Research Directions in Software Composition

OSCAR NIERSTRASZ AND THEO DIRK MEIJLER

Institut für Informatik, University of Bern, CH-3012 Bern, Switzerland, ⟨oscar, meijler@iam.unibe.ch⟩.

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 [Nierstrasz and Dami 1995]. 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*.

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:

- **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.
- **Composition level.** Specific applications are specified as compositions of generic components defined in the framework and new components obtained by specializing the generic ones.
- **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 macro expansion, higher-order functional composition, or binding of communication channels. Composition in the GenVoca model [Batory et al. 1994] works essentially by syntactic expansion of parameterized components. Composition with mixins [Bracha 1992] resembles higher-order programming. Composition in Darwin [Magee et al. 1995] 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 [Magee et al. 1995].

Given the three levels of software composition, a language should be applicable not only to defining specific architectures, but also to defining frameworks. It should be possible to define the generic components and the way they can be specialized and linked [Batory et al. 1994]. Furthermore, it must be possible to specify rigorously composition mechanisms, particularly in concurrent settings [Allen and Garlan 1994; Magee et al. 1995]. 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 specify software composition formally. Given a particular software framework, it is an open question as to 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* [Nierstrasz and Dami 1995] 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" [Gamma et al. 1995] 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

ALLEN, R., AND GARLAN, D. 1994. Formal connectors. Tech. Rep. CMU-CS-94-115, Carnegie Mellon University, March.

BATORY, D., SINGHAL, V., THOMAS, J., DASARI, S., GERACI, B., AND SIRKIN, M. 1994. The GenVoca model of software-system generators, *IEEE Softw.*, Sept., 89–94.

BRACHA, G. 1992. The programming language Jigsaw: mixins, modularity and multiple inheritance, Dept. of Computer Science, Univ. of Utah, Ph.D. Thesis, March.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns*, Addison Wesley, Reading, Mass.

MAGEE, J., DULAY, N., AND KRAMER, J 1995. Specifying distributed software architectures In *Proceedings of the European Software Engineering Conference*. Lecture Notes in Computer Science, Springer Verlag, to appear.

NIERSTRASZ, O., AND DAMI, L. 1995. Component-oriented software technology. In *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis, Eds., Prentice Hall, Englewood Cliffs, N.J., 3–28.