

6. Agent Coordination via Scripting Languages

Jean-Guy Schneider¹, Markus Lumpe², and Oscar Nierstrasz³

¹ School of Information Technology, Swinburne University of Technology
P.O. Box 218, Hawthorn VIC 3122, Australia

² Department of Computer Science, Iowa State University
113 Atanasoff Hall, Ames IA 50011-1041, USA

³ Institute of Computer Science and Applied Mathematics (IAM),
University of Berne, Neubrückestrasse 10, CH-3012 Bern, Switzerland

Summary.

In recent years, so-called *scripting languages* have become increasingly popular as they provide means to build quickly flexible applications from a set of prefabricated components. These languages typically support a single, specific architectural style of composing components (e.g. the *pipes and filters* architectural style), and they are designed to address a specific application domain. Although scripting languages and coordination languages have evolved from different roots and have been developed to solve different problems, we argue that both address similar *separations of concerns*. Scripting languages achieve a separation of *components* from the *scripts* that configure and compose them, whilst coordination languages separate *computational entities* from the *coordination* code that manages dependencies between them. In this chapter we will define coordination in the context of a conceptual framework for component-based software development. Furthermore, we will discuss main properties and abstractions of scripting languages and will compare selected scripting languages with respect to the identified core concepts. Finally, using a small set of sample applications, we will illustrate the power and the limitations of these concepts in order to define agent coordination.

6.1 Introduction

It is widely accepted today that closed and proprietary software systems cannot keep up with the pace of changing user requirements. In order to overcome the problems of these systems, modern applications are being built as collections of distributed software agents. Since these agents run in a distributed environment and concurrently access resources, they not only need to exchange information, but they must coordinate their actions to achieve the required functionality. Unfortunately, the corresponding coordination code is often intermixed with computational code, which reduces the flexibility and adaptability, hence the reusability of distributed agents.

Talking about agents, it is often not clear what kind of entities we should consider as being agents. In this chapter, we will not worry too much about giving a precise notion of software agents, but simply adapt a definition given in [28]. From our point of view, an agent should be considered as a software

program that can act autonomously on behalf of a human or another software or hardware system. Agents represent well-defined services, but are not required to be either mobile or intelligent [32].

There are several approaches to separate the coordination part of agents from the computational part. Common to all these approaches is the goal to make a *clear separation between computational entities and their relationships*.

Software architectures, for example, focus on describing software systems at a level beyond simple algorithms and data structures, including global organization and control structure. They express the structure of applications in terms of processing elements, data elements, and explicit connecting elements (also known as connectors). Furthermore, architectural styles abstract over a set of related software architectures and define a set of rules how the elements can be combined [38, 46].

A similar approach is taken in the field of component-based programming, where applications are expressed as compositions of plug-compatible software components [31]. Of particular interest is the fact that coordination aspects can be encapsulated into reusable coordination components [48]. This approach not only enhances the explicit separation of coordination and computational code, but also allows application developers to reuse coordination aspects in different settings.

A third approach in this direction is the concept of aspect-oriented programming (AOP). AOP aims at separating properties of software systems which can be cleanly encapsulated in a generalized procedure (i.e. components) from properties for which the implementation cannot be cleanly encapsulated (i.e. aspects) [23]. Aspects and components generally cross-cut each other in a system's implementation.

Naturally, it is not enough to separate different concerns of systems into deployable entities, but one needs a way to build applications as assemblies of such entities (i.e. to express applications as compositions of composable elements).

In recent years, *scripting languages* have become increasingly popular for quickly building 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 the Bourne Shell [7]), and they are designed with a specific application domain in mind (system administration, graphical user interfaces etc.). Furthermore, scripting languages are extensible as new abstractions can be integrated into scripting environments. Finally, it is often possible to embed scripts into existing components, offering a flexible way for adaptation and extension. Hence, scripting languages seem to be ideal tools for building open, distributed systems in general and for both implementing and coordinating software agents in particular.

In this chapter, it is not our goal to focus on basic coordination models and abstractions of scripting languages alone. We would like to view coordination from a different perspective, set the relation to other approaches which aim at separating independent concerns into deployable entities, in particular to component-based software development, and discuss the influence of scripting on building applications as assemblies of these entities. Furthermore, we would like to stress the fact that scripting languages do not only allow us to coordinate distributed agents, but also to implement the agents themselves as scripts.

The remaining parts of this chapter are organized as follows: in section 6.2, we clarify important terms used throughout this chapter and define coordination in the context of a conceptual framework for component-based software development. In section 6.3, we discuss the main properties and abstractions of scripting languages, compare selected scripting languages, and illustrate some important concepts of each of these languages. In section 6.4, we show how the concepts discussed previously are applied in practice by illustrating a small set of sample applications and discuss limitations of existing scripting languages. Finally, we conclude this chapter in section 6.5 with a summary of the main observations and a discussion about related and future work.

6.2 A conceptual framework for software composition

It is generally accepted that modern software systems are increasingly required to be open, flexible conglomerations of distributed software agents rather than monolithic heaps of code. This places a strain on old-fashioned software technology and methods that are based on the maxim

$$\text{Applications} = \text{Algorithms} + \text{Data}.$$

This maxim has some relevance for well-defined and delimited problems only and is often applied in imperative programming languages that focus on top-down decomposition.

Coordination approaches, on the other hand, view systems as (i) computational entities that encapsulate well-defined functionality and interact with each other in order to achieve a common goal and (ii) coordination entities that manage the corresponding interactions [19]. These approaches can be best described by the maxim

$$\text{Applications} = \text{Computation} + \text{Coordination}.$$

Coordination can be seen as the management of dependencies between computational entities, or as the “glue” between distributed software agents [37].

Recent work in the area of coordination has focused on the development of particular coordination languages that realize a particular model of coordination (the interested reader may find corresponding overviews in chapters 2 and 4 as well as in [37]). Coordination problems, however, cannot always

be solved by solely using a particular model. Furthermore, data-driven coordination approaches such as Linda [9] do not enforce a clear separation of concerns as a mixture of coordination and computation code within an agent is still possible. Finally, coordination languages generally do not allow the definition of reusable coordination abstractions at a higher level than the basic mechanisms and paradigms supported by the underlying model. In order to tackle the problems related to the development of open distributed systems, we need an approach which not only enforces a clear separation between computational and compositional entities, but also overcomes other problems with existing coordination models and languages (e.g. the interleaving of coordination and computation code in Linda-based systems).

In the last decade, component-based software technology has emerged as an approach to cope with the advances in computer hardware technology and rapidly changing systems requirements [47]. Component-based systems aim at achieving flexibility by clearly separating the stable parts of the system (i.e. the components) from the specification of their composition (i.e. scripts). Hence, a component-based engineering style can be best described by the maxim

$$\text{Applications} = \text{Components} + \text{Scripts.}$$

Components are black-box entities that encapsulate services behind well-defined interfaces whereas scripts define how the components are composed. More precisely, scripts specify connections between the services of components. In contrast to data-driven coordination models, which only make the distinction between computation and coordination *functionality*, component-based software technology aims at encapsulating coordination functionality into independent units of deployment (i.e. components) [48].

It is important to note that scripts may not only be used for composition, but also for implementing components (i.e. a composition of components is again a component). Hence, if we use the term *scriptability*, we either mean components offering an interface for scripting (scriptable components) or the possibility to implement components or agents as scripts.

The importance of component-based engineering in the context of open systems development is probably best underlined by the following quote:

What I think is quite important, but often underrated, is the dichotomy that scripting forces on application design. It encourages the development of reusable components (i.e. “bricks”) in system programming languages and the assembly of these components with scripts (i.e. “mortar”).

Brent Welch

Our experience in developing component-based systems has shown that components and scripts are only half of the truth if we want to build flexible and extensible distributed systems. It is necessary that we also think in terms of *frameworks*, *architectures*, and *glue* [43]. In this section, we will define and

clarify the relevant terms and propose a conceptual framework where these five techniques are combined.

A *software component* is a static abstraction with plugs and can be seen as a kind of black-box entity that hides its implementation details [31]. It is a static entity in the sense that it must be instantiated in order to be used. A software component has plugs which are not only used to *provide* services, but also to *require* them. It is important to note 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 a composable element of a component framework [26].

A *component framework* is a collection of software components and architectural styles that determine the interfaces that components may have, the connectors that may be used to plug components together, and the rules governing component composition. In contrast to an object-oriented framework where an application is generally built by subclassing framework classes that respond to specific application requirements (also known as *hot spots* [40]), a component framework primarily focuses on object and class (i.e. component) composition (also known as *black-box* reuse).

The main idea behind component-based software development is that an application developer only has to write a small amount of *wiring code* in order to establish connections between components. This wiring, or *scripting*, can take various forms, depending on the nature and granularity of the components, the nature and problem domain of the underlying framework, and the composition model. Composition may occur at compile-time, link-time, or run-time, and may be very rigid and static (like the syntactic expansion that occurs when C++ templates are composed [30]), or very flexible and dynamic (like that supported by Tcl or other scripting languages [36]).

In an ideal world, there are components available for any task an application has to perform and these components can be simply plugged together. However, it is sometimes necessary to reuse a component in an environment different than the one it has been designed for and that this environment does not match the assumptions the component makes about the structure of the system to be part of. In such a situation, *glue* code is needed to overcome the mismatched assumptions and to adapt components in order to be composable.

The question arises how and where coordination concerns fit into this conceptual framework. The purpose of a coordination model is to separate computational entities from their interactions and, therefore, needs to provide abstractions for controlling synchronization, communication, creation, and termination of concurrent and distributed computational activities. The main idea behind coordination in component-based software development is to encapsulate the first two concerns of a coordination model (i.e. synchronization and communication) into separate components (also known as *coordination components* [48]) and to handle the other two concerns (i.e. cre-

ation and termination of activities) in scripts. This approach enhances the flexibility to exchange individual synchronization and communication concerns (e.g. different network protocols) as “ordinary” components are simply plugged together using appropriately selected coordination components (refer also to the Bourne Shell example given below). Therefore, *coordination can be considered as scripting of concurrent and distributed components* [43].

At present, however, there does not exist a programming language or system that supports general-purpose software composition based on the conceptual framework presented in this section. Although scripting languages and 4GLs (such as Visual Basic [27]) go a long way in the direction of open systems development, they mainly focus on special application domains and offer only rudimentary support for the integration of components not built within the system. The reason for this situation is not only the lack of well-defined (or standardized) component interfaces, but also the ad-hoc way the semantics of the underlying language models are defined.

In order to illustrate how the conceptual framework illustrated above is applied in practice, consider the Bourne Shell script given below which prints the names of all users who were recently working on a UNIX machine. The Bourne Shell defines a simple component framework where UNIX commands (usually called *filters*), files, and character streams are the components and the pipe operator ‘|’ as well as the other stream redirectors (such as ‘<’, ‘>’ etc.) are the corresponding connectors. The standard input stream and the command line arguments of a filter can be considered as required services whereas the standard output and error streams as provided services, respectively.

```
last | awk '{ print $1 }' | sort -u | rsh server expand | awk '{ print $1 }'
```

Analyzing this Bourne Shell script, it is easy to identify components and connectors, the underlying architecture, as well as other interesting properties:

- the script consists of five components (the filters `last`, `awk`, `sort`, `expand`, and `awk`), each fulfilling a well-defined task,
- a data source (i.e. a system file read by the filter `last`) and a data sink (the standard output stream of the second `awk` filter),
- successive components are connected by a pipe (indicated by the ‘|’ symbol) and interact using character streams,
- the filter `rsh` is used to communicate with the filter `expand`, which runs on a remote server,¹
- the components have to be instantiated, and the functionality of each component can be specialized at instantiation by passing different (command line) arguments (e.g. the filter `sort` is used with the argument `-u`).

¹ The information about the full names of users is only available on a dedicated server (indicated by `server` in the script above).

The components and the character streams of the script form a pipeline, where each component only depends on the output of its predecessor. Since many Bourne Shell scripts fulfill similar restrictions, they are often associated with a *pipe and filter* architectural style [46]. However, scripts are not restricted to this style, and it is possible to define more complex unidirectional data-flow architectures [6] by connecting the standard error stream of a filter to the standard input stream of another filter using the connector '|&'. The composition of filters using the pipe connector again leads to a filter (i.e. a UNIX process which reads from the standard input stream and writes to the standard output and error streams).

In the script given above, the filter `rsh`, which is used to communicate with the filter `expand` running on a remote host, deserves a special attention. From the perspective of the conceptual framework, `rsh` acts both as a generic *glue component* and a *coordination abstraction*, as we will explain in the following.

Since the information about the full names of users is only available on a dedicated server, the filter `expand` cannot be run locally, but must run on the corresponding remote server. However, the pipe connector of the Bourne Shell only allows connections between filters running on the same host. In order to overcome the resulting architectural mismatch [18], we need a way to (i) start the process `expand` on the remote server and (ii) connect with its input and output streams. This is exactly what the filter `rsh` does: it starts a process on the remote server and forces this process to use `rsh`'s input and output streams as standard I/O medium. This has the effect that the remote process `expand` created by `rsh` reads the output of the filter `sort` and produces the input for `awk`, hence ensuring the correct communication between the two hosts and the three processes involved. Note that `rsh` is a *generic glue abstraction* as it can be used to instantiate and communicate with any UNIX process running on a different host.

On the other hand, `rsh` can also be considered as a *coordination abstraction*. Due to the inherently concurrent nature of UNIX processes, the character streams act as coordinators (synchronizers, buffers) between the filters, as UNIX commands generally do not specify any particular synchronization model; they simply read from/write onto their I/O streams. In addition to local synchronization, `rsh` has to open a network connection, synchronize the communication between the remote process, itself, and the local I/O streams, and close the network connection upon termination of the processes. Hence, `rsh` takes care of all coordination-related concerns we have previously given in this section. Due to the fact that `rsh` encapsulates all this functionality as a single unit of deployment, it must be considered as a coordination abstraction or as a coordination component.

The reader should note that the filters of the example given above could also be plugged together using a system programming language like C. However, C only offers low-level (library) abstractions for connecting the standard output stream of a command to the standard input stream of another com-

mand or for invoking a process on a remote server. Hence, it is not easily possible to make the architecture of the application explicit, especially as C does not offer such a convenient syntax for expressing connections of UNIX commands as the Bourne Shell.

6.3 Scripting languages at a glance

6.3.1 What is scripting?

Unfortunately, this question does not have a generally accepted answer. Consider the following two definitions:

“A script language is a language that is primarily interpreted, and on a UNIX system, it can be invoked directly from a text file using #!.”
Anonymous Usenet User

and

“A scripting language introduces and binds a set of software components that collaborate to solve a particular problem [33].”

These two definitions mark the lower and upper bound of possible applications of scripting languages. However, they fall short of giving us an intuition when and how we should use a scripting language to solve a given problem. Furthermore, these definitions are rather vague and lack a precise characterization in terms of provided features or supported application domains.

Many researchers have been working on a characterization of scripting languages. We will summarize the most prominent contributions in this field and elaborate our own definition.

John Ousterhout argues that scripting languages are designed for “gluing” applications [36]. They provide a higher level of programming than assembly or system programming languages, much weaker typing than system programming languages, and an interpreted development environment. Scripting assumes the existence of powerful components and provides the means to connect them together. However, scripting languages sacrifice execution efficiency to improve speed of development. Please note that in this context, Ousterhout uses the term glue in a much broader sense than we have defined it in the previous section.

Guido van Rossum, the inventor of Python [50], defines the main characteristics of scripting languages as follows: scripting languages should (i) provide text processing primitives, (ii) offer some form of automatic memory management, (iii) not require a mandatory separate compilation phase, (iv) favour high-level expressiveness over execution speed, and (v) interface well with the rest of the system.

Brent Welch puts emphasis on two aspects: *embeddability* and *extensibility*. In his view, scripting languages should (i) be interpreted, not compiled,

(ii) be dynamically typed, (iii) offer abstractions for introspection, (iv) be embeddable and extensible, and (v) have a simple syntax.

Embeddability and extensibility are two important properties of scripting languages, which tells them apart from other programming languages. Extensibility is needed in order to incorporate new abstractions (components or connectors) into the language, encouraging the integration of legacy code. Embedding a script into an existing component or application (e.g. Visual Basic [27] as scripting facility for Microsoft Word or Excel) offers a flexible way for adapting and extending this component, enable to configure applications to user defined needs, or simplify repeated complex editing as in case of Microsoft Word.

Finally, Clemens Szyperski claims that scripting is quite similar to application building, but unlike mainstream (component) programming, scripts usually do not introduce new components [47]. Scripts are used to simply plug existing components together: scripts can be seen as introducing behaviour but no new state. Therefore, *scripting* aims at late and high-level gluing.

Summarizing the definitions given above, we argue that scripting languages can be characterized as follows:

- The purpose of a scripting language is the development of applications by plugging existing components together (i.e. the primary focus is on *composition*).
- Scripting languages generally favour *high-level programming* over execution speed.
- Scripting languages are *extensible* and *scalable*: they are designed for extending the language model with new abstractions (e.g. new components or connectors) and for interoperating with components written in other languages.
- Scripting languages are *embeddable*: it is possible to embed them into existing components or applications (e.g. Microsoft Word), offering a flexible way for *adaptation*, *configuration*, and *extension*.
- Scripting languages are in general *interpreted* and offer *automatic memory management*.
- Scripting languages are *dynamically* and *weakly typed* and offer support for runtime *introspection*.

Using these properties, we define scripting and scripting languages as follows:

“Scripting is a high-level binding technology for component-based application development. Scripting itself is done using a high-level programming language (i.e. a scripting language) that allows us to create, customize, and assemble components into a predefined software architecture [42].”

6.3.2 Characterization of scripting languages

We have now identified the most distinguished properties of scripting languages. However, not all scripting languages aim to address the same application domain (e.g. Perl [51] is primarily used for text manipulation, whereas AppleScript [11] is used to configure and control “Part Editors” in the *Open Scripting Architecture* (OSA) of OpenDoc). In order to appropriately characterize scripting languages, we also have to classify them by the features they provide.

Like system programming languages (such as C or Java), scripting languages are generally not categorized according to their syntax or their semantic domain, but according to the language constructs (or *features*) they offer (or do not offer). For example, depending on the provided features, a scripting language is better suited for text-processing (e.g. Perl) than for building graphical user interfaces. For Tcl [35], on the other hand, the opposite holds.

We have carried out a detailed analysis of existing scripting languages. In the following, we will identify the most important features that modern scripting languages support. We will, however, distinguish between *essential features* and *characterizing features*. Essential features are those that must be supported by any scripting language while characterizing features classify scripting languages in terms of the language design space.

We have identified two concepts which are essential for scripting languages: (i) *encapsulation and wiring* and (ii) *external interoperability*.

Encapsulation and wiring. In order to build an application as a composition of components, a scripting language must support some notion of components and connectors. More precisely, a scripting language must offer mechanisms for encapsulation and wiring.

Besides the notion of components, a scripting language must offer a set of mechanisms which allow one to connect provided and required services of corresponding components. Such mechanisms can be as “low-level” as a function call or as higher-level as the pipe-operator in the Bourne Shell language.

An important aspect of encapsulation and wiring is whether a language is *compositionally complete* (i.e. it is possible to encapsulate a composition of components as a composite component). For example, a Bourne Shell script can be used as a component of another shell script.

External interoperability. In order to use components not written in the language itself, it is necessary that a scripting language provides features to interoperate with components written in other languages. We denote these features as external interoperability.

Interoperability features are important if a component is implemented as a composition of other components, but does not completely fulfill all actual requirements (e.g. it does not have the required run-time performance). In this kind of situation, it must be possible to reimplement

this component with more favourable run-time behaviour and integrate this new component into the scripting environment using interoperability features.

Characterizing features are summarized in the following list. The reader should note that these features are not only important for scripting languages, but for programming languages in general.

Embeddability. Scripting languages may be directly embedded into an application or component (e.g. JavaScript [15] or Visual Basic [27]) while others offer an interface to embed them into other programming languages (e.g. Python offers an interface for C/C++).

Extensibility. Scripting languages often provide an approach to extend themselves with additional abstractions (new components and connectors). As an example, the core of Tcl [35] does not offer the concept of classes and objects, but the `stooops` extension (which is fully written in Tcl) introduces abstractions for object-oriented programming [16].

Objects. A comparison of popular scripting languages reveals that they either directly support the notion of objects (e.g. Python or JavaScript) or there are extensions which introduce objects (e.g. `stooops`).

Exceptions. For programming in the large and for testing applications, it is useful if a language has features to explicitly cope with errors and exceptions.

Execution model. A criterion to distinguish scripting languages is whether they are *event driven* or *data driven*. In the case of an event driven language, it is important to know what kind of call-back mechanisms it supports (e.g. the concept of event listeners in Java) and how closures [45] can be specified.

Concurrency. Some scripting languages are inherently concurrent (e.g. Bourne Shell or Piccola [2]), while others provide abstractions for concurrency (e.g. threads, monitors). In both cases, the kind of built-in *coordination abstractions* are of interest.

Introspection. Scripting languages generally offer features for run-time introspection or even reflection, although these features often only have a limited functionality. From our point of view, both dynamic creation and execution of code (often referred to as an *eval*-feature) and the concept of *call-by-name* are part of this dimension. Whereas languages like Tcl only offer low-level introspection mechanisms, Python goes a step further and offers a meta-level protocol.

Typing. According to Ousterhout [36], scripting languages tend to be weakly typed. However, an analysis of popular languages reveals differences in the type system: some languages are untyped (e.g. Bourne Shell) or dynamically typed (e.g. Perl) whereas others have a mixture of static and dynamic typing (e.g. Visual Basic). This analysis also revealed different strategies for resolving type mismatches (e.g. implicit type conversions vs. exceptions).

Scoping rules. The scope of a name (variable, function etc.) is the range of program instructions for which the name is known. The scoping rules of a language defines the strategy how name-value bindings are established. Most scripting languages tend to be *dynamically scoped* (e.g. Python), although there are languages which also offer *static scoping* (e.g. Visual Basic).

Built-in data abstractions. Besides low-level data abstractions such as integers and strings, many scripting languages offer built-in *higher-level data abstractions*. Examples of such abstractions are key-based data abstractions (e.g. dictionaries), ordered data abstractions (e.g. lists), or data abstractions without a particular order or access strategy (e.g. sets).

In addition, many languages offer specialized operations on high-level data abstractions (such as iterations), and Perl even has a special syntax for these operations.

Persistence. Only few scripting languages (e.g. AppleScript) offer general-purpose support for making complex configurations or properties of applications and components persistent.

6.3.3 Selected systems and languages

In the following, we will briefly illustrate important concepts and features found in selected scripting languages.

Bourne Shell is an interpreted scripting language for the UNIX operating system and offers a simple component model based on commands and character streams. Commands can be connected by using higher-level connectors (e.g. the pipe operator '|'), which makes the architecture of a Bourne Shell script explicit in the source code. The language is compositionally complete (i.e. a composition of commands is again a command) and supports a declarative style of programming.

Tcl is a dynamically compiled, string-based scripting language and is available on all popular platforms. The basic abstraction in Tcl is a command (comparable to a procedure in an imperative programming language), and since every programming construct is achieved with commands (and not special syntax), commands are the unifying concept of the language. The concept of commands allows a user to extend the language using the same syntactical framework as is used for all built-in commands.

Perl can be considered as a uniform selected merge of sed, awk, csh, and C. It offers higher-level data abstractions such as lists, arrays, and hashes and syntactic sugar for processing instances of these higher-level data abstractions. Perl introduces the notion of contexts for evaluating expressions, offers support for operator overloading based on contexts, and has both lexical and dynamic scoping rules.

Table 6.1. Functional properties of selected scripting languages

Language	Domain and/or Paradigm	Extensible	Embeddable	Reflection support
Bourne Shell	administration, commands	any	no	no
Tcl	GUI, commands	C, Java	yes	yes
Perl	text-processing, object-oriented	C/C++	yes	yes
Python	object-oriented	C/C++	yes	yes
AppleScript	object-oriented, events	any	yes	no
JavaScript	object-based, events	Java	yes	yes
Visual Basic	object-based, events	C/C++	yes	no
Haskell	functional	COM	no	no
Manifold	coordination, process-based	C, (Java)	yes	no
Piccola	process-based, object-based	Java	no	yes

Python is an object-oriented scripting languages that supports both scripting and programming in the large. Objects are the unifying concept (i.e. “everything is an object”) and, therefore, all abstractions are first-class values. Python offers a meta-level protocol which can be used for extending and adapting existing abstractions as well as for operator overloading. Finally, the language model supports keyword-based parameter passing.

AppleScript is a dynamically typed, event- and object-oriented scripting language which only runs on the MacOS platform. In fact, AppleScript is not a scripting language on its own, but it should be considered as a front-end to a framework based on scriptable applications (also known as component parts). The concepts of AppleScript are heavily based on similar concepts defined in the Open Scripting Architecture (OSA) for OpenDoc [14]. The main purpose of AppleScript is to automate, integrate, and customize scriptable applications. The language is compositionally complete, but in contrast to many other scripting languages, it does not offer an equivalent to an “eval” feature (i.e. it is not directly possible to create and execute scripts at runtime).

AppleScript comes with an application called *Script Editor* which can be used to create and modify scripts. Although this editor has access to the dictionary of scriptable applications (i.e. the set of messages a scriptable applications supports), to our knowledge AppleScript scripts cannot introspect these dictionaries (refer also to Table 6.1).

JavaScript is a (general-purpose) object-based scripting language embedded into a web browser [15]. The main predefined components in JavaScript are windows, forms, images, input areas, and menus. In general, JavaScript is used to control the browser and web documents. JavaScript scripts are attached to events and executed when the corresponding event occurs. JavaScript enables the interaction of the user with a web-document by providing means to read and write content of document elements. However, JavaScript does not provide any graphic facilities, network operations other than URL loading, or multithreading.

Visual Basic is a visual programming environment for object- and component-based application development, focusing on wiring components. In particular, in Visual Basic one defines the wiring-code (e.g. event handling) while objects and components are usually developed in a language like C, C++, or Java. Visual Basic programs can be compiled to native code. However, components and the runtime system are packaged into separate run-time libraries (DLL's). Visual Basic provides a static typing scheme for variables and a dynamic typing scheme for components. Furthermore, Visual Basic supports keyword-based parameters (enabled by the `IDispatch` interface of COM-components). At present, Visual Basic is only available on Windows operating systems.

Haskell is a pure functional programming language that provides a COM binding [39]. Haskell is not really a scripting language, but due to its features like a polymorphic type system, higher order functions, lazy evaluation, or convenient syntax it is an attractive language for scripting components. The COM integration into Haskell is strongly typed. Haskell is an interpreted language and the Haskell system provides garbage-collection. Haskell provides an unconventional and new way for scripting.

Manifold is a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, and cooperating processes [5]. It should be considered as a scripting language for concurrent and distributed components. It is particularly suitable for specifying and implementing reusable, higher-level coordination abstractions and protocols as well as for dynamically evolving architectures.

Piccola is a simple untyped language and has been designed to be a general purpose “composition language” [2, 3]. Piccola is primarily used to express how components are composed, i.e. it is used to define the connectors, coordination and glue abstractions needed for an actual composition. Piccola uses the unifying concept of *forms* (immutable, extensible records), which represent almost everything in Piccola, including *namespaces*, *interfaces*, *parameters*, *scripts*, and *objects*. This unification results in an extremely simple but expressive language. Since Piccola is

Table 6.2. Non-functional properties of selected scripting languages

Language	Platform portability	Implementation technique	Module concept	Application area
Bourne Shell	UNIX	Interpreter	(yes)	adm. tasks
Tcl	major	Bytecode	yes	GUI
Perl	major	Bytecode	yes	Text
Python	major	Bytecode	yes	adm. tasks
AppleScript	MacOS	Interpreter	no	configuration customization
JavaScript	Netscape	Interpreter	(yes)	WWW
Visual Basic	Windows	Bytecode	yes	GUI/COM
Haskell	major	Bytecode	yes	COM
Manifold	UNIX	Compiled/PVM	yes	coordination
Piccola	major	Virtual machine	(yes)	composition coordination

based on a formal process semantics [25, 42], it is a prime candidate for coordination and configuration of internet-based agents.

The reader should note that other scripting languages (e.g. DCL [4], Icon [20], Lua [22], Obliq [8], Rapide [24], or Rexx [12] also support some of the features that we have illustrated in this section, but a detailed discussion is beyond the scope of this work.

6.4 Scripting in practice

In the previous section, we have discussed the main properties of scripting languages and compared selected scripting languages using these properties. In this section, we show how scripting languages are used in distributed applications where coordination concerns are of importance. More precisely, we illustrate CyberChair [49], an on-line submission and reviewing system for scientific conferences, a case study carried out in the context of the FAMOOS project,² where CORBA [34] is used as a medium to connect heterogeneous components [21], and the implementation of a Wiki server in Piccola [2].

Common to first two sample applications is that Python [50] is used as the underlying scripting language. These examples illustrate the application of a data-driven (i.e. CyberChair) and a control-driven coordination model (i.e. scripting of CORBA components) in Python, respectively. The Wiki server shows how user-defined operators are used to make the architecture of an application explicit.

² FAMOOS was an industrial ESPRIT Project (No 21975) in the IT Programme of the Fourth ESPRIT Framework Programme on reengineering object-oriented legacy systems towards component-based frameworks.

In this section, we will only focus on selected aspects related to scripting and coordination; a detailed description of the three sample applications is beyond the scope of this work. We forward the interested reader to the corresponding references for details.

6.4.1 CyberChair: a conference management system

Most scientific communities have established policies and mechanisms aiming at minimizing the organizational efforts of conference management, while keeping a high quality of accepted papers and a fair selection process. Generally, authors interested in presenting their work at a conference submit their papers to the program chair. Under the guidance of the program chair, a program committee (i) reviews all submitted papers and (ii) selects the papers to be accepted. Finally, the authors are notified about the evaluation process, and authors of accepted papers are invited to submit a camera-ready copy. The whole process of submission, reviewing, selection, and notification requires many activities, and the corresponding workload can be substantially reduced by the aid of an appropriate software system.

An example of such a software system is CyberChair, which was first developed in 1996 at the University of Twente in order to offer automatic conference management support for ECOOP, the annual European Conference on Object-Oriented Programming. In the past years, CyberChair has evolved and matured, covers most administrative tasks of the submission and review process, and has been successfully adapted for about ten different conferences.

The activities supported by CyberChair roughly correspond to the activities of a series of related case studies used to evaluate various coordination models and languages in real world problems (refer to [41, 44] for details). A detailed description of all activities has therefore been omitted here.

The overall structure of CyberChair conforms to a client-server architecture organized around a central data repository, where all the data of the submission process is stored. In order to enhance the access to the data repository and to reduce conflicting accesses as much as possible, the information for each paper submission, the status of each paper (accepted, rejected, withdrawn etc.), the distribution of papers to reviewers, and each review report are stored in separate files on the server system.

For each of the management task supported by CyberChair, there exist an agent which implements the corresponding functionality. Most of these agents are simple CGI-programs (in particular those agents which directly communicate with users) while other agents run as batch processes and periodically update HTML pages used to access specific services.

To transparently access the data repository and to encapsulate the concrete structure from client agents, CyberChair offers a set of basic Python routines which are used to create, modify, and access the data repository (i.e. the access functionality is encapsulated into a single Python module).

The agent for submitting abstracts, for example, is a CGI-program and can be activated using a simple web-browser (the corresponding URL is available from the main web-page of the conference). This agent creates an HTML form for submitting information about the authors, their affiliations, the title, and the abstract of a paper. The information submitted using this form is checked by another agent, which (i) stores the necessary information in the data repository and (ii) notifies via email both the contact author of the paper and the program chair about the submission. This email message also contains an identification and an URL which are needed to submit the electronic version of the paper in a later stage. The identification for each submission is again stored in the data repository.

In order to fulfill the security policy imposed by the conference management, access to restricted services (e.g. services only available to members of the program committee or the program chair) is given by password-enabled agents or password-protected web-pages. Note that these web-pages are periodically updated by the system in order to reflect changes in the data repository (e.g. all review reports of a paper are made available to those reviewers who review the same paper).

From a coordination point of view, there are three main concerns which must be considered by a conference management system like CyberChair: (i) concurrency concerns (i.e. ensure correct transaction protocols), (ii) security concerns (i.e. granting and revoking access rights to agents and/or data elements), and (iii) dependency concerns (i.e. tasks depend on the termination of other tasks). In the following, we will discuss each of these concerns in further detail.

Concurrency concerns. Generally, web-servers allow several CGI-programs to run concurrently. Hence, it might be possible that more than one active agent wants to update data stored in the same file (e.g. all identification keys for the authors are stored in a single file). At the level of accessing the files representing the persistent data repository, a two-phase locking protocol (based on POSIX file locks) prevents concurrent modification of files and ensures correct transaction protocols. Other access protocols (such as immutable data entries) are ensured by checking the presence or absence of the corresponding files.

Security concerns. As mentioned above, the access to several services is restricted to a particular set of users only. In order to achieve a fine-grained security policy, password-enabled web-pages are created which contain the access points (i.e. hypertext links) to dedicated services (agents or other web-pages). The access points given in the password-enabled web-pages use an identification mechanism to prevent users from “guessing” the access points to other restricted services. The system periodically updates the contents of the access pages in order to allow for an up-to-date view of the data repository (e.g. additional hypertext links are created when new review reports have

been submitted). Finally, the access of particularly critical services is logged by a security agent.

Dependency concerns. It is obvious that some management tasks depend on the termination of other tasks. The coordination of these dependencies can be easily achieved by (i) defining deadlines for the critical tasks, (ii) preventing access to the corresponding services once the deadline has exceeded, and (iii) only granting access to the dependent tasks from this moment on (e.g. access to all abstracts is only granted when the submission phase has been completed, electronic versions of papers can only be downloaded when all papers have been submitted). In CyberChair, this concept is implemented using simple boolean flags in the corresponding Python scripts.

6.4.2 Scripting CORBA components

The second sample application we will discuss was conducted in the context of an industrial ESPRIT project our research group participated in. The goal of this project (called FAMOOS) was to support the evolution of first generation object-oriented software, built with current analysis and design methods and programming languages, to frameworks – standard application architectures and component libraries which support the construction of numerous system variants in specific domains. Methods and tools were developed to analyse and detect design problems with respect to flexibility in object-oriented legacy systems and to transform these systems efficiently into frameworks based on flexible architectures.

Each of the FAMOOS project partners was interested in different aspects of reengineering and developed their own tools to conduct experiments in their field of interest. These tools were implemented in a heterogeneous environment using various programming languages (C/C++, Smalltalk, Java, and Python). Hence, there was a need to find a way to easily combine these tools in order to perform experiments where several tools were involved.

As a first step towards an integration of these tools, a common data exchange model was defined which represented object-oriented source code in a language-independent format [13]. Using such a language-independent exchange format allowed developers to implement their tools with a much broader functionality (i.e. metric tools could handle source code written in any object-oriented language). Furthermore, the character-based CDIF format the exchange model was based upon reduced the need for data marshaling between different operating system platforms and enhanced the interoperability between the tools.

All tools were either written in a language where a CORBA binding was defined or had an API in such a language. Hence, it was a natural decision to convert all tools into CORBA components, i.e. to define a wrapper for each tool with a CORBA compatible interface. These wrappers were defined

using standard wrapping technology and, therefore, details of the wrapping process have been omitted here (refer to [21] for details).

However, wrapping the tools as CORBA components was not enough as languages like C++ or Java do not offer abstractions to wire CORBA components in a flexible and extensible way. Hence, there was the need for an environment fulfilling this requirement.

Fnorb is an experimental CORBA ORB [10] which allows users to implement and compose CORBA components using Python. Of particular interest in the context of the FAMOOS project was the second property as it made Fnorb an ideal tool for experimenting with CORBA architectures, for scripting CORBA components, and for building test harnesses for CORBA development projects.

In order to enhance the transparent scripting of CORBA components, it was necessary to encapsulate CORBA-related protocols as much as possible. An approach to achieve this goal was to represent CORBA components in the scripting environment using proxy objects [17]. Hence, an application programmer was freed from directly referring to the components themselves, but could instead refer to local representatives (i.e. the proxy objects), being responsible for (i) instantiating the (possibly remote) CORBA components and (ii) coordinating the communication between the components and their clients. Proxy objects either forward requests to the actual components or, in case of “intelligent” proxies [21, 29], handle request themselves.

It is important to note that an implementation of component proxies in Python substantially benefits from the dynamic typing and the underlying metalevel protocol, in particular the way attribute accesses in objects are performed (refer to [42, 50] for details). As a consequence, it was possible to implement a single generic proxy class (and not a new class for each component). This proxy class heavily uses the *dynamic invocation interface* (DII) of CORBA for a correct handling of requests.

Using a middleware environment such as CORBA for interconnecting heterogeneous components has the advantage that several “lower-level” coordination aspects (such as intercomponent communication and synchronization, data marshaling etc.) are directly handled by the involved ORBs. At a higher level, the usage of generic component proxies enables a complete abstraction from the underlying middleware and application programmers are freed from CORBA-related protocols. If further coordination concerns have to be taken care of, it is possible to define specialized component proxies and implement these concerns in the corresponding proxies. As a result, all coordination concerns can be encapsulated into independent units of deployment and computational entities are freed from coordination code.

The efforts to (i) wrap tools as CORBA components and (ii) integrate them into a scripting environment resulted in the desired framework for conducting various reengineering experiments. The approach taken also reflects the elements of the conceptual framework given in section 6.2, in particular

the usage of generic glue abstractions for interoperation with components not being part of a scripting environment.

6.4.3 Observations

Although the two sample applications do not cover all aspects of open systems development, they, nevertheless, indicate some benefits and limitations of scripting languages for coordinating distributed software agents.

The two sample applications we have given in this section show that Python is a suitable scripting language for both a control- and data-driven coordination models. It has enough expressiveness to plug existing components together, to implement the missing functionality where needed, and to encapsulate (low-level) coordination concerns into independent units of deployment.

Due to dynamic scoping rules and the lack of static typing, many scripting languages (including Python) ease the implementation of generic glue abstractions. In particular the lack of static typing makes it easier to interoperate with components written in other languages and, therefore, encourages the use of legacy components.

Application development using scripting languages naturally leads to the maxim of viewing applications as combinations of components and scripts and, therefore, a scripting approach encourages the development of reusable components highly focused on the solution of particular problems, and the assembly of these components by means of scripts.

However, scripting languages also have their limitations. Consider the fact that the overall architecture of both sample applications given above is not made explicit in the underlying source code. It is important to note this observation can be made in many applications where scripting languages are used to wire components together.

Furthermore, scripting languages typically support a single, specific architectural style, which makes them ideal tools for solving problems where this style is appropriate. However, as soon as a different style is required, most scripting languages fall short of giving the required flexibility to adapt to the new style (refer also to the discussion about the Perl Wiki given in the next section).

Only a limited number of scripting languages offers built-in support for concurrency; most languages only provide concurrency by libraries. Therefore, it is not possible to express concurrency issues directly in the source code. Finally, scripting languages are suitable to build small applications, but only offer limited support (if at all) for programming in the large. Interested readers may find further details concerning scalability issues in chapter 12 (Coordination and Scalability).

6.4.4 A Piccola Wiki

In the previous section we have illustrated why scripting languages alone do not allow for flexible composition, adaptation, and configuration of existing components and to explicitly represent higher-level design elements (such as software architectures or coordination concerns) in applications. What we would need is something we call a *composition language* that (i) supports application configuration through a structured, but nevertheless flexible wiring technology and (ii) enforces a clear separation between computational elements and their relationships. In the following, we will briefly illustrate why Piccola can be considered as a step into this direction.

A Wiki Wiki Web Server (Wiki for short) is a simple hypertext system that lets users both navigate and modify pages through the World-Wide-Web, and was originally 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 internal links, bulleted lists, emphasized text etc. and are dynamically translated to HTML by the Wiki server. Generally, a Wiki is used as a medium to collaborate on documents and information webs.

Although the original Perl implementation is perfectly robust and widely used for many Wiki installations around the world, the style of programming that Perl encourages poses some problems for extensibility. In the available Perl implementation, it is not easy to understand the flow of control as the procedural paradigm of Perl is mixed with the stream-based processing of the web pages. Execution is sensitive to the sequence in which the declarations are evaluated. In particular, the architecture of the scripts are not evident, which makes it hard to adapt them to new functionality.

The Piccola Wiki implementation illustrates how the architecture of a scripted application can be made explicit by means of a data-flow architectural style (refer to [2] for details). More precisely, it shows that (i) a clear separation between components and their connections is enforced and (ii) glue abstractions adapt components that are not part of the underlying component framework. In particular, the Piccola Wiki implementation

- illustrates the implementation of an object-oriented (white-box) framework incorporating *streams*, *transformers*, and *files* that corresponds to a pull-flow stream-based architectural style [6]. Note that Java streams are integrated into this framework by means of gateway agents.
- substantially benefits from the fact that (i) the language offers user-defined operators which allow us to use a syntax that highlights the architectural style of the component framework and (ii) users can create new transformer and stream components without subclassing framework classes (i.e. the component framework supports *black-box extension*).
- integrates components of a push-flow architectural style (i.e. components which *push* data downstream instead of *pulling* it from upstream). Glue

components are used to adapt push-flow components so they can work within a pull-flow architecture.

- defines a top-level script that implements the Wiki by composing components that conform to the architectural style mentioned above.

Maintaining a system such as the Piccola Wiki is much easier than the original Perl implementation and substantially benefits from an explicit architecture. Changing requirements, for example, may be addressed by reconfiguring individual components (i.e. replacing their required services), reconfiguring interconnections between components (i.e. adapting the scripts), incorporating additional external components (i.e. using glue abstractions), and deriving new components from old ones (i.e. support for black-box extension).

6.5 Summary, conclusions

Although scripting languages and coordination languages have evolved from different roots and have been developed to solve different problems, we argue that there is a strong affinity between them. In particular, scripting and coordination address similar *separation of concerns*. Scripting languages (ideally) achieve a separation of *components* from the *scripts* that configure and compose them, and coordination languages separate *computational entities* from the *coordination* code that manages dependencies between them.

This affinity, we believe, will become only more important in the coming years as pervasive computing is becoming a reality, and agents representing hardware devices, internet services, and human clients will have to spontaneously interact and coordinate with other agents as a matter of routine. Building such flexible and dynamically evolving agent systems, on the other hand, will not be a trivial task, and we believe that scripting languages offer the only paradigm that has any hope of meeting the challenge.

If we examine the state of the art of today's scripting languages, however, we see that, although they offer many useful concepts and mechanisms for high-level programming, they are still a long way from fulfilling the coordination needs of tomorrow's agent systems. In particular, we see the need for improvement in the following areas:

- *Abstraction*: most scripting languages offer relatively weak abstraction mechanisms. To implement high-level abstractions, one is often forced to resort to techniques like generating code on-the-fly. As a consequence, higher-level coordination abstractions are cumbersome to implement in most scripting languages, if they are not already built into the language.
- *Software architecture*: the compositional (or "architectural") style supported by any given scripting language is usually fixed in advance. A truly general purpose scripting language would support the definition of new kinds of composition mechanisms for different application domains.

- *Concurrency*: concurrency, if supported at all, is provided by libraries, and is not explicitly supported by the language (both the Bourne Shell and Manifold are notable exceptions).
- *Coordination styles*: the fact that many different coordination models and languages have recently been proposed suggests that there does not exist a single approach suitable for all coordination problems. Similar to architectural styles, a catalogue of coordination styles that exhibit properties more suitable for some problems than others could be expressed in terms of components, connectors, and composition rules.
- *Reasoning*: scripting languages, as a rule, have no formal semantics, making it hard to reason about scripts. Since the behaviour that is coordinated by scripts is provided by components typically written in a separate programming language, a formal semantics for the scripting language is not enough to reason about overall behaviour, but it is certainly a prerequisite for making any progress.

In the Software Composition Group, we are attempting to address several of these problems with Piccola, a small language with a formal process calculus semantics, that is intended to be used as a general-purpose “composition language” for different application domains. In particular, we are currently working on formalizing and implementing various coordination styles based on the idea of a component algebra [1].

Acknowledgements

We thank all members of the Software Composition Group for their support of this work, and Richard van de Stadt for helpful comments on CyberChair. This research was supported by the Swiss National Science Foundation under Project No. 20-53711.98, “A framework approach to composing heterogeneous applications.”

References

1. F. Achermann, S. Kneubühl, and O. Nierstrasz. Scripting Coordination Styles. In A. Porto and G.-C. Roman, editors, *Coordination Languages and Models*, LNCS 1906. Springer, Sept. 2000.
2. F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. Piccola – a small composition language. In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing: A Survey of Object-Oriented Approaches*, chapter 18, pages 403–426. Cambridge University Press, Dec. 2001.
3. F. Achermann and O. Nierstrasz. Application = Components + Scripts – A tour of Piccola. In M. Aksit, editor, *Software Architectures and Component Technology*. Kluwer Academic Press, 2001.
4. P. C. Anagnostopoulos. *Writing Real Programs in DCL*. Digital Press, 1989.
5. F. Arbab. *Manifold Reference Manual*. Department of Software Engineering, CWI, Amsterdam, NL, June 1998.
6. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.
7. S. Bourne. An Introduction to the UNIX Shell. *Bell System Technical Journal*, 57(6):1971–1990, July 1978.
8. L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.
9. N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, Apr. 1989.
10. M. Chilvers. *Fnorb User Guide*. CRC for Distributed Systems Technology, University of Queensland, AU, Apr. 1999.
11. A. Computer. *AppleScript Language Guide*. Apple Technical Library. Addison-Wesley, 1993.
12. M. F. Cowlshaw. *The REXX Language: A practical Approach to Programming*. Prentice Hall, 2nd edition, 1990.
13. S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - The FAMOOS Information Exchange Model. Technical report, University of Bern, Institute of Computer Science and Applied Mathematics, Aug. 1999.
14. J. Feiler and A. Meadow. *Essential OpenDoc*. Addison-Wesley, 1996.
15. D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, 2nd edition, Jan. 1997.
16. J.-L. Fontaine. Simple Tcl Only Object Oriented Programming. Available at <http://www.multimania.com/jfontain/stoop.htm>, 1998.
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
18. D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 12(6):17–26, Nov. 1995.
19. D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, Feb. 1992.

20. R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, Dec. 1996.
21. M. Held. Scripting für CORBA. Master's thesis, University of Bern, Institute of Computer Science and Applied Mathematics, Apr. 1999.
22. R. Ierusalimschy, L. H. de Figueiredo, and W. Celes Filho. Lua – an Extensible Extension Language. *Software: Practice and Experience*, 26(6):635–652, 1996.
23. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, LNCS 1241, pages 220–242. Springer, June 1997.
24. D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, Apr. 1995.
25. M. Lumpe. *A π -Calculus Based Approach to Software Composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, Jan. 1999.
26. M. Lumpe, J.-G. Schneider, O. Nierstrasz, and F. Achermann. Towards a formal composition language. In G. T. Leavens and M. Sitaraman, editors, *Proceedings of ESEC '97 Workshop on Foundations of Component-Based Systems*, pages 178–187, Zurich, Sept. 1997.
27. Microsoft Corporation. *Visual Basic Programmierhandbuch*, 1997.
28. D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF, the OMG mobile agent system interoperability facility. In *Proceedings of Mobile Agents '98*, 1998.
29. T. J. Mowbray and R. C. Malveau. *CORBA Design Patterns*. Wiley, 1997.
30. D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.
31. O. Nierstrasz and L. Dami. Component-Oriented Software Technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.
32. O. Nierstrasz, J.-G. Schneider, and F. Achermann. Agents Everywhere, All the Time. In J. Bosch, C. Szyperski, and W. Weck, editors, *Proceedings of the Fifth International Workshop on Component-Oriented Programming (WCOP 2000)*, pages 97–100. Blekinge Institute of Technology, Oct. 2000.
33. O. Nierstrasz, D. Tsichritzis, V. d. Mey, and M. Stadelman. Objects + Scripts = Applications. In *Proceedings Esprit 1991 Conference*, pages 534–552, Dordrecht, NL, 1991. Kluwer Academic Publisher.
34. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1996.
35. J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
36. J. K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, 31(3):23–30, Mar. 1998.
37. G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press, Aug. 1998.
38. D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, Oct. 1992.
39. S. Peyton Jones, E. Meijer, and D. Leijen. Scripting COM Components in Haskell. In *Proceedings of the Fifth International Conference on Software Reuse*, Victoria, British Columbia, June 1998.
40. W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

41. D. Rossi and F. Vitali. Internet-Based Coordination Environments and Document-Based Applications: A Case Study. In P. Ciancarini and A. L. Wolf, editors, *Coordination Languages and Models*, LNCS 1594, pages 259–274. Springer, Apr. 1999. Proceedings of Coordination '99.
42. J.-G. Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, Oct. 1999.
43. J.-G. Schneider and O. Nierstrasz. Components, Scripts and Glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures – Advances and Applications*, chapter 2, pages 13–25. Springer, 1999.
44. A. Scutellà. Simulation of Conference Management Using an Event-Driven Coordination Language. In P. Ciancarini and A. L. Wolf, editors, *Coordination Languages and Models*, LNCS 1594, pages 243–258. Springer, Apr. 1999. Proceedings of Coordination '99.
45. R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1989.
46. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Apr. 1996.
47. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
48. S. Tichelaar. A Coordination Component Framework for Open Distributed Systems. Master's thesis, University of Bern, Institute of Computer Science and Applied Mathematics, May 1997.
49. R. van de Stadt. CyberChair: An Online Submission and Reviewing System for Conference Papers. Available at <http://trese.cs.utwente.nl/CyberChair/>, 1997.
50. G. van Rossum. Python Reference Manual. Technical report, Corporation for National Research Initiatives (CNRI), Oct. 1996.
51. L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, 2nd edition, Sept. 1996.