Research Directions in Software Composition [*]

Oscar Nierstrasz and Theo Dirk Meijler
University of Berne[†]

October 17, 1995

What is Software Composition?

Software composition is the construction of software applications from components that implement abstractions pertaining to a particular problem domain. Raising the level of abstraction is a time-honored way of dealing with complexity, but the real benefit of composable software systems lies in their increased *flexibility*: a system built from components should be easy to recompose to address new requirements [6]. A certain amount of success has been achieved in some well-understood application domains, as witnessed by the popularity of user-interface toolkits, fourth generation languages and application generators. But how can we generalize this?

Frameworks and Architectures

So far progress has been slow, perhaps because too much emphasis has been put on components and too little on how they are composed. A (software) architecture is a description of the way in which a specific system is composed from its components. A flexible application can be achieved if its architecture allows its components to be removed, replaced and reconfigured without perturbing other parts of the application. We see this phenomenon at work in object-oriented development: flexible classes of applications can be defined using *frameworks*, which are specified as hierarchies of reusable, abstract classes. Frameworks are powerful, not because they provide libraries of reusable object classes, but because they define the responsibilities, the collaborations and the interfaces of the fundamental objects of a system, that is, because they define *generic software architectures*.

[*]In *ACM Computing Surveys*, vol. 27, no. 2, June 1995, pp. 262-264.

[†]*Authors' address:* Institut für Informatik (IAM), Neubrückstrasse 10, CH-3012 Bern, Switzerland. Tel: +41 31 631.4618, Fax: +41 31 631.3965, E-mail: oscar, meijler@iam.unibe.ch, WWW: http://iamwww.unibe.ch/~scg

Generalizing from successful approaches to software composition, we see that the notion of a framework has a much broader interpretation. We can then understand component-oriented development in terms of the following three levels:

- **The Framework Level:** A *generic software architecture* is a description of a class of software architectures in terms of component interfaces, composition mechanisms, and composition rules. A *framework* is a generic software architecture together with a set of generic software components that may be used to realize specific software architectures.
- **The Composition Level:** Specific applications are specified as compositions of generic components defined in the framework and new components obtained by specializing the generic ones.
- **The Instance Level:** A composition is instantiated to a running system. System evolution may or may not be possible at this level.

The three-level view of software composition maps well to many successful approaches, but does not tell us anything about what features of those approaches make them work well, or how these approaches can be generalized to work for different domains. Let us consider the research problems in the development of composition models, composition languages, and tools and methods.

Composition Models

Different languages, tools and environments that realize some degree of component-oriented development support various kinds of components and notions of composition, but no common model exists. This makes it hard to describe component frameworks and their architectures in a uniform way, hard to compare approaches, and hard to reason about interoperability between languages and frameworks.

If we consider examples of composable software entities, such as macros, functions, mixins, classes, templates, modules, processes and even complete applications, we can note that composition ultimately boils down to one of: (i) macro expansion; (ii) higher-order functional composition; or (iii) binding of communication channels. Composition in the GenVoca model [2] works essentially by syntactic expansion of parameterized components. Composition with mixins [3] resembles higher-order programming. Composition in Darwin [5] is based on dynamic interconnection of distributed processes.

A comprehensive model of software composition would provide standards for designing, specifying and reasoning about component frameworks.

Composition Languages

Languages for software composition should support the description of software architectures at a sufficiently high level of abstraction. By making the architecture of an individual application explicit, we can achieve a high degree of flexibility and maintainability [5].

Given the three levels of software composition, a language should not only be applicable for defining specific architectures, but also for defining frameworks. It should be possible to define the generic components and the way they can be specialized and linked [2]. Furthermore, it must be possible to rigorously specify composition mechanisms, particularly in concurrent settings [1, 5]. Although a common language for describing all aspects of both frameworks and specific compositions is attractive, a practical composition language must accommodate the use of existing and heterogeneous component libraries and applications.

Tools and Methods

Composition models and languages only give us the means to formally specify software composition. Given a particular software framework, it is an open question how the framework can "drive" software development through all phases of the software lifecycle. This suggests that an important complement to any framework consists of documentation and guidelines that aid developers during requirements specification and analysis to achieve a mapping from the problem domain to the abstractions provided by the framework. Clearly it is not enough to search for reusable software components in a repository late in the development lifecycle: a *software information system* [6] supports the entire lifecycle by providing domain knowledge, requirements models, design guidelines and component frameworks.

During implementation, software composition boils down to editing and combining structures. Successful graphical composition tools are already available for various specific application domains and composition models, but so far general-purpose graphical composition tools have been elusive, due to the difficulty in finding intuitively understandable presentations for arbitrary software abstractions.

A very different issue is how to support the development of frameworks themselves. Experience in developing object-oriented frameworks shows that the process is imprecise and iterative. Currently, only so-called "design patterns" [4] provide any methodological support for creating object-oriented frameworks. By formalizing composition models and emphasizing the role of composition in the software process, we may indirectly arrive at better methods.

References

[1] Robert Allen and David Garlan, "Formal Connectors," technical report,

CMU-CS-94-115, Carnegie Mellon University, March 1994.

[2] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci and Marty Sirkin, “The GenVoca Model of Software-System Generators,” *IEEE Software*, Sept. 1994, pp. 89-94.

[3] Gilad Bracha, “The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance,” Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[4] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.

[5] Jeff Magee, Naranker Dulay and Jeffrey Kramer, *Specifying Distributed Software Architectures*, Proceedings European Software Engineering Conference, Springer Verlag, Lecture Notes in Computer Science, 1995, to appear.

[6] Oscar Nierstrasz and Laurent Dami, “Component-Oriented Software Technology,” in *Object-Oriented Software Composition*, ed. O. Nierstrasz and D. Tsichritzis, Prentice Hall, 1995, pp. 3-28.