# Software Architecture Based Composition of Components

Matthias Anlauff
German National Research Institute
for Computer Science, GMD FIRST
Rudower Chaussee 5
D-12489 Berlin, Germany
+49-30-6392-1827
ma@first.gmd.de

Asuman Sünbül
Technical University Berlin
Computation and Information Structures (CIS)
Sekr.E-N7, Einsteinufer 17,
D-10587 Berlin, Germany
+49-30-31479463
asu@cs.tu-berlin.de

Component-oriented software development is becoming more and more important for the production of large-scale software applications. Today's systems are constructed by combining different software components rather than built from scratch. Therefore, one of the most challenging tasks during the development process of such systems is the one of the software architect: He or she is responsible for the combination of software components on the highest design level. One of the most important problems in this connection is, that in most cases the components themselves are not designed for playing the role of building blocks of bigger systems. The software architect has to deal with components having different characteristics concerning, for example, their heterogeneity, distribution, and interfaces. Especially legacy systems are hard to handle in this respect, because they are not designed to be reused at all. Often, different *styles* are used to describe the architecture of components, so that a direct combination of these components is impossible. Examples for these architectural styles are *pipe-and-filter* style to describe input/output behavior, *event-based* style to describe how a component reacts on certain events, *process-oriented* style to describe the component as independent interacting processes, etc., just to mention a few. Because of these low-level, technical problems concerning the component composition it is very important to provide means that allow system design on a high level of abstraction. Otherwise, if only the technical problems are solved on the implementation level to perform the composition, one might run into danger to neglect the overall design problems of the system.

Software architecture is a discipline that addresses the design of these high-level structures of complex systems. According to several researchers [4], there are many definitions for software architecture around. This partly reflects the immaturity of this active area within software engineering. However, we understand this term as a framework for describing system components and their relationships, especially those attributes, that are consistent across time and implementations and can be regarded as a specification of a class of systems at the same level of abstraction. In the following, we will describe how we solve the problem of composing heterogenous components on the level of architectural descriptions.

Because of the earlier mentioned heterogeneity and complexity of many of todays software developments, it is often not possible to fix a certain architectural style for the design process of the whole software system. The need for multiple styles can come from either the problem domain or the subparts used to construct the system. For example, if a robot control system shall be combined with a component providing a graphical user interface, the robot control component may be described using a process-oriented style, while the architecture of the user interface component is given in an event-based description. In general, trying to combine different software components implies solving the problem of composing different architectural styles. However, only a few approaches cover this aspect of system design; software architects mostly have to rely on ad hoc methods in trusting their own personal experiences while composing different architectural styles [6]. Current practice relies on little more than informal diagrams and notations between these styles [7]. Because of this informality, there is wide space for individual interpretation of these diagrams.

## Service Layer For Component Composition

On the implementation level, the problem of composing components has already been solved by numerous communication protocol implementations as, for example, remote procedure call, remote method invocation, tuple spaces, etc.. Once we know the overall structure of our system including the interplay between the different components, we have reached our goal; the remaining problems can be solved on a technical implementation level. Therefore, our approach addresses the much more challenging task of deriving the component structure of a system from its overall

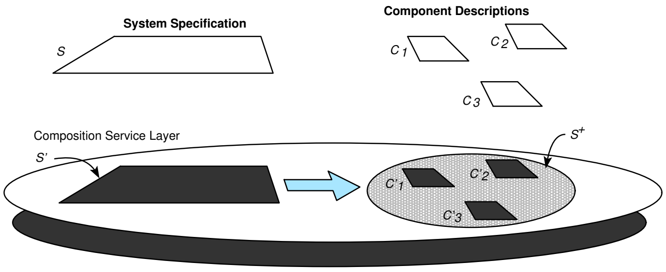


**Figure 1. Service layer for the composition of components**

specification having in mind the existing components that can be used. In order to realize this approach we propose the concept of a *composition service layer* as sketched in Figure 1. Assuming, that we have access to the overall system specification $S$ as well as to the descriptions of the existing components that shall be integrated in the system $C_1, \ldots, C_n$, the functioning of a service layer can be described as follows: Depending on the problem domain, we select a *composition description language* (CDL) which we will use for performing the composition task. *Architectural description languages* (ADLs) like ControlH [5], Rapide [9], Darwin [10], GenVoca [3], ACME[7] may typically be chosen as CDL, but also more general description methods like Petri-Nets, Abstract State Machines [8], or Z are candidates for being chosen as CDL. It is also possible, that a domain-specific specification language, that has explicitly been designed for that composition task, is given as CDL [2]. As the next step the description $S$ and $C_1, \ldots, C_n$ are mapped to corresponding CDL descriptions $S', C'_1 \ldots, C'_n$ which must be consistent with their unmapped versions. This mapping can be done by using standard techniques, like language translation, or the definition of adaptors and wrappers. Finally, as the most challenging task, we transform the overall system specification $S'$ step by step in such a way that it finally contains explicit references to the interfaces of the existing components $C'_1 \ldots, C'_n$. An example of this kind of transformation is the use of refinement techniques in formal description methods [1].[1] The result of these stepwise transformations represents the composition specification of the system. We will call this final specification $S^+$ in order to emphasize that it realizes the "sum" of the components. Finally, we can now analyze the resulting specification $S^+$ aiming at the identification of new components that need to be developed besides the existing ones. As a side-effect, the specification of these newly identified components can then automatically be obtained from the specification $S^+$ and developed accordingly.

We think, that in contrast to existing approaches for component composition, the idea presented here takes the overall system specification directly into account while most other approaches focus on composition of components on a technical level. In case of safety critical software systems, we think that our approach is also of great relevance, because it reduces the number of possible error sources in the process of transforming overall system specifications into realistic system architectures.

[1] In our project, we are using the Abstract State Machine refinement techniques to perform these transformations.

## References


[1] ANLAUFF, M., AND SÜNBÜL, A. Component based software engineering for telecommunication software. In *Proceedings of the SCI/ISAS Conference, Orlando,Florida* (1999). (to appear).

[2] ANLAUFF, M., AND SÜNBÜL, A. Domain-specific languages in software architecture. submitted for publication.

[3] BATORY, D. Intelligent components and software generators. Tech. Rep. 97-06, Department of Computer Sciences, University of Texas at Austin, April 1997. Invited presentation to the Software Quality Institute Symposium on Software Reliability.

[4] BOEHM, B. W., GARLAN, D., KRAMER, J., KRUCHTEN, P., LUCKHAM, D., SALASIN, J., AND WOLF, A. L. ICSE98 Panel: Are new-generation architecture description languages useful. In *ICSE98* (1998).

[5] ENGLEHART, M., AND JACKSON, M. ControlH: A fourth generation language for real-time GN&C applications. In *Proceedings of the CACSD* (Tucson, AZ, March 1994).

[6] GACEK, C., ABD-ALLAH, A., CLARK, B., AND BOEHM, B. On the definition of software architecture. In *ICSE17 Software Architecture Workshop* (Apr. 1995).

[7] GARLAN, D., AND SHAW, M. *Software Architecture: Perspectives On An Emerging Discipline*. Prentice Hall, 1995.

[8] GUREVICH, Y. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1995, pp. 9–36.

[9] LUCKHAM, D., AUGUSTINE, L., KENNEY, J., VEERA, J., BRYAN, D., AND MANN, W. Specification and analysis of system architecture using rapide. In *IEEE Transactions on Software Engineering, Special Issue on Software Architecture* (Apr. 1995), vol. 21(4), pp. 336–355.

[10] NG, K., AND KRAMER, J. Automated support for distributed software design. In *Proceedings of 7th International Workshop on Computer-aided Software Engineering (CASE 95)* (1995).