# Visual Component Composition Using Composition Patterns

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

***Abstract***

*We improve current visual component composition environments by introducing composition patterns as first class objects that can be defined, stored and reused independently of the components. We document both components and composition patterns using an extended sequence diagram notation. For a component, typical usage protocols are specified while a composition pattern specifies how a set of roles interact. From this documentation, we check whether a component can work as described by a composition pattern using finite state automata theory and we generate glue-code for the composition. In this paper we present our approach, explain the checking algorithms and glue code generation and discuss the tool support we developed.*

## 1. Introduction

Current visual programming environments offer a variety of composition possibilities. Most of these environments provide drag and drop facilities of components on a form and let the interaction between these components be defined by "overriding" the event handlers with free code. I.e. for every component, the programmer has the possibility to select an event and to provide code for it. Typically, code that is written for such events involves calls to methods on other components. This means that the programmer has to know which methods to call, what these methods do and in what order these methods should be called to accomplish the task at hand. With the advent of Java Beans and more precisely with the Java Bean Box, a higher abstraction level was introduced. This environment allows the same drag and drop facilities as the classic environments but are extended with a code generation wizard for the basic interactions. Such an environment allows you to connect a component event with a method of the same or another component. Mind that a developer still needs to know all the components, the meaning of the operations and the order in which these have to be called.

This means that even with the latest visual composition tools, expert knowledge is required to compose a set of components into a working application. The process of selecting suitable components on the contrary does usually not require expert knowledge. Just browsing a catalogue with natural text descriptions will do. If one needs expert knowledge to select suitable components, it is knowledge about the domain not about the technical details of the component. We state that the main reason of this situation lies in the lack of abstraction at the composition level.

### 1.1. Our approach

Therefore, we propose the concept of composition patterns. A composition pattern formally specifies how a set of roles interacts using an extended sequence diagram notation. We provide

---

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

[2] Supported by the FWO

tool support to check compatibility between a component and a role in a composition pattern. Furthermore, glue code is generated automatically to make the components work together as prescribed by the composition pattern.

To achieve automatic checking and glue-code generation we document typical usage protocols for components explicitly. For this documentation, we propose to use a special kind of MSC's (Message Sequence Charts) [1]. Typical composition patterns are documented in a similar way. Mapping the API and events of the components on a common set of messages allows us to check compatibility between the protocol offered by the components and the protocol as specified by the composition.

Checking incompatibilities is based on finite automata theory. In short, we calculate the intersection between the protocols as specified by the parallel composition of all components with the protocol specified by the composition. The result is a new state machine describing the "compatible" behaviour between this set of components, constrained by the protocol specified by the composition pattern. During code generation we generate the source code that implements this result automaton. The details of this process are further explained in sections 3 en 4.

All these technical matters are transparent for the user. During component composition a user works with a visual composition editor that contains a palette of components and a palette of compositions. We developed a prototype of such a tool called PacoWire. 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 roles of a composition pattern are filled, this tool checks the composition and glue-code is generated. This implies that we have a new way to start the development of component-based applications. The classic way to develop component based applications starts with the selection of a set of suitable components, followed by a manual wiring process. By introducing composition patterns we add the possibility to select the interaction scenarios first (based on the design) and search for components that are compatible with these compositions

### 1.2. Component model

The research presented here ignores component model issues and starts from a world where there exists only one component model. This is not a strong constraint as this work is mainly useful to build construction kits for a given domain. I.e. one component builder provides a set of components and a set of typical compositions. It is possible that the component builder reuses a third party component but he needs to bring these in the same component model and provide suitable documentation. The user is only confronted with a set of components using the same component model and a consistent documentation. In practice we use the Java Bean component model throughout this text. We have chosen this component model just for practical reasons. There is no fundamental reason why we could not apply the concepts of this work on other component models.

## 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).

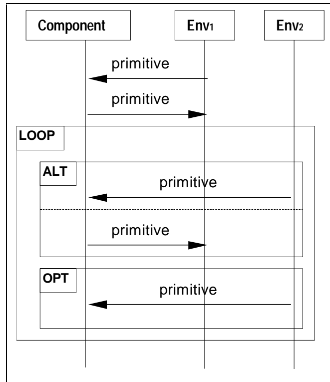


**Figure 1: Summary of the scenario syntax**

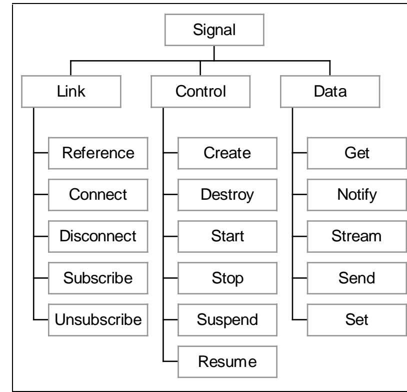


**Figure 2: Set of Primitives**

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. 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.

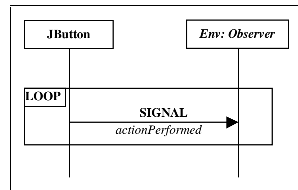


**Figure 3: Usage Scenario of JButton bean.**

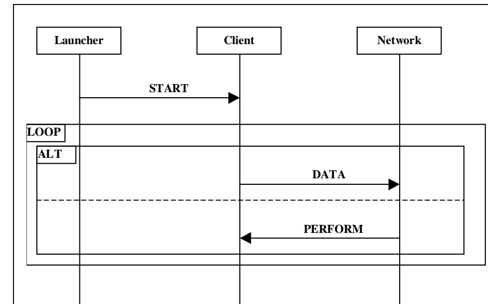


**Figure 4: Example of a composition pattern from the exam service construction kit.**

### 2.2. 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. The following sections describe how we perform compatibility checking of a component with a role in a composition pattern and the compatibility of a set of components with its composing pattern.

### 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.

We consider composition patterns as reusable entities. This means that a generic composition pattern often provides several alternatives. Suppose that the composition pattern supports two different kinds of observer connections. One based on notification and another one based on polling. It is clear that we do not want components to implement both 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.

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 4 for details.

### 3.2. Local match

During the local check we need to compare the behaviour of the component with the behaviour of the corresponding role in the composition. This requires the calculation of the projection of the corresponding role in the composition pattern, followed by the conversion of the usage scenario of the component and the projection of the composition pattern to a (non) deterministic finite automaton ((N)DFA), then we calculate the intersection (or the difference) between these automata and check for a start-stop path in the intersection. These steps are now further explained.

One component only copes with the interactions with its own role. Therefore, as a first step we strip the composition pattern from all interactions between roles other than the mapped role. Next we convert the component and the projection of the composition pattern to a (N)DFA. As the scenarios we use are directly compatible with regular expressions, this conversion is straightforward. The interested reader can find proves of equivalence and efficient implementations in [2].

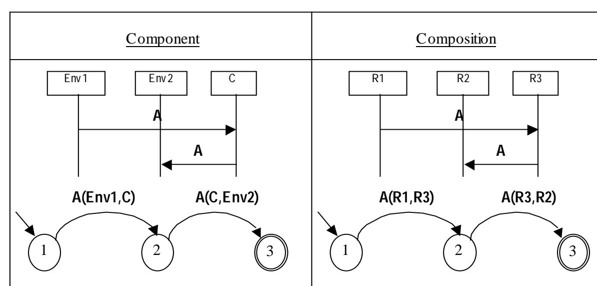


**Figure 5: Adding direction to automata labels during the conversion.**

However, a straightforward conversion loses the “direction” of the arrows. Adding source and destination names to every label does not solve the problem, as we do not know the mapping of the “env” participants of the component. I.e. the alphabet of the component DFA will be different from the alphabet of the projected composition DFA, making it impossible to calculate the intersection in the end. Suppose we do the conversion as in Figure 5 and we want to check that the component on the left hand side fits in role R3 of the composition pattern on the right hand side. At this moment we know that R3 in the composition state-machine matches with C in the component state-machine, but we have no indication for the matching of Env1 and Env2 on R1 or R2. This problem is solved using “relative” direction instead of absolute direction information. I.e. instead of adding source and destination to the labels, we add “In” or “Out”.

For the component conversion we add “Out” for outgoing messages or “In” for incoming messages from the viewpoint of the component participant (a component usage scenario has exactly one component participant, so this is always possible). For the composition projection we add “Out” or “In” from the viewpoint of the mapped role. We also keep the information

about the source and the destination of a message in the state diagram. This information is used later to check the mapping of the “env” participants (see section 3.4 for details).

The next step is to calculate the intersection. This is again straightforward using the algorithms described in [2]. It is interesting however to discuss the difference between taking the intersection and taking the difference between the component and the projection of the composition. We introduced our notion of compatibility in the previous section. Current research in this area tends to put a stronger constraint on compatibility. They demand that the component implements all traces of the role it needs to fill. If we want to strengthen our compatibility definition to this level, we could do that by calculating the difference between the automata instead of the intersection.

Finally we check for a start-sop path in the intersection. Theoretically, a match is found if any path is found. In practice, we need to check the length of the start-stop path to be strictly larger than zero. (One could also argue that the start-stop path should be infinite depending on the interpretation of the life cycle of the used components). It is possible for a product automaton to render a result where the start-state is also a stop-state and where this "path" of length zero is the only start-stop path in the intersection. This is obviously not a valid solution in practice because it means that the component fits in the pattern as long as no events take place at all. When taking the difference, the automaton should be empty to have a match. If the behaviour of a component includes the behaviour of the composition, taking the difference between these behaviours renders the empty set.

### 3.3. Global match

During the global check we combine the behaviour of all selected components first and take the intersection with the composition pattern afterwards. This requires the conversion of the usage scenario of the components to deterministic finite automaton (DFA), then calculating the shuffle automaton of these automata, followed by the post-processing of the shuffle automaton and the addition of component mapping information, subsequently we calculate the intersection of the post processed automaton with the composition pattern and check for a start-stop path in the intersection. These steps are now further explained.

The conversion to deterministic finite automata is done exactly as it was done for the local check. Adding direction to the labels is also done in the same way (i.e. adding “Out” or “In” from the viewpoint of the component to the labels). Next we calculate the shuffle automaton. It corresponds to the total interleaving of the state machines and is obtained by advancing in one automaton at the time and adding the transition to the result. Note now that even without taking the composition into account, we already know that many of the traces in the shuffle automaton are “invalid”. It is clear that in the combined behaviour of a set of components any one interaction between two components complies with the template as shown in Figure 6.

The reason is that in a valid trace a component first sends a message and this message has to be accepted by another component immediately. The shuffle automaton doubles every message in the overall sequence diagram by splitting every message in an “Out” and an “In” part. Thus, all traces that send a message out first and receive *another* message afterwards and all traces that receive a message first and send it afterwards are in practice not valid.

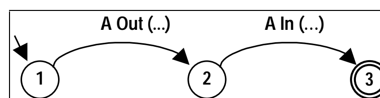


**Figure 6: Template for a single message in the shuffle automaton**

As a last step, we contract these “Out/In” couples in the shuffle automaton to one transition. 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 “(C1,C2) A”. 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 resolved shuffle automaton now contains transitions labelled with a primitive, the originating component, the destination component and the implementation mapping for the (set of) events that needs to be translated in (the set of) method calls. To calculate the intersection between this automaton and the automaton constructed form the composition pattern we need to bring them both in the same alphabet. To do this we use the role-component mapping as provide by the user (recall that we let the user drag components into the roles of the composition pattern). The intersection automaton is now calculated using a match on the primitive and the source and the destination. If there exists a match, we know that the corresponding implementation mapping needs to be implemented in this state in the resulting glue code. Finally we search for a start-stop path in this intersection. If there exists a path with length more than zero we have common behaviour between all components and the composition.

### 3.4. Miscellaneous

Due to space constraints, we cannot describe all the odds and ends that need to be dealt with to make this work. This includes a more efficient algorithm (though less intuitive) to do the global compatibility checking and the resolution of "super types". We need to define the result of matching a message with another message that is lower in the hierarchy (the more specific one gets precedence). It is also possible to obtain non-deterministic behaviour in the automaton used to generate the glue code. This occurs if the same implementation mapping is used for two different messages (user input is needed). We also worked on mismatch feedback. We developed an algorithm based on the intersection of a special kind of non deterministic automata to generate adapters. We further constructed a set of introspection tools so that a user easily gets a view on the problem. We also developed an algorithm based on dynamic programming techniques that automatically calculates a possible role-component mapping for a given set of components and one composition pattern. I.e. the user no longer needs to define what component will be used for what role. In case of multiple solutions our tools asks the user which one to choose. Finally, we also perform a second check on the resolution of the environment participants. I.e. the environment participants of a given component are implicitly mapped by our resolving algorithm on other components. We need to check if one environment is not split over multiple components. However, we do allow that one component is mapped on several alternatives.

## 4. Code generation

As stated in the introduction 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 resolve 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.

## 5. Tool support

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*. *PacoDoc* is a graphical editor that allows drawing the documentation of both components and composition patterns. The MSC's are stored in XML.

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 and optionally mismatch feedback is given to the user. 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 is 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.

## 6. Validation: an exam construction kit

As validation for our theory, we created a construction kit for exam services. This construction kit was created in the context of the Advanced Internet Access (AIA) project. Our work package concentrated on the construction of a service creation platform targeted to end-users. I.e. the environment should be usable by users without a technical background in computer science. In cooperation with our industrial partner Alcatel we developed the higher defined construction kit approach. The application we illustrate here is a distributed exam service. The exam service provides a teacher with the possibility to set up an exam server that provides a set of multiple-choice questions and handles the interaction with the students during the exam. Every student has an exam client that allows him or her to login and connect to this exam server. After login, the student receives the first question from the server. The student selects an answer and sends it back to the exam server. The exam server stores the answer in a database and sends the next question. Once all questions are answered, the exam server produces a report for the teacher that gives an overview of the performance of the student. During the exam, the teacher can follow the progress of all examinees.

Figure 7 demonstrates a snapshot during the development process of our exam application. The user has already filled in all roles, except for the launcher role of the first *StartAndUseExam* composition pattern. He is about to drag the *JButton* component onto this role (indicated by the circle). Using the local checking algorithm we check whether the *JButton* component is compatible with the corresponding role. The drag is refused in case of a mismatch. In this case the drag is accepted, but for example dragging a *UserExamControl* component onto Launcher will be refused. The composition pattern at the top represents the client side of the exam and the composition pattern at the bottom is the administrator or teacher side of the application. Notice that the second button is used in two compositions, i.e. when the button is pressed it will start both the network server and the administrator interface. When the user initiates the glue-code generation, all compositions are checked as a whole using the global checking algorithm and glue-code to make the compositions work is generated. The resulting application is then launched.

To illustrate the power of our approach we show how simple it is to add chat functionality to the exam so that the teacher can chat with its students during the exam. We add two console components and four composition patterns. Both Console components are used as chat window and connected to the network component of the examinee interface and teacher interface through a *use_network* composition pattern. We use a *show* composition pattern to connect the two buttons with their respective consoles so that when the button is pressed the console will become visible.

This is everything needed to extend this application with chat functionality. This little example indicates that the concept of composition patterns allows us to improve on state of the

art visual component composition tools by lifting the abstraction level of the compositions. It allows users to concentrate on the domain knowledge rather than on the technical details. This leads to a construction kit that is easy to use without sacrificing flexibility. The exam construction kit was demonstrated both for our industrial partner Alcatel and during the final review of the Advanced Internet Access (AIA) project. The tool and the demonstration were very well received on both occasions.

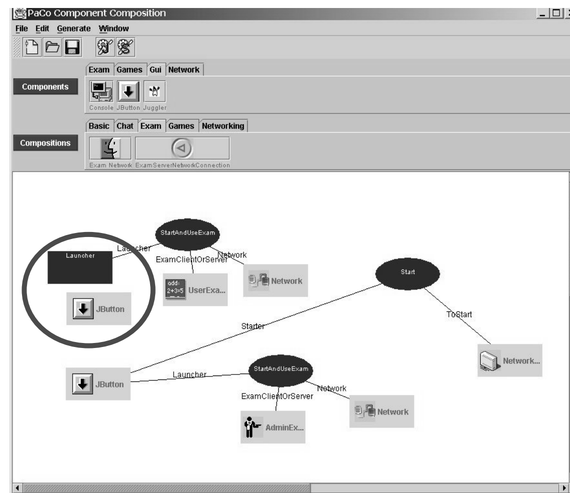


**Figure 7: Snapshot of the development of an exam application.**

## 7. Related 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 [3] introduced the idea of augmenting interface descriptions with sequence constraints already in 1974. More recent work includes the Rapide system [4] or the PROCOL system [5]. 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 gluons by Pintado et al. [6], and the interface adaptors of Thatte [7]. This work is extended and improved by Yellin and Strom [8] 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 [9] [10] [11]. 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. The closest related work can be found in Allen and Garlan's work [12] as well as in the work on contracts by Helm et al. [13]. 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. They use a stronger compatibility rule that allows them to deduce deadlock and livelock freedom using the local checks alone. We extended this work with glue code generation that allows a more flexible compatibility check leading to more generic and more reusable compositions. Using a compact set of known terms to document components and compositions also improves the reusability of our connectors. However, their work has an interesting advantage above ours in the report on

mismatches. Using theorem provers allows one to generate a trace with an explanation where the match went wrong. It has proven to be very difficult to integrate similar feedback in our state machine algorithms.

## 8. Conclusions

In this work, we build on the work of architectural languages to improve current visual component composition environments. This is done using the concept of composition patterns. This concept lifts the abstraction level of current composition techniques to the same level of the components. I.e. compositions are now first class objects that can be defined, stored and reused independently of the components. We further developed and implemented algorithms to perform automatic compatibility checking based on finite automata theory. Finally, we describe how glue code can be generated that constrains incompatible and unwanted behaviours of components based on the constraints specified by the composition pattern. This glue code allows us to use a more flexible compatibility check that leads to more generic and more reusable compositions. This work is mainly useful to build very flexible construction kits. It allows the developers of such a kit to provide default compositions together with their set of components without touching the ability of the users of these construction kits to build very complex compositions that were not foreseen by the developers.

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

[4] 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.

[5] 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.

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

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

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

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

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

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

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

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