Component Interoperation

Johannes Sametinger

Johannes Kepler University
CD Laboratory for Software Engineering
Institut für Wirtschaftsinformatik
A-4040 Linz, Austria

E-mail: sam@swe.uni-linz.ac.at
Tel: ++43-732-2468-9435
Fax: ++43-732-2468-9430

Abstract

A variety of component types can be reused. Yet different composition techniques and different forms of interoperation of components make reuse difficult or impossible in many situations. We briefly introduce components and component composition. Then we describe forms of interoperation. We propose a classification by introducing an interoperation matrix where we consider control and data aspects of interoperation and distinguish among sixteen different forms of interoperation.

Keywords: software components, software composition, software interoperation, component taxonomy, software reuse

Workshop Goals: discussion/definition of component/composition/interoperation

Working Groups: component certification tools, frameworks and processes; domain engineering tools

1 Background

The author's main research interests include software engineering, software maintenance, software documentation, object-oriented programming, component-oriented programming, and software reuse. Results in the area of reuse include a proposal for reuse documentation [Sam96a], a method for documentation reuse [CS96a, Sam94], a measure for ad-hoc reuse [CS96b], a contribution on a component taxonomy [Sam96b], and empirical studies on reuse [CS96b, CS97].

2 Position

Software components are self-contained, clearly identifiable pieces that describe and/or perform specific functions. This is a broad and general definition. We can have a variety of components like functions, classes, applications, subsystems, design documents, distributed processes, Ada packages, etc. We cannot necessarily reuse any of these components in a given reuse context. A classification system is important to assign components to specific categories and to limit component retrieval to categories appropriate in a certain context.

2.1 Software Composition

Constructing software systems from software components is called software composition. Composable software has a higher degree of flexibility than monolithic software. Different languages and environments realize software composition to different degrees and support different notions of components and compositions. Component-oriented software development requires that we have a selection of reusable components that are plug-compatible. No general model of software composition exists yet [NM95].

2.2 Software Interoperation

If two components interoperate, we have a *sending* component (initiating the interoperation) and a *receiving* component. The sending component activates the receiving component and passes *control* to it. The receiving component *reacts* to the control input; it performs some action and, depending on whether communication is synchronous or asynchronous, returns control to the sending component. Some amount of information is usually passed along with interoperation. If more extensive data exchange is needed, components may use another component for that purpose.

The receiving component may or may not be known to the sending component. This has a major influence on the flexibility of compositions. Interconnection can be between two components (*peer-to-peer*), to a fixed set of components (*multicast*), or to a dynamic set of components (*broadcast*). Static interconnections are peer-to-peer. Dynamic interconnections can be either peer-to-peer, multicast or broadcast. The data component also may or may not be known to both the sender and receiver of interoperation. Fig. 1 gives an overview of the categories resulting from these distinctions. We distinguish *no*, *static*, *dynamic* and *broadcast* for *control* and *data*, which leads to sixteen categories. Some forms of interoperation in this table seem somewhat exotic, for example, the combination *no control* and *dynamic data*. However, they do have practical applications.

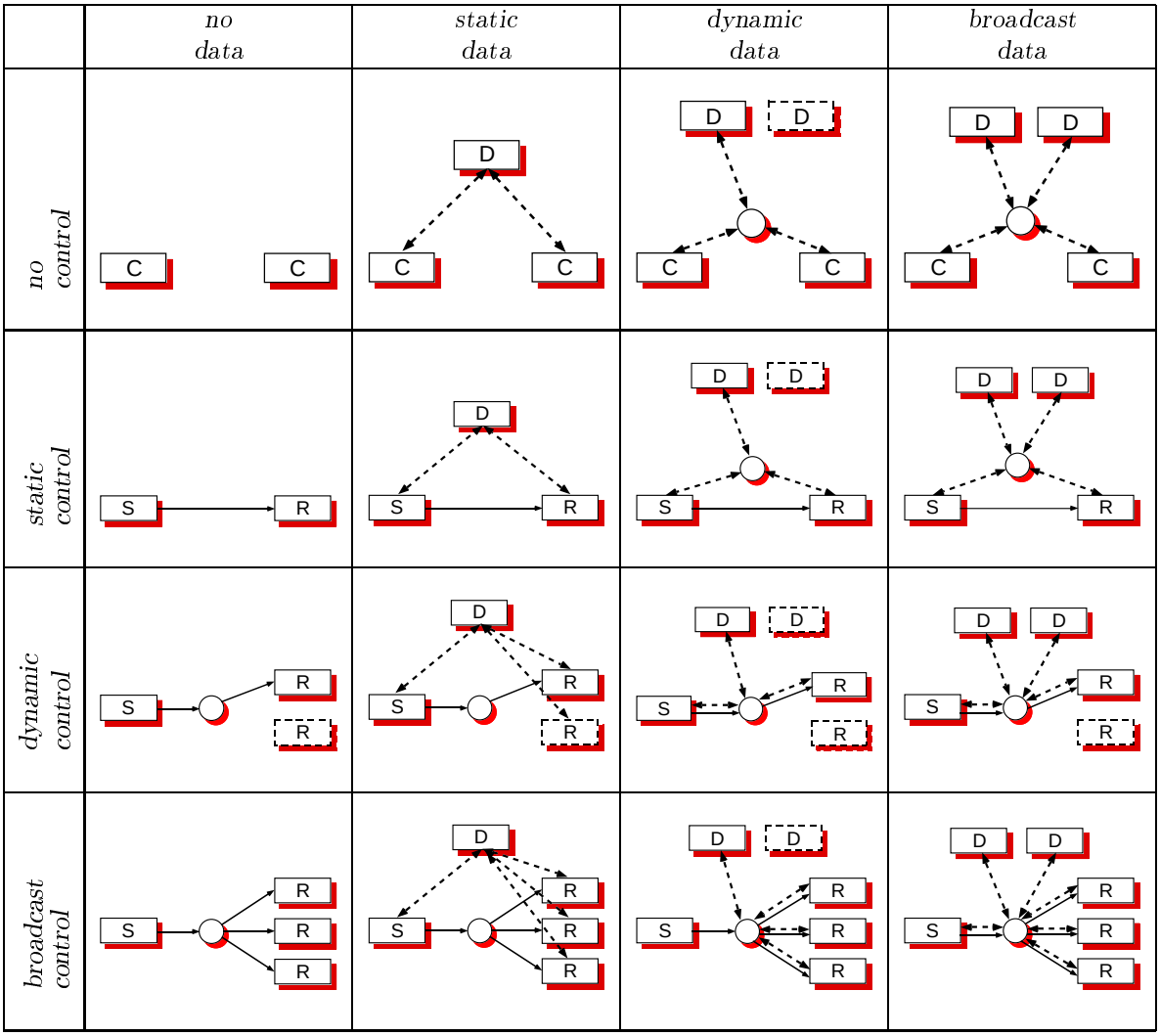

Figure 1: Interoperation Matrix

For software reuse it is essential that components can be composed without having to know each other. This allows component composition without modifying components (*dynamic control*). For example, a function calls a *sort* function. In order to call a function *shellsort* instead, the program text in the calling function has to be modified. Object-oriented programming provides more flexibility through dynamic binding; a calling object does not know the receiver of a call. This makes this object work with a variety of other objects without being modified. Component composition is easiest and most flexible when interconnections among components are not *point-to-point*. Reusing components is easy in environments where each component can react to events generated by any other components and create new events without being aware of any recipients.

Standardized interoperation mechanisms of components on different levels of complexity and granularity are important for software reuse. We must also increase interchangeability of components beyond programming language boundaries and benefit from the availability of different programming paradigms.

3 Comparison

Several definitions of components and reusable components have been provided in the literature. We distinguish two approaches. Components can be seen as some part of a software system that is identifiable and reusable; functions and classes are examples of such components. Components can also be seen as the next level of abstraction after functions, modules and classes. The term *component-oriented programming* (as a successor to *object-oriented programming*) is used in this context.

Examples of definitions in the former category have been proposed by Booch [Boo87], by Holibaugh *et al* [HP88], by Nierstrasz and Dami [ND95], in the *NATO Standard for the Development of Reusable Software Components* [NAT94], by Wegner [Weg93], etc. The definitions are more or less general. Szypersky provides an example of the latter category. He sees component-oriented programming as a refined variation of object-oriented programming [Szy95]. Without explicitly defining a component, Szypersky considers information hiding, polymorphism, late binding, and safety as crucial aspects for component-oriented programming.

Categories of components have also been suggested by Booch, Holibaugh, Nierstrasz, etc. Additional taxonomies have been proposed by Margono and Berard (a modification of Booch's taxonomy) [MB87], by Bradford Kain [BK96], by McGregor et al. [MDK96], and by Dusink and van Katwijk [DvK87]. The presented component taxonomies were created in different contexts and with different motivations. Many concentrate on source code and do not consider higher levels of abstractions.

Component types strongly influence how they are composed. Functional composition and object-oriented composition are typical for functions and classes. Nierstrasz and Dami make a distinction among *functional composition*, *blackboard composition* and *composition by extension* [ND95] and among *macro expansion*, *high order functional composition* and *binding of communication channels* [NM95]. In the literature the terms composition and interoperation are mostly used interchangeably.

References

[BK96] J. Bradford Kain. Components: The Basics: Enabling an Application or System to be the Sum of its Parts. *Object Magazine*, 6(2):64–69, April 1996.

[Boo87] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems.* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[CS96a] Bart Childs and Johannes Sametinger. Literate programming and documentation reuse. In Murali Sitaraman, editor, *4th International Conference on Software Reuse*, pages 205–214. IEEE Computer Society Press, Orlando, Florida, April 23–26, 1996. 1996.

[CS96b] Bart Childs and Johannes Sametinger. Reuse measurement with line and word runs. In *TOOLS Pacific '96*, Melbourne, Australia, November 1996.

[CS97] Bart Childs and Johannes Sametinger. Analysis of literate programs from the viewpoint of reuse. *Software—Concepts and Tools*, to be published, 1997.

[DvK87] E.M. Dusink and J. van Katwijk. Reflections on Reusable Software and Software Components. In Tafvelin [Taf87], pages 113–126. 1987.

[HP88] Robert Holibaugh and J. Perry. Phase I testbed description: Requirements and selection guidelines. Technical Report CMU/SEI-88-TR-13, Software Engineering Institute, Carnegie Mellon University, September 1988.

[MB87] J. Margono and E.V. Berard. A Modified Booch's Taxonomy for Ada Generic Data-Structure Components and their Implementation. In Tafvelin [Taf87], pages 61–74. 1987.

[MDK96] John D. McGregor, Jim Doble, and Asha Keddy. A Pattern for Reuse: Let Architectural Reuse Guide Component Reuse. *Object Magazine*, 6(2):38–47, April 1996.

[NAT94] NATO. NATO standard for the development of reusable software components, volume 1 (of 3 documents), 1994. Public Ada Library: *http://wuarchive.wustl.edu/languages/ada/docs/nato_ru/*.

[ND95] Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. In Nierstrasz and Tsichritzis [NT95], pages 3–28. 1995.

[NM95] Oscar Nierstrasz and Theo Dirk Meijler. Research Directions in Software Composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[NT95] Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice Hall International (UK), December 1995.

[Sam94] Johannes Sametinger. Object-oriented documentation. *ACM Journal of Systems Documentation*, 18(1):3–14, January 1994.

[Sam96a] Johannes Sametinger. Reuse documentation and documentation reuse. In Richard Mitchell, Jean-Marc Nerson, and Bertrand Meyer, editors, *TOOLS 19: Technology of Object-Oriented Languages and Systems*, pages 17–28. Prentice Hall, Paris, France, 1996.

[Sam96b] Johannes Sametinger. On a taxonomy for software components. In WCOP-96. *Workshop on Component-Oriented Programming*, Linz, Austria, July 8, dpunkt, 1996.

[Szy95] Clemens Szyperski. Component-oriented programming: A refined variation on object-oriented programming. *The Oberon Tribune*, 1(2):1, 4–6, December 1995.

[Taf87] Sven Tafvelin, editor. *Ada Components: Libraries and Tools*, Ada-Europe International Conference, Stockholm, Cambridge University Press, May 26–28, 1987.

[Weg93] Peter Wegner. Towards component-based software technology. Technical Report No. CS-93-11, Brown University, 1993.

4 Biography

Johannes Sametinger is assistant professor at the *Johannes Kepler University* in Linz, Austria. His research interests include software engineering, software documentation, software maintenance, software reuse, object-oriented programming, and programming environments. He was a visiting researcher at Texas A&M University and at Brown University for one year each in 1995 and 1996, respectively. He received a Ph.D. in 1991 in computer science from the *Johannes Kepler University*. Dr. Sametinger has also worked with Siemens in Germany on the development of compilers and programming environments.