Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica en Toegepaste Informatica

Software Composition through Linguistic Symbiosis

Dissertation submitted in partial fulfillment of the requirements for the degree of Licenciaat in de Informatica.

Lieven De Keyzer

Promotor: Prof. Dr. Theo D'Hondt

Advisors: Dr. Wolfgang De Meuter & Kris Gybels

August 2006

Samenvatting

Het is moeilijk om grote, complexe softwaresystemen te ontwikkelen en nog moeilijker om ze te onderhouden. In component-gebaseerde applicatie-ontwikkeling, worden applicaties gecreëerd door een aantal reeds bestaande componenten te hergebruiken. Deze componenten worden aangepast en samengeplugd in een hogere-orde *scripting* taal. Deze taal moet een manier voorzien om componenten geschreven in andere talen te kunnen gebruiken.

Dit probleem lijkt erg op het definieren van een taalsymbiose tussen twee programmeertalen, zodat de talen op een transparante manier data kunnen uitwisselen en functionaliteit geschreven in de andere taal kunnen uitvoeren.

We beschouwen het gebruik van zo een taalsymbiose als manier om een *scripting* taal componenten geschreven in andere talen te laten gebruiken. We vergelijken deze aanpak dan met de aanpak van Piccola, een taal die specifiek werd ontwikkeld om componenten aan te passen en samen te pluggen.

We stellen ook een manier voor om een onderscheid te maken tussen aanpassingen aan componenten die enkel een interface *mapping* uitvoeren en aanpassingen die nieuwe functionaliteit aan een component toevoegen.

Abstract

Large and complex software systems are hard to build and even harder to maintain. In component-based development, applications are built by reusing a number of already existing components. These components are adapted and composed in a high-level scripting language. To be able to also use components written in other languages, the scripting language should provide a way for exchanging data with other languages.

This problem is very similar to the one of defining a symbiotic relationship between two languages, such that the languages can transparently exchange data and invoke each other's behaviour.

We contemplate the use of linguistic symbiosis for a scripting language to be able to access components written in another language. This is contrasted with inter-language bridging, the approach used by Piccola, a language specifically designed for adapting components and expressing compositions.

An approach to discriminate between adaptation code that purely maps interfaces and adaptation code that adds new functionality to a component is introduced.

Acknowledgments

The completion of this thesis would not have been possible without the help of many people. Therefore, I would like to express my gratitude towards:

Prof. Dr. Theo D'Hondt for promoting this thesis.

Dr. Wolfgang De Meuter and Kris Gybels for being excellent advisors. Apart from coming up with the subject, they also kept me on the right track through every stage of this work, from preparation to proofreading.

Jan Meskens for linguistic proofreading of this document.

All members of the Programming Technology Lab for their valuable comments during the thesis presentations.

My friends and fellow students for their support and for distracting me once in a while when writing this thesis.

The *Vrije Universiteit Brussel* and *Departement Informatica* for providing an excellent education.

My parents for supporting me and giving me the opportunity to study in the best possible circumstances.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Building large, complex software systems is a hard and time-consuming task. The best way to increase productivity when constructing such software, is to reuse already existing software. There exist various techniques to put this idea of software reuse into practice, one of the more dominant ones being component-based development. In this approach, the existence of a large collection of reusable software components is assumed. It will typically not be possible to reuse such a component "as is"; adaptation of the component, by means of glue code, will be necessary to make it compatible with other selected components. Thus, building applications becomes a matter of selecting, adapting and composing components.

Specifying how components are to be composed is done using scripting languages because they allow for fast development, are dynamic and provide high-level abstractions. Most of these languages, however, only allow composition according to a single compositional style. Consider Unix shell languages, for example, which are based on a pipes and filters approach.

The Piccola programming language [AN01, ALSN01] is designed especially to adapt components and express compositions. It does not constrain these components to adhere to a certain compositional style, but rather allows the programmer to specify different styles of composition. As such, Piccola is said to be a general-purpose composition language.

The architecture of Piccola is based on a formal semantic model, called the $\pi\mathcal{L}$-calculus [Lum99]. This is an extension of the π-calculus, which was introduced by Milner et al. [MPW92] to describe concurrent computational processes. Therefore, it is possible to reason about component compositions specified in Piccola.

When adapting and composing components, we do not want to limit the available components to those written in the scripting language itself. The language should provide abstractions that allow accessing components written in other languages.

This problem is very similar to the one of defining a symbiotic relationship between two languages [IMY92]. Such languages can transparently exchange data and invoke

each other's behaviour. Gybels et al. [GWDD06] introduce a conceptual model that allows implementing linguistic symbiosis between two languages.

Piccola does indeed allow components to be written in a separate implementation language. It employs an approach, called bridging by Schärli [Sch01], to pass data between Piccola and other languages.

1.1 Contributions

We contemplate the use of linguistic symbiosis for a scripting language to be able to access components written in other languages We contrast this with the approach used by Piccola to pass data from another language to Piccola, and vice versa, to pass data from Piccola to another language.

Consequently, we identify some problems in how external components are accessed when using Piccola as a composition language. We solve these problems by using Sic% [Gyb04], a language that implements the conceptual symbiosis model, and as such is in linguistic symbiosis with Smalltalk. This means we define the properties a composition language should possess for manipulating external components.

We also introduce an approach to discriminate between adaptation code that purely maps interfaces and adaptation code that adds new functionality to a component.

1.2 Dissertation Roadmap

In chapter 2, we first give an outline of how component-based development originated. Then some terminology and common techniques are discussed. Finally, a conceptual framework for composition [Sch99] is presented. The Piccola programming language is designed with this framework in mind, and an overview of its syntax and semantics is given in chapter 3. We also discuss the approach that Piccola implementations should use for accessing components written in a different language. Chapter 4 introduces the Pico programming language [MGD99] and its object-oriented variation Pic% [DM03]. This leads to the presentation of a specific implementation of Pic% called Sic%, which engages in a symbiotic relationship with Smalltalk. An high-level overview of this feature's implementation and the conceptual model behind it finishes the chapter. In chapter 5, Piccola's bridging strategy is evaluated against the linguistic symbiosis model, and we present an approach to separate interface mapping adaptations and behavioural adaptations. We conclude this dissertation with some final remarks in chapter 6.

Chapter 2

Component-Based Software Development

In this chapter we describe what is meant by component-based software development. The origin of the field is sketched in section 2.1. What is meant exactly by the term component is explained in section 2.2. Components are adapted and composed by means of respectively glue code and scripts, which are discussed in sections 2.4 and 2.3. Section 2.5 presents a conceptual framework for composition.

2.1 Origin

An ideal way of developing software, would be for the developer to identify some needed code modules, what order they should be executed and what information should be passed between them. Freshly written modules would then be combined with existing ones to form new applications. A similar way of working can be seen in other fields, like electronics engineering. Hardware components are the smallest units making up an electrical circuit. They are both interchangeable and reliable.

This kind of component programming is not a recent notion. A proposed solution for the software crisis in the late sixties was called *component-oriented software construction*. It introduced the idea that software should be built from prefabricated components, which are black-box entities [McI68]. The goal was to reuse these components for different applications to lower development costs and to establish a market for software components. This vision could not be established at the time. Some of the reasons being the ideas that components should be built system independently or that a component catalogue must be available, allowing application developers to choose the right component for a specific problem [Sch99].

2.2 Components and Frameworks

Today, component-oriented programming still receives much attention from both the industrial and research point of view. Nevertheless, it is not always clearly understood what is exactly meant by the term *software component*. Schneider [Sch99] claims it is closely related to the term *component framework* and the two cannot be defined in isolation. Indeed, Lumpe et al. [LSNA97] define a component this way:

> A software component is a *composable element of a component framework*.

Although this seems to be a circular definition, Schneider claims it does a very good job capturing the crucial properties of components: components are designed to work together with other components. A component that is not part of a component framework is a contradiction in terms. Furthermore, a component cannot function outside a well-defined framework. We will see a formal definition of the term component framework later in this section.

Just as is the case in electronics engineering, a component could function by itself, but it is of much more value when combined with other components. The whole intention of designing components is to plug them together, so we could say:

> A software component is a *static abstraction with plugs*. [ND95].

By *static*, it is meant that a component is a long-lived (i.e. stable) entity that can be stored in a software base, independently of the applications it has been used in before. It is an *abstraction*, because it puts a more or less opaque boundary around the software it encapsulates and provides some priorly known functionality. Finally, a component has *plugs*, which are not only used to provide services but also to require them. All features or dependencies of a component are exposed by means of such public plugs; there exist no hidden dependencies. Plugs are the most important prerequisite for composition: required services of a component are connected with suitable provided services of another component. The nature of the interface, and how these interfaces may be plugged together will differ from one component framework to another.

Some other aspects of components are given by Szyperski [Szy02] through the following definition:

> A software component is a unit of composition with contextually specified *interfaces* and explicit context *dependencies* only. A software component can be *deployed independently*, is subject to *third-party composition*, and has *no persistent state*.

We have already mentioned that for a component to be composable (by third-parties), it must explicitly specify the services it provides (i.e. interfaces) but also the services it requires (i.e. dependencies). This means the component has to be sufficiently *self-contained*. For a component to be independently deployable, it needs to be well separated from its environment and other components. Therefore, a component *encapsulates* its features and can never by deployed partially. Thus, a component needs to

encapsulate its implementation and interact with its environment through well-defined interfaces. The last statement blurs the distinction between stateless component factories and stateful component instances. As such it might cause some confusion since components, like buttons or windows, usually do have state. We are thus not working with the abstract component classes, but with their instances. It's therefore useful to distinguish between design-time and runtime of a component [Ach02]. At design-time, the developer chooses the properties that are set when the component is instantiated at runtime.

We have used the term component framework a few times without properly defining it, although we have seen that the definition of a component and the notion of a component framework are closely related to each other. Schneider [Sch99] gives the following definition:

> A component framework is a *collection* of software components with a *software architecture* that determines the interfaces that components may have and the rules governing their composition.

In an object-oriented language, a realization of a framework might be an abstract class hierarchy, but there is no reason for components to be classes or for frameworks to be abstract class hierarchies. In such an object-oriented framework, an application is generally built from subclassing framework classes that adhere to certain application requirements, a component framework on the other hand focuses on object and class composition (i.e. black-box reuse).

As is also the case for components, there exist more than one definition of the term component framework. Szyperski [Szy02] describes a component framework as a set of interfaces and rules of interaction that govern how components plugged into the framework may interact. He points out that an overgeneralization of that scheme has to be avoided in order to keep actual use of frameworks practicable.

All of the above definitions no longer regard components as isolated parts. Components adhere to a particular component architecture or architectural style that defines the plugs, the connectors and the corresponding composition rules [Sch01].

2.3 Scripting

Components and frameworks alone are not sufficient for building real applications. We need a mechanism to specify which components are to be plugged together (i.e. *express compositions*). Think of a playscript that tells actors how to play various roles in a theatrical piece. The whole idea behind component-based development is that only a small amount of such wiring code has to be written to create connections between components. Flexibility is obtained by detaching the components from the specification of their composition.

This wiring technology, also called *scripting*, can take various forms depending on the nature and granularity of the components, the nature and problem domain of the framework, and the composition model [Sch99]. Composition may occur at different

times in the development process: at compile-time, at link-time or at run-time. It may be very rigid and static (e.g. the syntactic expansion that occurs when C++ templates are composed [MDS01]) or very flexible and dynamic (e.g. the composition supported by Tcl [Ous94] or other scripting languages [Ous98]). It's not easy to give a generally accepted definition of the terms scripting and scripting language. Nevertheless, in the following sections, we will try to give an overview of what is commonly meant when they are used.

2.3.1 What is Scripting?

The essence of scripting is that of performing routine operations with existing tools. The main purpose of CGI-scripts for example, is to dynamically generate web pages, however, they do not perform all the necessary computations themselves. Various components, residing on the server system the scripts are run on, are used. Most of the code of the CGI-scripts just sets off and coordinates computations of the components.

Sometimes scripts are described as *glue* between components [NTdMS91]. This is a metaphor to emphasize that scripting is done using a *high-level* language that takes entities *outside* the programming language (e.g. system facilities) to do the work of an application. Thus, the glue is a *high-level of abstraction*. However, the term glue code is used in a much narrower sense in most references, as we will see in section 2.4.

Szyperski [Szy02] says that scripting is quite similar to application building. Scripting admits that the actual wiring may need more than just connections: scripting allows small programs (i.e. scripts) to be attached to connections (i.e. connectors). This can be either at the source end (e.g. for events) or at the target end (i.e. hooks) of connections. Scripts usually do not introduce new components, but simply plug already existing ones together: they introduce behaviour, but no state. Or in other words:

> Scripting aims at *late and high-level gluing*.

Summarizing the main properties of scripting given above, we could say that the principal purpose of scripting is to build applications by connecting a set of already existing components. Cox [Cox86] generalizes this notion by defining scripting as follows:

> Scripting is a *high-level binding technology* for component-based systems.

This definition implies that there exist other binding technologies for component-based systems. It also doesn't name any language features needed for scripting. We will discuss and analyze these in the next paragraph.

2.3.2 What is a Scripting Language?

There are two major directions used by researchers for describing scripting languages: by their usage and by their features.

As mentioned above, the purpose of a script is to coordinate a programmable system and to establish connections between components. Therefore, one might argue that any programming language that supports these activities can be called a scripting language. We could label any language that is used to drive another system as a scripting language, as opposed to a programming language, wherein the program itself is the main action.

Of course, the way a language is used depends strongly on the features it supports. By describing a language's usage, as well as its features and characteristics, we can analyze *why* it is used in that specific way.

Kanavin [Kan02] gives the following definition of a scripting language, which already includes a lot of typical features:

> A scripting language eliminates the need for compilation, manages memory automatically and includes high-level data types. Its power is connecting existing components together into a working application.

One of the most important properties of scripting languages is that it should be relatively easy to interconnect components not written in the scripting language itself.

Other important aspects are covered by the following definition:

> A scripting language should i) be interpreted, not compiled, ii) be dynamically typed (so that a variable can have different types during its lifetime), iii) offer abstractions for introspection, iv) be embeddable and extensible, and v) have a simple syntax. *Brent Welch* [Sch99].

Embeddability and extensibility are two important properties of scripting languages because they make reuse a lot easier. A versatile way for adapting and extending an existing component is embedding a script into this component. Extensibility on the other hand is needed in order to incorporate new abstractions (i.e. components and connectors) into the language, making it easier to integrate legacy code.

An interesting reference about scripting and scripting languages is a paper by Ousterhout [Ous98]. He categorizes programming languages into three major groups. Assembly languages, system programming languages, and scripting languages.

In assembly languages, virtually every aspect of the machine is reflected in the program. Each statement represents a single machine instruction and programmers must deal with low-level details such as register allocation and procedure calling sequences. As a result, it is difficult to write and maintain large programs in assembly language.

System programming languages differ from assembly languages in two ways: they are higher level and they are strongly typed. The term higher level means that many details are handled automatically by the programming environment (e.g. register allocation), so that programmers can write less code to get the same job done. The functionality of a single instruction in a system programming language, usually takes several instructions in an assembly language. Ousterhout defines typing as the degree

to which the meaning of information is specified in advance of its use. In a strongly typed language, the programmer declares how each piece of information will be used and the language prevents the information from being used in a different way. In a weakly typed language there are no a priori restrictions on how information can be used: the meaning of information is determined solely by the way it is used, not by any initial purposes. System programming languages are designed to handle the same tasks as assembly languages, namely creating applications (and components) from scratch.

Ousterhout defines a scripting language as follows:

> Scripting languages are designed for gluing applications. 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 languages sacrifice execution efficiency to improve speed of development [Ous98].

Ousterhout claims scripting languages represent a very different style of programming than system programming languages. They aren't intended for writing applications from scratch: their primary purpose is plugging together components. Scripting languages are also rarely used for implementing complex algorithms and data structure, as features like these are usually provided by the components. Scripting languages are sometimes referred to as glue languages or system integration languages.

Scripting languages tend to be weakly typed, in order to simplify the task of connecting components. A weakly typed language makes it easier to hook together components, even in different ways for different purposes not foreseen by the designer.

Scripting languages are higher-level than system programming languages, in the sense that a single statement does more work on average. A typical statement in a scripting language executes hundreds or thousands of machine instructions, much more than a typical statement in a system programming language. Much of this difference is because the primitive operations in scripting languages have greater functionality than those in system or assembly programming languages.

Since performance and resource usage will be dominated by the components and not the scripts, the performance of the scripting language will usually not be a problem. It is however much more important that high-level abstractions for connecting components are provided; the language should give a high-level view of services implemented in a lower-level language.

Figure 2.1 shows a graphical comparison of these three programming language categories. Table 2.1 delivers anecdotal support for the claim that scripting languages speed up the development process.

Above definitions already make up a list of certain characteristics of scripting languages:

- The main purpose of a scripting language is to plug together existing components in order to build applications.

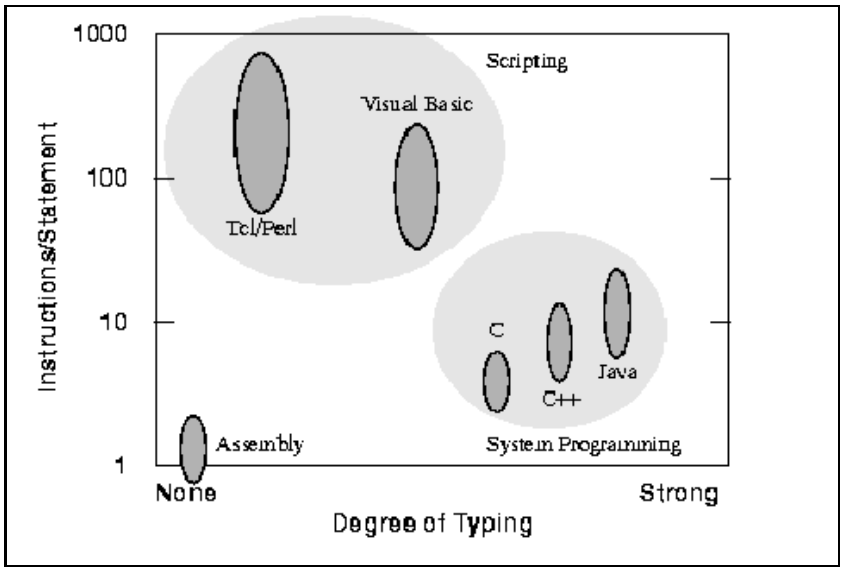

Figure 2.1: Comparison of programming languages, based on their level of abstraction (higher-level languages execute more machine instructions for a typical language statement) and their degree of typing [Ous98].

- Scripting languages prefer high-level abstractions to execution speed.
- Scripting languages are interpreted and offer automatic memory management.
- Scripting languages are dynamically and weakly typed and offer support for runtime introspection.
- Scripting languages are extensible: adding new abstractions (e.g. new components and connectors) to the language and incorporating components implemented in other languages should be straightforward.
- Scripting languages are embeddable: they can be embedded into existing components, offering a versatile way for adaptation and extension.
- Scripting languages offer explicit support for architectural styles and can therefore be considered as executable architectural description languages (ADLs).

Not all of the features listed above are essential for scripting. Considering only the essential properties, Schneider defines a scripting languages as follows:

> A scripting language is a high-level language used to create, customize, and assemble components into a predefined software architecture [Sch99].

Application (Contributor)	Comparison	Code Ratio	Effort Ratio	Comments
Database application (Ken Corey)	C++ version: 2 months; Tcl version: 1 day		60	C++ version implemented first; Tcl version had more functionality
Security scanner (Jim Graham)	C version: 3000 lines; Tcl version: 300 lines	10		C version implemented first; Tcl version had more functionality.
Query dispatcher (Paul Healy)	C version: 1200 lines; Tcl version: 500 lines	2.5	4-8	C version implemented first, uncommented; Tcl version had comments, more functionality.
Spreadsheet tool	C version: 1460 lines; Tcl version: 380 lines	4		Tcl version implemented first.

Table 2.1: Comparison of some applications implemented twice, once using a scripting language and once using a system programming language [Ous98].

2.4 Compositional Mismatches and Glue

2.4.1 What is Glue?

In practice, it is often not possible to just select some components with the needed functionality and plug them together. Many researchers [HÖ3, YS97, Sam97] have identified that "as-is" reuse is very unlikely to occur, and that in almost all cases, a component has to be adapted in some way to fit the compositional requirements of an application or a system. For this adaptations, glue techniques are required. To understand in what ways a component can fail to match the compositional requirements, first consider the following definition:

> *Software composition* is the process of constructing applications by interconnecting software components through their plugs [ND95].

It might not always be possible to interconnect some components in a desired way: their plugs could not be *plug-compatible* [HÖ3]. A nice analogy is a traveller who is unable to plug his razor he uses at home into the plugs of various other countries. These kind of situations are known as *architectural mismatches* and *adaptors* are needed to bridge the different interfaces.

Even if components can be successfully interconnected, this does not mean they will be able to interoperate successfully:

> *Interoperability* is the ability of software components to communicate and cooperate with each other [Kon95].

Reconsider the example of the razor; the form of the plugs is not a problem anymore, but different countries may use different voltages. Even when using an adaptor, the components are not compatible: composition is possible, but interoperability is not. We need a *transformer* to solve this kind of problems, known as *interoperability mismatches*.

Architectural and interoperability mismatch both belong to a problem domain that can be referred to as *compositional mismatch* [Sam97]. A compositional mismatch occurs whenever it is impossible to successfully interconnect components with existing connectors. As we will see in section 2.4.2, architectural and interoperability mismatch are not the only situations where a compositional mismatch may occur.

Schneider [Sch99] formally defines glue as follows:

> *Glue* is the part of an application that overcomes compositional mismatches.

We use this definition of glue because it plays a role in the composition language Piccola and the conceptual framework the language is based on. We will take a look at Piccola in chapter 3 and at the framework in section 2.5. Other references, however, might use different definitions; the notion of glue often refers to any kind of abstraction that can be used to plug components together. We will differentiate between the notions of *scripting* and *glue*: the former refers to abstractions for connecting components while the latter makes mismatched components composable.

2.4.2 Glue problems

Schlapbach [Sch03] identifies different levels on which compositional mismatch can occur: at the *architecture platform level*, where components are not designed for the platform they are supposed to run on. At the *cross-platform level*, where components are running on different component platforms. At the *interaction level*, where components use different protocols. At the *architectural level*, where components make different assumptions about the architecture on which they are supposed to run, leading to architectural mismatch. *Versioning* conflicts can lead to compositional mismatch as well.

There exist various techniques for adapting components to overcome compositional mismatch. They can be categorized either as black-box or white-box techniques. White-box techniques adapt mismatched components by changing or overriding their internal specifications while black-box techniques only adapt their interfaces.

2.5 A Conceptual Framework for Composition

Today, the object-oriented programming paradigm is the most dominant one. The languages and design techniques utilizing it are nearly ideal for *implementing* components, but it seems they hinder component-based development in a number of significant ways [SN99, Ach02]:

- **Reuse comes to late:** object-oriented analysis and design methods are largely domain-driven. This leads to designs based on domain objects and non-standard architectures. Most of these methods make the assumption that an application is being written from scratch and they incorporate the reuse of existing components too late in the development process (if at all).
- **Overly rich interfaces:** Instead of sticking to small, restricted and plug-compatible interfaces and standard interaction protocols, OOA and OOD lead to rich interfaces and interaction protocols.
- **Lack of explicit architecture:** For a programming language to support component-based development, it must offer a way to state both what is to be computed by a component (i.e. *computational*) and the way components interoperate (i.e. *compositional*) [ND95]. Object-oriented source code exposes the inheritance hierarchy instead of the object interaction. How the objects are plugged together is typically distributed amongst the objects themselves. As a consequence, adapting an application to new requirements requires detailed analysis.
- **Little code reuse:** Instead of providing reusable abstractions for object collaborations, object composition is often implemented according to design patterns. While we can (and should) reuse design, we often cannot reuse the actual code.

Schneider [Sch99] claims that complex software systems are increasingly required to be open, flexible aggregations of heterogeneous and distributed software components rather than monolithic heaps of code. He says this places a strain on old-fashioned software technology and methods that are based on the maxim

> Applications = Functions + Data.

The object-oriented approach already went a step further and does a fairly good job encapsulating state and behaviour. It is based on

> Applications = Objects + Messages.

However, as we have seen above, object-oriented technology is often applied in a way that hinders component-based development. Achermann et al. [ALSN01] say that the flexibility and adaptability needed for component-based applications to cope with changing requirements will be improved if we not only think in terms of components, but also in terms of architectures, scripts, coordination and glue. It is claimed that the following paradigm should be applied for application development:

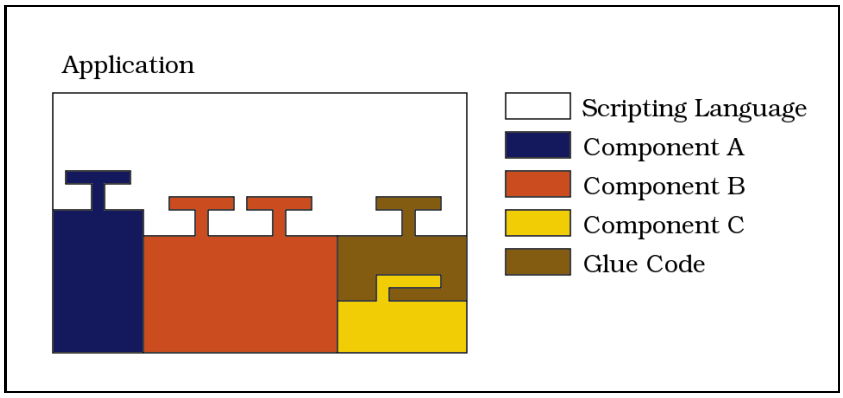

Figure 2.2: A framework for component-based development [Sch03].

Applications = Components + Scripts.

Components act as black-box entities, encapsulating services behind well-defined interfaces. Scripts on the other hand, specify how the components are related to each other.

Schlapbach [Sch03] extends the equation to also contain the glue code, needed for overcoming compositional mismatch as we have seen in section 2.4.

Applications = Components + Scripts + A Drop of Glue.

Figure 2.2 illustrates this framework for component-based development.

Chapter 3

The Piccola Programming Language

This chapter gives an overview of the language Piccola. Piccola's purpose and architecture are presented in section 3.1. Section 3.2 discusses forms, which are Piccola's only first-class values. Section 3.3 focuses on the original approach used to access external components and the problems this approach causes. A revised version of the approach is presented in section 3.4.

3.1 An Overview of Piccola

3.1.1 What is Piccola?

Piccola is a scripting and composition language. It is designed with the conclusions of section 2.5 in mind. It cannot only be used to express how software components written in a separate implementation language should be configured, but also to provide the connectors, coordination abstractions and glue abstractions to plug those components together on a higher level [AN01, ALSN01].

Most of the scripting languages of the the fourth generation, have a rich set of programming constructs and built-in features that facilitate composition of components according to a predefined compositional style [AN01]. For instance, Unix shell languages are designed around the pipes and filters approach; the composition rules tell us which compositions are valid (e.g. it is impossible to make circular pipes and filter chains). Another example is Perl [WS91], which provides regular expressions to work on a number of string buffers. Piccola, on the other hand, is a *small*, *pure* and *general-purpose* composition language [Sch01]:

- **Small:** Piccola has only a small syntax and a limited number of primitives, needed for specifying different types of compositional styles.

- **Pure:** Piccola is a pure composition language, because there is only a small set of primitives providing the necessary composition abstractions. All the other features of the language are provided by exchangeable components. Even basic programming entities such as numbers and strings are represented by dynamically reconfigurable components.
- **General-purpose:** Piccola is a general-purpose composition language because it supports composition of components corresponding to different kinds of compositional styles. This means that Piccola allows us to specify our own styles that define a kind of component algebra. According to Achermann [AN01], another example of a general-purpose scripting language is the popular language Python [vR95].

3.1.2 Language Model

In order to have the simplest possible framework to define compositional styles, Piccola has a small set of primitives that unify various concepts [AN00]:

- **Forms embody structure:** A form is an immutable set of bindings that associate labels with values. They can be extended with additional bindings, which yields a new form. Forms unify objects, services, keyword-based arguments, namespaces and interfaces.
- **Agents embody behaviour:** Agents are concurrent, communicating entities whose behaviour is specified by a script. Agents implement the connections between components, and they unify communication and concurrency.
- **Channels embody state:** Channels are the mailboxes that agents use to communicate. They unify synchronization and communication.

3.2 Forms

Forms are Piccola's only first-class values, and consequently they represent all first-class entities. We start this section by presenting the syntax of forms and the operations defined on them. Then, we show how they are used to model different language aspects. Finally, we compare them to the object-oriented model.

3.2.1 Semantics of Forms

Forms are immutable sets of bindings that associate labels with values. Syntactically speaking, there are two ways to define forms: using nested, comma-separated, parenthesized lists of bindings or by indenting bindings, where the indentation indicates the nesting level. Semantically, there is no difference between the two. Both styles are exemplified in listing 3.1.

```
# A form with 2 bindings: x and y
aPoint = (x=1, y=2)

# Indentation indicates nesting levels
aCircle =
  centre =
    x = 3
    y = 4
  radius = 5

# The previous definition is equivalent to this one
aCircle = (centre=(x=3, y=4), radius=5)
```

Listing 3.1: Defining forms in Piccola

The following operations are defined on forms [Sch01]:

Polymorphic Extension. Polymorphic extension `F, G` of a form `F` with a form `G` yields a new form containing all the bindings of the form `G` and the bindings of the form `F` whose labels are not used within the form `G`. This means that bindings of the form `G` override bindings with the same label of the form `F` in the resulting form. Polymorphic extension is illustrated in listing 3.2. When extending a form, it is of course also possible to use the indentation syntax that is shown in listing 3.1.

Projection. Projection allows us to retrieve the form bound to a certain label. This means `F.t` returns the form bound to the label `t` within the form `F`. A runtime exception will be thrown if the form `F` does not contain a binding labeled `t`. An example of projection is given in listing 3.3.

Application. Services are Piccola abstractions, which represent functions or procedures. Because everything in Piccola is a form, services are also represented as forms. They are bound with `:` instead of `=` and might introduce named arguments. The application `F G` invokes the service represented by the form `F` with the form `G` as argument and yields the resulting form. An alternative syntax that can be used, is `F(G)`. A form can have bindings and represent a service at the same time. Application is exemplified in listing 3.4.

Restriction. Removing a binding `t` from a form `F` is possible through restriction. If no binding labeled `t` exists within form `F`, an error is generated. A label (e.g. the left side of `t = 1`) is a first-class value in Piccola; it can be passed to and returned from services. In order to get hold of a label with a certain name, the primitive service `label` can be used. This service takes a form as argument and returns an arbitrary label that is bound in that form. So when passed a form that contains only one binding, the label of that binding will be returned. Such a label then provides a `restrict` service, that when invoked with a form as argument, will return the argument form minus the label. The code snippet in listing 3.5 gives an example of restriction.

Inspection. To find out whether a form contains bindings, represents a service or

```
# A nested form with 3 bindings:
# name, age and size
F =
  name = "Lieven"
  age = 23

  # The label size is bound to the
  # form (m = 1, c = 80)
  size =
    m = 1
    c = 80

# A nested form with 2 bindings:
# age and size
G =
  age = 24
  size = m = 1, c = 82

println (F, G) # prints (name = "Lieven",
               #         age = 24,
               #         size = (m = 1, c = 82))
```

Listing 3.2: Example of polymorphic extension in Piccola

is the empty form, inspection is used. See listing 3.6 for an example. The primitive service `inspect` is curried. As a first argument it takes the form that gets inspected. The second argument should contain three services, labeled `isLabel`, `isEmpty` and `isService`. Depending on the structure of the inspected form, the right service gets invoked. If a form contains bindings, inspection can be used to retrieve an arbitrary first-class label that is available within the inspected form.

There exist no syntactical structures for restriction and inspection, above mentioned primitives have to be used. Note that iteration over the bindings of a form can be accomplished by combining restriction and inspection.

3.2.2 Unification of Concepts

Since forms are the only first-class citizens in Piccola, they are used to model different language concepts. Schärli [Sch01] gives the following overview:

- **Data structures:** Piccola uses (nested) forms to define data structures. These data structures are basic objects that may consist of structure and behaviour (services).
- **Services:** Services represent functions or procedures. Internal services are defined by Piccola scripts, external services are provided by external components.

```
# A nested form with 3 bindings:
# name, age and size
F =
  name = "Lieven"
  age = 23

  # The label size is bound to the
  # form (m = 1, c = 80)
  size =
    m = 1
    c = 80

F.name   # Prints: Lieven
F.size   # Prints: (m = 1, c = 80 )
F.size.m # Prints: 1
F.weight # Error! (F does not contain a
         # binding labeled weight)
```

Listing 3.3: Example of projection in Piccola

```
# The form F gets defined as a service
# taking an argument X
F X:              # Alternative definition:
                  # F = \X: ...
  value = X
  predecessor = X - 1
  successor = X + 1

println (F 3) # Prints: (value = 3,
              #          predecessor = 2,
              #          successor = 4)
```

Listing 3.4: Example of application in Piccola

```
# A form with two bindings labeled name and age
F =
  name = "Lieven"
  age = 23

# The service label returns an arbitrary first
# class label bound in the argument form.
# Here, it returns age because this is the only
# label in the argument form
labelAge = label(age = ())
G = labelAge.restrict F       # Form restriction

println F     # Prints: (name = Lieven,
              #          age = 23)
println G     # Prints: (name = Lieven)
```

Listing 3.5: Example of restriction in Piccola

```
# Define the three services for the second
# argument of the inspect service
Cases =
  isEmpty = println "Form is empty"
  isService =
    println "Form is a service and has no bindings"
  isLabel = println "Form with label" + L.name()

inspect () Cases      # Prints: "Form is empty"

inspect (\X: X) Cases # Prints: "Form is a service
                      # and has no bindings"

inspect (a = 5) Cases # Prints: "Form with label a"
```

Listing 3.6: Example of inspection in Piccola

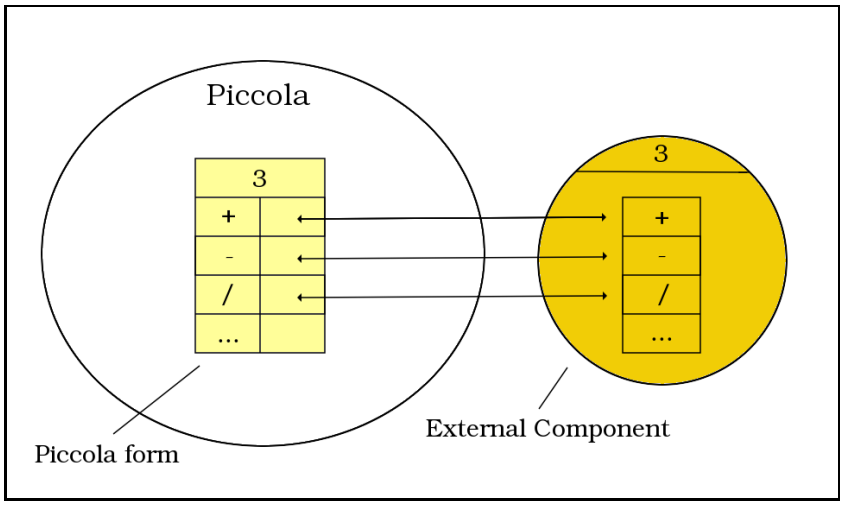

Figure 3.1: A Piccola form as interface to an external component.

Both internal and external services are represented as forms. This allows us to define higher-order services.

- **Keyword-based arguments:** The structure of forms permits the strictly monadic Piccola services to use keyword-based arguments with optional default values.
- **Namespaces:** Piccola supports both static and dynamic namespaces, which are modeled as nested forms.
- **Channels:** As we have mentioned in section 3.1.2, channels are the mailboxes that agents use to communicate. New channels are created by the primitive service `newChannel`, which returns a form that gives access to a channel. This form consists of two services for sending respectively receiving.
- **Labels:** Piccola has the notion of first-class labels, which provide a non-syntactic alternative for form extension (`bind`), restriction (`restrict`) and projection (`project`). They can also be used to find out whether a form contains a specific label (`exists`).
- **External components:** Piccola represents external components (respectively their interfaces) as forms. Figure 3.1 shows an external Piccola form that represents the object 3. All the (public) methods of the object are mapped to the corresponding labels of the form. Thus, the external object can be considered an interface or a proxy for the associated object. We will discuss the mechanisms Piccola uses to provide external objects as Piccola forms in sections 3.3 and 3.4.

3.2.3 Forms vs. Objects

- **No self:** There is kept no reference to the active form (i.e. the form where the currently executed service is defined).

```
printDate X:
  println X.month + "/" + X.day + "/" + X.year

# Defines a form representing a date
date =
  month = "6"
  day = 1
  year = 2006

printdate date      # Prints: "6/1/2006"

# Defines a form representing a date and a time
dateAndTime =
  date
  hour = 7
  minute = 22       # Prints: "6/1/2006"

printdate dateAndTime
```

Listing 3.7: Dynamic extension in Piccola

- **Immutability:** Forms are extensible but immutable data structures. There is no need for a copy semantics. Altering a form in any way (e.g. extension or restriction) will always yield a new form and leave the original one unchanged.
- **Prototype-based instantiation:** Forms may be built by adding bindings to or removing bindings from an already existing form. There is no need to specify a class to instantiate it. This approach is similar to the one used in prototype-based object-oriented languages.
- **Dynamic extension:** Polymorphic extension can be used as a primitive subclassing mechanism. As is the case with traditional subclassing in object-oriented languages, extended forms are compatible with the original forms. This means an extended form can play the role of the original one. Form extension is completely dynamic and directly applies to forms as runtime entities.

  The example in listing 3.7 uses a service `printDate` that prints the date represented by its argument form. This service expects that its argument provides at least the bindings `month`, `day` and `year`. First, we invoke the service with a form that represents a date and contains only the required bindings. Next, we extend the form with bindings specifying the time and show that it is still compatible with the service.

3.3 Accessing External Components From JPiccola 2

Piccola is originally implemented on top of Java and this implementation is appropriately called JPiccola. This means the parser and virtual machine are both written

in Java, whereas other parts (e.g. simple development environment, small library) are built in Piccola by using Java components. Piccola is designed to be a composition language, so using external components is a core principle of JPiccola, which consequently influences its implementation.

3.3.1 JPiccola's Virtual Machine

The JPiccola virtual machine consists of a special part called the inter-language bridge, which reflects the fact that Piccola is a composition language. It allows accessing external components and their methods from within Piccola. Instead of providing a large set of primitives to perform basic system operations such as integer arithmetic, like most virtual machine implementations do; the JPiccola virtual machine delegates these operations to external components via the inter-language bridge.

Altogether, the JPiccola virtual machine consists out of the following parts:

- Interpreter
- Runtime data structures (forms)
- Primitive services
- Inter-language bridge

The interpreter directly operates on the parse trees. Interpretation of these parse trees result in forms, which are the only first-class values in Piccola. Every form is represented by a Piccola runtime data structure that is an instance of a Java class providing the five basic form operations shown in section 3.2.1.

When a Piccola service is invoked, the interpreter executes the Piccola code that is associated with the service. This is not true for all services however. Some of them trigger the execution of a virtual machine primitive. Such primitive services are typically used by virtual machines to perform basic operations that cannot be performed or can only be performed inefficiently without a primitive. But while most virtual machine implementations provide lots of primitives for arithmetic operations, arrays and streams, input/output, storage management, and system operations, Piccola only needs the four primitives that are shown in table 3.1.

Every other basic operation of Piccola is delegated to external components that are accessed through the inter-language bridge. This part of the Piccola virtual machine allows passing of runtime entities across the language boundary; not only from Java to Piccola but also from Piccola to Java.

[1]Note that the primitive to access external components depends on the host language. Thus, the name and the semantics of this service are implementation dependent.

Primitive	Description
`run`	Spawns a new asynchronous agent executing the service that is bound to the label `do` of its arguments.
`newChannel`	Creates a blocking communication channel.
`inspect`	Inspection is used to find out whether a form contains bindings, represents a service or is the empty form. If the form contains bindings, inspection returns an arbitrary label that is used within the form.
`external`	Provides access to external components [1].

Table 3.1: The Piccola primitives [Sch01].

3.3.2 Bridging between Two Nested Language Models

Because Piccola is implemented on top of Java, we are dealing with two nested language models. The Piccola model one the one hand, where forms are the only runtime entities. On the other hand there is the Java model, where everything is an object. Because Piccola is running in the Java model, every Piccola form is actually a Java object. Java objects are, however, incompatible with the form-based Piccola model and so they cannot be accessed within Piccola.

Terminology

To discuss this concrete situation at a more abstract level, we will first define some terminology. The *down level* refers to the (object-oriented) implementation language, in this case Java. The term *up level* denotes the level of the language that is implemented and evaluated by the down level. A more standard name of the down level is the *meta level*, while the up level is more commonly referred to as the *base level*. The down level is assumed to supply some object-like first-class entities, which will be called *objects*. The first-class entities of the up level are named *forms*. When an object is passed from the down level to the up level, the object is said to be passed upwards. Consequently, a form is passed downwards when it is passed from the up level to the down level. The two language models and the terminology are illustrated in 3.2.

Passing objects upwards

The language model of the up level cannot handle the generic objects of the down level, so they have to be converted into forms to be of any use. As such, the inter-language bridge has to provide an appropriate representation for every object that crosses the language boundary upwards. If the passed object is already a down level representation of a form, nothing has to be done and the inter-language bridge can simply forward the form.

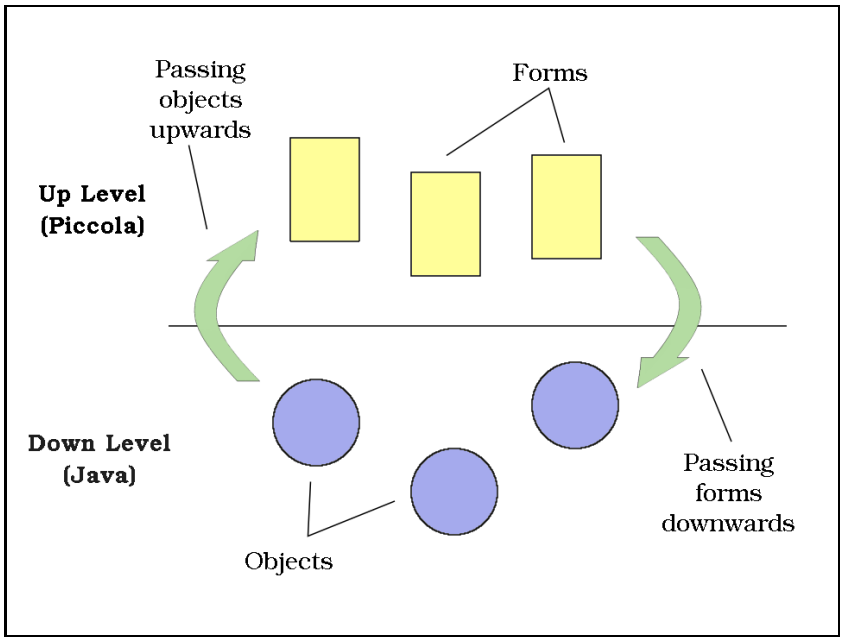

Figure 3.2: Nested language models (up level and down level) [Sch01].

As a result, their exists two different kinds of forms in the up level. The simplest ones are the forms that are *ex nihilo* created within the up level. A form of this kind might be the empty form or one that is created by binding values to labels within Piccola. The other forms are the ones that are automatically created by the inter-language bridge whenever objects are passed upwards. Schärli [Sch01] calls the former kind *plain forms* and the latter one *external forms*. External forms are actually just up level representations of down level objects, thus every external form has an *associated object*.

Passing forms downwards

Because a form is just an object in the down level's language model, a form does not need to be converted and can be passed downwards as it is. Although this is what we want for plain forms, it is often not the expected behaviour for external forms, which represent down level objects. We want the down level to operate on the associated object of the external form rather than on the form itself. The inter-language bridge thus has to decide which one of the two entities it has to pass down.

3.3.3 JPiccola's Bridging Approach

In this section, we look at the bridging approach that is used in JPiccola [Sch01, ALSN01].

Up. Passing Objects from Java to Piccola.

A. If the object already represents a form, it is passed directly to the Piccola language.

B. Otherwise, the following happens:

   B1. The object is converted into a form that contains a label for every public method of the object's class, whether it is implemented or inherited. Each of these labels is than bound to a service that represents this method for the given object. The object serves as *self* when the service is called.

   B2. Forms representing special objects like numbers, booleans or strings are extended with additional bindings to make them more appropriate from a Piccola point of view.

The external forms we have discussed before are the ones created by step *Up.B* of the inter-language bridge. Note that a form that has been yielded by extending or restricting an external form is no longer regarded as an external form.

Down. Passing Forms from Piccola to Java.

A. If the form is an external one, the associated object is passed down to Java.

B. Otherwise the form itself (actually, the object representing it) is passed down to Java.

Figure 3.3 illustrates how forms are passed down.

3.3.4 Problems with JPiccola's Bridging Approach

As we have seen in the previous section, JPiccola's inter-language bridge does what it is supposed to do: making external components accessible by wrapping them up as forms. However, Schärli [Sch01] points out that the employed strategy is still not flexible enough. The external forms as provided by the bridge cause many incompatibilities with Piccola's core concepts. The problems are coupled in such a way that users cannot work around some of them without running into other ones.

Incoherent Behaviour of External Forms

Because Piccola uses external components even for basic operations, the programmer should be able to use forms transparently, whether they are internal or represent such an external component. This is especially important for extension. As we have seen in section 3.2.3, form extension can be used as a very simple but dynamic subclassing mechanism in Piccola. This means we can extend a form with new bindings, where the newly created form remains compatible with the original form. This extended form can play the role of the original one in a way that resembles a subclass playing the role

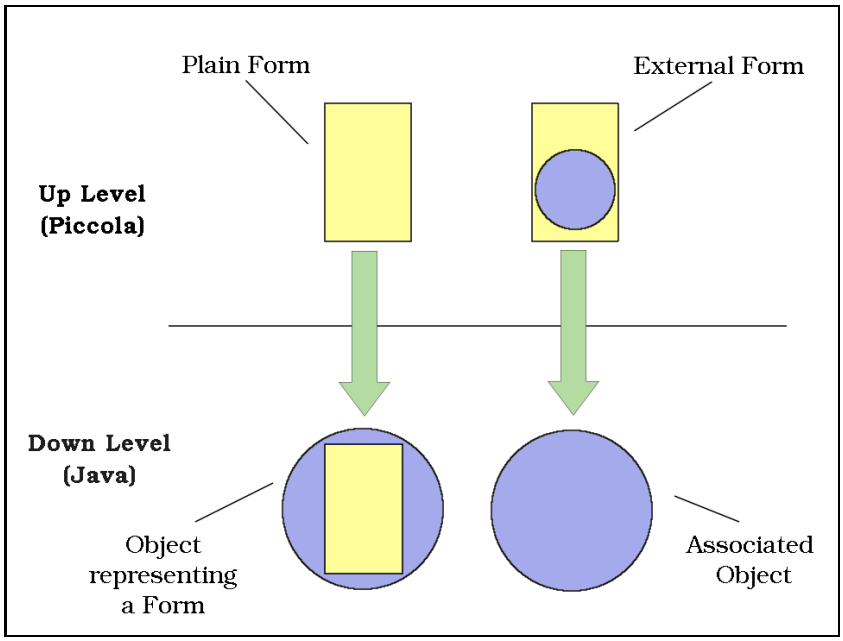

Figure 3.3: Passing down Piccola forms.

of its base class. Piccola's semantics assure this is always the case for a form, provided it is used inside Piccola.

Unfortunately, this basic Piccola concept does not hold for forms that are passed to external services. Once an external form is extended, it will not be considered an external form anymore and it will behave differently when passed to the down level. While an external form gets converted into the associated external object (*Down.A*), a plain form is not converted, and the object representing the form is passed to the down level (*Down.B*). It is not possible for the user to find out whether a certain form is external, and as such it is also impossible to know whether extending the form will influence the way it is dealt with by the bridge.

The code snippet in listing 3.8 demonstrates this problem. First, a service `newButton`, which builds an external form representing a button and extends it with a service `setText`, is created. Then, we invoke the service `newButton` to create a new button and set its label with the `setText` service. Eventually, we want to add the button to a Java panel, but Piccola does not behave as one might expect. Because the argument `okButton` has been modified, it is not regarded an external form anymore, and the inter-language bridge passes the object representing the form (*Down.B*) and not the associated button object (*Down.A*) to the down level.

```
# This service creates a new button and extends
# it with an additional interface binding
newButton:
  'button = javaClass("java.awt.Button.).new()
  button
  setText(S):
    button.setLabel(val = "Piccola-Button: " + S)

okButton = newButton()
okButton.setText("Ok")   # Uses interface binding to
                         # set the label

# XPiccola.piccolaPanel.buttons is a Java panel
# of the Piccola user interface
panel = XPiccola.piccolaPanel.buttons

# Unexpected behaviour! The object representing the
# Piccola form and not the original button object is
# passed down
panel.add(val = okButton)
```

Listing 3.8: Problems with external Piccola forms [Sch01].

Direct Mapping

When an external component is converted into a Piccola form, this is done in a very direct way. JPiccola maps the method interface of the object entirely onto the resulting form. Consequently, Piccola almost operates on the level of the host language. This leads to a couple of problems:

No separation between the language levels. Most of the components used by JPiccola are Java objects. Because of the direct bridging strategy, handling these components can quickly become "Java programming within Piccola". Because both languages have very distinct philosophies, this results in code that does not fit the Piccola paradigm. Moreover, such code is inherently Java dependent and cannot be used on other Piccola hosts.

External forms are cluttered with lower level services. Due to the rich object interfaces Java usually provides, the corresponding Piccola forms contain many bindings. This makes them very complex and contradicts Piccola's philosophy. Within Piccola, Java objects are considered components that contain a small set of services used to plug them together according to a certain compositional style. Thus, most of a Java object's public methods should not be visible on the composition level.

Hard to use components with incompatible interfaces. Because of the lack of abstraction for accessing external components, it is not possible for the programmer to use components with incompatible interfaces. Instead of directly using these interfaces, the bridge should convert them according to the current compositional style.

Hardcoding

In step *Up.B2*, the inter-language bridge adds some special bindings to external forms representing some of the more commonly used Java object. Although this provides a solution for some of the direct-mapping problems, it is not a flexible solution, because the extension is hardcoded in the virtual machine. Piccola is designed to be a general-purpose composition language, which means it should be able to use it for varying problem domains. This domains might have very different requirements on the used components. When the structure is hardcoded in the virtual machine, it is completely static, and no changes are possible without substituting the entire virtual machine.

3.4 JPiccola's Bridging Approach Revisited

After having analyzed the problems described in the previous section, Schärli [Sch01] presents a bridging strategy that adheres more closely to the philosophy of Piccola. This strategy is based on two main concepts:

- Separating the various aspects of external forms.
- Moving the variable part of the inter-language bridge onto Piccola's meta level.

3.4.1 Overview of the Revised Strategy

As in the previous sections, we use the term *external form* to denote a form that represents an external object within Piccola. All other forms are called *plain forms*. As a first change to the bridging strategy, Schärli says an external form must have a nested structure consisting out of two parts. The top level part represents the Piccola interface of the object and hence is called the *interface form* or shortly *interface*. This form contains a label `peer`, which is bound to a nested form representing the identity of the external object. This subform is called the *peer form* or just *peer*. Only forms that satisfy these structural conditions are considered external forms. A form with a label `peer` that is not bound to a peer form (i.e. a form representing an external component), for example, is not an external form.

To make the inter-language bridge more flexible, it is separated into two parts. The *generic part* is situated in Piccola's virtual machine while the *variable part* can be found inside Piccola. If an external object is passed up to Piccola, both of these parts may build an interface for this object. Note that these interfaces usually include glue code. The interfaces built by the generic and the variable part of the bridge are called the *generic interface* and the *specific interface*, respectively. Consequently, an external form created by the generic part of the bridge is named a *generic form* and one created by the variable part is called a *specific form*.

3.4.2 Illustration of the Revised Bridging Strategy

Figure 3.4 illustrates the structure of the inter-language bridge and how entities of both levels are passed across the language boundary. We can see that the bridge is divided into two parts. The generic part is implemented in the virtual machine, which is part of the down level while the variable part is situated in Piccola's meta level.

On the left side of figure 3.4 is shown what happens when an object that does not represent a form is passed upwards. In the generic part of the inter-language bridge, the object is converted to fulfill the requirements for external forms that were set in section 3.4.1. This external form consists of a generic interface and the peer form that represents the identity of the object. This form is then passed to the variable part of the inter-language bridge on Piccola's meta level. Here, the generic interface gets replaced by a specific one that can be declared by the programmer. This resulting external form can be used in Piccola. The variable part is not obliged to provide a specific interface. It can just pass an external form with a generic interface to Piccola.

In the middle is illustrate how forms are passed downwards. In the first step, the inter-language bridge picks out the nested form bound to the `peer` label if it exists. In all other cases, it takes the form itself. If the currently handled form is a peer form, the associated object is passed to the down level, otherwise, the form itself is passed downwards. Thus, an external form is converted to its associated external object while a plain form is passed down as it is.

On the right side, an object that represents a form is passed upwards. This is a trivial case, and the bridge directly passes the form to Piccola.

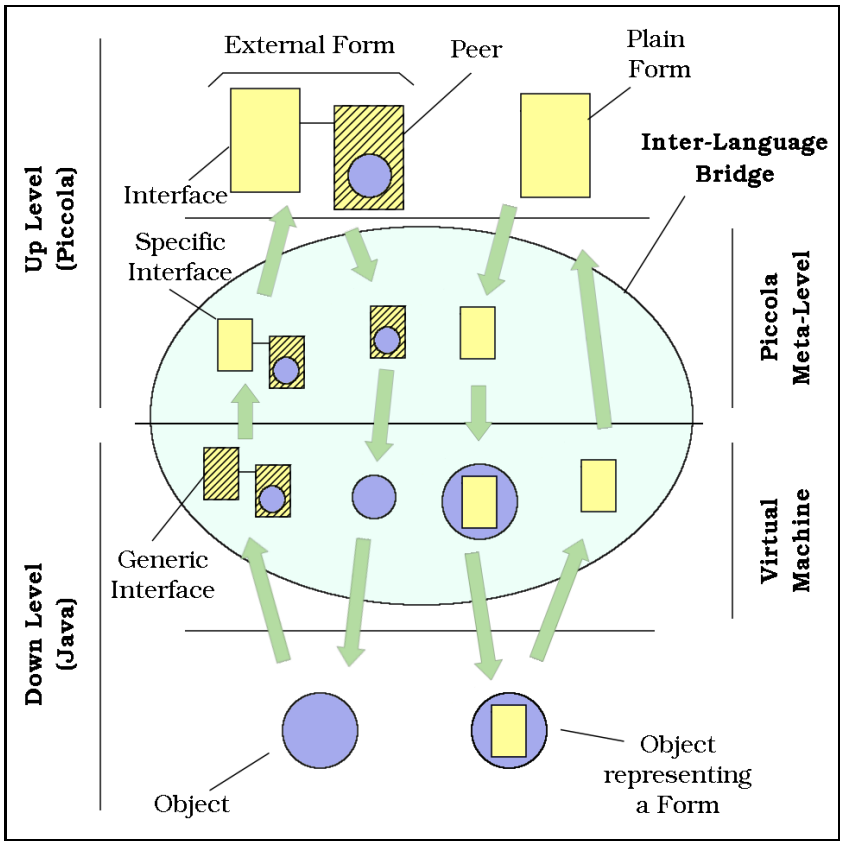

Figure 3.4: The revised inter-language bridge [Sch01].

Chapter 4

The Pico Programming Language

Sections 4.1, 4.2, and 4.3 give an overview of the language Pico. A prototype-based extension of Pico, called Pic%, is presented in section 4.4. Section 4.5 introduces Sic%, a Pic% implementation, which is in linguistic symbiosis with Smalltalk. A conceptual linguistic symbiosis model, on which Sic% is based, is discussed in 4.6.

4.1 Properties and Purpose of Pico

Pico [MGD99] is a high-level programming language, featuring strong and dynamic typing, garbage collected tables (i.e. arrays), higher-order functions and reflective meta-programming. But as its name underlines, Pico is above all very small.

Pico was originally designed to teach computer science concepts to students in other sciences than computer science. Pico can be seen as a derivative of the Scheme programming language, combining the latter's simple semantics and powerfulness (in the sense that everything is first class) and the standard infix notation students are used to from ordinary calculus. De Meuter et al. [MDD04] claim that there are indications that essential programming concepts can be acquired using Pico in less than half the time it takes using Scheme.

From a research point of view, Pico is an ideal candidate for extension. The compactness of its implementation makes it very easy to add new experimental language features. It has been used in experiments related to distributed programming [CM04, CMM+04], prototype-based inheritance [DM03], mobility of software agents [BFVD01, Meu04], language symbiosis [Pee03, Lie05] and actor-based programming [Ded06].

```
{
  fac(n):
    if(n <= 0,
      1,
      n * fac(n - 1));
  display("fac(10) = ", fac(10), eoln)
}
```

Listing 4.1: A recursive implementation of the factorial function in Pico

4.2 Syntax and Semantics

The syntax of Pico is exemplified in listing 4.1, with an implementation of the factorial function in Pico. The first expression defines a function called `fac`, which takes one argument. The body of this function contains only one expression, an application of the `if` conditional [1]. The second expression calls the `fac` function and prints the result to the standard output. Although it may seem like it, this code snippet does not consist out of two *statements* since Pico does not make a distinction between expressions and statements. All Pico expressions return a value and syntactically, any expression can be nested in any other.

Pico has a small and easy to remember syntax. A key notion is the invocation, which is either a variable, a tabulation or an application. Such an invocation is used in three modes: reference, definition and assignment. By combining the three invocation types into the tree modes, nine different Pico expression types are constructed, as is shown in the simple 3 by 3 grid in figure 4.1. Some of Pico's language extensions have added additional operators. Version two of the Pico language, for example, defines a dictionary qualification operator.

An important property Pico shares with Scheme is that it both are *homoiconic* [McI60] programming languages. This means the primary representation of a program, is also a data structure in a primitive type of the language itself. Scheme programs are represented as *lists*, whereas Pico uses *tables* for this purpose. Compiled languages like C++ and Java usually lack this property since their program trees are compiled and translated to machine code. Because of this feature, code and data can be treated in exactly the same way and this lies at the basis for reflection in Pico.

4.3 Lazy Evaluation through Call-by-Function

The majority of programming languages differentiate between ordinary expressions and special forms. Tasks like defining a new variable or assigning a new value to one are usually taken care of by special forms. For example, Scheme makes a difference

[1] Note that the `if` conditional is syntactically expressed as a function. As will be shown in section 4.3, this is also the case semantically.

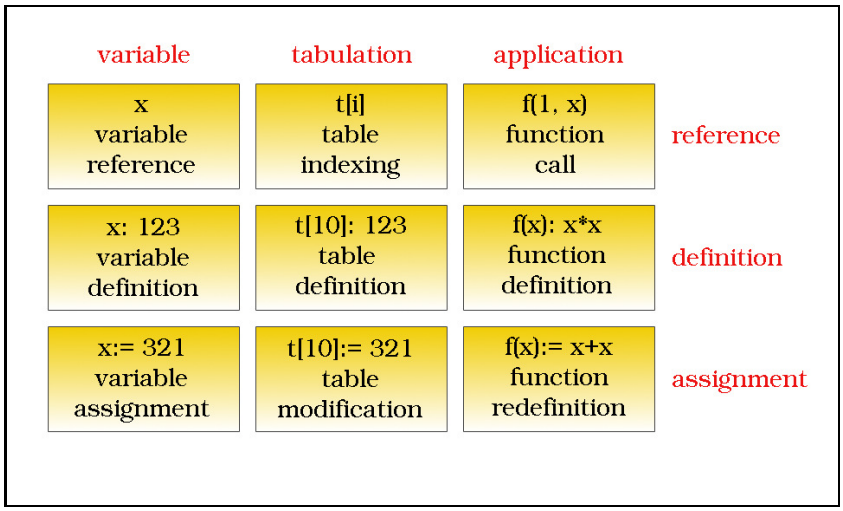

Figure 4.1: The Pico semantic grid [MGD99].

between *procedure calls* and *special forms*. Whereas procedure calls are evaluated using applicative-order evaluation [2], Scheme does something special for special forms. Consider for example the `if` control construct: `(if (< 1 2) x y)`. Scheme will evaluate the first argument passed to `if`. Depending on the returned value, either the second or the third argument will be evaluated while the other one remains unevaluated.

Contrary to Scheme, Pico does not require special forms. Equivalent expressions are in Pico defined as functions. This is possible because it has richer parameter binding semantics than Scheme. When a function is defined, a formal parameter can be specified as an invocation. If the parameter is specified as a reference, it will be bound by value at application time (i.e. *call-by-value*); but if it is specified as an application, we will have an extension of *call-by-name*, which is called *call-by-function* [MDD04]. Listing 4.2 contains an example: the function `maps` behaves like its Scheme counterpart. In this example the call-by-value parameter `table` will be bound to the value of `[1, 2, 3, 4, 5]` while the call-by-function parameter `func(val)` will be bound to the expression `val*val`. This means that during the application of `map`, a local variable `func` will be bound to a closure consisting of the parameterlist `(val)`, the body `val*val` and the calling environment of `map` [3].

Even boolean arithmetic in Pico is implemented in Pico itself using this lazy evaluation mechanism. The code in listing 4.3 implements the `true`, `false` and `if` functions. As can be seen in this code snippet, Pico's boolean system is based on the famous *Church booleans*, which were first introduced in Alonzo Church's λ-calculus [Chu41]. Although slightly adapted for imperative languages, the idea remains the same: to define booleans as functions that choose between two options supplied as arguments. The `if` implementation takes advantage of this model by passing its `then` and `else` branches to the Church boolean representing the `if` condition. Since only

[2] Languages using applicative-order evaluation evaluate an operator and all of its supplied arguments before executing the operation.

[3] Note that `val` has dynamic scope

```
{
  map(func(val), table):
    {
      result[size(table)]: void;
      for(i: 1, i <= size(table), i:= i + 1,
        result[i]:= func(table[i]));
      result
    };

  'Prints [1, 4, 9, 16, 25]'
  display(map(val * val, [1, 2, 3, 4, 5]))
}
```

Listing 4.2: Example of call-by-function in Pico

```
{
  true(yes(), no()): yes();
  false(yes(), no()): no();
  if(cond, then(), else()): cond(then(), else())
}
```

Listing 4.3: Boolean arithmetic in Pico

one of the branches has to be evaluated, they have to be passed in a lazy way instead of as an evaluated value. As we have already seen, this can be achieved in Pico by declaring the parameter functional (i.e. by using the `()` syntax after the parameter name). This is a very elegant solution to a problem normally solved by wrapping code in *lambda* expressions (Lisp, Scheme) or *blocks* (Smalltalk). What is more, the user need not to perform the wrapping at the calling level since the laziness of evaluation is already specified at the definition level. This leads to a much cleaner syntax and semantics.

Consider the implementation of a Pico `while` iterator shown in listing 4.4 as a last example of call-by-function parameter binding.

4.4 Pic%: Adding OO to Pico

Pic% [DM03] is the object-oriented extension of the Pico programming language. It is a prototype-based language, based on Agora [Ste94].

```
while(predicate(), expression()):
  {
    loop(value, boolean):
      boolean(loop(expression(), predicate()), value);
    loop(void, predicate())
  }
```

Listing 4.4: A while iterator in Pico

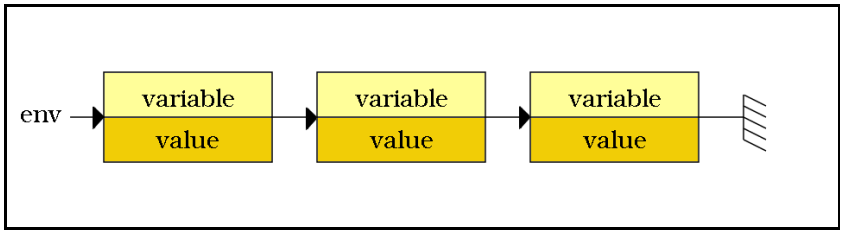

Figure 4.2: A simple environment layout.

4.4.1 A Simple Object Model

Object-orientation in Pic% is obtained by letting *first-class dictionaries*, a feature which is already present in Pico, act as objects. Pico uses these dictionaries to store variable bindings into. A typical implementation would organize them as linked lists, as is graphically illustrated in figure 4.2. The only thing that needs to be added is a native function `capture` that returns the current environment as a first-class value.

Listing 4.5 shows how an object definition in Pic% might look like. An expression like `c: Counter(0)` will now define a variable `c`, bound to a counter object with attribute `n` initialized to zero and two methods `incr` and `decr`, which change the state of the object. These will be stored in the closure that is created when the function `Counter` is applied.

Scheme users may notice the similarity with the object system presented by Abelson and Sussman [ASS96]. The only difference is the absence of a `dispatch` function.

```
Counter(n): {
  incr(): n:= n + 1;
  decr(): n:= n - 1;
  capture()
}
```

Listing 4.5: A simple object definition in Pic%.

```
(define (make-counter n)
  (define (incr)
    (set! n (+ n 1)))
  (define (decr)
    (set! n (- n 1)))
  (define (value)
    n)
  (define (dispatch msg)
    (cond
      ((eq? msg 'incr) incr)
      ((eq? msg 'decr) decr)
      ((eq? msg 'value) value)
      (else
        (error "Message " msg " not understood."))))
  dispatch)
```

Listing 4.6: A analogous object definition in Scheme [ASS96].

```
{
  ‘ invocation of a method in a dictionary ‘
  object.aMethod(10, 100);

  ‘ variable reference in a dictionary ‘
  object.attribute;

  ‘ table indexing in a dictionary ‘
  object.collection[2]
}
```

Listing 4.7: Message sending in Pic% by means of Pico dictionary qualifiers.

In an environment-based object model, such a function returns an appropriate function when a message is sent to an object, as can be seen in listing 4.6. Scheme needs this `dispatch` function because it does not support first-class dictionaries.

Since dictionaries act as objects, the dictionary qualification operator serves as a message sending mechanism in Pic%. A qualification instruction evaluates an invocation (variable reference, table indexing or function application, see section 4.2) in the given dictionary object. To follow the fashion of popular object-oriented languages like C++, Java and Python, Pic% uses the dot-operator syntax to express qualification. For a few examples, take a look at listing 4.7.

This simple object model, however, has a notable disadvantage. When a *constructor* function like `Counter` creates a new object, this object will have bindings for its instance variables, as well as for all the methods associated with it. Thus, the method bindings are duplicated in each instance. Figure 4.3 illustrates this code duplication

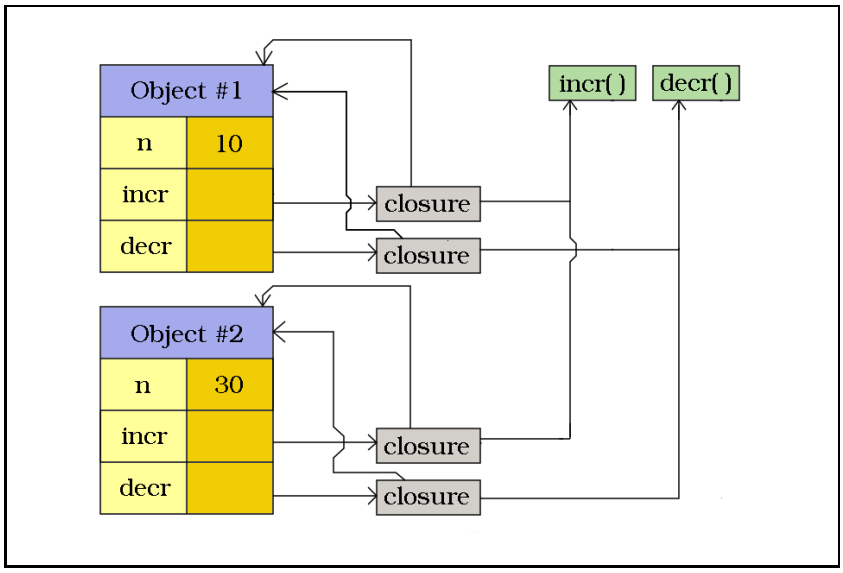

Figure 4.3: Object state duplication with the simple object model [Lie05].

graphically.

4.4.2 A More Advanced Object Model

The problem of method entry duplication is solved in Pico% by using a *prototype-based* model. In such a model, some objects contain method entries, while other objects contain a reference to the object holding those method entries. The object containing the entries is called a prototype. The reference to the prototype is typically called the *delegation link* or *parent* link [Lie86]. In some languages, it is the responsibility of the programmer to create this link, but in most languages this reference is created automatically through a process called *cloning*, a reuse mechanism comparable with instance creation in class-based languages. The clone operator creates a new object, and copies the attributes of the prototype into this new object. However, as we have seen in the previous section, not all of them need to be copied. Some should remain shared with the prototype, so prototype-based languages that use this approach need a mechanism to distinguish between the two types of bindings.

Pic% achieves this by differentiating between *declaration* and *definition* for binding variables in a dictionary. The semantics of the definition operator (`name:value`) is not altered; just as in Pico it adds a mutable key-value binding to the current environment dictionary. Dictionaries, however, are organized differently in this model. Pic% dictionaries consist of a *variable* and a *constant* part. As their names point out, variable entries can be altered, whereas constant entries are immutable. Adding an entry to the constant part of a dictionary is done by using the declaration operator (`name::value`). When a dictionary object is cloned, its variable part is deeply copied, whereas the constant part is linked using a reference. The result of this cloning

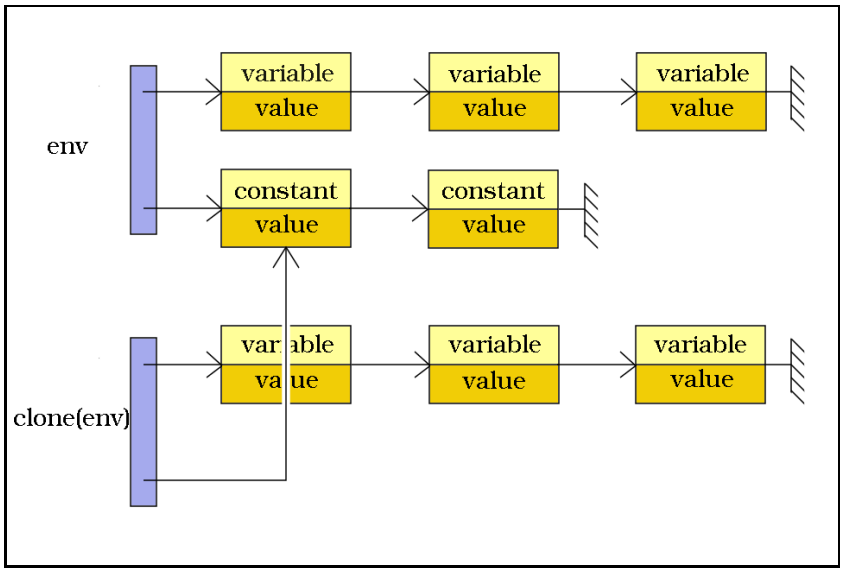

Figure 4.4: Cloning Pic% objects [DM03].

process is illustrated in figure 4.4. To handle this change in dictionary structure, the `clone` native is introduced.

To allow for object state encapsulation, Pic% also restricts message sends to variables and methods that are declared, as opposed to defined, thus aligning declaration with public visibility. Aside from the zero argument clone operator, a second `clone(object)` is provided, to clone an arbitrary given object.

In the model described above, the constant part of a dictionary contains function closures, which are bound to the object that they are defined in. Thus, when an object is cloned, these constant functions should be bound to the clone. Rebinding the closures, however, would require a deep copy of the constants, and we would once again have no code sharing. Pic% solves this problem by removing the lexical bindings of functions to their defining environment. An environment, upon reception of a message, will activate the function bound to that message with respect to itself (i.e. `self`). In the case of a self-send (i.e. the absence of an explicit receiver), `self` will be bound to the current environment. Hence, Pic% uses *dynamic scoping* in contrast with the *static scoping* of Pico.

4.4.3 Object inheritance

Inheritance in Pic% is introduced by using nested *mixin methods*. To see what mixin methods, sometimes called *modular inheritance*, are, consider the example in listing 4.8.

The code snippet defines a typical *stack* data structure. The parameter `n` specifies the maximum size of the stack, the variable `T` holds the stack's contents, and the variable

```
Stack(n):
{
  T[n]: void;
  t: 0;
  empty():: t = 0;
  full():: t = n;
  push(x):: T[t:= t + 1]:= x;
  pop()::
    {
      x: T[t];
      t:= t - 1;
      x
    };
  makeProtected()::
    {
      push(x)::
        if(full(),
           error("overflow"),
           .push(x));
      pop()::
        if(empty(),
           error("underflow"),
           .pop());
      clone()
    };
  clone()
}
```

Listing 4.8: Inheritance in Pic% through nested scopes.

`t` keeps track of the top index. The semantics of the methods `empty`, `full`, `push` and `pop` should be obvious. At the end of `Stack`'s body, the `clone` native is called to return the object.

The stack definition also contains a mixin method `makeProtected`. When `makeProtected` is called, a new environment will be created, extending the stack object it is called from. The declarations within `makeProtected` will extend the stack object with new bindings for `push` and `pop`, thus *overriding* their implementations. At the end of the definition of `makeProtected`, `clone` returns a new protected stack object. Super sends are also possible, by using the `.message()` syntax, which starts method lookup in the parent object.

4.5 Sic%

Sic% [Gyb04] is a Smalltalk implementation of the Pic% language. As such, it implements all the properties of Pic% we have discussed in the previous sections. An additional feature of Sic% is its linguistic symbiosis with the underlying Smalltalk system, which allows Pic% and Smalltalk objects to seamlessly send each other messages. We will discuss the conceptual model for linguistic symbiosis on which Sic% is based in the next section.

4.6 A Conceptual Model for Linguistic Symbiosis

As we have seen in section 3.3, it is sometimes necessary for different computer languages to interact and establish some inter-language communication. There exist different techniques to accomplish this; some are bidirectional while others work in only one direction. In all techniques, however, a mutual intent to make the interoperability between the languages as *transparent* as possible, can be identified. If this need for transparency becomes a key concern, the participating languages can become so closely entangled that they engage in a *symbiotic* relationship [Lie05].

The term *linguistic symbiosis* was originally defined by Ichisugi et al. in the work on `RbCl` [IMY92], a concurrent programming language that engages in a symbiotic relationship with its implementation language. This concept was further refined by Steyaert [Ste94].

4.6.1 Overview of the Model

Gybels et al. [GWDD06] define linguistic symbiosis between two languages as follows:

> Two languages are in linguistic symbiosis when they can transparently exchange data and invoke each other's behaviour.

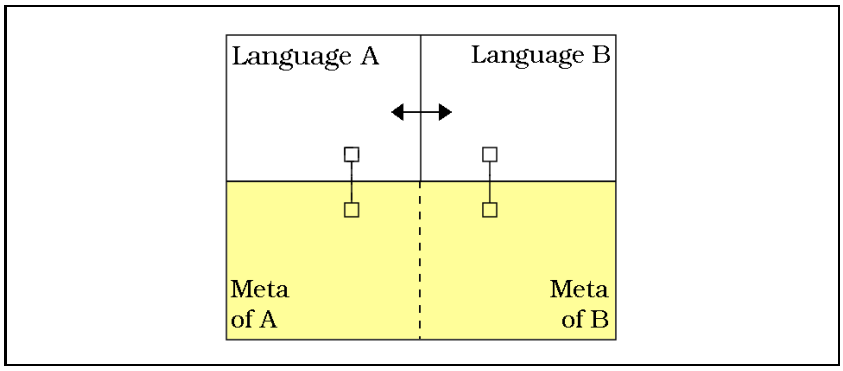

Figure 4.5: Conceptual overview of linguistic symbiosis between two languages A and B, showing both base and meta levels [GWDD06].

This means that through linguistic symbiosis (see figure 4.5), data of one language can be represented in the other, and behaviour specified in one language can be invoked from the other. They propose a model to achieve linguistic symbiosis, consisting of two elements, a *data mapping* and a *protocol mapping* that is needed between the two languages:

Data mapping. The process of passing data between programs written in different languages should be as transparent as possible. Therefore it is important that data of language B, when passed to language A is, syntactically speaking, not distinguishable from native data of language A. Thus, from within a language it should be possible to apply operations on such passed data as if it was native data. This requires a translation of these operations to the language from where the data was passed.

Protocol mapping. Making the data of language B accessible for programs written in language A is accomplished by making it possible for the meta representations of that data to be passed between the interpreters of the languages. Since the meta representations of both languages will usually understand different protocols, it is necessary to make the passed meta representations understand the meta operations of the interpreter they are passed to. The data mapping at the syntactic level thus actually boils down to a protocol mapping at the language implementation level.

This linguistic symbiosis model is mere a conceptual framework that needs to be instantiated. It does not say how protocol mappings for concrete languages can be established; to show how this might be accomplished, we will discuss the data and protocol mapping for a concrete case.

4.6.2 Linguistic symbiosis between Pico% and Smalltalk

For Pic% and Smalltalk to participate in a symbiotic architecture, there should exist transparent ways for exchanging data and invoking behaviour, according to the definition of linguistic symbiosis. Exchanging data means it should be possible to pass Pic% objects to Smalltalk and the other way around to pass Smalltalk objects to Pic%. It

```
Smalltalk defineClass: #NeonSign
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames:
    classInstanceVariableNames: ''
    imports: ''
    category: ''

NeonSign>>receiveAdvertisement: ad
    Transcript show: (ad getText)
```

Listing 4.9: A Smalltalk implementation of a `NeonSign` component.

```
{
  sign: NeonSign.new();

  MakeAd():
    { getText():: "lieven";
      clone()
    };

  anAd: MakeAd();
  sign.receiveAdvertisement~(anAd)
}
```

Listing 4.10: Example of the language symbiosis between Smalltalk and Pic%

should also be possible to send messages to Pic% objects from within Smalltalk and to send messages to Smalltalk objects from within Pic% to satisfy the requirement of the definition concerning behaviour invocation. These processes should be transparent: within Pic%, a Smalltalk object should appear as a Pic% object that can be sent messages in the same way native Pic% objects can be sent messages. The same should be true for Pic% objects in Smalltalk programs.

An example

Listing 4.10 shows an example of a Pic% program that uses linguistic symbiosis with Smalltalk, using the Smalltalk class shown in listing 4.9:

- The first expression defines a variable `sign` to hold a newly created instance of the Smalltalk class `NeonSign`: the reference `NeonSign` will return the Smalltalk class as a Pic% object to which the message `new` is sent.

- The second expression defines a function `MakeAd` which can be called to produce an object with only one method: `getText`.
- The third expression defines a variable `anAd` to hold an instance of a new Pic% object.
- The last expression sends the message `receiveAdvertisement` to the sign with the Pic% object as argument. Since the `sign` variable contains a Smalltalk object, the message and its argument are passed to Smalltalk, which means a Pic% object is passed to Smalltalk.

Data Mapping

Accessing Smalltalk objects from Pic%. Smalltalk classes are accessible from within Pic% as *regular* Pic% objects. To create new instances, the constructors are invoked through Pic% methods. Listing 4.10 shows how a Smalltalk class is accessed from Pic%. When the name of a Smalltalk class is used as a reference, the variable lookup will return this class as a Pic% object. New instances can then be created by using Pic% messages like `new()`. Smalltalk objects can also appear in Pic% if they are passed as arguments to messages to Pic% objects.

Passing Pic% objects to Smalltalk. The only possibility for Pic% objects to wind up in Smalltalk is when they are used as arguments in messages to Smalltalk objects from within Pic%. In the last expression of the code of listing 4.10, the message `receiveAdvertisement~()` is sent to the Smalltalk object `NeonSign`, with the Pic% object `anAd` as argument.

4.6.3 Linguistic Symbiosis at the Meta Level

As we have already explained in section 4.6.1, linguistic symbiosis provides a data mapping at the base level which is implemented as a protocol mapping at the meta level. The interpreters of the languages should be able pass their meta representations to each other, and apply their own meta operations on meta representations coming from another interpreter. This means the protocols of the different representations have to be mapped to each other. This is graphically illustrated in figure 4.6: the meta level contains the meta representations of data of language A and language B, and on this meta level, the protocol differences of these representations are to be resolved.

We will now show how this is accomplished in Pic% and Smalltalk. The choice of the meta language in which this interpreter are written doesn't really matter for showing how the protocol mapping at the meta level works. We will however show, in the next section, how the conceptual model explained here is used in the actual implementation. In that case one of the two languages is implemented in the other one and there is no clear separation between the meta level and base level. To clearly show the difference in how the mappings occur then, we already use one of the two base languages, namely Smalltalk, on the meta level as well. The important point here is that there is a clear separation of the base and meta levels, and that a common language is used on the meta level for the two base level languages.

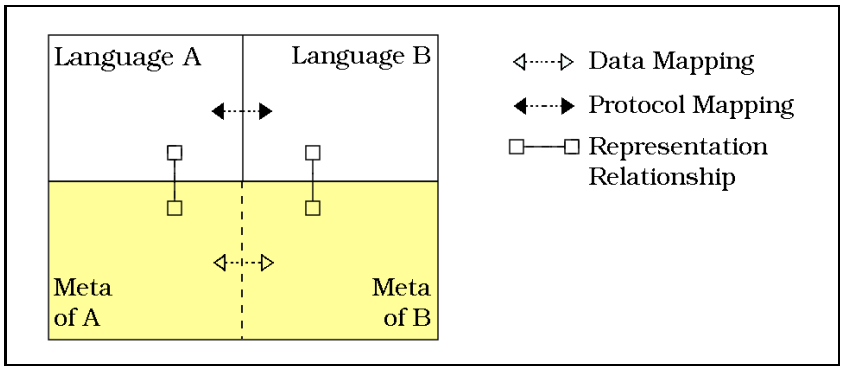

Figure 4.6: Linguistic symbiosis between two languages A and B at the meta level: A and B have meta-level representations that have different protocols that need to be bridged [GWDD06].

On the meta level, there are two interpreters, one for Pic% and one for Smalltalk. As we have assumed Smalltalk to be the meta level language for the implementation of these interpreters as well, these interpreters are written as a number of cooperating objects of different classes. Two important classes are the ones that implement the base level objects themselves: a class `SmalltalkObject` and a class `PicooObject`. Instances of these classes are thus meta level objects that represent base level objects.

Each of the two classes of meta objects understands a fairly similar protocol that implements the message sending of the base level. Both `SmalltalkObject` and `PicooObject` have methods that are the implementations of base level message sending. Of course, this protocol is similar but not entirely the same: the class `SmalltalkObject` supports the meta operation `sendSelector: withArguments:` while the class `PicooObject` supports the meta operation `sendMessage:withArguments:` [4].

As is shown in figure 4.7, the data mapping of the base level can be split in *left* and *right* relationships which allow base language data of one language to appear in the other language. On the meta level, there are meta representations for this base data, and the *left* and *right* relationships of the base level require equivalent protocol mapping relationships at the meta level. A clean equivalent relationship and way of implementing the symbiosis is to introduce wrapper classes that take care of mapping the protocol differences: in the case of Pic% and Smalltalk, a class `SmalltalkWrappedPicooObject` and a class `Pic%WrappedSmalltalkObject` can be introduced. Instances of these classes repectively wrap around a `PicooObject` instance and support the `SmalltalkObject` protocol, or wrap around a `SmalltalkObject` instance and support the `PicooObject` protocol. So for example in the figure, the base level Smalltalk object labeled *(1)* is represented by the meta object labeled *(2)* and appears in Pic% as a Pic% object *(3)*, which is implemented as a `Pic%WrappedSmalltalkObject` wrapper around the `SmalltalkObject` instance *(2)*. One desirable property of the *left* and *right* relationships is that they cancel each other out: applying the *right* relationship to a wrapped meta object produced by the *left* relationship should yield the original meta object, and

[4] There is not really a protocol difference here, but the different names will suffice to explain the concept of the model and how the mappings should be specified.

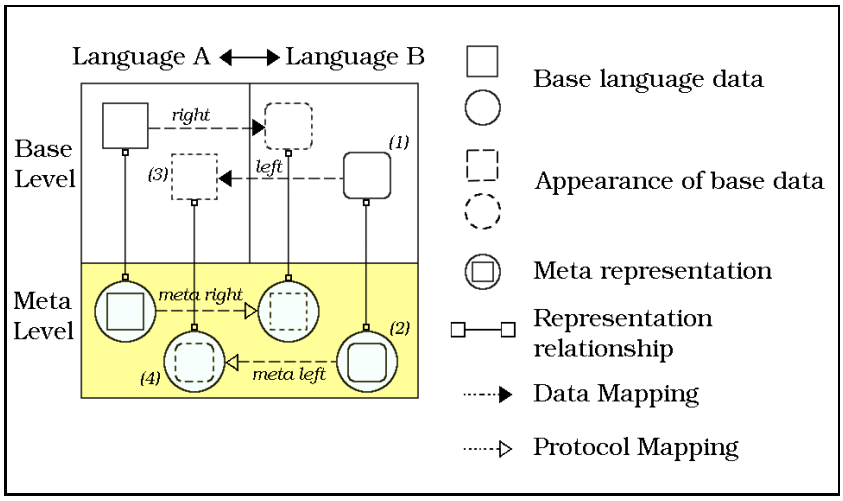

Figure 4.7: Linguistic symbiosis in more detail, focusing on the left and right appearance relationships and their equivalent relationships on the meta level [GWDD06].

vice versa.

The protocol mappings used on the meta level between Pic% and Smalltalk meta objects should have the following effect on the base level for messages between Pic% and Smalltalk objects:

Sending messages from Pic% to Smalltalk objects. In Pic% variables, Smalltalk objects appear as regular Pic% objects, and can be sent messages in the same way as other Pic% objects. For this to work, a mapping is needed that maps messages sent to Smalltalk objects in Pic% to Smalltalk messages, taking into account the different syntax used for messages in the two languages. Smalltalk messages consist of multiple keywords, whereas Pic% messages consist of a single name. An Pic% message is thus constructed from a Smalltalk message by concatenating the keywords. This is normally done in Smalltalk by appending colons to each keyword. For instance, the message `at: index put: value` is represented by `at:put:`. It is however impossible to use this symbol as a message qualifier in Pic%, because the parser cannot handle colons in symbols, as the colon is reserved for definition. As a replacement character, the tilde character (∼) is adopted. As such the Smalltalk message `data at: 10 put: 'lieven'` is translated into Pic% as `data.at put (10, "lieven")`. The same mapping is used for invoking constructors on classes.

Sending messages from Smalltalk to Pic% objects. When contained in Smalltalk variables, Pic% objects can be sent messages to by Smalltalk objects in the same way as the latter would send messages to other Smalltalk objects. A Smalltalk message sent to a Pic% object is constructed from a Pic% message in the inverse way as described above.

A critical point in the mappings performed by the protocol wrappers is to ensure that the appropriate left and right relationships are applied when mapping arguments from one protocol to the other. When a `SmalltalkWrappedPicooObject` maps a

receiver sendSelector: *#name*
withArguments: {*argument1*, *argument2*, ... *argumentn*}
==
right[*result*]

⇳

left[*receiver*] sendMessage: *#name*
withArguments: {left[*argument1*], left[*argument2*], ...
left[*argumentn*]}
==
result

Figure 4.8: Semi-formal description of meta operation mapping from Smalltalk to Pic%

Smalltalk `sendSelector:withArguments:` operation to the Pic% `sendMessage:withArguments:` operation, the arguments involved in the message send are Smalltalk meta objects which also need to be converted to Pic% ones. Thus, the mapping done by this wrapper semi-formally comes down to what is shown in figure 4.8.

The rule simply describes the same protocol mapping solution for sending Smalltalk messages to Pic% objects as described above, but illustrates the point of needing to convert the receiver and arguments to Pic% objects. Applying the left relationship on the receiver, which is in this case the `SmalltalkWrappedPicooObject` wrapper, simply results in the unwrapped Pic% meta object. Similarly, the left relationship applied to the arguments either wraps them or unwraps them, depending on whether they were wrapped Pic% meta objects produced by the right relationship, or plain Smalltalk meta objects in the first place. As also illustrated, the result of the mapped message also needs to be mapped back using the *right* relationship to turn in from a Pic% object into a Smalltalk object.

The converse rule for mapping Pic% messages to Smalltalk messages is very similar and can be without further explanation as illustrated in figure 4.9. Note that this rule is easily derived from the rule above using the fact that the *left* and *right* relationships cancel each other out (i.e. $left[right[x]] = x$).

4.6.4 Actual Implementation

The conceptual model for linguistic symbiosis explained in the previous section is readily applicable to actual implementation schemes where the two languages in symbiosis are implemented as interpreters in a third common implementation language. There is however a differing scheme possible, namely that the interpreter of one language is written in the other language, and that a linguistic symbiosis is defined between the first language and its implementation language rather than with a language that is also implemented in that implementation language. In this section we will explain how to conceptual model maps to this scheme.

receiver sendMessage: *#name*
withArguments: {*argument1*, *argument2*, ... *argumentn*}
==
left[*result*]

⇳

right[*receiver*] sendSelector: *#name*
withArguments: {right[*argument1*], right[*argument2*], ...
right[*argumentn*]}
==
result

Figure 4.9: Semi-formal description of meta operation mapping from Pic% to Smalltalk

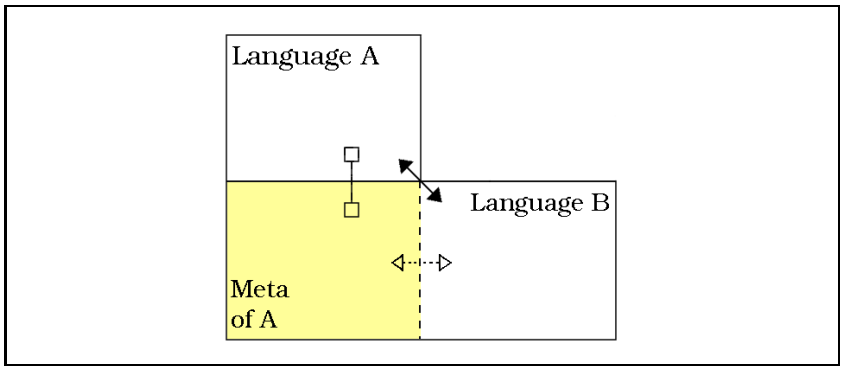

Figure 4.10: Implementation of language symbiosis [GWDD06].

This scheme essentially means that the meta level of one language is made to overlap the base level of the other language. As is illustrated in figure 4.11, the interpreter for the one language is written in the other language, and there is no interpreter for the other language at this level. One reason for this variation of that in practice it is typically easier to implement a new language in the one with which it should be in linguistic symbiosis, rather than in a common language, or that such an implementation already exists. Note that while we already used one of the base languages as meta language as well in the explanation of the conceptual model, we still made a distinction between the meta level and the base level. The variation we are referring to here is that, as illustrated in figure 4.11, the meta representations of one language - Pic% in the figure - exist one the same level as the values of the other language. This deviation of the conceptual model has an effect on how the linguistic symbiosis is actually implemented , as we will discuss in more detail in the remainder of this section.

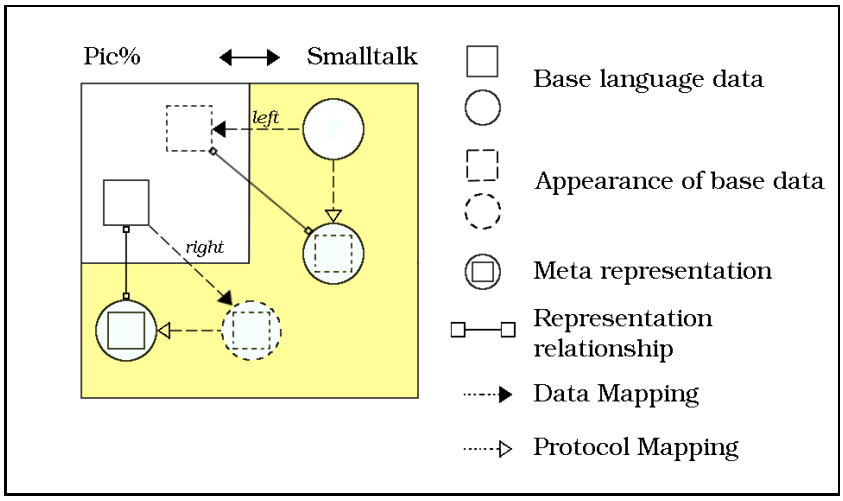

Figure 4.11: Folding of language symbiosis in the actual implementation of Pic% in Smalltalk [GWDD06].

Linguistic Symbiosis Implementation

One effect of the overlap is that the *right* relationship maps a Pic% value *directly* to a wrapper. Contrast this with the pure conceptual model of figure 4.7, where the *right* relationship allows a Pic% value to appear in Smalltalk, and this appearance is implemented as a wrapper around the meta object representing the value. Here, the *right* relationship maps directly to the wrapper. Furthermore, this wrapper is a base level Smalltalk object, rather than a meta level object as in the pure conceptual model. Thus the wrapper translate *base* level Smalltalk messages to *meta* operations on the Pic% meta object.

The important point to note about this difference in how the mappings work is that the mappings in the conceptual model more clearly show the protocol difference that is being solved. This is the reason for clearly separating the base level and meta level in the conceptual model. Because in the actual implementation, wrappers map between base level and meta level operations, this difference is no longer as obvious. The mapping performed by the *right* wrappers in the actual implementation for Pic% and Smalltalk is for example the one given in figure 4.12. Note again that the *base* level Smalltalkmessage with name `name:` is mapped to the Pic% meta operation `sendMessage:withArguments:`, while previously it was the Smalltalk *meta* operation `sendSelector:withArguments:` that was mapped to the Pic% meta operation.

The *left* operation is similarly affected. Making a Smalltalk object appear in Pic% for example involves wrapping the Smalltalk object in a wrapper that maps the Pic% meta protocol to the Smalltalk base level, instead of as in the conceptual model where it maps it to the Smalltalk meta protocol.

receiver name: {*argument1*, *argument2*, ... *argumentn*}
==
right[*result*]

⇳

left[*receiver*] sendMessage: *#name*
withArguments: {left[*argument1*], left[*argument2*], ...
left[*argumentn*]}
==
result

Figure 4.12: Semi-formal description of base and meta level operation mapping in actual implementations with overlap of base and meta levels.

Chapter 5

Pic% as a Scripting Language

In chapter 3, we discussed Piccola, a programming language especially designed for adapting and composing software components written in other languages. Despite software composition being the language's primary purpose, there are a few potential pitfalls, which might introduce unexpected behaviour or lead to malfunctioning compositions. We will discuss these problems and the situations in which they might occur in the following sections. Comparable experiments as those that cause trouble when using Piccola are performed using Pic%, the small, prototype-based language, which we have discussed in chapter 4. More particularly, we will use Sic%, introduced in section 4.5, because it has a symbiotic relationship with the underlying Smalltalk system.

Piccola was originally implemented in Java, but we will use the same Smalltalk components for the Piccola experiments as for their Pic% counterparts. This is not a problem, since there exists a Smalltalk implementation of Piccola as well, called SPiccola, which can access external Smalltalk components instead of Java ones.

5.1 Piccola Black-Box Problem

As we have seen in section 3.4, a Piccola form representing an external component has a nested structure consisting of two parts. The top level part represents the Piccola interface of the object. This form contains a label `peer`, which is bound to a nested form representing the identity of the external object.

The programmer can declare and implement a specific interface in Piccola, in which the `peer` label can be used to send messages to the peer form and as such to the external component This glue code is thus actually nothing more than a simple wrapper around the external component. Such a wrapper uses *message forwarding*, also known as *consultation* [KRC91], to interact with the object it wraps. Since most of the components used in Piccola are external *objects*, it is possible to run into the so called *self-problem* [Lie86].

In inheritance systems, the pseudo-variable `self` is automatically bound to the receiver of a message when the code of the method associated with that message is executed. During method lookup, the self variable is not rebound when going from a class to its superclass. As such, self sends that occur in methods implemented in superclasses will go to the initial object. When a user sends a message, however, self is always rebound.

To see why this can lead to problems, consider the Smalltalk classes `Button` and `Juggler` shown in listing 5.1. The purpose of a `Button` object is to attach another object to it, which the `Button` object will notify when it gets clicked. Attaching an object is possible by using the method `attach:`, which assigns its argument to `Button`'s instance variable `attachedObject`. Since the `Button` class is a subclass of `UI.ApplicationModel`, it has to specify a method that will be invoked when a displayed instance gets clicked. The `push` method is implemented for this purpose. This method just forwards the message `push` to the attached object held by the `attachedObject` variable.

The `Juggler` class contains three methods, `startJuggling`, `stopJuggling`, and `dropBall`. To keep it clear and simple, their implementations only print a string to the `Transcript`. Note that `dropBall` contains a self send.

We want to attach an instance of the `Juggler` class to a `Button` object such that every time the button is clicked, the juggler receives either the message `startJuggling` or the message `stopJuggling`, depending on which one of these it received last [Van04]. Merely passing the `Juggler` instance as an argument to `Button`'s `attach:` method will cause problems because the `Juggler` class does not contain a method called `push`. Furthermore, if the juggler receives a `startJuggling` or `stopJuggling` message, it will execute the implementations of these message selectors even if the juggler respectively has already started juggling or is not juggling. We thus want to adapt a `Juggler` object such that it keeps track of the fact whether it is juggling or not, and contains a push method that invokes either the `startJuggling` method or the `stopJuggling` method according to the current state.

The Piccola code shown in listing 5.2 shows how a specific interface is declared. The wrapJuggler service will receive the generic form created by the inter-language bridge as an argument. Note that we extend this generic form with the specific interface form, so all the labels of the generic interface that are not overridden by the specific interface can still be projected on. After registering the wrapper service, every `Juggler` component instance that is passed from Smalltalk to Piccola is adapted to contain the services `push` and `alterState`. The wrapper also adds the local label `state`, which contains a variable that will alternately be bound to true or false depending on whether the juggler is juggling or not. The bindings `startJuggling` and `stopJuggling` override the equally labeled services of the generic interface. They first update the juggler's state and then invoke the external component's methods with the same name through the peer form.

Now suppose the `startJuggling` service of the form contained by the `juggler` label has been invoked. The service `dropBall` of the generic interface was not overridden, so when this service gets invoked, the message `dropBall` will be forwarded to the external component. As can be seen in listing 5.1, the execution of the appropriate method in the Smalltalk class will send a `stopJuggling` message to the external

```
Smalltalk defineClass: #Button
    superclass: #{UI.ApplicationModel}
    indexedType: #none
    private: false
    instanceVariableNames: 'attachedObject'
    classInstanceVariableNames: ''
    imports: ''
    category: ''

Button>>attach: anObject
    attachedObject := anObject
Button>>push
    attachedObject push

Smalltalk defineClass: #Juggler
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: ''

Juggler>>startJuggling
    Transcript show: 'Started juggling'
Juggler>>stopJuggling
    Transcript show: 'Stopped juggling'
Juggler>>dropBall
    Transcript show: 'Dropped my ball'.
    self stopJuggling
```

Listing 5.1: Implementation of Smalltalk classes `Button` and `Juggler`.

object itself. Consequently, the `state` variable in the glue code will not reflect the real state of the juggler any longer.

We can adapt the Smalltalk `Juggler` component of listing 5.1 in a similar way using Pic%, as is illustrated in listing 5.3. The native function `extend`, introduced by Lievens [Lie05], returns a new object that is an extension of a given object, extended with the code given as a second parameter, thus basically acting as a mixin [BC90]. As we have discussed in section 4.4.2, Pic% uses prototype-based delegation as a reuse mechanism. As a consequence, the extended object will delegate to the Smalltalk object. Due to the late binding of self, a `dropBall` message sent to the `jugglerToggle` object will invoke the `stopJuggling` method of `jugglerToggle` instead of the one of the external object. The difference between the Piccola mechanism and the Pic% mechanism is illustrated in figure 5.1. It is however not possible to let these adaptations be carried out automatically when the object passes the language barrier.

```
wrapJuggler Juggler:
  'state = newVar()
  'state.set(false)
  alterState:
    state.set(state.get().not())
  startJuggling:
    alterState()
    peer.startJuggling()
  stopJuggling:
    alterState()
    peer.stopJuggling()
  push:
    if state.get()
      then: stopJuggling()
      else: startJuggling()

registerWrapper "Smalltalk.Juggler" wrapJuggler

juggler = Host.class("Smalltalk.Juggler").new()
button = Host.class("Smalltalk.Button").new()
button.add(juggler)
```

Listing 5.2: Adapting and plugging together external components in Piccola.

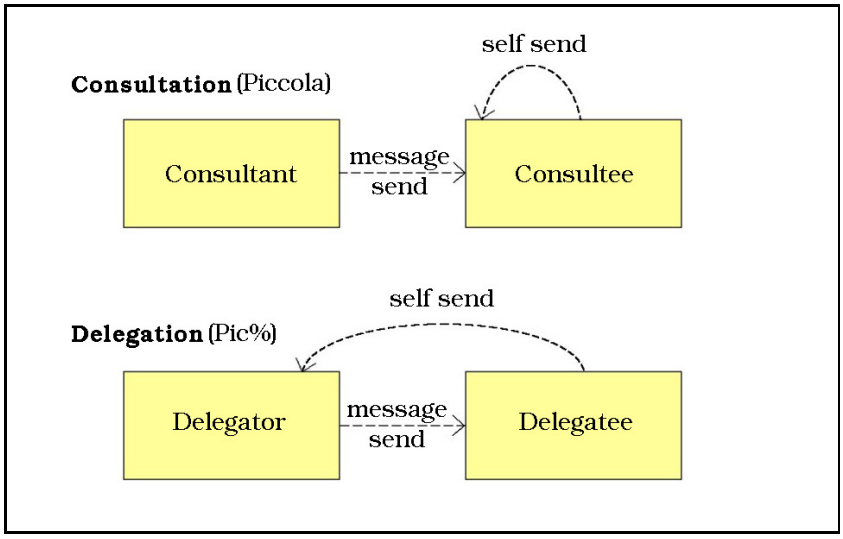

Figure 5.1: Self-sending semantics in Piccola and Pic%.

5.2 Piccola's Bridging Strategy vs. Linguistic Symbiosis Model

The bridging strategy used in Piccola is conceptually similar to the linguistic symbiosis model. Programs written in Piccola should be able to access components written

```
{
  juggler: Juggler.new();

  jugglerToggle: extend(juggler,
    { state: false;
      push():: {
        if(state, stopJuggling(), startJuggling())
      };

      alterState():: { state:= not(state) };
      startJuggling():: if(not(state),
                           { alterState();
                             .startJuggling() },
                           false);
      stopJuggling():: if(state),
                           { alterState();
                             .stopJuggling() },
                           false);
      clone()
    });

  button: Button.new();
  button.add~(jugglerToggle)
}
```

Listing 5.3: Component adaptation and composition using Pic%.

in Smalltalk. There still are, however, a few problems with the revised bridging approach we have seen in section 3.4. These problems do not occur when implementing linguistic symbiosis as proposed by the conceptual model discussed in the previous chapter. We will therefore, in this section, compare Piccola's inter-language bridge with the model, and analyze what is wrong with the Piccola approach. These are not mere theoretical problems, since they affect the way adapted components can be used or plugged together.

The Piccola bridging strategy and the linguistic symbiosis model utilize different terms for the processes of passing data of one language to another language: the *up* and *down* operations we have discussed in section 3.3 are respectively equivalent to the *left* and *right* operations of the linguistic symbiosis model.

Piccola has a symbiotic relationship with the language in which it is implemented, and as such we should compare it with the variation of the conceptual model, which we have described in section 4.6.4.

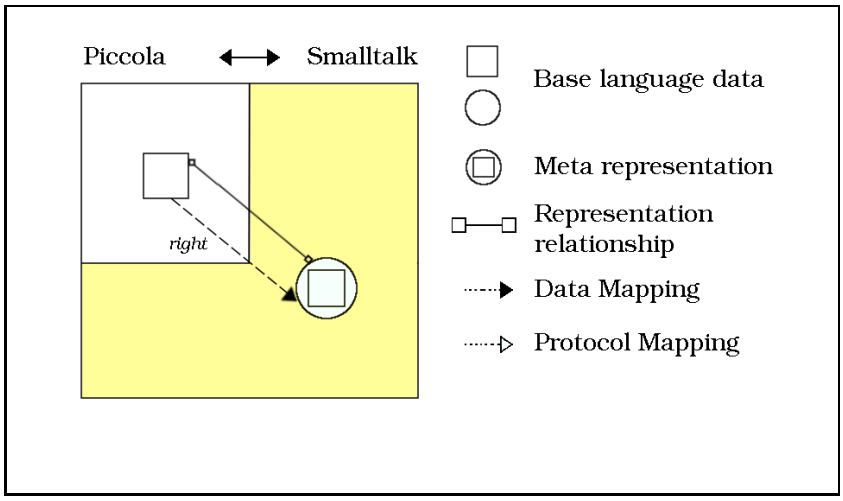

Figure 5.2: Passing a plain Piccola form to Smalltalk in linguistic symbiosis terms.

5.2.1 Passing Plain Forms to Smalltalk

Plain Piccola forms are forms that do not represent an external component, and were created inside Piccola. The process of passing down such forms to Smalltalk is specified in the revised bridging strategy as "a plain form is passed down as it is". In terms of the linguistic symbiosis model, "as is" means that when the form is passed to Smalltalk, its meta representation will be used on this level. This is graphically illustrated in figure 5.2. Contrast this with the linguistic symbiosis between Pic% and Smalltalk as shown in figure 4.11.

To understand why this can be a major problem, consider the snippet of Piccola code shown in listing 5.4. A Piccola form `person` is defined with the local label `name` and the service `getName`, which returns the value bound to the label `name`. The Smalltalk class `VisitingCardMaker` is instantiated and the service `printCard` is invoked with the `person` form as argument. Because no specific interface is specified for the external form, the message will be forwarded to the external object. This means the argument form will be passed to Smalltalk. Now suppose the external object's implementation of `printCard` sends the `getName` message to its argument, in this case the `person` form. While this would work in Pic%, which is based on the linguistic symbiosis model, it does not have the expected behaviour in Piccola.

When the `person` form is passed to Smalltalk, it winds up there as its meta representation. This representation can receive Piccola meta operations but can neither understand the Smalltalk message `getName` nor map this message to the `getName` Piccola service.

This problem arises from the fact that Piccola exchanges data with the same language as it is implemented in: Smalltalk. Piccola's meta representations thus exist on the same level as the regular Smalltalk values. We have seen that for this to work, the Piccola meta representations should be wrapped by base level Smalltalk objects

```
person =
  'name = "lieven"
  getName: name

vcm = Host.class("Smalltalk.VisitingCardMaker").new()
vcm.printCard(person)
```

Listing 5.4: Passing a plain Piccola form as an argument to a Smalltalk method.

when passed down. These wrappers are to map base level Smalltalk messages to meta operations on the Piccola meta object.

Because the meta representation of a passed down form is not wrapped, it only understands Piccola meta operations. This means a Piccola form cannot be sent regular Smalltalk messages as if it was a Smalltalk object.

5.2.2 Passing an External Form back to Smalltalk

The Piccola code shown in listing 5.2, has another flaw besides the one mentioned in the previous section. If the button generated by that piece of code would be clicked, no `push` method would be found in `juggler`, which was assigned to the instance variable of the `button` component. Although we have written code to make sure `Juggler` components are adapted if they are passed to Piccola, these adaptations get lost when passing the adapted component back to Smalltalk.

The problem is that the glue code to adapt an external component, is contained in the interface form, which is never passed to Smalltalk. If an adapted external component is passed back to Smalltalk, first the interface form is stripped off, and only the peer form, the Piccola level representation of the external component, is passed back to Smalltalk. This results in the original component winding up in Smalltalk again. These processes are illustrated in terms of the linguistic symbiosis model in figure 5.3.

Compare this illustration with figure 5.4, which shows what happens when an external component adapted in Pic% is passed down to Smalltalk again. We have already seen an example of this behaviour in the Pic% code of listing 5.3. A component is adapted by creating a new *regular* Pic% object, which delegates to the Smalltalk component. Thus, when passing this regular Pic% object to Smalltalk, its meta representation will be wrapped to map regular Smalltalk messages to Pic% meta operations. If a method cannot be found in the regular Pic% object, it will delegate to the external component. This means the adaptations specified in Pic% are preserved when the adapted component is passed to Smalltalk again.

A related problem is that the *up* and *down* operations for respectively passing Smalltalk objects to Piccola and passing these objects back from Piccola to Smalltalk do not always have the property of cancelling each other out. Schärli [Sch01] says that the *up* operation, or *left* operation from the viewpoint of the linguistic symbiosis model, can

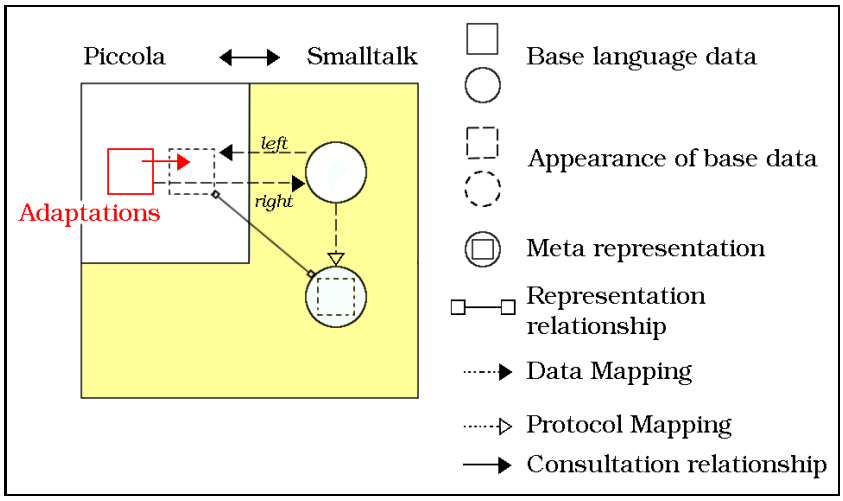

Figure 5.3: Passing an external component between Piccola and Smalltalk in linguistic symbiosis terms.

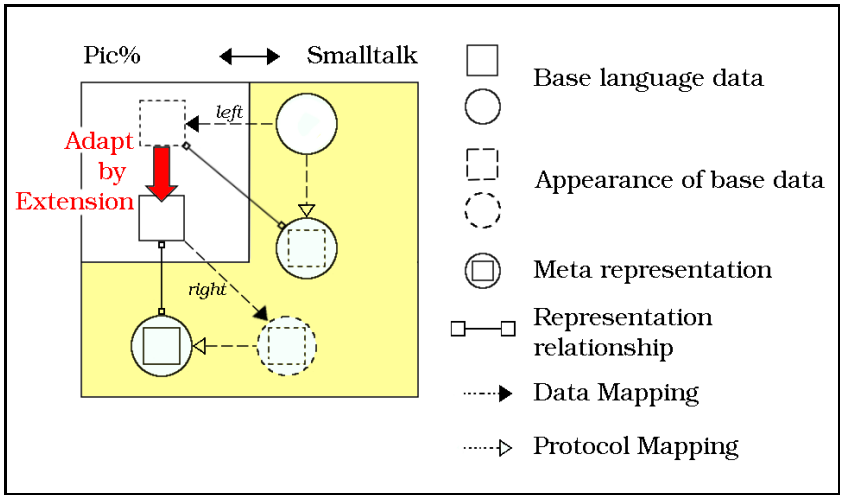

Figure 5.4: Adapting an external component in Pic% and passing it back to Smalltalk.

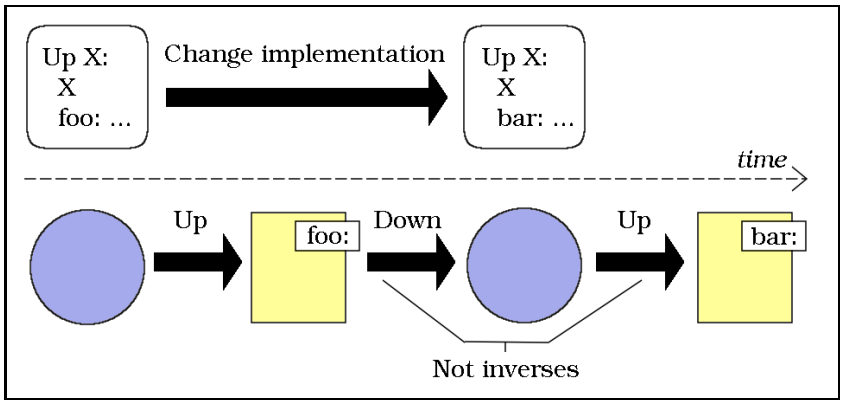

Figure 5.5: Passing a component up and down while changing the *up* operation.

be altered dynamically. Now suppose an external component is passed up to Piccola, back down to Smalltalk and finally up again to Piccola. If in between the down process and the last up process, the specification of the up operation is changed, these two last operations will not be inverses.

This problem is exemplified in figure 5.5: The `up` operation is altered to add a label `bar` instead of a label `foo`. Passing down a form that was created with the first version of the `up` operation, and then passing it up to Piccola again, after the changes to the operation, will not result in the same form that was passed to Smalltalk.

5.3 Separation of Interface Mapping and Behavioural Adaptations

As we have seen in section 3.4.1, Piccola provides a generic interface for components that are passed from another language. It is however possible to alter the *up* operation in a reflective manner, so that this generic interface is replaced by a specific one. Such reflective control over the *up* operation might also prove very valuable in Pic%, when using the language and its linguistic symbiosis relationship for component composition.

However, despite the fact that the form created by the altered *up* operation is called the specific interface form, such a form may also adapt the component to contain new functionality. We believe that when adapting components, it might be useful to be able to separate glue code that purely implements interface mappings and glue code that adds new functionality to components, like the code in listing 5.2.

To understand when this feature might prove valuable, consider the illustration in figure 5.6. An external component instance *(1)*, some `Collection` type, is passed up to Pic% and appears as a regular Pic% object at that *(2)*level. We first extend this object with code containing an interface mapping, resulting in a new Pic% object *(3)*. This object will understand the message `add`, which is mapped to the method `at~put~`.

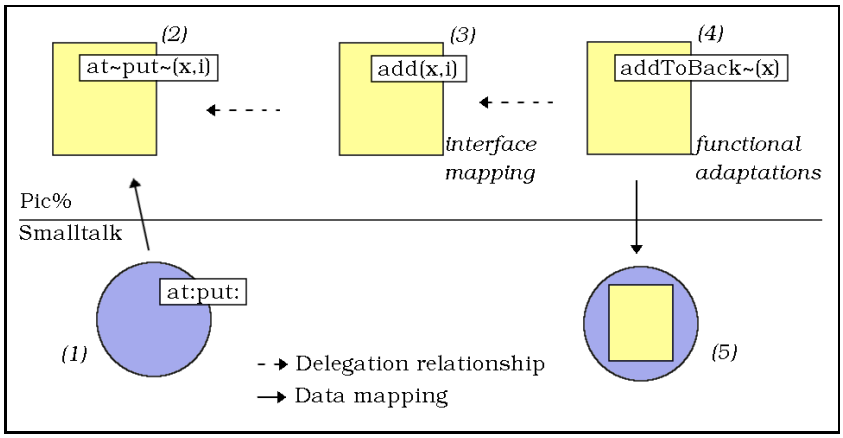

Figure 5.6: Separation of interface mappings and behavioural adaptations.

We then again extend this new object, now with glue code that adds new behaviour to the component. The method `addToBack~` will add its argument to the back of the collection and is implemented in terms of the `add` method. Finally, the resulting object *(4)* is passed to Smalltalk again and winds up there as a wrapped Pic% meta representation *(5)*.

We now want the regular Pic% object that contains the behavioural adaptations *(4)* to understand the messages declared in the interface mapping. A reason for declaring such an interface mapping might be to provide the component with an interface that suits the used compositional style. When this object is passed to Smalltalk *(5)*, however, we do not want the object to understand the messages declared in the interface mapping. It makes more sense, on this level, to send the Smalltalk messages from the original component when we want to invoke that behaviour. This prevents the interfaces on the Smalltalk level from getting cluttered.

When sending a message to the adapted component *(5)* from within Smalltalk, the Pic% object *(4)* will not delegate to the object containing the interface mappings *(3)*, but directly to the representation of the original component *(2)*, and as such to the original component. When sending a message to the adapted component from within Pic%, the interface object *(3)* will be delegated to, making the methods that perform the mappings accessible.

Chapter 6

Conclusions

6.1 Summary

The aim of this dissertation was to discuss the benefits of using linguistic symbiosis to let scripting and composition languages transparently exchange data with other languages.

In chapter 2, we discussed component-based development. We presented a brief history of the field and clearly defined what is meant by the terms component and component framework. The term scripting was defined and an overview of the features that a scripting language should provide was given. We also discussed the notion of glue code, which is different from other scripting code in that it adapts components that do not satisfy certain requirements. Finally, a conceptual framework for composition, which specifies that applications should be defined in terms of components, scripts, and glue, was presented.

Chapter 3 explained Piccola, a composition language that is designed with the conceptual framework for composition in mind. We first discussed the syntax and semantics of the language, followed by a presentation of its original approach to access external components. This strategy, called bridging, was identified to have some problems and so a revised version of it was also presented.

Chapter 4 presented Pico, a high-level programming language that can be seen as a derivative of Scheme. We explained its purpose, syntax, and semantics. We also introduced Pic%, a prototype-based extension of Pico, and an implementation of of Pic%, called Sic%. A special property of Sic% is its linguistic symbiosis with the underlying Smalltalk system, which allows Pic% and Smalltalk objects to seamlessly send each other messages. This led to an explanation of the conceptual model on which this linguistic symbiosis implementation is based.

In chapter 5, we contemplated the use of linguistic symbiosis to adapt and compose components written in an external implementation language. We contrast this with the bridging strategy used by Piccola, which is another approach to access external components.

6.2 Conclusions

We have seen that, when adapting an component, the glue code should not merely forward messages to this component. If new operations are invoked in the behaviour executed by the message forward, only the original component will be aware if these operations, rather than the whole adapted component. This is due to the self-problem and can be solved by using proper delegation. The glue code thus should delegate to the external component instead of consulting the latter.

When using a scripting language to compose components written in another language, we do not only want to access these components from within the scripting language. We also want to be able to pass data created in the scripting language to the component's implementation language. Typically, this is done by using such data as arguments to operations invoked on the external component. It is important to ensure that from within the component's implementation language, the behaviour specified on the passed scripting language data can be invoked. Some needed functionality can then be implemented in the scripting language itself, for example to create a prototype of a component, and be invoked from within the component language when passed to it.

After we have adapted an external component, we would like our modifications to still be reachable when the adapted component is passed back to the original component's implementation language. An approach to achieve this is by combining the features discussed in the previous paragraphs. The glue code that adapts an external component should be encapsulated as regular scripting language data, delegating to the external component. Now, if the behaviour specified in the scripting language can be invoked from within the component's implementation language, the adaptations will also be visible on that level.

Thus, when passing an adapted component back to the implementation language of the original component, the adaptations will be preserved. We can, however, differentiate between two types of adaptations. On the one hand those that are made purely to map the interface of an external component to another interface, for example to better suit the scripting language's philosophy. On the other hand, the adaptations that add new functionality to a component. It does not make much sense however, to also maintain adaptations of the former category when an adapted component is passed to the original component's implementation language, as this will only result in cluttered interfaces. We showed an approach that allows to differentiate between the two categories when implementing glue code, and will disregard the interface mappings if messages are sent from within the component's implementation language.

Bibliography

[Ach02] Franz Achermann. *Forms, Agents and Channels — Defining Composition Abstraction with Style*. PhD thesis, University of Bern, January 2002.

[ALSN01] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola — a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing — A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[AN00] Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages*, volume 1897 of *LNCS*, pages 77–89, Zürich, Switzerland, September 2000. Springer-Verlag.

[AN01] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts — A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[ASS96] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[BFVD01] Werner Van Belle, Johan Fabry, Karsten Verelst, and Theo D'Hondt. Experiences in mobile computing: the cborg mobile multi-agent system. In *TOOLS '01: Proceedings of the 38th conference on Technology of Object-Oriented Languages and Systems*. IEEE Computer Society Press, 2001.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, NJ, USA, 1941.

[CM04] Tom Van Cutsem and Stijn Mostinckx. A prototype-based approach to distributed applications. Licenciaatsthesis, Vrije Universiteit Brussel, June 2004.

[CMM+04] Tom Van Cutsem, Stijn Mostinckx, Wolfgang De Meuter, Jessie Dedecker, and Theo D'Hondt. On the performance of soap in a non-trivial peer-to-peer experiment. In *Proceedings of 2nd International Conference of Component Deployment.*, pages 215–218. Springer, 2004.

[Cox86] Brad J. Cox. *Object-Oriented Programming: an Evolutionary Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Ded06] Jessie Dedecker. *Ambient-oriented Programming*. PhD thesis, Vrije Universiteit Brussel, 2006.

[DM03] Theo D'Hondt and Wolfgang De Meuter. Of first-class methods and dynamic scope. In *Actes de LMO'03: Langages et Modèles à Objets*, 2003.

[GWDD06] Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-language reflection – a conceptual model and its implementation. *Journal of Computer Languages, Systems and Structures*, 32(2-3):109–124, July 2006.

[Gyb04] Kris Gybels. Sic%: Smalltalk implementation of pic% [online]. `http://prog.vub.ac.be/~kgybels/Sicoo/sicoo.html`, 2004.

[HÖ93] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 36–56, London, UK, 1993. Springer-Verlag.

[IMY92] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. Rbcl: A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pages 24–35, Tama City, Tokyo, November 1992.

[Kan02] Alexander Kanavin. An overview of scripting languages [online]. `http://www.sensi.org/~ak/impit/studies/report.pdf`, 2002.

[Kon95] Dimitri Konstantas. Interoperation of object-oriented applications. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 69–95. Prentice-Hall, 1995.

[KRC91] Günter Kniesel, Mechthild Rohen, and Armin B. Cremers. A management system for distributed knowledge base applications. In *Verteilte Künstliche Intelligenz und kooperatives Arbeiten, 4. Internationaler GI-Kongress Wissensbasierte Systeme*, pages 65–76, London, UK, 1991. Springer-Verlag.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPLSA '86: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 214–223, New York, NY, USA, 1986. ACM Press.

[Lie05] Wouter Lievens. A symbiosis between delegation-based and inheritance-based object-oriented programming languages. Licenciaatsthesis, Vrije Universiteit Brussel, June 2005.

[LSNA97] Markus Lumpe, Jean-Guy Schneider, Oscar Nierstrasz, and Franz Achermann. Towards a formal composition language. In Gary T. Leavens and Murali Sitaraman, editors, *Proceedings of ESEC '97 Workshop on Foundations of Component-Based Systems*, pages 178–187, Zurich, September 1997.

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

[McI60] M. Douglas McIlroy. Macro instruction extensions of compiler languages. *Commun. ACM*, 3(4):214–220, 1960.

[McI68] M. Douglas McIlroy. Mass-produced software components. In J. M. Buxton, Peter Naur, and Brian Randell, editors, *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, pages 88–98. NATO Science Committee, October 1968.

[MDD04] Wolfgang De Meuter, Theo D'Hondt, and Jessie Dedecker. Pico: Scheme for mere mortals. In *ECOOP 2004: Lisp and Scheme Workshop*, 2004.

[MDS01] David R. Musser, Gilmer J. Derge, and Atul Saini. *STL tutorial and reference guide, second edition: C++ programming with the standard template library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Meu04] Wolfgang De Meuter. *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks*. PhD thesis, Vrije Universiteit Brussel, 2004.

[MGD99] Wolfgang De Meuter, Sebastián González, and Theo D'Hondt. The design and rationale behind pico. Technical report, Vrije Universiteit Brussel, 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[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.

[NTdMS91] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey, and Marc Stadelmann. Objects + scripts = applications. In *Proceedings, Esprit 1991 Conference*, pages 534–552, Dordrecht, NL, 1991. Kluwer Academic Publishers.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[Ous98] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[Pee03] Adriaan Peeters. Language symbiosis through a joint abstract grammar. Licenciaatsthesis, Vrije Universiteit Brussel, August 2003.

[Sam97] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[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, October 1999.

[Sch01] Nathanael Schärli. Supporting pure composition by inter-language bridging on the meta-level. Diploma thesis, University of Bern, September 2001.

[Sch03] Andreas Schlapbach. Enabling white-box reuse in a pure composition language. Diploma thesis, University of Bern, January 2003.

[SN99] Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures — Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[Ste94] Patrick Steyaert. *Open Design of Object-Oriented Languages*. PhD thesis, Vrije Universiteit Brussel, 1994.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Van04] Wim Vanderperren. *Combining Aspect-Oriented and Component-Based Software Engineering*. PhD thesis, Vrije Universiteit Brussel, May 2004.

[vR95] Guido van Rossum. *Python Reference Manual*. Amsterdam, The Netherlands, The Netherlands, 1995.

[WS91] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1991.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.