1

PICCOLA – a Small Composition Language

Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz
Software Composition Group,
Institute of Computer Science and Applied Mathematics,
University of Berne, Switzerland

Abstract

Although object-oriented languages are well-suited to implement software components, they fail to shine in the construction of component-based applications, largely because object-oriented design tends to obscure a component-based architecture. We propose to tackle this problem by clearly separating component implementation and composition. In particular, we claim that application development is best supported by consciously applying the paradigm "Applications = Components + Scripts." In this chapter, we propose PICCOLA, a small "composition language" that embodies this paradigm. PICCOLA models components and compositional abstractions by means of communicating concurrent agents. Flexibility, extensibility, and robustness are obtained by modeling both interfaces of components and the contexts they live in by "forms", a special notion of extensible records. Using a concrete example, we illustrate how PICCOLA offers explicit support for viewing applications as compositions of components and show that separating components from their composition improves maintainability.

1.1 Introduction

Component-based software development offers a plausible solution to one of the toughest and most persistent problems in software engineering: how to effectively maintain software systems in the face of changing and evolving requirements. Software systems, instead of being programmed in the conventional sense, are constructed and configured using libraries of components. Applications can be adapted to changing requirements by reconfiguring components, adapting existing components, or introducing new ones.

We argue that the flexibility and adaptability needed for component-based applications to cope with changing requirements can be substantially enhanced if we think not only in terms of *components*, but also in terms of *architectures*, *scripts*, *coordination*, and *glue*. In particular, we claim that application development is best supported by consciously applying the paradigm

Applications = Components + Scripts.

Components are black-box entities that encapsulate services behind well-defined interfaces whereas scripts encapsulate how the components are composed. This paradigm helps to make a clear separation of computational elements and their relationships.

However, currently there exists no general-purpose composition language that (i) offers explicit support for the paradigm introduced above and (ii) fulfills the requirements for a composition language elaborated previously [NM95a, NM95b, NSL96]. Object-oriented programming languages and design techniques, for example, go a long way towards supporting component-based development, and the languages are nearly ideal for *implementing* components, but current practice actually hinders component-based development in a number of significant ways:

- **Reuse comes too late:** object-oriented analysis and design methods are largely domain-driven, which usually leads to designs based on domain objects and non-standard architectures. Most of these methods make the assumption that applications are being built from scratch, and they incorporate reuse of existing architectures and components in the development process too late (if at all).
- **Overly rich interfaces:** being domain-driven, OOA and OOD lead to rich object interfaces and interaction protocols, but component composition depends on adherence to *restricted, plug-compatible interfaces* and *standard interaction protocols*.
- **Lack of explicit architecture:** object-oriented source code exposes *class hierarchies*, but not *object interactions*. How the objects are plugged together is typically distributed amongst the objects themselves. As a result, adapting an application to new requirements typically requires *detailed study,* even if the actual needed changes are minimal.

In order to solve these problems, we argue that it is necessary to define a language specially designed to compose software components and to base this language on an appropriate semantic foundation. Although to some extent the concepts we identified can be applied in traditional object-oriented languages, we believe that a specially-designed language is better for explaining, highlighting, and exploring compositional issues as opposed to general-purpose programming issues. Furthermore, if we can understand all aspects of software components and their composition in terms of a small set of primitives, then we have a better hope of being able to cleanly integrate all required features for software composition in one unifying concept.

We are currently developing PICCOLA, a prototype of an experimental composition language. We explore two approaches: a first approach based on an imperative style of programming [Lum99, Sch99] (similar to the PICT programming language [PT97]) and a second approach emphasizing a more functional and declarative style of programming (which is the topic of section 1.4). Experiments have shown that existing paradigms do not fully address the abstractions required for component-based development. Therefore, by combining the main concepts of existing paradigms, we hope to (i) discover the right abstractions for software composition and to (ii) define an unified paradigm which fulfills our requirements.

Common to both approaches mentioned above is the fact that all language features are defined by transformation to a core language that implements the $\pi\mathcal{L}$-calculus [Lum99], an inherently polymorphic variant of the π-calculus [Mil90, HT91], in which agents communicate by passing *forms* (a special notion of extensible records) rather than tuples. By this approach, we address the problem that reusability and extensibility of software components is limited due to position-dependent parameters.

Besides forms, which have their analogues in many existing programming languages and systems (e.g., HTML, Visual Basic, Python), the $\pi\mathcal{L}$-calculus also incorporates *polymorphic form extension*, a concept that technically speaking corresponds to asymmetric record concatenation [CM94], as a basic composition operation for forms. As we will show in sections 1.4 and 1.5, both forms and polymorphic extension are the key mechanisms for extensibility, flexibility, and robustness as (i) clients and servers are freed from fixed, positional tuple-based interfaces, (ii) abstractions are more naturally polymorphic as interfaces can be easily extended, and (iii) environmental arguments (such as communication policies or default I/O-services) can be passed implicitly.

This chapter is organized as follows: in section 1.2, we summarize our requirements for PICCOLA in terms of a conceptual framework for software composition. In section 1.3, we illustrate the ideas behind the $\pi\mathcal{L}$-calculus, the formal foundation of PICCOLA. We introduce PICCOLA in section 1.4 and present an extended example that illustrates how PICCOLA supports the conceptual framework for composition in section 1.5. We conclude with a comparison of related work and present some perspectives on future work in sections 1.6 and 1.7, respectively.

1.2 Components, Scripts, and Glue

Component-based applications, we argue, provide added value over conventionally developed applications, since they are easier to adapt to new and/or changing requirements. This is the case since we can (i) configure and adapt individual components, (ii) unplug components and plug in others, (iii) reconfigure the connections between sets of components at a high level of abstraction, (iv) define new, plug-compatible components from either existing components or from scratch, (v) take legacy components and adapt them to make them plug-compatible, and (vi) treat a composition of components as a component itself. In the following, we introduce a few important terms and illustrate that a composition language has to provide support for the following key concepts.

Components. A *component* is a "black-box" entity that both *provides* and *requires* services. These services can be seen as "plugs" (or, more prosaically, interfaces). The added value of components comes from the fact that the plugs must be standardized (i.e. a component must be designed to be composed [ND95]). A "component" that is not plug-compatible with anything can hardly be called a component. The plugs of a component take many different shapes, depending on whether the component is a function, a template, a class, a data-flow filter, a widget, an application, or a server. It is important to note that components also require services, as this makes them individually configurable (e.g., consider a sorting component that behaves differently given different containers or comparison operators [MS96]).

Architectures. Components are by definition elements of a *component framework*: they adhere to a particular *component architecture* or "architectural style" that defines the plugs, the connectors, and the corresponding composition rules. A *connector* is the wiring mechanism used to plug components together [SG96]. Again, depending on the kind of components we are dealing with, connectors may or may not be present at run-time: contrast C++ template composition to Unix pipes and filters. The composition rules tell us which compositions of components are valid (e.g., we cannot make circular pipes and filters chains). A so-called *architectural description language* (ADL) allows us to specify and reason about architectural styles [SG96]. Note that we adopt here a very restricted view of component architecture, ignoring such issues as module architecture or configuration management [Kru95].

Scripts. A *script* specifies how components are plugged together [NTMS91]. Think of the script that tells actors how to play various roles in a theatrical piece. The essence of a scripting language is to configure components, possibly defined *outside the language*. A "real" scripting language will also let you treat a script as a component: a Unix shell script, for example, can be used as a Unix command within other scripts. At the minimum, a scripting language must provide (i) an encapsulation mechanism to define scripts, (ii) basic composition mechanisms to connect components, and (iii) abstractions to integrate components written outside the language (i.e. a *foreign code concept*) [Sch99]. Note that a script makes architectures explicit by exposing exactly (and only) how the components are connected.

Coordination. If the components are agents in a distributed (or at least concurrent) environment, then we speak of *coordination* rather than scripting. A coordination language is concerned with managing dependencies between concurrent or distributed components. Classical coordination languages are Linda [CG89], Darwin [MDK92], and Manifold [Arb96].

Glue. Although we claimed that components must be designed to be composed, the simple fact is that we are often constrained to use (legacy) components that are not plug compatible with the components we want to work with. These situations are referred to as *compositional mismatches* [Sam97], and *glue code* overcomes these mismatches by adapting components to the new environment they are used in. Glue adapts not only interfaces, but may also adapt client/server contracts or bridge platform dependencies. Glue code may be *ad hoc*, written to adapt a single component, or it may consist of generic abstractions to bridge different component platforms.

From our point of view, a *composition language* is a combination of the aspects of (i) ADLs, allowing us to specify and reason about component architectures, (ii) scripting languages, allowing us to specify applications as configurations of components according to

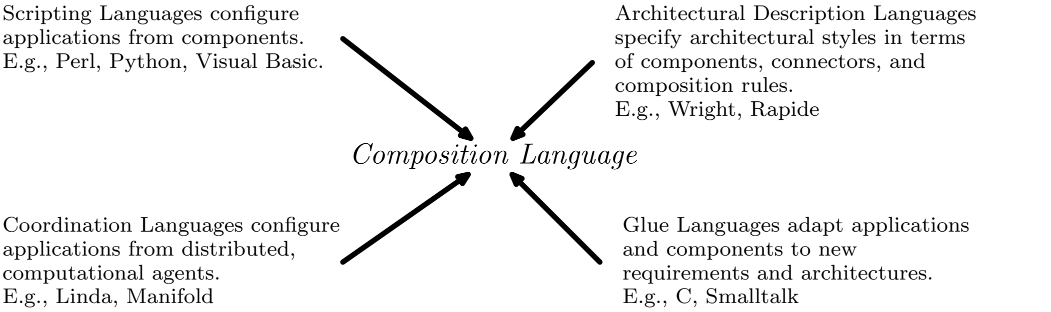

Fig. 1.1. Conceptual framework for software composition.

a given architectural style, (iii) glue languages, allowing us to specify component adaptation, and (iv) coordination languages, allowing us to specify coordination mechanisms and policies for concurrent and distributed components. Figure 1.1 illustrates this point.

A particular challenge for a composition language is the ability to define new, higher-level composition and coordination abstractions in terms of the built-in ones [Nie93a]. Consider, for example, the difficulty of defining in a conventional object-oriented language, a generic synchronization policy, such as a readers-writers policy, that can be applied to existing, unsynchronized objects. Typically, this is either not possible, would require a language extension, or is only possibly by using meta-level abstractions. A composition language does not only let us instantiate and compose components, but also provides the means to define higher-level abstractions to compose and coordinate components.

1.3 Foundations for Software Composition

In order for a composition language to meet our requirements, it must be based on a semantic foundation that is suitable for modelling different kinds of components and compositional abstractions. A precise semantics is essential if we are to deal with multiple architectural styles and component models within a common, unifying framework.

The simplest foundation that seems appropriate is that of communicating, concurrent agents. For this reason we have extensively explored the asynchronous polyadic π-calculus [Mil90, HT91] as a tool for modelling objects, components, and software composition [LSN96, SL97]. The tuple-based communication of the π-calculus, however, turns out to restrict extensibility and reuse. These observations have led us to explore communication of *forms* – a special notion of extensible records – instead of tuples. In the rest of this section, we illustrate briefly the nature of problems the $\pi\mathcal{L}$-calculus solves and show how forms are the key concept to extensibility, flexibility, and robustness; a detailed discussion of π and $\pi\mathcal{L}$ is beyond the scope of this chapter (refer to [Mil90, Lum99] for details).

Let us consider the following expression in the polyadic π-calculus as an example to highlight the difference between π and $\pi\mathcal{L}$:

$$!w(a, b, r).(\nu\, r')(\, \overline{f}\langle a, b, r'\rangle \mid r'(x, y).\overline{r}\langle x, y\rangle\,)$$

This models a process providing a service at a channel w, and acts as a wrapper for another process providing a service at channel f. The process listens repeatedly at channel w for a triple (a, b, r) where, by convention, a and b are service parameters and r is a channel to which the reply will be sent. After receiving a message, the process creates a new private channel r' and forwards the message to f, substituting the new reply channel. In parallel, it starts a new process that listens at channel r', picks up the response, and forwards it to the original client along channel r. This particular wrapper does nothing exciting, but the same pattern can be used for more interesting wrappers. The important point is that the wrapper code *hard-wires* the protocol, so it will not work if the service at f extends its interface to accept more (or less) parameters or to return a result with a different arity.

In the $\pi\mathcal{L}$-calculus, the same example can be encoded as follows:

$$!w(X).(\nu\, r')(\, \overline{f}(X\langle \mathit{reply} = r'\rangle) \mid r'(Y).\overline{X_{\mathit{reply}}}(Y)\,)$$

Instead of expecting a tuple as input, the wrapper receives a single form X. The original service is requested by overriding the binding of the reply channel to r'. Finally, when the result (Y) is obtained, it is forwarded to the original client by looking up the reply binding in the original form X. The interesting point to note is that the wrapper in the $\pi\mathcal{L}$-calculus is completely generic, only assuming that the message received contains a reply channel.

As a second example, consider the specification of invariants (e.g., default arguments) using polymorphic form extension. Let us assume that a service located at channel g provides a query interface for a simple database. This service requires a binding for $output$ in order to display a query result. To facilitate the usage of this service, we define a wrapper located at channel u guaranteeing the invariant that the query result is passed to a default output service located at channel p:

$$!u(X).\overline{g}(\langle output = \mathrm{p}\rangle X)$$

Using this scheme, we guarantee that (i) by default, query results are passed to channel p (as desired) and (ii) the default output behaviour can be overridden by providing an additional binding for label $output$ (denoting a new output service) in the query arguments. Note that the same behaviour cannot be expressed without polymorphic form extension.

Similar schemes can be used to simplify the modelling of numerous object-oriented and component-based abstractions [Sch99]. For example, it is much easier to model generic synchronization policies (such as a readers-writers mutual exclusion policy) in the $\pi\mathcal{L}$-calculus than in the polyadic π-calculus [SL97].

Although the $\pi\mathcal{L}$-calculus makes a fundamental modification to the π-calculus, it is possible to translate $\pi\mathcal{L}$-agents to π-processes and back, *preserving behavioural equivalence both ways*. Furthermore, the concept of expressing computation by means of exchanging messages is computationally complete [Mil90] and, therefore, any programming scheme and model can be encoded in the $\pi\mathcal{L}$-calculus. This is of major importance in the context of adapting and composing components defined in different programming environments.

1.4 PICCOLA in a Nutshell

Although the $\pi\mathcal{L}$-calculus has been designed for reasoning about concurrency and communication, it turns out to be extremely low-level as a programming language. The natural style of interaction described by the $\pi\mathcal{L}$-calculus is that of directed channel communication. Other types of interaction such as event based communication or failures can be encoded, but they often turn out to be awkward.

PICCOLA addresses this shortcoming by defining language constructs to simplify these encodings. These constructs are functions, infix operators to support an algebraic notion of architectural style, and the explicit notion of a (dynamic) context to encapsulate required services.

Higher-level abstractions can then be defined as library functions on top of this core language much in the same way that CLOS is defined on top of Common Lisp [KdRB91]. In both systems, we can define an abstraction `Class` that allows the programmer to build classes for object oriented programming [Sch99]. However, the fundamental difference with respect to CLOS is that PICCOLA's abstractions are defined in terms of a formal foundation of agents, forms, and channels, instead of functions and lists.

1.4.1 Core elements

In this section we give a brief overview of the PICCOLA language elements. The version of PICCOLA described in this chapter conforms to the functional programming paradigm. More precisely, the main language element of PICCOLA is a so-called *form expression* that represents a unified concept of both $\pi\mathcal{L}$-agents and $\pi\mathcal{L}$-forms. In fact, form expressions are sequences of form terms (e.g., synchronous and asynchronous function calls, binding extensions). Using form expressions, PICCOLA programs or scripts can be defined without using the low-level primitives of the underlying $\pi\mathcal{L}$-calculus. The parallel composition operator, for example, is modeled by asynchronous function calls whereas the rendezvous of input- and output-prefixes is achieved by a synchronous function call.

PICCOLA has a syntax similar to that of Python and Haskell (e.g., newlines and indentation, rather than braces or **`end`** statements, are used to delimit forms or blocks). Forms, however, may also be specified on a single line by using commas and brackets as separators.

PICCOLA consists of the following core elements:

- `ident = e` – binds the form expression `e` to the name `ident`.
- **`export`** `ident = e` – extends the current context with the binding `ident = e`. The extension is done by a functional update. Therefore, an expression **`export`** `ident = e,` **`global`**, where **`global`** denotes the current context, is equivalent to **`global`** `= (`**`global`**`, ident = e)`.
- `e.ident` – yields the value that label `ident` binds in form expression `e`.
- **`def`** `ident(`$\texttt{ident}_1$`)`...`(`$\texttt{ident}_n$`) = e` – defines a parameterized abstraction over form expression `e`. More precisely, this construct is used to define a function `ident` with the formal parameters $\texttt{ident}_1$...$\texttt{ident}_n$. Functions are first-class values. The way

function arguments are specified is a useful device to keep things separated. In fact, form expressions are flat values. Using the parameter specification `(ident`$_1$`)`...`(ident`$_n$`)` allows us to add additional information about the structure of the arguments, i.e., it allows us to make the structure of the expected arguments explicit.

Functions are translated to $\pi\mathcal{L}$-agents that wait for requests at a channel that, by convention, is associated to the name of the function.

- **`return`** `e` – returns the expression `e`. In general, this term is used to specify an early return [Gen81].
- **`run`** `e` – invokes a function denoted by the form expression `e` asynchronously, i.e., it does not yield a result.
- `ident(e`$_1$`)...(e`$_n$`) [`**`in`** `e`$_m$`]` – invokes the function `ident` with the actual arguments `e`$_1$`, ..., e`$_n$ synchronously. If **`in`** `e`$_m$ is specified, the function is invoked using `e`$_m$ as actual context, otherwise the current context is used.
- *Infix operators* – Operators like `+`, `-`, `|`, `>` are syntactic sugar to denote designated functions; they are encoded as label bindings `__+`, `__-`, `__|`, and `__>` that map the corresponding operations. For example, the expression `e`$_1$`|e`$_2$, read `e`$_1$ pipe `e`$_2$, denotes the call of the pipe function within the context `e`$_1$ using `e`$_2$ as argument.

The reader should note that the basic $\pi\mathcal{L}$-calculus operations like creation of a new channel or the input- and output-prefix are represented by built-in functional abstractions. For example, the function `newChannel` creates a fresh $\pi\mathcal{L}$-channel and returns a form with the bindings `send` and `receive`. To send or to receive a value to/from the $\pi\mathcal{L}$-channel, one has to use the corresponding bindings of the returned form. Similarly, the function `concat(F)(G)` implements the polymorphic extension of the form `F` by `G`.

Finally, constants like numbers or strings can be represented in the pure $\pi\mathcal{L}$-calculus using the scheme presented by Milner [Mil91] or Turner [Tur96] for the π-calculus. Therefore, adding *constant values* to the PICCOLA language does not change the underlying semantics. However, if constant values are available, then calculations involving such values are more efficient.

1.4.2 Implementation of PICCOLA

We have implemented PICCOLA in Java. Furthermore, in order to use external components in PICCOLA we have also defined a corresponding Java gateway interface. Using the gateway interface, external components can be transparently integrated into the PICCOLA system. In fact, external components are represented by a PICCOLA form expression that defines the bindings for the provided and required services of the external component. Internally, we use the reflection package of Java. With this approach it is possible to embed arbitrary Java objects into PICCOLA scripts.

When a PICCOLA scripts is executed, the initial context provides access to the basic I/O services, in particular for loading other PICCOLA scripts. Moreover, PICCOLA scripts can be embedded into stand-alone Java applications, applets, or servlets.

1.4.3 Example: a compositional abstraction

The following example illustrates several key concepts of PICCOLA and shows how higher-level abstractions can be defined. Suppose we have a `Multiselector` and a `GUIList` component. The `GUIList` component provides two services `paint` and `close` whereas the `Multiselector` provides the services `select`, `deselect`, and `close`. A composition of these two components offers the union of both sets of services, and, in order to close the composite component correctly, an invocation of `close` must be forwarded to both components. Furthermore, we assume that the component `GUIList` is the master component whereas `Multiselector` is a client component (i.e. the client's close service must be activated first). The following specification implements the close dispatch:

```
def dispatchclose (L)(R) =
    def close () = (L.close(), R.close())
```

The function `dispatchclose` expects two arguments and yields a new function `close` which invokes the close functions on both arguments `L` and `R`.

Now, to compose the components `GUIList` and `Multiselector`, we can define the following function:

```
def fixedcompose (L)(R) =
    paint = L.paint
    select = R.select
    deselect = R.deselect
    close = dispatchclose(L)(R).close
```

The function `fixCompose` implements the union of both sets of services and yields the correct composition of `GUIList` and `Multiselector`.

Unfortunately, this function explicitly refers to the services of the composite component. Therefore, this function cannot be used in a context where the composition should also provide possible extensions of the involved components. For example, the `GUIList` component may be extended with a `resize` service and the `Multiselector` component may define a new service `selectall`. In such a case the above abstraction would not reflect these extensions and the extra services would not be available. This problem, however, can be solved if we use polymorphic extension to define the composition:

```
def compose (L)(R) =
    concat(L)(R)
    close = dispatchclose(L)(R).close
```

Given the original `GUIList` and `selector` components, the new abstraction returns exactly the same composite component as the old version. However, due to the usage of polymorphic form extension, the resulting composite component also reflects extensions of the argument components like `resize` or `selectall`. The `compose` abstraction is more generic than `fixedcompose` as it only assumes that both arguments offer a `close` service. Note that if both arguments offer other services with the same name, only that of the right argument will be available in the composite component.

Our experiences have shown that polymorphic form extension is a fundamental concept for defining adaptable, extensible, and more robust abstractions. It is also used in several PICCOLA library abstractions for object-oriented programming (e.g., in the `Class` abstraction we use in following section).

1.5 Applications = Components + Scripts

In this section, we illustrate how PICCOLA supports our conceptual framework for composition using an example of a Wiki Wiki Web Server (Wiki for short). A Wiki is a simple hypertext system that lets users both navigate and modify pages through the world-wide-web. The original Wiki was implemented by Ward Cunningham as a set of Perl scripts (available at c2.com). Wiki pages are plain ASCII text augmented with a few simple formatting conventions for defining, for example, internal links, bulleted lists, and emphasised text. Wiki pages are dynamically translated to HTML by the Wiki server. A Wiki allows its users to collaborate on documents and information webs.

In the available Perl implementation, it is not easy to understand the flow of control since, as is typical in Perl, the procedural paradigm is mixed with the stream-based processing of the web pages. Execution is sensitive to the sequence in which the declarations are evaluated. To make a long story short, the architecture of the scripts are not evident, and that makes it hard to extend the functionality. Typical extensions that users ask for are reversing the order of new entries to the `RecentChanges` log (so that the latest changes appear at the top instead of at the bottom), extending the formatting rules to allow embedded HTML, support for version control, or an additional concurrency control mechanism (optimistic transaction control, access control, etc.).

We do not argue that the available Perl implementation is weak. The Perl scripts simply make use of the style provided by Perl (i.e. sequentially modifying buffers using regular expressions) which, however, generally does not make the underlying architecture explicit. In the following, we will present the implementation of a component framework supporting a pipe-and-filter architectural style that allows us to make the architecture of the Wiki application explicit. This framework gives a user the feeling of using a specific scripting language for composing filters and streams. However, we would like to point out that a similar approach can be used in any object-oriented programming language that supports operator overloading (e.g., C++ and Python).

The PICCOLA Wiki illustrates how the architecture of a scripted application can be made explicit. In particular, it shows that there is a clear separation between the computational elements and their relationships. Furthermore, glue and coordination abstractions are used that adapt and coordinate components which are not part of the component framework. The Wiki application is presented as follows:

- We define the top-level PICCOLA script that implements the Wiki by composing components that conform to a pull-flow stream-based architectural style [BCK98].
- We illustrate the implementation of an object-oriented (white-box) framework incorpo-

```
def getRequest(F) =
   file = repository.getFile(F)
   body = byParagraphs < file | mkStrong | mkEmphasis |
          mkLinks | mkList
   return mkHead(F) + body + mkTail(F) # stream concatenation

def editRequest(F) =
   file = repository.getFile(F)
   return mkEditHead(F) + file.getStream() + mkEditTail(F)
```

Fig. 1.2. Scripting streams.

rating *streams*, *transformers*, and *files* that corresponds to the architectural style mentioned above. Java streams are integrated into the framework by means of gateway agents. We extend the framework with black-box abstractions that allow us create transformers and streams without subclassing.

- We integrate components of a push-flow architectural style (i.e. components which *push* data downstream instead of *pulling* it from upstream). A coordination layer is used to adapt push-flow components so they can work within a pull-flow architecture.

1.5.1 Scripting the Wiki in a pipes and filters style

The PICCOLA Wiki script is embedded into a Java servlet [Hun98] that delegates its HTTP Requests to the corresponding agents. The script defines the following services:

- A `repository` service that manages files. The contents of a file can be read or written. Each file must be protected against concurrent write access.
- A `doGet` service that handles HTTP GET requests. Depending on whether the request is to view a Wiki file or to edit it, the service returns the appropriate HTML document.
- A `doPost` service that handles HTTP POST requests to update a Wiki page. After modifying the file, it forwards the request to `doGet` so that a user sees the updated page. Finally, a log entry is appended to a recent changes file.
- Several *transformers* that translate the stream of stored ASCII text into HTML documents.

In general, stream composition is done using transformers. Furthermore, we use files as sources and sinks for streams. Using this approach makes it easier to add, remove, or substitute transformers (thus changing the formatting rules) since their interconnections are made explicit in the source code.

Figure 1.2 shows the definition of the two functions `getRequest` and `editRequest`. These functions convert ASCII files into streams of HTML text. The service `doGet` (not

$S \mid T \rightarrow S$	A stream may be piped into a transformer, yielding a new stream.
$T \mid T \rightarrow T$	A transformer may be piped into another transformer, yielding a new transformer.
$T < F \rightarrow S$	A file may be piped into a transformer, yielding a stream.
$S > F \rightarrow F$	A stream may be dumped into a file, yielding the file.
$S >> F \rightarrow F$	A contents of a stream may be appended to a file, yielding the appended file.
$S + S \rightarrow S$	Two streams may be concatenated, yielding a stream.

Table 1.1. *Composition rules for the stream style.*

shown in Figure 1.2) uses these functions to serve the HTTP requests. Note the pipes-and-filter composition with the intentional syntactic resemblance to Unix shell scripts. The main differences in comparison with shell scripts are that transformers (i) work record-at-a-time rather than byte-at-a-time and (ii) are purely pull-flow components, using the `next` method of the previous component, whereas Unix filters read and write simultaneously. The component framework adopted here also supports stream concatenation using the `+` operator.

1.5.2 Stream composition in PICCOLA

An architectural style defines a set of components and the rules governing their composition. We define a pull-flow stream architectural style whose components are streams (S), Transformers (T), and Files (F). The components can be composed (or "connected") using the operators $|$, $+$, $<$, $>$, and $>>$. The corresponding compositions rules are given in Table 1.1.

Compositions like $F + S$ are not permitted. Furthermore, we require that the operators $|$ and $+$ are associative, making it possible to consider compositions of streams and transformers as first-class values. The *algebraic* notation emploied by the framework provides a compact formalism of describing the architectural style the component framework conforms to.

In order to ensure the correct behaviour of composite components, each component of the framework offers an interface that enables low-level interaction. The interface of a stream component, for example, provides three services to access the elements of the stream: `next`, `isEOF`, and `close`. Note that these services must not used by the application programmer; they are only used in the "internal" protocol of composed stream and transformer components.

In order to illustrate the framework implementation, we show the implementation of the abstract superclass for streams `AbstractInputStream` in Figure 1.3.

The abstraction `Class` in Figure 1.3 denotes an abstraction to create class metaobjects with a Smalltalk-like inheritance and method dispatch semantics. We will not show the corresponding code here (refer to [Sch99] for details), but it is important to note that `Class`

```
AbstractInputStream = Class
    parent = Object
    def delta(X) =
        def next() = global.raise("subclass responsibility")
        def isEOF() = global.raise("subclass responsibility")
        def close() = ()
        # infix operators: Stream 'op' Other
        def __|(Right) = Right.prefixStream(X.self())
            # NB: double dispatch for |
        def __+(Right) = ConcatStreams
            first = X.self()
            second = Right
        def __>(File) =
            File.write(X.self())
            return File
        def __>>(File) =
            File.append(X.self())
            return File
```

Fig. 1.3. The class `AbstractInputStream`.

takes a parent-class metaobject (`Object` in the case of `AbstractInputStream`) and a `delta` function as parameters in order to create a class metaobject. The formal parameter `X` in `delta` provides access to `self`.

The composition interface of a transformer component is similar to that of a stream, but since (unconnected) transformers do not contain any elements, there are no services to access them. Transformers implement a service `prefixStream` which is used to compose a transformer with an input stream (or a file) and yields a new (transformed) stream of elements.

Since PICCOLA itself is implemented in Java, we can benefit from Java's reflection support in order to directly integrate (embedded) instances of Java classes into applications. In fact, file streams are not implemented in PICCOLA, but are actually wrapped Java objects. Therefore, we can use Java methods to implement the services `next`, `isEOF`, and `close`. More precisely, we implement `next` and `isEOF` in terms of the `read` method offered by the Java class `java.io.Reader`. Note that this embedding requires only little glue code, mainly that of renaming the services.

1.5.3 From white-box to black-box composition

In order to use the stream framework without subclassing abstract framework classes, we have implemented various components that, appropriately parameterized, yield components with the required behaviour. As an example, consider the class `newTransformer`. It requires a parameter `transformElement`, denoting a function which transforms each

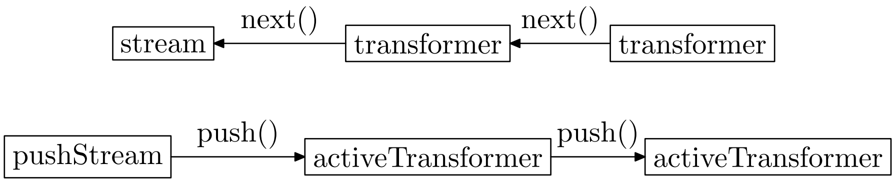

Fig. 1.4. Pull and push streams.

element of the stream. In fact, this parameter can be viewed as a *required* service of a component.

The following example makes use of the `newTransformer` class. It instantiates the transformer which is responsible for translating the intentional links of a Wiki page into a HTML link. In this case, all words starting with a question mark are substituted by the appropriate HTML fragment to make it a hyperlink. Note that the abstraction `substituteAll` is again a wrapped Java class which is part of the `gnu.regexp` package.

```
mkLinks = newTransformer
   def transformElement(Elem) = substituteAll
      regexp = "\\?(\\w+)"
      text = Elem
      by = "<A HREF='$1'>$1</A>"
```

1.5.4 Overcoming compositional mismatch

In its current form, the Wiki components strictly adhere to the pull-flow architectural style illustrated in section 1.5.2. As we extend the functionality of the Wiki, however, we may need functionality offered by external components that do not conform to this style. In many cases, it will not be possible to simply adapt methods by renaming or adding parameters, and some components are more naturally specified in terms of push rather than pull operations (i.e. rather than having upstream components “passively” waiting for downstream components to ask for the next element, upstream components push elements to downstream components; see Figure 1.4).

In the Wiki application, we use wrapped Java output streams for writing HTML. However, these output streams are push-flow and not pull-flow streams, and components conforming to these two styles cannot be freely mixed. Consider the following function that emits a HTML header in our Wiki server:

```
def printHeader(File) =
   global.print("<HEAD>")
   global.print("<TITLE>" + File + "</TITLE>")
   ...
```

Printing is essentially a push operation, and it is not immediately obvious how to define

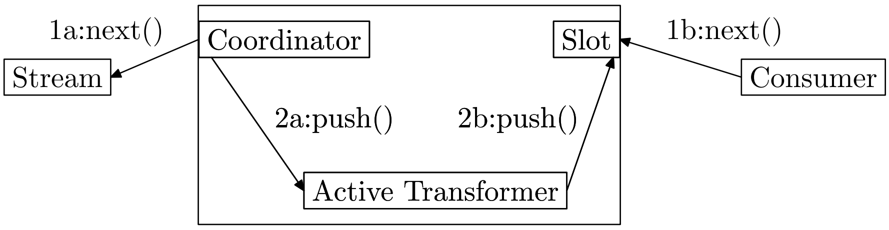

Fig. 1.5. Coordinating push and pull streams.

this functionality as a pull stream. The occurring compositional mismatch can be solved, however, by (i) adapting `printHeader` as a push stream and by (ii) applying a generic glue abstraction (i.e. a *mediator*) that bridges the gap between push and pull streams. The corresponding glue abstraction illustrated in Figure 1.5 and 1.6 consists of a coordinator and a one-slot buffer. The coordinator pulls elements from the upstream component and pushes them into the downstream active transformer, which in turn pushes elements into the slot. The downstream consumer can then pull elements from the slot. Note that the push stream requires a `push` service in its context. Furthermore, the service `processElement` is executed in a context where `push` is bound to the `push` service of the slot (as required).

The glue abstraction is defined as an abstract class that instantiates and binds the coordinator and the slot. The coordinator runs a loop that pulls elements and processes them with an abstract `processElement` method. The class `ActiveWrapper` creates the slot, adapts it to the stream interface, and starts the coordinator. The coordination agent is defined in a loop: while the element read is not empty, the active transformer can process it in its own context. When the loop terminates, the slot and the stream are closed, and the hook service `done` of the client is called.

Note that the coordinator is open for future adaptations and extensions in the sense that it makes only a few assumptions about the context. We simply use form extension to map `push` onto the slot's `push` method and do not change any other external services for the function `processElement`.

Instead of having to subclass `ActiveWrapper`, we again use black-box composition. In our case, the abstraction `asStream` requires a `start` service. Now we can apply our glue abstraction to the given `printHeader` service. Additionally, we wire the `print` service in its context to the `push` label that is provided by `ActiveWrapper`.

```
def mkHead(File) = asStream
    def start() =
        export print = global.push # wire print to push
        printHeader(File)
```

```
ActiveWrapper = Class
   parent = AbstractInputStream
   def delta(X) =
      slot = global.newSlot()     # create Buffer
      atEOF = global.newRefCell(0)
      return                      # adapted Stream Interface
         next=slot.pull, isEOF=atEOF.get
         close=X.init.IS.close

      # start coordinator
      agContext = (global, push = slot.push)
      def loop() =
         elem = X.init.IS.next()
         isNotEmpty(elem)
            then:
                X.init.processElement(elem) in agContext
                loop()
      loop()
      slot.close()
      X.init.done()               # hook for client
      atEOF.set(1)
      X.init.IS.close()           # close stream when done
```

Fig. 1.6. Generic glue abstraction.

1.5.5 Lessons learned

The Wiki example illustrates a number of principles that we claim can also be applied to other contexts. We started by selecting an architectural style that was appropriate for our problem. The fact that PICCOLA offers user defined operators allows us to use a syntax that highlights our style. External components like Java streams can be integrated by means of ad hoc wrappers. In PICCOLA, adapting interfaces is simply done by composing forms since the interface of a service is represented as a label in a form.

Components that do not correspond to the required architectural style of a framework may be integrated by means of glue abstractions. We have shown a generic coordinator that mediated the compositional mismatch between push-flow and pull-flow streams largely because there was a simple, unifying foundation of agents and forms in which we could model both styles.

The Wiki application also embodies natural guidelines for maintenance. Changing requirements may be addressed by reconfiguring individual components (i.e. replacing their required services), reconfiguring interconnections between components (i.e. adapting the scripts), introducing new external components (i.e. possibly using glue or coordination abstractions), and deriving new components from old ones.

The PICCOLA Wiki can be easily extended in a number of interesting ways. For example, we can make file streams thread-safe by applying a generic readers-writers synchronisation policy them [Lea96]. Writing and appending files requires exclusive access, whereas several readers may be concurrently active. Another possible extension is to replace the repository with one using a version control system like RCS.

We do not pretend that all possible changes in requirements can be addressed while maintaining a single architectural style. A style itself may have to evolve with time, or eventually have to be replaced if it no longer provides a suitable metaphor for the problem domain.

We have shown using the Wiki example that a pipes-and-filters architectural style can be made explicit in PICCOLA. At the very end, this means that we have modeled streams in the $\pi\mathcal{L}$-calculus. This is not surprising per se since the $\pi\mathcal{L}$-calculus is Turing-equivalent. However, encodings of higher-level interaction types, like event based notification, often turn out to be quite awkward in the $\pi\mathcal{L}$-calculus itself. When we enrich the calculus to a language which defines forms, functional applications, and contexts as primitives, these encodings turn out to be more compact, understandable, and composable.

1.6 Related Work

In the past twenty years, there has been considerable research into the foundations of concurrency, and much of this research has focused on process algebras (i.e. equational theories of communicating processes) and process calculi (i.e. operational theories of evolving systems of communicating processes). The π-calculus has proven to be successful for modeling object-oriented concepts [HT91, Jon93, Vas94, BS95, Wal95], and Sangiorgi has demonstrated that Abadi and Cardelli's first-order functional *Object Calculus* [AC96] can be faithfully translated to the π-calculus [San96].

The design of PICCOLA owes a great deal to the experimental programming language PICT [PT97]. PICT's programming constructs are provided as syntactic sugar and as library abstractions on top of a core language that implements the asynchronous π-calculus. We have used PICT extensively to experiment with different ways to model compositional abstractions in the π-calculus [LSN96, SL97]. These experiments led us to conclude that form-based communication is a better basis for modeling composition than tuple-based communication, which resulted in the development of the $\pi\mathcal{L}$-calculus.

PICCOLA differs from PICT in significant ways. First, PICT was primarily developed to experiment with type systems, whereas PICCOLA was developed to experiment with abstractions for software composition. As a consequence, PICCOLA is an untyped language and provides different abstractions than PICT. Second, record-like structures (i.e. forms) in PICCOLA are part of the underlying calculus whereas the are defined as syntactic sugar on top of the core of PICT. Furthermore, PICCOLA supports asymmetric record concatenation which is not available in PICT. Finally, the runtime system of PICT is implemented in C and, therefore, offers a simple interface to integrate C functions. The runtime system of PICCOLA, on the other hand, is implemented in Java and allows for interoperation with Java objects.

The class abstractions implemented in PICCOLA are based on object encodings defined in PICT [PT95, LSN96]: an object is viewed as an agent containing a set of local agents and channels representing methods and instance variables, respectively, whereas the interface of an object is a form containing bindings for all exported features. Classes are reified as first-class entities (i.e. class metaobjects), which allow us to integrate features such as controlled object instantiation, class variables and methods, inheritance, reusable synchronization policies, and different method dispatch strategies into the model. In contrast to the object model defined in PICT and other object-oriented programming languages, PICCOLA's object model makes a stronger separation between functional elements (i.e. methods) and their compositions (i.e. inheritance), which allows us to define multiple objects models supporting different kinds of inheritance and method dispatch strategies [Sch99].

The syntax of the PICCOLA version presented in this chapter deliberately resembles that of *Python*, an object-oriented scripting language that supports both scripting and programming in the large [vR96, WvRA96]. It supports objects, classes as first-class values, single and multiple inheritance, modules as well as a runtime (meta-)object protocol. In fact, Python has a unifying concept: everything is an object, including functions and classes. Functions (and methods) can be defined in a way that they support positional parameters (i.e. tuples) or keyword arguments (i.e. à la forms). Python provides operator overloading based on features of the (meta-)object protocol, which can be used to make the architecture of an application explicit in the source code [Sch99], similar to the approach we have described in section 1.5. Furthermore, the (meta-)object protocol offers limited support to change the underlying object model, although it does not have a meta-reflective architecture like Smalltalk [GR89]. Finally, Python is not inherently concurrent, although there is a POSIX-dependent threads library, and some researchers have experimented with active object models for Python [PHMS97].

PICCOLA can also be compared to numerous coordination languages. *Linda* is generally considered to be the prototypical coordination language, although it is not a language on its own, but a *coordination medium*, consisting of a tuple space (i.e. a blackboard) to which agents may put and get tuples using primitives added to a host language [CG89]. The main problem with Linda is that computational and coordination code are typically intertwined, making it difficult or impossible to define separate coordination abstractions. *Darwin* is a “configuration language” for distributed agents that models composition in terms of dataflow [MDK92]. The composition primitives of Darwin have a formal semantics specified in terms of the π-calculus [EP93]. *Manifold* is a “pure coordination language” that models external components as processes [Arb96]. A manifold is a process that can dynamically connect input- and output-ports depending on its current state. Therefore, it is particularly suitable for specifying reusable higher-level coordination abstractions and protocols as well as for implementing dynamically evolving architectures. Manifold has some interesting successes in parallelizing sequential legacy code by splitting monolithic applications into parallel components that are coordinated by a Manifold layer [Arb95].

Forms have appeared in countless shapes and guises in programming languages over many years, as dictionaries, records, keyword arguments, environments, and URLs. Although forms are clearly not a new idea, we believe that PICCOLA is the first language that adopts forms as a basic mechanism for concurrent programming, and in particular as the key concept for modeling extensible and composable systems.

Aspect-Oriented Programming is an approach for separating certain aspects of programs that cannot be easily specified as software abstractions, and there exists an Java implementation of an *aspect language* called ASPECTJ which allows to specify aspects which can be weaved into Java source code [KLM+97]. Initial experiments have shown that certain aspects can be nicely expressed in PICCOLA. For example, Readers and Writers synchronization policies cannot be factored out as software abstractions in Java [Lea96], but it is relatively straightforward to achieve this in both ASPECTJ and in PICCOLA. Whether aspects in general can be addressed by PICCOLA's compositional paradigm of agents and forms, however, is still an open question.

1.7 Concluding remarks

In this chapter, we have argued that the flexibility and adaptability needed for component-based applications to cope with changing requirements can be substantially enhanced if we think not only in terms of *components*, but also in terms of *architectures*, *scripts*, *coordination*, and *glue*. Furthermore, we have presented PICCOLA, a small language for specifying applications as compositions of components, that embodies the paradigm of "Applications = Components + Scripts" and fulfills the requirements for a general-purpose composition language.

PICCOLA's language constructs are translated into the $\pi\mathcal{L}$-calculus, an inherently polymorphic variant of the π-calculus. A component is viewed as a set of interconnected agents. The interface of a component is represented as a form, a special notion of extensible records. PICCOLA models composition in terms of agents that exchange forms along private channels whereas higher-level compositional abstractions are introduced as sets of operators over sorts of components. Using such an approach, we hope to cleanly integrate all required features for software composition in one unifying concept (i.e. the concept of agents and forms) and to reason about components, compositions, architectures, and architectural styles.

The PICCOLA prototype we have presented demonstrates that:

- the architecture of a component-based application can be made explicit by separately specifying components, the architectural styles they conform to, and the script that composes them,
- separating an application into components and scripts enhances its configurability, extensibility, and maintainability.
- a composition language generalizes scripting languages by providing additional support for specifying architectural styles, compositional abstractions, coordination abstractions as well as glue abstractions,

- a composition language can be directly built on top of a unifying foundation of agents and forms,
- this foundation provides a good basis for specifying higher-level components and connectors; forms are needed to model extensible interfaces and contexts, and agents are needed to model coordination abstractions,
- multiple object models can be represented, which makes it possible to bridge compositional mismatches in heterogeneous applications.

Ultimately we are targeting the development of a general-purpose composition language as well as a formal model for component-based application development. In order to achieve this goal, future work in the following areas is needed:

Language. As mentioned in section 1.1, we explore two approaches in the development of a composition language: an approach based on an imperative style of programming and another approach emphasizing a functional and declarative style of programming. As one of the next steps, we intend to further validate our experiments, analyze the advantages and disadvantages of both approaches, and to define an appropriate unification. Furthermore, the similarity of agents and forms in PICCOLA suggests another opportunity for unification: can we simplify the language by unifying these two concepts and by viewing an agent as an expression that evaluates to a form? Can the language be easily extended to explicitly model the location of distributed agents, as in the ambient calculus [CG98]? Open systems allow components to be plugged in at run-time – what reflective features are needed in PICCOLA to compose components dynamically? The current implementation is stable enough to be used for non-trivial experiments, but it is far from being a product. As the language design stabilizes, we will attempt to improve the tools and composition environment, with a particular focus on visualization [Cri99].

Applications. Although we claim that PICCOLA can be used to compose applications according to different architectural styles, we have only demonstrated a single, well-understood style, namely that of pipes-and-filters. We plan to experiment with specifying other architectural styles as operators over sorts of components. In particular, we plan to investigate GUI composition, other forms of event-based composition, blackboard-based composition, and domain-specific composition (e.g., for workflows).

OBJECT MODELS. PICCOLA does not have a built-in object model, but can support multiple models as library abstractions. We further plan to investigate how PICCOLA can be used to mediate between different external object and component models (such as those of different programming languages and middleware platforms). We are particularly interested in identifying necessary glue and coordination abstractions for bridging compositional mismatches.

Reasoning. The original motivation for developing PICCOLA "bottom-up" from a process calculus foundation was to ensure that the interaction of high-level compositional abstractions has a precise semantics in terms of a simple computational model. This goal has been reached. In addition, however, we wish to exploit the established theory and techniques for reasoning about software composition. The next steps are to formally express the contracts that are often implicit in an architectural style, in order to reason about valid compositions and about compositional mismatches (e.g., protocol mismatches [Nie93b]).

PICCOLA is an attempt to design a language that supports a particular paradigm for software composition in terms of components, architectural styles, scripts, coordination, and glue. In this chapter, we have mainly focused on technical issues. This work, however, should be understood in a broader context of component-based software development [ND95, NM95b]. There are just as many, and arguable equally 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 existing object-oriented applications* in order to arrive at a flexible component-based design? Given a problem domain and a body of experience from several applications, how do we re-engineer the software into a component framework? As we develop a component framework, how do we select a suitable architectural style to support black-box composition? 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?

Although we do not pretend to have the answers to all these questions, we believe that separating applications into components and scripts (i.e. making a clear separation between computational elements and their relationships) is a necessary step towards a methodology for component-based software development.

Acknowledgements

We thank all members of the Software Composition Group for their support of this work, especially Juan Carlos Cruz, Serge Demeyer, Robb Nebbe, and Tamar Richner for helpful comments. We also express our gratitude to the anonymous reviewers of an earlier draft of this chapter.

This work has been funded by the Swiss National Science Foundation under Project No. 20-53711.98, "A framework approach to composing heteregeneous applications".

Bibliography

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[Arb95] Farhad Arbab. Coordination of massively concurrent activities. Technical report, Centrum voor Wiskunde en Informatica (CWI), 1995.

[Arb96] 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.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[BS95] Manuel Barrio Solorzano. *Estudio de Aspectos Dinamicos en Sistemas Orientados al Objecto*. PhD thesis, Universidad de Valladolid, September 1995.

[CG89] Nicholas Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[CG98] Luca Cardelli and Andrew. D. Gordon. Mobile Ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computational Structures*, LNCS 1378, pages 140–155. Springer, 1998.

[CM94] Luca Cardelli and John C. Mitchell. Operations on Records. In *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

[Cri99] Cristina Gheorghiu Cris. Visualisierung von π-programmen. Informatikprojekt, January 1999. University of Bern.

[EP93] Susan Eisenbach and Ross Paterson. Pi-Calculus Semantics of the Concurrent Configuration Language Darwin. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, volume 2. IEEE Computer Society Press, 1993.

[Gen81] Morven Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software – Practice and Experience*, 11:435–466, 1981.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, September 1989.

[HT91] Kohei Honda and Mario Tokoro. An Object Calculus for Asynchronous Communication. In Pierre America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 133–147. Springer, July 1991.

[Hun98] Jason Hunter. *Java Servlet Programming*. O'Reilly & Associates, Inc, 1998.

[Jon93] Cliff B. Jones. A Pi-Calculus Semantics for an Object-Based Design Notation. In Eike Best, editor, *Proceedings CONCUR '93*, LNCS 715, pages 158–172. Springer, 1993.

[KdRB91] Grégor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KLM$^+$97] Grégor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, LNCS 1241, pages 220–242. Springer, June 1997.

[Kru95] Philippe B. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, November 1995.

[Lea96] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. The Java Series. Addison-Wesley, October 1996.

[LSN96] Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Using Metaobjects to Model Concurrent Objects with PICT. In *Proceedings of Langages et Modèles à Objets '96*, pages 1–12, Leysin, October 1996.

[Lum99] Markus Lumpe. *A π-Calculus Based Approach to Software Composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[MDK92] Jeff Magee, Naranker Dulay, and Jeff Kramer. Structuring Parallel and Distributed Programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, March 1992.

[Mil90] Robin Milner. Functions as Processes. In *Proceedings ICALP '90*, LNCS 443, pages 167–180. Springer, July 1990.

[Mil91] Robin Milner. The polyadic Pi-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Computer Science Department, University of Edinburgh, UK, October 1991.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[ND95] 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.

[Nie93a] Oscar Nierstrasz. Composing Active Objects. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 151–171. MIT Press, 1993.

[Nie93b] Oscar Nierstrasz. Regular Types for Active Objects. In *Proceedings OOPSLA '93*, volume 28 of *ACM SIGPLAN Notices*, pages 1–15, September 1993.

[NM95a] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a Composition Language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer, 1995.

[NM95b] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[NSL96] Oscar Nierstrasz, Jean-Guy Schneider, and Markus Lumpe. Formalizing Composable Software Systems – A Research Agenda. In *Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 271–282. Chapmann & Hall, 1996.

[NTMS91] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey, and Marc Stadelman. Objects + Scripts = Applications. In *Proceedings Esprit 1991 Conference*, pages 534–552, Dordrecht, NL, 1991. Kluwer Academic Publisher.

[PHMS97] Michael Papathomas, Juan Hernandez, Juan Manuel Murillo, and Fernando Sanchez. Inheritance and Expressive power in Concurrent Object-Oriented Programming. In Roland Ducournau and Serge Garlatti, editors, *Proceedings of Langages et Modèles à Objets '97*, pages 45–60, Roscoff, October 1997. Hermes.

[PT95] Benjamin C. Pierce and David N. Turner. Concurrent Objects in a Process Calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP)*, LNCS 907, pages 187–215. Springer, April 1995.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, March 1997.

[Sam97] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.

[San96] Davide Sangiorgi. An interpretation of Typed Objects into Typed Pi-calculus. Technical Report RR-3000, INRIA Sophia-Antipolis, September 1996.

[Sch99] Jean-Guy Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, 1999. to appear.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[SL97] Jean-Guy Schneider and Markus Lumpe. Synchronizing Concurrent Objects in the Pi-Calculus. In Roland Ducournau and Serge Garlatti, editors, *Proceedings of Langages et Modèles à Objets '97*, pages 61–76, Roscoff, October 1997. Hermes.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, Department of Computer Science, University of Edinburgh, UK, 1996.

[Vas94] Vasco T. Vasconcelos. Typed Concurrent Objects. In Mario Tokoro and Remo Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 100–117. Springer, July 1994.

[vR96] Guido van Rossum. Python Reference Manual. Technical report, Corporation for National Research Initiatives (CNRI), October 1996.

[Wal95] David J. Walker. Objects in the Pi-Calculus. *Information and Computation*, 116(2):253–271, 1995.

[WvRA96] Aaron Watters, Guido van Rossum, and James Ahlstrom. *Internet Programming with Python*. MIS Press, October 1996.