from very different sources. Mapping a set of API calls from one component on for example the primitive "CONNECT" indicates that these API calls correspond with a set of other API calls on another component that are also mapped on the primitive "CONNECT". Figure 2 shows the set of primitives we use in our experiments. These primitives are classified in a simple hierarchy. This hierarchy is used during the matching process described further in this text in the sense that we allow subtypes to map on super types and vice versa.

Important note:

The set of primitives we use here is just a prove 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.

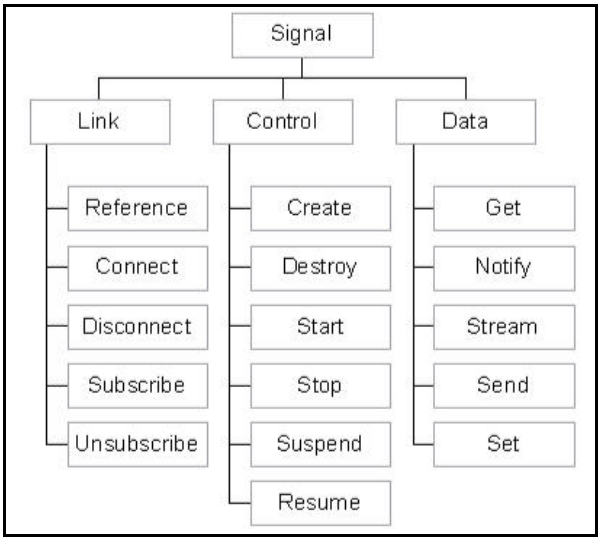

Figure 2: Set of Primitives

From our limited 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.

3.1.3. Control Blocks

We use the OPT, ALT and LOOP keywords 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.

3.2. Component and Composition Documentation

The documentation introduced in the previous sections is used to document both components and compositions. The documentation for components is straightforward. For every component a usage scenario describes the interaction of the component with its environment. Thus our component documentation contains exactly one main participant and a set of environment participants. It also contains an implementation mapping for every message used in this usage scenario. This implementation mapping consists of the real API calls that perform the primitive.

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 part of an application. As a composition describes an interaction between roles, it does not contain environment participants nor implementation mappings. A composition pattern is just a high level description of the cooperation between several roles without any indication on how this cooperation will be implemented. The next section describes how components are checked on their compatibility with a role in such a composition pattern and how glue-code can be generated based on this abstract composition documentation.

4. Methodology

The service development methodology we propose is illustrated in Figure 3. The methodology uses two repositories, namely one with components and one with

composition patterns. In the proposed methodology we not only document components, but we also document composition patterns explicitly. Components are documented with usage scenarios that reflect the typical use of the component. Section 3 explains our documentation in more detail.

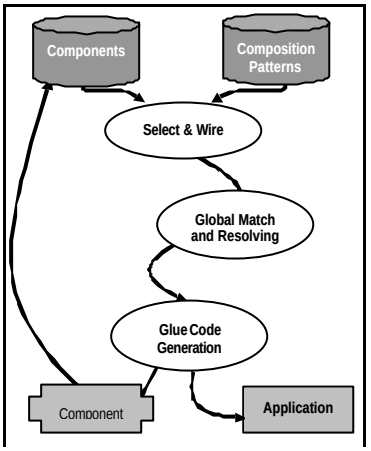

Figure 3: The Component Composition Methodology

Development of a new application or component can start in two ways. Component-centered development starts with the selection of some components, followed by the selection of a compatible composition pattern. The second possibility consists of choosing a composition pattern from the pattern database and then filling in the component roles. This is called pattern-centered development. The component composer can explicitly select a component for each participant in a composition pattern. When the component composer selects a component for a role, the component is verified to be compatible with the selected role of the composition. When all the component roles of a composition are filled, the composition as a whole is checked for validity. If the component composer did not explicitly map each role on a component, an additional step is required. During this step the role-component mappings are resolved automatically. Glue-code that makes this composition work is then automatically generated. This glue-code reflects the behavior of the composition pattern and makes the interacting components cooperate as the composition pattern prescribes. After the glue-code generation is done, the development of the new component or application is finished. If a new component is generated, it can be added to the component repository.

In the next sections, the three major steps in our methodology are explained in more detail.

4.1. Select & Wire

When the component composer selects a component for a role in a composition pattern, this component is checked for compatibility with that role. Our checking algorithm is based on finite automata theory and consists of four steps.

First a projection of the composition pattern with the corresponding role is taken. This projection is needed because interactions between other roles are not relevant for this component.

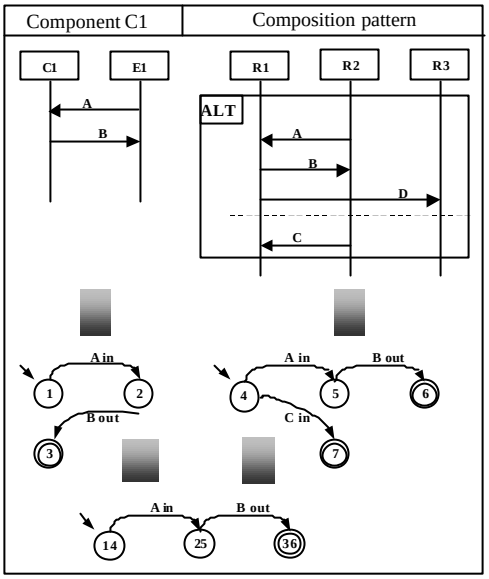

Figure 4: The local checking process.

Next, both scenarios are translated to non-deterministic finite automata (NDFA). Using the standard translation algorithm, the direction of messages between participants is lost. Because we cannot ignore direction, we had to adapt the translation algorithm slightly to take direction into account. Each message or primitive in our case is augmented with a direction tag, namely "out" or "in". These direction tags are in terms of the component in the component documentation and in terms of the role where the component maps on in the composition pattern.

The third step involves the translation of these NDFA's to deterministic finite automata (DFA). Finally, the product automaton of both DFA's is calculated. The calculation of the product automaton is a well-known process. The interested reader can find efficient implementations in [4,18]. If this product automaton is non-empty the MSC's have common behaviour. So, the component is compatible with the selected role of the composition pattern.

Figure 4 illustrates the working of the local checking algorithm. The usage scenario of component C1 is shown on the left. The component is mapped on the second role of the composition pattern. The next step shows the DFA's calculated from both MSC's. The DFA of the composition pattern is restricted to the projection of the composition

with the second role. Notice that each label of a transition is augmented with a direction tag. The product automaton of both DFA's is not empty, so this component can work as the selected role of the composition prescribes.

4.2. Global Check

The next step in the composition process consists of validating the complete composition. This additional check is needed because it is possible that all components match with their intended role, but fail in cooperating with each other. Figure 5 depicts a theoretical situation where all the local checks for the three components at the left hand side succeed but where there is clearly no trace in the three components together that matches with the required trace of the composition. The two first component scenarios select the first alternative and the third component scenario selects the second alternative. It is obvious that this composition is invalid.

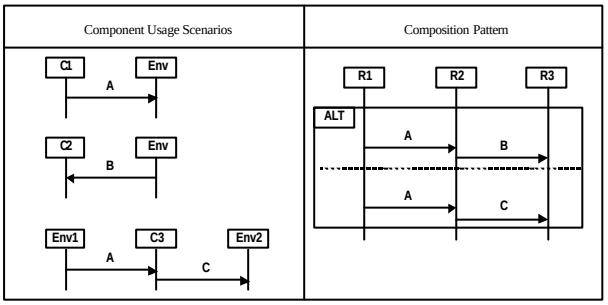

Figure 5: Why a global check is needed

This example clarifies why a global check is needed. In a local check, parts of the composition pattern are "selected" by (or unified with) the component. It is possible for different components to "select" different traces trough the composition pattern that are not necessarily compatible.

During the global check we combine the behaviour of all selected components first and take the intersection with the pattern afterwards. This requires the following steps:

1. Convert the usage scenario of the components to deterministic finite automata (DFA).
2. Calculate the shuffle automata of these automata.
3. Post-process the shuffle automata
4. Add component mapping information
5. Calculate the product automata with the composition pattern
6. Check for a start-stop path in the intersection

These steps are now further explained.

First we convert the usage scenario of the components to deterministic finite automata. This conversion is done exactly as it was done for the local check. The only difference is that we no longer only add relative direction to the transitions but also absolute direction. I.e. we no longer write "A out" but "A out (C1, E1)". This information is needed in the following steps.

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 [5] for details. The resulting automaton contains many traces that are "invalid" without even considering cooperation between components or compatibility with the composition pattern. It is clear that in the combined behaviour of a set of components, any interaction between two components complies to the template as shown in Figure 6.

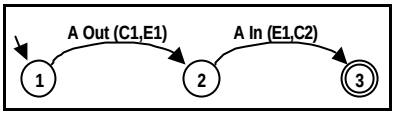

Figure 6: Template for a "valid" trace in the shuffle automaton

The reason is that the shuffle automaton doubles every message in the overall sequence diagram by splitting every message in an "Out" and an "In" part. Therefore 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 "A out (C1,E1)" followed by a transition labelled "A In (E1,C2)" becomes one transition "A (C1,C2) ". This step is needed to calculate the intersection with the composition automaton. As long as every message is split up in two parts we would obtain an empty intersection.

The composition automaton now 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 the selected set of components are able to provide the behaviour as specified by the composition pattern.

4.3. Glue-code generation

As a final step in the composition process, glue-code is generated. This glue-code reflects the behavior of the composition pattern and makes the interacting components

cooperate as the composition pattern prescribes. The glue-code generation happens automatically except for the more involved parameter mappings.

To simulate the composition, we use the automaton generated from the previous global checking process, because that automaton represents the behavior of the composition as a whole. The components do not interact directly because obviously there are some incompatibilities (otherwise there is no need for glue-code). Instead, a number of adapters are generated for each component. These adapters represent the environment the component expects. They just forward all incoming messages to the state machine. Depending on the next state of the state machine, one or more outgoing messages are then sent to other components.

5. Discussion

This section discusses two issues concerning the algorithms above. A first issues concerns the performance of the global check and a second one explains how automatic role component mappings can be deduced from the automata.

5.1. Asymmetric Cross Product

Notice that all the steps in the composition process except for the glue-code generation require calculations that are exponential. Especially the calculation of the shuffle automaton is very expensive (shuffling 10 component automaton with 10 states each results in an automaton with 10^{10} states). Now make the following two observations.

Observation 1: Recall that in the global checking process we need to prune the generated automaton so that only out/in couples remain. Thus the generated automaton is much bigger than needed.

Observation 2: As the global checking process ends with the calculation of the intersection between this pruned shuffle automaton and the composition automaton, it is clear that the result needs to be a restricted version of the composition automaton.

These observations have inspired the construction of a new algorithm. The idea is to skip the calculation of the shuffle automaton and restrict the composition automaton incrementally with the component automaton by calculating a special kind of intersection. It is clear that calculating the intersection of the composition automaton with the automaton of one of the participating components removes all behavior that is not relevant for this particular component. If we then calculate the cross product of a second component with the result we end up with an empty automaton.

Therefore we propose to calculate the product automaton for all related transitions *only* (i.e. all transitions that stand for messages that are send from or to the role mapped on the component we are intersecting with). All other transitions are ignored and left were they are.

Figure 7 gives an example for a very simple component (named C1) and a composition pattern. We calculate the asymmetric cross product between this component automaton and the composition automaton. The result contains a transition A(C5,C6) because this transition has no relation with the component. It is thus left intact. It also contains a transition B(C1,C3) because this transition is related to the component but occurs in both automatons. The transition C(C1,C2) of the composition automaton is pruned because it is related to component C1 but component C1 has no corresponding transition.

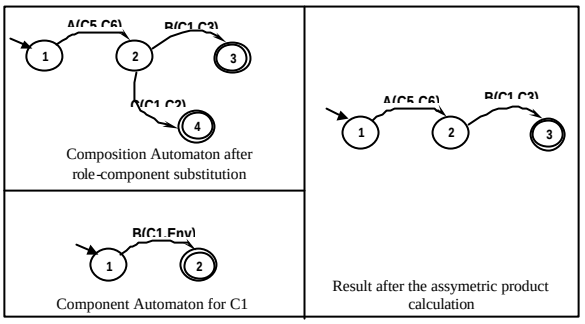

Figure 7: The asymmetric cross product.

Theoretically, both algorithms are exponentially and the performance of both algorithms is equal for the worst-case scenario. However, the new algorithm never performs worse than the shuffle algorithm and in practice the incremental algorithm performs far better.

Another advantage of this algorithm is its incremental nature. This algorithm renders an automaton for partially filled composition patterns. This makes it very well suited for "component generation". I.e. it is possible to take a composition pattern, fill it in partially and use the unfilled part as new environments for a super component. Details about this process are left out of the paper due to space constraints.

5.2. Automatic Role-Component Mapping

All the algorithms in this paper are based on the assumption that the developer provides a role-component mapping for each of the roles in the composition pattern. This is not always easy to do for the developer. We developed an algorithm that calculates all mappings for a set of components and a composition pattern that render a solution (i.e. the components fit in the role and provide the behavior as specified by the composition pattern).

This algorithm is based on the algorithm that calculates the shuffle automaton of the components first and

calculates the intersection afterwards (thus not the incremental algorithm). The only point where a role-component mapping is necessary in this algorithm is to substitute role names with components in the composition automaton before the calculation of the product automaton.

The straightforward algorithm to find these mappings is to calculate the intersection for all permutations of role-component mappings and keep all these permutations that render a non-empty product automaton. This algorithm is clearly too expensive. We developed an algorithm based on dynamic programming to circumvent this performance problem. Due to space constraints we cannot further discuss this in detail.

6. Prototype tool

We implemented a prototype tool to do component composition according to our methodology. Our tool is entirely written in JAVA and consists of two programs, namely PacoDoc and PacoWire. Figure 8 shows screenshots of both tools. PacoDoc is a graphical editor that allows drawing the documentation of both components and composition pattern in a user-friendly manner. The MSC's are stored in XML.

Figure 8: Screenshots of the prototype tools

The PacoWire tool is our actual composition tool. It uses a pallet of both components and composition patterns. 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. It is possible to drag a component on more than one role, so that the same component can be shared among different composition patterns. When all the component roles are filled, the composition is checked as a whole and glue-code can be generated. The tool has an option dialog to select the shuffle or the incremental algorithm and to choose between automatic and manual role-component mappings.

7. Relation with other work

In this article we propose to augment the component interface description with protocols, to document component composition patterns with the same kind of documentation and to use state machine theory to perform protocol compatibility checks and glue-code generation.

Campbell and Habermann's [6] introduced the idea of augmenting interface descriptions with sequence constraints already in 1974. More recent work includes the Rapide system [7] or the PROCOL system [8]. These approaches differ from our proposal because they use unidirectional protocols only. I.e. components are used as a class library where functions are called and output is never actively sent. Other work concerning the translation of interfaces includes the work on glu-ons by Pintado et al. [9], and the interface adaptors of Thatte [10].

This work is extended and improved by Yellin and Strom [11] who use similar ideas as ours to check component compatibility. Their approach is however restricted to two parties. The component composition model used in their approach allows an interface in one component to be bound to an interface in a second component. It does not allow an interface in one component to be bound to multiple interfaces (in several components) as our system does.

Other interesting work regarding protocols can be found in protocol conversion literature [12,13,14]. In this work, protocols are used to specify interfaces and an algorithm is described that synthesizes a converter given the protocols and the specification. The goal of this work is to generate converters from one protocol to another rather than checking compatibility.

Closely related work can be found in Allen and Garlan's work [15] as well as in the work on contracts by Helm et al. [16]. In both models, components may have one or more interfaces, each with its own formal specification based on finite state protocols. Their connectors are first-class reusable components in their own right and can support n-party interactions. However, they support the local checking process only. They provide no mechanism to check whether a set of components can be used to implement the wanted connector (the global checking process). Their work has an interesting advantage above ours in their "mismatch" report. Using theorem provers allows one to generate a trace with an explanation where the match went wrong. We are currently trying to reverse engineer the resulting product automaton on the original composition pattern to provide similar feedback.

Reussner also uses finite automata theory in his "Coconut" project [17] to perform component matching. His work is very similar to ours as far as the local check is involved. At the moment he does not perform a global

check. He uses the incremental algorithm to generate adapters for mismatching components.

Finally, we use parts of the adaptive programming library [18] for efficient implementations of the cross product. This library allows us to calculate the cross product of two non-deterministic automata directly without the need for a (expensive) conversion to deterministic automata first.

8. Conclusions

In the previous we introduce documentation and algorithms that allow us to check a set of components against a specified mini architecture. Building a component-based application can now be done starting from these composition patterns instead of the components. In practice this will be an iterative process where composition patterns are selected based on available components and components are selected based on the specification given by the composition pattern. This process fits in the tradition of Software Development life cycles such as the waterfall model, the iterative model and the spiral model. Composition patterns can be viewed as a kind of use-case for the application. As use cases are developed very early in the software life cycle they provide an excellent link between the different phases.

The whole process is supported by automatic tool support to suggest compatible components for a given composition pattern or compatible composition patterns for a given set of components.

9. Acknowledgements

We like to thank Luc Goossens for his help on several topics, but especially for his idea concerning the incremental algorithm. We also want to thank Bart Michiels, who worked with us on this research, but recently went to the industry. We owe our gratitude to Ralf Reussner, who gave us interesting feedback. In addition, we like to thank Kurt Verschaeve who participated in this research and for proof reading the paper. Finally, we like to thank Prof. Dr. Viviane Jonckers for her invaluable help during our research.

10. References

[1] Mezini, M., Seiter, L. & Lieberherr, K. *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer. *2000*. Available at http://www.ccs.neu.edu/research/demeter/biblio/ComponentIntegration.html

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

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

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

[5] Shaw, A.C. Software descriptions with flow expressions. *IEEE Transactions on Software Engineering*, 4(3):242-254, May 1978.

[6] Campbell, R. & Habermann, A. The specification of process synchronisation by path expressions. In *Proceedings of an International Symposium on Operating Systems*, pages 89-102. SpringerVerlag, April 1974.

[7] Luckham, D., Kenney, J., Augustin, L., Vera, D., Bryan, D. & Mann, W. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering 21*, 1995.

[8] den Bos, J. V. & Laffra, C. Procol a concurrent object-oriented language with protocols delegation and constraints. *Actua Inf. 28*, pages 511-538, June 1991.

[9] Pintado, X. & Junod, B. Gluons: Support for software component cooperation. *Object Frameworks*, pages 311-346, July 1992.

[10] Thatte, S. Automated Synthesis of interface adapters for reusable classes. *In ACM SIGPANSIGACT POPL 94 Conference proceedings.*, pages 174-487, 1994.

[11] Yellin, D. & Strom, R. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292-333, 1994.

[12] Lam, S. Protocol conversion. *IEEE Trans. Softw. 14*, pages 353-362, March 1988.

[13] Okumura, K. A formal protocol conversion method. *In proceedings of the ACM SIGCOMM '86 Symposium*, pages 30-37, July 1992.

[14] Shu, J. & Liu, M. A synchronization model for protocol conversion. In *Proceedings of IEEE Infocom '89*, 1989.

[15] Allen, R. & Garlan, D. Formalising architectural connection. In *Proceedings of the Sixteenth International Conference of an International Symposium on Operating Systems.*, pages 71-80, Sorrento, Italy, May 1994.

[16] Helm, R. Holland, I.M. & Gangopadhay D. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proc. Of the OOPSLA/ECOOP-90: Conference on Object-Oriented Programming: Systems*, pages 169-180, 1990.

[17] Reussner R. Dynamic types for software components. In *ACM Conf. OOPSLA*, 1999.

[18] Lieberherr, K.J. & Patt-Shamir, B. *Traversals of Object Structures: Specification and Efficient Implementation.* Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, July 1997.