Components, Scripts and Glue[*]

Jean-Guy Schneider and Oscar Nierstrasz

Software Composition Group, University of Berne,
Institute for Computer Science and Applied Mathematics (IAM),
Neubrückstrasse 10, CH-3012 Bern, Switzerland.
{schneidr,oscar}@iam.unibe.ch
http://www.iam.unibe.ch/∼scg

August 5, 1999

Abstract

Experience has shown us that object-oriented technology alone is not enough to guarantee that the systems we develop will be flexible and adaptable. Even "well-designed" object-oriented software may be difficult to understand and adapt to new requirements. We propose a conceptual framework that will help yield more flexible object-oriented systems by encouraging explicit separation of computational and compositional elements. We distinguish between *components* that adhere to an architectural style, *scripts* that specify compositions, and *glue* that may be needed to adapt components' interfaces and contracts. We also discuss a prototype of an experimental composition language called PICCOLA that attempts to combine proven ideas from scripting languages, coordination models and languages, glue techniques, and architectural specification.

1 Introduction

The last decade has shown that object-oriented technology alone is not enough to cope with the rapidly changing requirements of present-day applications. One of the reasons is that, although object-oriented methods encourage one to develop rich models that reflect the objects of the problem domain, this does not necessarily yield software architectures that can be easily adapted to changing requirements. In particular, object-oriented methods do not typically lead to designs that make a clear separation between computational and compositional aspects; this separation is common in component-based systems.

Component-based systems, on the other hand, achieve flexibility by clearly separating the stable parts of the system (i.e., the components) from the specification of their composition. Components are black-box entities that encapsulate services behind well-defined interfaces. These interfaces tend to be very restricted in nature, reflecting a particular model of plug-compatibility supported by a component-framework, rather than being very rich and reflecting real-world entities of the application domain.

[*] In *Software Architectures – Advances and Applications*, Leonor Barroca, Jon Hall, and Patrick Hall (Eds.), pp 13–25, Springer, 1999.

Components are not used in isolation, but according to a *software architecture* [30] that determines the interfaces that components may have and the rules governing their composition.

This separation of concerns can be seen more clearly when we consider scripting languages. Whereas conventional programming languages are perfectly suitable for implementing software components, scripting languages are designed for configuring and connecting components. The use of scripting languages encourages the development of reusable components ("bricks") highly focused on the solution of particular problems, and the assembly of these components by means of scripts ("mortar").

Still, it may be necessary to adapt the behaviour of components in order to compose them. Such adaptations are needed whenever components have to be used in systems that they have not been designed for. Adaptations of this kind, however, are often nontrivial (if not impossible) and considerable glue code may be needed to reuse components coming from different frameworks [12]. In general, glue code is rather application specific and cannot be reused in different settings, unless well-understood glue abstractions can be used.

We are currently developing an experimental composition language that allows one to express applications as compositions in terms of components, scripts, and glue. In order to support formal reasoning about architectural styles and concrete compositions, this language is being developed with a formal semantics based on $\pi\mathcal{L}$, a variant of the polyadic π-calculus [21].

This chapter is organized as follows: in Section 2, we summarize the state-of-the-art in component technology and analyze problems with existing approaches. In Section 3, we introduce a conceptual framework for software composition as an approach to overcome these problems. We present our ongoing research on the composition language PICCOLA in Section 4, and discuss related work and open problems in Section 5. We conclude with some remarks on future directions.

2 Motivation and State-of-the-Art

In order to cope with the advances in computer hardware technology and rapidly changing requirements, there has been a continuing trend in the development of software applications towards so-called *open systems* [32]. Open systems differ from closed, proprietary systems in the sense that they are not only open in terms of topology (distributed systems) and platform (heterogeneous hardware and software), but particularly in terms of changing requirements: they assume that requirements evolve rapidly and are neither closed nor stable. The essential point is that open systems define a generic (hence reusable) architecture for a family of applications. An individual application may either be considered as an instance of a generic family of applications or a snapshot in time of an evolving application [23]. By viewing open systems as compositions of reusable and configurable software components, we expect to cope better with the requirements of present day applications in general and rapidly evolving requirements in particular.

Component-based software development is always driven by an underlying component framework. A component framework offers a predefined set of reusable and plug-compatible components and defines a set of rules how components can be instantiated, adapted, and composed. Component-based software development has the advantage that applications do not have to be developed from scratch: new systems can benefit from well-understood properties and important design decisions of previous systems and increased flexibility and adaptability during maintenance and evolution.

Object-oriented programming languages and analysis and design methods provide a well-suited tool-box for component-based software development, but current practice shows that the technology is often applied in a way that hinders the development of open systems.

Object-oriented analysis and design methods are domain-driven, which usually leads to a design based on domain objects. Most of these methods make the assumption that an application is being built from scratch and they incorporate the reuse of existing components and architectures too late in the development process (if at all) [28].

In order to successfully plug components together, it is necessary that i) the interface of each component matches the expectations of the other components and ii) that the "contracts" between the components are well-defined. Therefore, component-based application development depends on adherence to restricted, plug-compatible interfaces and standard interaction protocols. However, the result of an object-oriented analysis and design method generally is a design with rich object interfaces and non-standard interaction protocols.

Object-oriented programming languages have been very successful for implementing and packaging components, but they offer only limited abstractions for flexibly connecting components and explicitly representing architectures in applications. Given the source code of an object-oriented application, it is quite easy to identify the components. However, it can be notoriously difficult to tell how the system is composed. The reason is that object-oriented source code exposes *class hierarchies*, not *object interactions*. In addition, the way objects are interconnected is typically distributed amongst the objects themselves, which hinders a clean separation between computational and compositional features.

Although object-oriented applications can often be adapted to new requirements with a minimal amount of new code, it can require a great deal of detailed study in order to find out where exactly the extension is needed. Unfortunately, object-oriented frameworks do not make their generic architecture explicit, which results in a steep learning curve before a framework can be successfully reused. Since object-oriented frameworks focus on subclassing of framework classes (aka white-box reuse), a detailed understanding of the generic architecture is needed in order not to break contracts between classes. In addition, changing framework classes often implies extensive modifications of application-specific code.

Visual application builders and scripting languages go a step further than object-oriented frameworks since they already incorporate important ideas and concepts needed for component-based application development (e.g., higher-level abstractions for composing components). They generally: focus on a specific application domain (e.g., Delphi focuses on database applications for the Windows platform [6]); offer a collection of reusable components tailored to their application domain; and make the generic architecture of applications much more explicit than object-oriented frameworks. However, due to their restriction to specific application domains, visual application builders and scripting languages are not flexible enough for general-purpose component-based development and lack a well-understood formal foundation.

3 A conceptual framework for software composition

As we have discussed in the previous section, there exist a variety of languages and tools for building software systems from reusable components. There also exists a considerable body of best practice, such as design patterns, standard software architectures, and various reflective techniques. However, it is not clear how these techniques can be productively combined in a disciplined way in order to build flexible and adaptable software systems. We propose an approach in which five of these techniques are combined, namely:

- *component frameworks* provide software components that encapsulate useful functionality;
- *architectural description languages* explicitly specify architectural styles in terms of interfaces, contracts, and composition rules that components must adhere to in order to be composable;
- *glue* abstractions adapt components that need to bridge compositional mismatches;
- *scripting languages* are used to specify compactly and declaratively how software components are plugged together to achieve some desired result; and
- *coordination models* provide the coordination media and abstractions that allow distributed components to cooperate.

Before we discuss this approach in further detail, we first have to introduce the exact meaning of the terms mentioned above.

3.1 Terminology

A *component framework* is a collection of software components and architectural styles that determines the interfaces that components may have and the rules governing their composition. In contrast to an object-oriented framework where an application is generally built by subclassing framework classes that respond to specific application requirements, a component framework primarily focuses on object and class (i.e., component) composition (aka black-box reuse).

A software component itself is a *static abstraction with plugs* and can be seen as a kind of *black box entity* that hides its implementation details [23]. It is a static entity in the sense that it must be instantiated in order to be used. Finally, a component has plugs which are not only used to *provide* services, but also to *require* them. The essential point is that components are never used in isolation, but according to a software architecture that determines how components are plugged together. Therefore, a software component has to be considered as an element of a component framework.

A *software architecture* describes a software (sub-)system as a configuration of components and connectors. A connector connects required ports of a set of components to provided ports of other components. A configuration of components and connectors can be used as a component of another (sub-)system. The main purpose of software architectures is to make a clear separation between computational elements (components) and their relationships (connectors) [30]. An *architectural style* is an abstraction over a family of software architectures. It defines a vocabulary of component and connector types and a set of rules defining how components and connectors can be combined [27].

An *architectural description language* (ADL) is a notation that allows for a precise description of the externally visible properties of a software architecture, supporting different architectural styles at different levels of abstraction. Externally visible properties refer to those assumptions other components can make of a component, such as its provided services, performance characteristics, error handling, and shared resource usage [4].

It is sometimes necessary to reuse a component in a different environment than the one it was designed for and which does not match the assumptions the component makes about the structure of the system that it was to have been a part of. Such a situation is sometimes referred to as *compositional mismatch* [29]. The nature of a compositional mismatch can be: i) incompatible assumptions about the architectural styles of the underlying component frameworks (i.e., architectural mismatch [12]); ii) different data formats (aka interoperability mismatch [15]); iii) different synchronization schemes, and many more. In such a situation, *glue* is needed to overcome a compositional mismatch and to adapt the "foreign" component to the new environment. Glue abstractions are generally divided into two categories: *adaptors* bridge different interfaces and architectural styles whereas *transformers* bridge incompatible data formats [29].

Naturally, it is not enough to have components and frameworks, for building real applications one needs a way to wire components together (i.e., to express *compositions*). In recent years, so-called *scripting languages* have become increasingly popular as they make it very easy to quickly build small, flexible applications from a set of existing components. These languages typically support a single, specific architectural style of composing components (e.g., the *pipe and filter* architectural style supported by UNIX shells), and they are designed with a specific application domain in mind (e.g., system administration, graphical user interfaces etc.) [26].

Historically, scripting languages were used for automating tasks (e.g., batch processing) or for programming-in-the-small. However, modern scripting languages offer other features as well.

First, scripting languages are extensible: new abstractions can be added to the language, encouraging the integration of legacy code into frameworks and applications [5].

Secondly, scripting languages are embeddable: it is possible to embed them into existing components, offering a flexible way to adapt and extend applications.

Finally, they offer high-level abstractions for flexibly connecting existing components and representing design elements in applications. Scripting can be considered as a higher-level binding technology for component-based systems [8]. One should note that in a complete environment, the composition of components using a script again leads to a reusable component. Therefore, components and scripts can be considered as a kind of *component algebra*.

A new class of formalisms has recently evolved for describing concurrent and distributed computations based on the concept of *coordination*. The purpose of a coordination model is similar to the one of software architectures: making a clear separation between computational elements and their relationships by providing abstractions for controlling synchronization, communication, creation, and termination of concurrent and distributed computational activities [2]. One can also consider coordination as the scripting of concurrent and distributed components.

3.2 Concepts in practice

In order to illustrate our approach of components, architectures, scripts, and glue, consider the following UNIX shell script which reverses the lines of a 7-bit character input stream:[1]

```
cat -n | sort -r -n | cut -b8-
```

Analyzing this shell script, we can immediately identify components and connectors as well as the underlying architecture: the script consists of: i) a data source (i.e. the standard input stream of `cat`); ii) three components (the UNIX processes `cat`, `sort`, and `cut`); iii) two connectors (i.e. character streams); and iv) a data sink (the standard output stream of `cut`). The components and the character streams of the script form a pipeline, where each component only depends on the output of its predecessor. Since all shell scripts fulfill similar restrictions, they all share a similar overall structure: they conform to a *pipe and filter*[2] architectural style [30].

Further analyzing the shell script, it is possible to detect other important properties of shell scripts. First, the composition of shell components using the pipe connector again leads to a shell component (i.e., a filter process which reads from the standard input stream and writes to the standard output stream). Therefore, all UNIX filters together with the pipe connector can be viewed as a component algebra. Second, it is obvious that using the `sort` filter to reverse lines might not be the most efficient way to solve this problem: reversing lines is a problem with linear complexity whereas the complexity of sorting is $O(n \cdot \log n)$. If efficiency is a major concern, it is possible to reimplement the line reverser with the same interface using a more appropriate language (e.g., Perl [33]). This is a common approach for application development using scripting languages [9]. Third, the line reverser scripts only works for 7-bit characters (the `cat` filter transforms non-printing characters). If it should be used for 8-bit characters, it is possible to build a wrapper around it which encodes 8-bit characters in 7-bit characters and decodes them after the lines have been reversed. This is a typical usage of: i) a transformer in order to bridge incompatible data formats; and ii) a component in an environment other than the one it has been designed for. Another possibility for solving the problem is to exchange `cat -n` by `nl -ba`. Finally, due to the inherently concurrent nature of the UNIX operating system, the character streams also act as coordinators (synchronizers) between the three filters.

3.3 Other aspects of the conceptual framework

The primary focus of software architectures is the identification of components that are necessary for the architecture, design, and implementation of a system. The importance of software architectures in the software life-cycle is often underestimated. Having reusable architectural styles is a precondition for successfully developing reusable components and frameworks [4] and, therefore, for building component-based applications.

There is a large variety of architectural description languages emerging from either industrial or academic research groups [20, 30]. However, most of these languages offer only a restricted set of architectural styles and are not sufficiently open to support new (i.e., user-defined) styles, since it is not possible to define new component or connector types. In addition, they focus on specific application domains and do not support dynamic reconfiguration of architectures.

[1]The usage and arguments of the UNIX programs used can be found in an appropriate UNIX manual.

[2]In UNIX terminology, a command like `sort` is usually called a filter.

As mentioned above, glue techniques are required to adapt components that do not really fit the system architecture. Today, almost all component glue is based on *wrappers* that pack the original component into a new one with a suitable interface. These wrappers usually have the form of an adaptor in order to bridge architectural mismatch or a transformer to overcome interoperability mismatch. If wrappers are used frequently, this technique can give rise to serious performance problems. However, most glue problems can be solved by using *behavioural reflection* [19]. The idea is to intercept the messages that are exchanged between the components and to manipulate them according to the requirements of the receiver of the corresponding message. In particular, messages can be transformed, delayed, or even delegated. Comparable approaches have been used in existing languages (Sina [1], CLOS [3], and Smalltalk [13] to name but a few), but none of these approaches focuses primarily on overcoming compositional mismatches.

Open applications often have to deal with new kinds of components at run-time (e.g., exchanging a component of a database system by a new version). It is obvious that a composition framework must offer some sort of introspection facilities in order to inspect the interface and capabilities of a new component and its features to dynamically connect it to other components. Modern scripting languages offer abstractions for: i) dynamically loading and exchanging components; ii) restricted introspection facilities based on interface declarations of components; and iii) for executing dynamically created code. The last feature is often referred to as an "eval" feature [26]. However, most of these abstractions are rather ad-hoc and lack a well-defined formal semantics.

Building component-based applications is seldom just a matter of plugging together components from a single component framework: it often entails building components as a composition of other (possibly lower-level) components which can be reused in a family of applications. In contrast to the fact that each software system has an architecture, it is often not possible to assign one architectural style to a system: a system may be a combination of several architectural styles (e.g., a filter of a pipe and filter pipeline is structured in a layered style). Such systems are called *heterogeneous* [4, 30]. Therefore, a composition language must provide support for packaging a composition of components as a new component and for simultaneously using multiple architectural styles.

4 PICCOLA – a Small Composition Language

Programming software components and combining them using scripts are very different activities that may well benefit from different kinds of tools. We expect that traditional programming languages will be best suited for programming components, whereas something we call a *composition language* may be better for specifying architectural styles, coordination abstractions, glue abstractions, and scripts.

PICCOLA is a small composition language currently under development that is intended to address these four aspects of composition. The language is being developed "bottom up" by adding thin layers of abstraction and syntax on top of the π calculus. The kernel of the PICCOLA prototype is an interpreter (written in Java) of $\pi\mathcal{L}$ – a conservative extension of the polyadic π calculus, inspired by Dami's λN calculus [10], in which agents communicate by passing *forms* rather than tuples. Forms are essentially extensible records – i.e., records with an extensible set of attributes – and

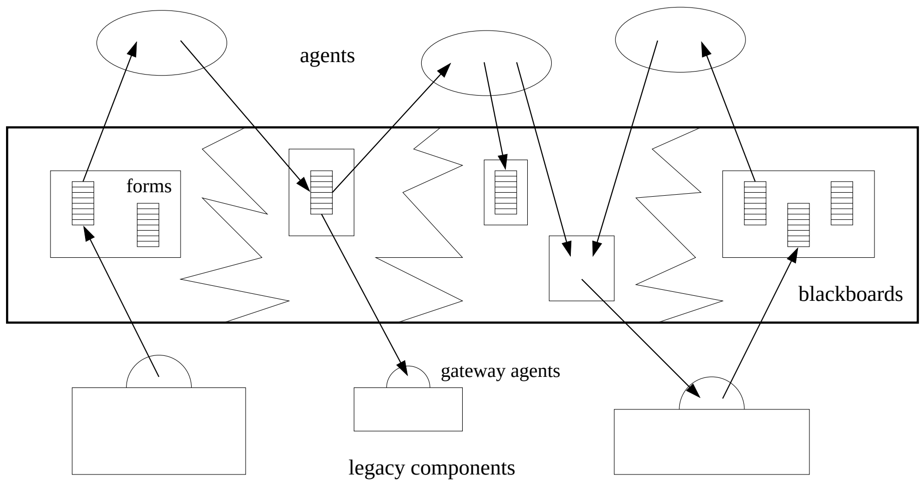

Figure 1: A distributed composition medium.

can be used to model dictionaries, first-class contexts (i.e., environments), objects, and keyword-based parameter-passing (as in Tcl [25] and Python [18]).

PICCOLA attempts to bring together a number of proven ideas from scripting languages, coordination models and languages, glue techniques, and architectural specification. Essentially, PICCOLA models everything in terms of *agents*, *blackboards*, and *forms*. Active entities (external components, glue, connectors) are modelled as agents. Agents communicate by posting forms to, and reading forms from, distributed blackboards. Blackboards resemble the communication media of coordination models and languages like Linda [7], but are formally modelled as *channels* in $\pi\mathcal{L}$. Finally, glue code is handled by special *gateway agents* that wrap external components and applications, and by *interceptors* – agents that intercept, transform, and adapt messages intended for other components [14] (see Figure 1).

Scripts are modelled as sets of interconnected agents. Abstractions over agent configurations are values, and can be stored in contexts or communicated in forms. By the same token, connectors may be specified as coordination abstractions. Consider the following abstraction in PICCOLA (the syntax is tentative, but friendlier than the equivalent $\pi\mathcal{L}$ code):

```
def future (service) (args) =          // Defined in the enclosing context
    slot = global.newBlackboard()      // requires newBlackboard
    return (val = slot.read)           // early return of access channel
    slot.write (service(args))         // write result into blackboard
```

Futures are a well-known abstraction from concurrent object-based programming that allow clients to progress in parallel with service providers, until the actual results are needed. In most programming languages, futures cannot be expressed directly as an abstraction, but must be either hand-coded or defined as a language extension, either because the language cannot directly express higher-order abstractions or because one cannot abstract over all the possible types of arguments of service providers. $\pi\mathcal{L}$ and PICCOLA support arbitrary abstractions over agents, and finesse the second problem by passing all arguments within forms. A future does not care how many arguments are expected by the service; they are simply bundled together as a single args form. Another example of a classic mechanism that is impossible to encapsulate as an abstraction in most programming languages is a *readers/writers* synchronization policy for concurrently accessed resources. This is relatively straightforward to express in $\pi\mathcal{L}$ and in PICCOLA by treating such policies as (first-class) wrappers around (first-class) methods [17].

PICCOLA is still an experimental language. As with the π calculus, there are various ways to model objects and components, each with their own advantages and drawbacks. These must be especially evaluated against the object models of standard component libraries and middleware (such as CORBA [24], JavaBeans [31], etc.).

The formal foundations are important, because it is notoriously difficult to reason about distributed, heterogeneous systems. We intend that PICCOLA support reasoning at the architectural level about properties such as service guarantees, and the degree of real concurrency that can be exploited, by means of mappings to the underlying π calculus foundations. We are working on the relationship between $\pi\mathcal{L}$ and the π calculus, on a flexible type system for $\pi\mathcal{L}$ (and PICCOLA) based on sorts and types for the π calculus, and we are exploring techniques for reasoning about behavioural properties of PICCOLA systems (such as the degree of real concurrency). Technical details of the formal mapping between π and $\pi\mathcal{L}$ are discussed in [16].

At the same time we are examining a number of practical problems, such as CORBA scripting from the point of view of available technology, how to wrap and script external components (such as JavaBeans), and how to interactively monitor and configure the composition medium.

5 Discussion

The concepts we have discussed in Section 3 define a framework for composing applications form component frameworks. One might argue that these concepts only apply to run-time composition, as scripting languages are typically dynamically compiled or interpreted. The same ideas, however, apply equally well to compile-time composition.

Consider, for example, the Standard Template Library (STL) [22]. STL provides a set of C++ container classes (such as vectors, lists, sets etc.) and template algorithms for common kinds of data manipulations on the container classes (e.g., searching, sorting, merging). STL has all the properties we have previously established for component frameworks: it focuses on component composition rather than white-box reuse, it incorporates a collection of reusable components, fixes the interfaces components may have, and defines a set of rules how components can be composed. All applications using STL therefore share a common architectural style.

The line reverser we have introduced in Section 3.2 may be implemented in C++ using STL, using similar concepts to those we have already used in the UNIX shell scripts: components (STL containers), connectors (generic functions), and glue (e.g., input/output stream adapters to make cin and cout look like containers). The major difference between the C++ program and the shell script is that: i) our C++ program does not make the underlying architecture of the program explicit; and ii) any C++ program using STL only works in a sequential environment and, therefore, does not require any coordination abstractions.

Although we can point out many examples of component-based systems that conform to the conceptual framework we have presented, we have not addressed the issue, *how do we migrate object-oriented applications to component-based architectures?* In parallel to our basic research on PICCOLA, we are participating in FAMOOS,[3] a European industrial research project on reengineering object-oriented legacy systems towards component-based frameworks. Early adopters of object-oriented technology now find themselves with large, object-oriented applications that are critical to their business interests, but are difficult to adapt to changing business needs.

Within FAMOOS we have found that a *pattern-based approach* is most promising, since similar reengineering problems seem to recur across applications, even with very different reengineering requirements. Reverse engineering patterns help to extract models from existing applications and source code, and reengineering patterns help to identify and resolve problems in legacy code [11]. Nevertheless, there are fundamental problems that are difficult to resolve: i) the components are not "just there for the taking" – it can be very hard to extract components from source code, even if the end-user functionality suggests that "they must be there somewhere!"; ii) it is hard to extract architecture from object-oriented source code (or dynamic traces), and hence it is hard to tell where the architecture may be "broken"; and iii) semantics-preserving transformations are not enough to get you from even a good object-oriented design to a flexible component-oriented design (more drastic measures may be needed).

6 Conclusions

Object-oriented programming alone is not enough to guarantee the development of flexible systems, but it provides a good set of tools and techniques that can be used for component-based application development. Components, however, are not enough either, since a component without an architecture is like a single lego piece – all by itself. CORBA, Delphi, JavaBeans, and D-Active-COM-X-++ are also not enough – each solves important technical problems, but does not go beyond a specific domain.

We have surveyed some of the problems with object-oriented technology – as it is used today – and argued that the flexibility and adaptability needed for applications to cope with changing requirements can only be achieved if we think *not only* in terms of *components*, but also in terms of *architectures*, *scripts*, and *glue*. We have also presented our ongoing research to develop a formal composition language to support these ideas.

In this chapter we have focussed mainly on technological issues, but there are just as many, and arguable more important, methodological issues: component frameworks focus on software solutions, not problems, so *how can we drive analysis and design so that we will arrive at the available solutions?* Frameworks are notoriously hard to develop, so *how can we iteratively evolve our object-oriented applications to arrive at a flexible component-based design?* Finally, and perhaps most important, software projects are invariably focussed toward the bottom line, so *how can we convince management to invest in component technology?*

[3]FAMOOS is an industrial ESPRIT Project (No 21975) in the IT Programme of the Fourth ESPRIT Framework Programme.

Acknowledgements

We thank all members of the Software Composition Group for their support of this work, especially Franz Achermann, Serge Demeyer, and Markus Lumpe.

References

[1] Mehmet Aksit. *On the Design of the Object-Oriented Language Sina*. PhD thesis, University of Twente, NL, 1989.

[2] Farhad Arbab. The IWIM Model for Coordination of Concurrent Activities. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, LNCS 1061, pages 34–56. Springer, April 1996. Proceedings of Coordination '96.

[3] Giuseppe Attardi, Cinzia Bonini, Maria Rosaria Boscotrecase, Tito Flagella, and Mauro Gaspari. Metalevel Programming in CLOS. In Stephen Cook, editor, *Proceedings ECOOP '89*, pages 243–256. Cambridge University Press, July 1989.

[4] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[5] David M. Beazley. SWIG and Automated C/C++ Scripting Extensions. *Dr. Dobbs Journal*, (282):30–36, February 1998.

[6] Borland International. *Borland Delphi Users Manual*, 1995.

[7] Nicolas Carriero and David Gelernter. *How to Write Parallel Programs: a First Course*. MIT Press, 1990.

[8] Brad J. Cox. *Object Oriented Programming – An Evolutionary Approach*. Addison-Wesley, 1986.

[9] Douglas Cunningham, Eswaran Subrahmanian, and Arthur Westerberg. User-Centered Evolutionary Software Development Using Python and Java. In *Proceedings of 6th International Python Conference*, pages 1–9, San Jose, October 1997.

[10] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, Centre Universitaire d'Informatique, University of Geneva, CH, 1994.

[11] Stéphane Ducasse, Robb Nebbe, and Tamar Richner. Type-Check Elimination: Two Reenegineering Patterns. In Jens Coldewey and Paul Dyson, editors, *Proceedings EuroPLoP '98*, pages 161–171, July 1998.

[12] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 12(6):17–26, November 1995.

[13] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, September 1989.

[14] Manuel Günter. Explicit Connectors for Coordination of Active Objects. Master's thesis, University of Bern, Institute of Computer Science and Applied Mathematics, March 1998.

[15] Dimitri Konstantas. Interoperation of Object-Oriented Applications. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 69–95. Prentice Hall, 1995.

[16] Markus Lumpe. *A π-Calculus Based Approach to Software Composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[17] Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. An Extensible Language for Composition. Submitted for Publication, 1998.

[18] Mark Lutz. *Programming Python: Object-Oriented Scripting*. O'Reilly & Associates, October 1996.

[19] Jeff McAffer. Meta-level Programming with CodA. In Walter Olthoff, editor, *Proceedings ECOOP '95*, LNCS 952, pages 190– 214. Springer, August 1995.

[20] Nenad Medvidovic and Richard N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In Mehdi Jazayeri and Helmut Schauer, editors, *Proceedings ESEC '97*, LNCS 1301, pages 60–76, September 1997.

[21] Robin Milner. The Polyadic Pi-Calculus: a Tutorial. Technical Report ECS-LFCS-91-180, Computer Science Department, University of Edinburgh, UK, October 1991.

[22] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[23] Oscar Nierstrasz and Laurent Dami. Component-Oriented Software Technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[24] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1996.

[25] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[26] John K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, 31(3):23–30, March 1998.

[27] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[28] Trygve Reenskaug. *Working with Objects: the OOram Software Engineering Method*. Manning Publications, 1996.

[29] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.

[30] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[31] Sun Microsystems. *JavaBeans Specification*, July 1997.

[32] Dennis Tsichritzis. Object-Oriented Development for Open Systems. In *Proceedings IFIP '89*, pages 1033–1040. North-Holland, August 1989.

[33] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, 2nd edition, September 1996.