CHAPTER 1

Applications = Components + Scripts

A tour of Piccola

Franz Achermann, Oscar Nierstrasz
Software Composition Group, University of Berne

Abstract. Piccola is a language for composing applications from software components. It has a small syntax and a minimal set of features needed for specifying different styles of software composition. The core features of Piccola are communicating *agents*, which perform computations, and *forms*, which are the communicated values. Forms are a special notion of extensible, immutable records. Forms and agents allow us to unify components, static and dynamic contexts and arguments for invoking services. Through a series of examples, we present a tour of Piccola, illustrating how forms and agents suffice to express a variety of compositional abstractions and styles.

1. Introduction

Piccola is intended to be a *general-purpose language for software composition*. Whereas existing programming languages appear to be suitable for implementing software components, and many scripting languages and fourth-generation languages have been developed to address the needs of particular component models, there has been relatively little work that attempts to develop a generalized approach that may span various architectural styles and component models.

We have argued elsewhere [1][24] that most object-oriented methods typically do not lead to pluggable component architectures (mainly because reuse is considered too late in the lifecycle) and that the resulting software systems can be hard to maintain and understand because they do not make the run-time architecture explicit (the source code describes the classes, not the objects). To address this problem, we have proposed a conceptual framework for software composition that can be summed up as:

Applications = Components + Scripts

Components must conform to *architectural styles* [26] that determine the *plugs* each component may have (i.e., exported and imported *services*), the *connectors* that may be used to compose them, and the *rules* governing their composition. *Scripts* define specific connections of the components. Additionally, *glue* abstractions may be required to bridge architectural styles, and adapt components that have not been designed to work together, and *coordination* abstractions may be required to manage dependencies between concurrent and distributed components.

Piccola's runtime model consists of communicating agents. The behaviours of these agents are specified by scripts. Agents invoke services and compose forms. Agents live in a *context* which contains the known services and forms for an agent. In this text we will show how components can be scripted in a declarative way by means of a *style* which defines a kind of "component algebra." Consider, for example, the well-known style of *pipes and filters:*

Components:	File, Stream, Filter	*Files and Filters are external components*
Connectors:	<, |, >	*Three kinds of pipe operators*
Rules:	Filter < File → Stream Stream | Filter → Stream Stream > File → *nil*	*A File piped into a Filter yields a Stream* *A Stream piped into a Filter is still a Stream* *A Stream can be piped into a File*

TABLE 1. **Pipes and Filters**

Pipes and filters are "algebraic" in the sense that the composition of two components yields another component.

Unlike scripting languages that offer only a fixed set of compositional styles, Piccola allows you to *define your own styles* for different application domains. Rather than develop Piccola as an extension to an existing language, we felt it was important and necessary to emphasize a *separation of concerns* between component implementation and component composition. Our goal is to identify a well-founded set of features necessary and sufficient for specifying software compositions as scripts, while supporting an open-ended set of architectural styles.

Piccola adopts a layered approach to achieve this goal. *External components* export services transparently to each layer. For example, the abstract machine layer sees these services as ordinary channels and agents.

Applications	components + scripts	*External components*
Architectural styles	streams, events, GUI composition, ...	
Core libraries	basic coordination abstractions, basic object model	
Piccola language	services, operator syntax, nested forms, built-in types	
πL **abstract machine**	agents, channels, forms	

The bottom level of the Piccola system provides an abstract machine in which *agents* asynchronously communicate *forms* through shared *channels*. This abstract machine implements the πL-calculus [13], a variant of the polyadic π-calculus [15] in which forms are communicated instead of tuples. The innovation at this level is the introduction of *forms*, which are immutable, extensible records (sets of bindings from labels to channels). Technically speaking, communicating forms rather than tuples does not alter the expressive power of the π-calculus, but it makes it much simpler to express higher-level abstractions in Piccola [25]. This simple foundation allows us to reason about complex and concurrent interactions using a well-developed formal model, and guarantees that the semantics of higher-level abstractions can always be precisely explained in terms of simple interactions.

The next layer defines the Piccola language syntax and semantics. We introduce *primitive values*, like numbers and strings, *higher-order abstractions* over agents, forms and channels, and *nested forms*. Abstractions and nested forms are defined simply by translation to the lower level model using hidden intermediate channels and agents. At this level we already begin to appreciate the expressive power of forms. Forms represent:

- Interfaces to components. Forms encapsulate a set of named services exported to clients.
- Arguments. Forms provide keyword-based arguments for services.
- Contexts. The static context represents all known services and components for any statement. The dynamic context collects services and capabilities that are passed from callers to callee.
- User-defined services.

As forms are immutable, operations on forms yield new forms with an enriched or reduced set of services. It is not possible to modify forms, thereby breaking by accident other agents using this form or component, but only to create new forms. We can see a form as a kind of "primitive object" with public and private features, but without any explicit notion of classes or inheritance. More elaborate object models can be encoded directly in Piccola. Piccola permits form labels to be accessed as overloaded infix operators, which is convenient for expressing compositional styles.

The third layer defines libraries of basic composition abstractions, including control abstractions (e.g., if-then-else, try-catch), coordination abstractions (e.g., blackboards, futures), and other utilities, such as an interface to the Java world. The interface wraps Java objects and represents them as forms so that they can be accessed by Piccola agents.

At the fourth layer, libraries of architectural styles may be defined, such as push-flow or pull-flow streams, GUI composition, and GUI event composition. This is done by implementing connectors for such a style as infix operators on components. A style may also define coordination abstractions to manage interactions between components, and glue abstractions to adapt external components to a particular style, or possibly to bridge gaps between different styles [6][27].

Finally, application programmers can script applications using the connectors of a particular style and the glue abstractions to use external components.

This paper is structured as follows. The next section presents an example that illustrates the top-level view of a Piccola script. Then, in sections 3, 4 and 5, we present the Piccola language layer, and describe respectively, forms, agents and contexts. In section 6 we show how Piccola can be used to define a simple architectural style, and in section 7 we show how classes and mixins can be scripted. Finally, section 8 discusses related work and section 9 concludes this paper.

2. Scripting Components

In this section we present a small example of a Piccola script that uses styles for GUI composition and GUI event composition. The specification of event style itself is presented later in section 6. The reader should not worry too much about details of the mechanics of the script on a first reading, but pay attention instead to how Piccola is used to develop a high-level, declarative view of how this application is composed. The same application written directly in an object-oriented language would typically be more procedural, and emphasize low-level wiring of observers and observables [5]. The Piccola script, on the other hand, expresses the wiring by using compositional operators defined as library abstractions supporting an architectural style

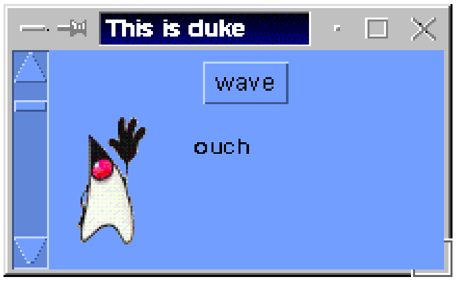

Figure 1 **Duke scripted in Piccola**

The script "`duke.picl`" in figure 2 uses an event style to wire the events and illustrates how the graphical layout is scripted. It also coordinates several agents. Running the script, a frame with Java's Duke appears (see figure 1). When we click on the `wave` button, duke waves at the speed controlled by the scrollbar on the left. When we click on Duke himself, he complains, issuing the message "ouch." After a short delay, the message disappears.

We now look at the individual parts of the script and identify the forms and agents when necessary:

1. We load a file "`nawt`" which defines several services we will use. The keyword **`root`** denotes a special form that represents the static context in which duke.picl is evaluated (see section 5). `load()` reads a set of definitions in a Piccola library script and returns a form containing those bindings. We then *extend* the static context by simply redefining **`root`** to be **`root`** extended by the result of `load()`.
2. Now our extended root context contains the service `awtComponent` defined in the loaded script. This service instantiates new AWT components and wraps them according to our style. We use it to create the duke component, a button, and a scrollbar. The form returned by `awtComponent` can be thought of as a kind of "primitive object" providing the service `set` (amongst others). This service allows us to send a form containing some properties. For example, we set the label of the `waveButton` component by invoking `set` with the argument form `Label = "wave"`. Note that `set` may be invoked either with a parameterized expression on the same line, or by passing an indented form on the subsequent lines. Either syntax can be used at any time. The arguments passed to `set` will cause these properties to be updated in the wrapped Java object. We do not change any default property of `duke`.

```
# File: duke.picl
# 1. load nawt services
root = (root, load("nawt")) # use event and AWT wrappers style
# 2. create AWT Components
duke = awtComponent("demos.duke.Duke")
 waveButton = awtComponent("java.awt.Button").set(Label = "wave")
 speedScrollbar = awtComponent("java.awt.Scrollbar").set
    Minimum = 1
    Maximum = 800
    Value = duke.getSpeed()
# 3. do the event wiring
speedScrollbar ? Adjustment
    do: (duke.set(Speed = speedScrollbar.getValue()))
 waveButton ? Action(do: duke.wave(val = 1))
# 4. click on Duke
counter = load("counter").newCounter(0)
sleep() = javaClass("java.lang.Thread").sleep(val = 2000)
duke ? MouseClicked
    do:
        duke.set(Message = "ouch")
        counter.inc()
        sleep()                     # sleep 2 seconds
        if (counter.dec() <= 0)     # if this was the last click
           then: duke.clearMessage()
# 5. arrange components in a panel
panel = newBorderPanel
    center = newBorderPanel
       north = Components + waveButton
       center = duke
    west = speedScrollbar
# 6. add panel into a frame and display it
exit() = javaClass("java.lang.System").exit(val = 0)
frame = awtComponent("java.awt.Frame").set(Title = "This is duke")
frame.add(val = panel.java, type = "java.awt.Component")
frame ? WindowClosing(do: exit())
frame.pack()
frame.show()
```

Figure 2 **Duke script**

3. Next, the events are wired using a compositional notation with infix operators, (see table 2). The style defines a set of *event types*, like `Adjustment` and `Action`. Each event type is modelled as an abstraction that takes a *response* (a form containing a `do` service) as a parameter and yields a *listener*. The resulting listener may be bound to a component with the infix `?` operator.

Components:	C E R L	*GUI-Component* *Event type* *Response* *Listener*
Connectors:	( ), ?	
Rules:	E(R) → L C ? L → `()`	*compose an event type with a response to get a listener* *connect a component to a listener*

TABLE 2. **GUI Event Composition style**

For example, when the scrollbar is adjusted, the new speed value is set in the `duke` component, whereas clicking on the button causes duke to wave.

4. When we click on duke, he displays a complaining message. The message disappears after a short delay. Each time the user presses the mouse on duke (`MouseClicked`) an agent runs the code given in the response. We do not see the agent directly, but we specify the script (`do: duke.set(Message="ouch", ...)`) he executes. The agent runs in a context which contains bindings for the forms `duke` and `counter`, as well as the services `sleep` and `if`.

   Note that the bindings returned by `load("counter")` are not used to extend **`root`**. We directly use the exported service `newCounter()` to construct a thread-safe counter.

5. The graphical layout uses a different composition style from the event wiring. We use the service `newBorderPanel` exported by “`nawt`.” We define a new `panel` by invoking service `newBorderPanel` which creates a new Java panel with a border layout manager. The argument is a form specifying sub-components with constraints north, south, west, east, or center, according to the border layout manager of Java [7]. A sub-component may itself be a an instance of `newBorderPanel` or even a list of components. In this case these components are arranged using a flow layout in an inner panel. This determines the stretch-

Components:	C List	*GUI-Component* *List of Components*

TABLE 3. **GUI Composition style**

Connectors:	+, newBorderPanel	
Rules:	List + C → List List + List → List newBorderPanel(Form)	*builds a new list with additional element* *concatenate lists* *layout Components in the form*

TABLE 3. **GUI Composition style**

ing properties of the sub-components. Component lists are built up by starting with an empty list (i.e. `Components`) and adding widgets using the + operator. Glue code maps the interfaces of Java objects to fit the style. Note that GUI composition in Piccola using an appropriate style is more declarative than what one would typically write in a conventional object-oriented language. Contrast it with the code fragment necessary to achieve the same layout in Java:

```
Panel panel = new Panel(new BorderLayout());
Panel innerPanel = new Panel(new BorderLayout());
Panel buttons = new Panel();            // using the default flow layout
buttons.add(waveButton);
innerPanel.add(buttons, BorderLayout.NORTH);
innerPanel.add(duke, BorderLayout.CENTER);
panel.add(innerPanel, BorderLayout.CENTER);
panel.add(speedScrollbar, BorderLayout.WEST);
```

6. Finally, the panel is put into a new frame, which is displayed. As the Piccola AWT style uniformly wraps AWT components from Java, we can use methods `pack()`, `show()` etc. directly from the underlying peer Java objects.

This simple example illustrates several important points about Piccola:

- Piccola syntax is extremely lightweight. There are only four keywords (**`root`**, **`dynamic`**, **`def`** and **`return`**) and six reserved operators.
- Forms are ubiquitous in Piccola. They are used to represent interfaces to components, arguments for services, and contexts for agents.
- Although Piccola is not designed as a Bean scripting language, one can use it to compose Beans — or any other kinds of components, for that matter — by defining a suitable architectural style.
- When styles are defined as “component algebras,” the resulting scripts are highly declarative and make the wiring of components explicit.

In the next three sections, we give an overview of all the features of Piccola, namely that of forms, communicating agents, and contexts.

3. What is a Form?

We have identified forms as a central concept needed for composition. A form is a mapping of labels to values. The empty form has no labels. Forms in Piccola are themselves values and may therefore be nested. Many data-structures have a natural embedding as forms. Forms are written as sequences of *bindings*, separated by commas or new-lines and structured using brackets or indentation:

```
baseForm =
    Text = "foo"
    Name = Text
    Size = (x = 10, y = 20)
```

The form `baseForm` contains three labels: `Text`, `Name`, and `Size`. The nested form `baseForm.Size` has labels `x` and `y`. *Projection* is used to fetch elements of a form. For example, the projection `Form.Size.x` yields 10.

Forms are built as a sequence of bindings. Each individual binding is added to the form it follows. At the same time, each binding also acts as a declaration for subsequent code. Thus, the identifier `Text` in the binding `Name` is bound to the string `"foo"` in the previous line. Forms and sequences of statements are unified in Piccola. The whole assignment defines a nested form bound to the label `baseForm` in the global form **`root`**.

3.1 Extending Forms

New forms can be built by *extension*. A form, or more precisely the list of its bindings, may be concatenated with other bindings, which yields a new form. We can extend `baseForm` with a binding for `Color`:

```
coloredForm =
    baseForm
    Color = "green"
```

Now the `coloredForm` has a label `Color` in addition to the labels of `baseForm`. We cannot detect in the extended form how and in what order the labels where added. Note that `baseForm` remains unchanged.

Bindings may also be overridden by new bindings. Clients using an extended form will only have access to the most recent binding for a label. The following example makes a new form with a modified `Size`:

```
modForm =
    baseForm
    Size = (baseForm.Size, x = 15)
```

This extension makes only minimal assumptions on the labels in `baseForm`. It only assumes the presence of label `Size` in `baseForm`. We add a binding for a new `Size`. The new size itself is an extension of `Size` in the original form with a overridden label `x`. Note that this extension would also work if the original size would contain different labels, say for example three parameters `x`, `y`, and `z`. Then, our modified form would also contain these bindings with a modified x value. We heavily use this feature of forms in building reusable abstractions.

It is also possible to extend one form by another, rather than just specifying individual labels to bind. This is an easy and compact way to have default parameters:

```
withDefaults =
    Font = aSystemFont
    baseForm
```

Now, we can project on `Font` in the form `withDefaults`. If `baseForm` already contains a binding for the label `Font`, this value is returned, otherwise the value `aSystemFont` is returned.

Projecting on an unbound label is a type error and yields an undefined value. (Using this value generates an exception.) Type systems for πL and Piccola have been explored [13] but are not presented in this paper.

3.2 Services

In Piccola, we represent everything as a form. Literal values like strings or numbers are forms in the same way strings and numbers are objects in pure object-oriented systems like Smalltalk. Forms are used to encapsulate sets of services. Services themselves are also represented as forms. A service can be invoked with a function-call syntax, but is actually a form with a hidden label that gives access to an agent that represents it. (We use the term *service* rather than “function” to emphasize the fact that the invoked behaviour is provided either directly or indirectly by an external component.)

As everything is represented as a form, the arguments for invoking services are also forms. Therefore, they have in general only one argument.

```
hello() =
    println("hello world")
```

This statement defines a service and assigns it to the form `hello`. The body of the service consists of a call to another service: `println`. When `hello` is invoked, it returns whatever `println` will return.

An alternative can be used when no formal parameter is needed. We can omit the brackets and write:

```
hello: println("hello world")
```

The colon signals that the right hand side is an abstraction. The colon notation sometimes makes code easier to read. Drawing from our earlier example in section 2, the following two forms are strictly equivalent in Piccola:

```
do: duke.wave(val = 1)
```

```
do() = duke.wave(val = 1)
```

To see that a service is just a form, consider the following, equivalent statement:

```
hello = \() = println("hello world")
```

Here, the label `hello` is bound to the anonymous lambda abstraction `\() = ...` Anonymous abstractions are sometimes convenient for defining coordination abstractions, but we will rarely use them directly. Most of the time, a form with a `do` service is more convenient to use.

External components export primitives services to Piccola, but higher-level services can be scripted in Piccola. We therefore speak of the body of a service as its *script*. For example, the script of the `hello` service above is `println("hello world")`.

When a service is invoked, its script (also a form) is evaluated by an agent. The **`root`** context this agent runs in provides access to statically bound services (like `load`) and a dynamic argument which gets passed at invocation time.

We can extend services like any other form and, for example, add labels documenting their interface. Piccola makes no assumption about such additional labels.

```
myhello =
    doc = "My hello world"
    hello
```

There are several ways to invoke services. The argument form can be enclosed in brackets or given by indentation. The following alternatives all invoke a *higher-*

order service `if`. When it is invoked with a boolean value as an argument, it returns a service taking as argument a form containing labels `then` or `else`.

```
if (name == "main")
    then: hello()

if (name == "main") (then: hello())    # a one liner!

branch = if(name == "main")            # curried: apply boolean
branch                                 # branch is a service:
    then: hello()                      # apply cases
```

As services are first class values, we could also directly bind `hello` to the label `then`:

```
if (name == "main")
    then = hello                   # bind then to (form) hello
```

Boolean values are encoded as forms that provide a `select` service. This service either selects a true or false binding of its argument:

```
true = (select(B) = B.true)
false = (select(B) = B.false)
```

Services in Piccola always take a single form as an argument. Since services are values, however, it is possible to define curried services (i.e., taking a single argument and returning a service). Consider the implementation of `if` as it is used above:

```
if(Boolean)(Cases) =     # curried: same as: if(B) = \(C) = ...
    withDefaults =
        then: ()
        else: ()
        Cases
    Case = Boolean.select              # select a case
        true = withDefaults.then
        false = withDefaults.else
    return Case()                      # evaluate branch
```

The service takes two forms as its arguments: `Boolean` and `Cases`. In the body of the service, we first provide `Cases` with default `then` and `else`. The defaults we supply are dummy services that return the empty form, written as `()`. Next, we use the boolean to select either the `then` case (the boolean is true) or the `else` branch. Finally we evaluate the case selected and return it as the result of the `if` service.

What would happen if we omitted the **`return`** keyword in the above definition? Then the result of an application `if(B)(C)` would be a form containing not only the bindings returned by `Case()`, but also those of `withDefaults` and `Case`! The use of the keyword **return** ensures that only the value of the expression that follows is returned. All prior bindings are strictly local. This same mechanism can be used to build objects with private and public features.

3.3 Operators

Piccola supports user defined operators. Any sequence of operator characters like `-`,`+`,`*`,`=`,`!`,... represents an infix or prefix operator. As is usual in object-oriented languages supporting infix operators, such operators are treated as projections on their left-hand side component with the right-hand side component as the argument. The label associated with the operator token has two underscores for infix and one for prefix-use in front of it. For instance: `name == "main"` is interpreted as `name.__==("main")`. Identifiers may also be infix operators when they are enclosed in single backquotes as in `5 ‘mod‘ 3` which is `5.mod(3)`. Similar: `- 4` is interpreted as `4._-()`. Sequences of infix terms associate to the left, i.e. `a | b | c` is `(a | b) | c` or, equivalently, `a.__|(b).__|(c)`.

Infix operators are used to syntactically present architectural styles in a more compositional or algebraical way, as illustrated by the example in section 2.

3.4 Scopes

So far we have only seen simple bindings of labels to expressions using labels bound in previous statements. The right-hand side of a binding can never refer recursively to the label being bound. In practical applications, however, we often need recursive services and forms. The keyword **def** defines such a binding. In definitions, the right-hand side can refer to the identifier being assigned to, provided it is used within an abstraction:

```
def fact(N) =
    if (N < 2)
        then: 1
        else: N * fact(N-1)
```

While **def** is not surprising for services, we also use it to construct fixpoints for plain forms. In this circumstance it allows us to define forms with a notion of self:

```
def cout =
    __<<(X) =
        print(X)
        return cout

nl = "\n"
```

```
cout << "Hello World" << nl
```

Evaluating the term `cout << X` prints `X` and returns `cout`. Therefore, we can write sequences of such terms as in C++.

Note that in each of these examples the recursion occurred within an abstraction. The following examples, by contrast, are not sound in Piccola:

```
def silly = (a = silly)

def sillier = sillier
```

and result in run-time errors. The agent that builds the fixpoint reads it before it is correctly set. The following service is uninteresting, but sound:

```
def sillyButOK() = sillyButOK
```

The **`def`** keyword can also be used to define mutually recursive services. When two or more services should refer each other, they can be enclosed in a common, recursive scope:

```
def myscope =
    a() =
        ...
        myscope.b()                   # call b in myscope
    b() =
        ...
        myscope.a()
```

Note that we could equally omit `myscope` in the body of service `b()` to call to `a()`.

4. Communicating Agents

The semantics of Piccola is given in terms of communicating agents. There are two predefined abstractions necessary to control these agents: one to asynchronously evaluate a `do` service by a new agent and one to synchronize running agents.

The `run` primitive evaluates the `do` service of a form as a separate agent. The result of `run(...)` is the empty form. This result is returned in parallel to starting the new agent. The term `newChannel()` creates a new channel. Channels provide atomic send and receive services to communicate forms. The sender cannot detect when and whether the value sent is received by a communication partner. Receiving a value from a channel blocks unless someone has sent a form to it. If one or more forms are sent, then an arbitrary one of them is received. There is no ordering on the values communicated along a channel. The following script creates a channel `ch` and starts two agents that communicate a form along it:

```
ch = newChannel()

run (do: ch.send("a form"))

run
    do:
        v = ch.receive()
        println("I received " + v)
```

Running this script, the second agent will eventually print out `I received a form`.

The library script “`pil`” provides a style that makes programming with channels and agents more convenient, and mimics the operators of the lower-level πL machine. The script redefines `newChannel` and equips new channels with infix operators `!`, `?` and `?*` instead of `send` and `receive`. The operator `?*` attaches a “replicated agent” to the channel. A replicated agent behaves like an endless supply

Components:	C A	*Channels* *Agents*
Connectors:	!, ?, ?*	*output, input, replicated input*
Rules:	C ! Form → A C ? Abstraction → A C ?* Abstraction → A	*send form along channel C* *receive form and run abstraction* *multiple receive from channel.*

TABLE 4. pil-style

of agents, always ready to receive another message. These operators send and receive forms in their own agents. Using the pil-style, the above script becomes:

```
root = (root, load("pil"))# redefines newChannel
ch = newChannel()
ch ! "a form"            # send the string
ch ? \(v) =              # receive a value, then run the service
    println("I received " + v)
```

The two predefined abstractions `run` and `newChannel` are enough to recover the expressive power of πL. For example, a stop service can be implemented as:

```
stop() =
    newChannel().receive()      # will never receive anything
```

Calling `stop()` will never return and therefore stop the client agent.

Another useful concurrency abstraction is one that evaluates two abstractions in parallel. It returns the result of one of the two passed abstractions. When both abstractions terminate, either result is returned. However, when we know that only one branch terminates and the other stops, the result of `OrJoin` is uniquely determined:

```
OrJoin(X) =
    ch = newChannel()
    run (do: ch.send(X.left()))
    run (do: ch.send(X.right()))
    return ch.receive()      # blocks unless there is one result
```

Here, we run two agents in parallel. The two agents execute the left and the right abstraction given. The service `ch.receive()` blocks, unless one value is sent on it. Once a value is sent to the channel, this value is returned. In the next section, we will use these services to implement an exception handling mechanism within Piccola.

`OrJoin` and `stop` are examples of coordination abstractions. For example, `OrJoin` is used to coordinate two agents such that only one agent returns a result.

5. Contexts

When an agent evaluates a script, it may make use of services defined in the current context (or “environment”). Piccola models contexts explicitly as forms. Since contexts are therefore first-class values, one can implement various abstractions to support modules and packages. In contrast to Piccola, most languages provide a predefined and fixed way to import modules and look up imported services.

The special form **`root`** denotes the (static) context in which identifiers are looked up. Instead of writing:

```
print("Hello")
```

we could equally say:

```
root.print("Hello")
```

Similarly, bindings also extend the **`root`** form for subsequent statements. It is also possible to assign any form as new root or to use root as an ordinary form. For example, `load()` locates a script and evaluates it. It returns the form defined by the script. Assume we have a script `"hello.picl"` with the contents:

```
# File: hello.picl
hello: println("This is the hello script")
```

We can now import the bindings into the **`root`** and use `hello` directly:

```
root = (root, load("hello"))       # extend our root with hello
hello()                            # call hello
```

or we can load the script and keep it in a separate form. This prevents cluttering up our **`root`** namespace:

```
x = load("hello")                  # bind hello to x
x.hello()                          # and use it
```

When the Piccola run-time system is initialized, **`root`** contains the services of the basic Piccola composition abstractions.

5.1 Dynamic Contexts

Statically compiled languages typically use static (lexical) scoping whereas dynamically compiled and interpreted languages often use dynamic scoping or a combination of static and dynamic scoping. Piccola is statically scoped, but offers *dynamic scoping on demand*. Although static scoping is good enough for most purposes, it turns out that certain kinds of coordination and control abstractions are next to impossible to define without dynamic scoping.

As an example, consider exception handling. Most languages that provide exception handling as a built-in construct allow an exception to be raised in the context of some service provider, and thereby cause an associated exception handler of the client to be invoked. In languages that do not provide exception handling, it can be very difficult to simulate. Let us see now how such an abstraction can be defined in Piccola by explicitly passing dynamic contexts between agents.

An example application is the `import` service, which is defined as:

```
import(F) =
    x = findFile(F.name)
    if (isEmpty(x))
        then: raise("Cannot locate Script: " + F.name)
    # otherwise x points to a valid file. We return its contents:
    return try
        do: builtinLoad(x)(F.context, scriptLocation = x)
        catch(E):
            raise("Error in Picclet " + x + "\n" + E)
```

Import tries to find a given file. When this file cannot be located, it raises an error. Otherwise, the location `X` is read and executed. The service `builtinLoad` loads, parses, and executes the script at location `X`. It is possible that this process raises an error. This error is caught and reported to the user. The service `builtinLoad(x)` returns a anonymous abstraction containing the script at `x` as its body and **`root`** as its argument. We invoke this context with the context passed (`F.context`) extended with the location of the script itself. When `builtinLoad` returns successfully, `import` returns the contents of the file.

Observe that `try` and `raise` are normal abstractions, whereas `do` and `catch` are ordinary labels in the argument to `try`. Here are the implementations of `try` and `raise`:

```
try (block) =
    exception = newChannel()
    result = OrJoin
        left:
            e = exception.receive()
            return block.catch(e)
        right:
            raise(e) =        # define a local raise abstraction
                exception.send(e)
                stop()
            dynamic = (dynamic, raise = raise)
            return block.do()
    return result

raise(E) =                    # use dynamic raise
    dynamic.raise(E)
```

Let us first look at the body of `try`. It creates two agents and waits for one of them to terminate. We have already seen `OrJoin` and `stop` in section 4. The `right` agent runs the `do` service of the argument to `try`. This service may terminate normally, causing the agent to return a result, or it may raise an exception, and transfer control to the `left` agent. The `left` agent blocks and waits if an exception is raised. If so, it evaluates the `catch` service of the argument to `try`. Otherwise it does nothing.

The difficulty here is that the client's `do` service knows nothing about the exception channel we want to use to coordinate the two agents. The solution is to define a *local* `raise` abstraction which will signal the exception and stop the `right` agent. This `raise` abstraction is injected into the dynamic context made available to the do service. When the do service calls the *global* raise abstraction, it in turn calls the dynamic one, and the right thing happens.

Whenever a service is called in Piccola, the form **dynamic** is passed implicitly together with the actual parameter. If the client has extended its dynamic context with any additional services, these will then be available to the called abstraction.

5.2 Passing the dynamic context

For readers with some background in the π calculus, it may be helpful to have a closer look at how services are invoked. For that purpose, we show the protocol that is used by service invocations. This protocol can be implemented nicely on top of Piccola using agents and channels. A service becomes a channel together with a replicated agent that implements its body and returns a result. An invocation consists in communicating a dynamic context to this agent along the service-channel. This context will contain the argument (args) and a result channel. The replicated agent will send its result along that result channel.

```
root = (root, load("pil"))  # redefines newChannel
fact = newChannel()         # the service channel

fact ?* \(Dynamic) =        # the service body...
    N = Dynamic.args        # Assign argument form
    if (N > 1)              # factorial:
        then:
            # invoke fact(N-1):
            h = newChannel() # the result channel
            fact ! (Dynamic, args = (N - 1), result = h)
            h ? \(Result) =
                Dynamic.result ! (N * Result)
        else:
            Dynamic.result ! 1
```

Note that we use our previously mentioned pil-style. In the code, we use the identifier Dynamic instead of the Piccola keyword **dynamic**. Observe the invocation of fact(N-1):

- We first create a reply channel h.
- We then send an invocation to the service channel (fact). The invocation consists of the context for the agent responsible to evaluate the service. The context at least contains the argument form and the result channel.
- We receive the result on the reply channel h. Once the service agent delivers a result, we fetch it and continue.

An invocation closely corresponds to the responsibilities the agent implementing the service has. The service is modelled by a replicated agent receiving invocations. An invocation consists of a form. The arguments are by convention bound by label `args`, the result channel is bound by label `result`. The result is returned by sending it along the result channel, from where the client will pick it up.

6. Implementing Styles

This section presents the implementation of the event composition style used in section 2. Participants transmit or receive pieces of information in response to events. Components that emit events are called informers, those that receive them are called listeners [2].

We show code to glue the services provided by objects of the Java AWT Event framework to the event composition style of table 2 which can be used as:

```
javaComponent ? EventType(Response)
```

The `?` with a given event type connects a `Response` to an event within the Java component. A `Response` is a form with a `do` service.

6.1 Interfacing to Java Components

The low-level bridge to Java objects from Piccola is done using the predefined abstractions `javaClass` and `javaObject`. These generic glue abstractions create Java objects and return forms giving access to the public methods of them. The methods are invoked like any other service but the arguments are given as nested forms with labels `val` or `val0`, `val1`, `val2`, etc. since arguments for Java are tuples instead of being keyword based. For overloaded methods, we must also give the type of the arguments in order to select a unique method implementation in Java.

The Piccola Java interface also provides some generic listener classes, like the class pi.piccola.bridge.GenericActionListener. These classes allow us to call Piccola services from Java. The generic action listener class, for example, implements the Java interface java.awt.event.ActionListener. An action listener that prints the events is created by:

```
listener =
javaClass("pi.piccola.bridge.GenericActionListener").new
    val = dynamic
    val1 = (actionPerformed = println)
```

The constructor for the listener class requires two parameters, the first is the dynamic context which will be passed to the listener service, in case the listener service makes use of services in the dynamic context. We need to pass this context explicitly, since Java does not offer a notion of context. The second parameter contains an abstraction to which the event is delegated. The handler for action listeners must be bound by the label `actionPerformed`. The Java constructor for `GenericActionListener` is given as:

```
    public GenericActionListener(Form context, Form delegate);
```

A listener object may be plugged into components using **void** addActionListener(java.awt. event.ActionListener). An event is then forwarded to the service `actionPerformed` within the dynamic context passed. For example, the listener can be added to a button:

```
button = javaObject("java.awt.Button")
button.addActionListener(val = listener)
```

6.2 The GUI Event Composition Style

To support the GUI event composition style, we need to (1) model event types as abstractions that take `do` services as arguments and return listeners, and (2) extend GUI components with a `?` operator to attach listeners. For example, the following code creates a listener for `Action` events and attaches it to a Java Button that has been wrapped to conform to the style.

```
myButton = awtComponent("java.awt.Button")

myButton ? Action(do = println)
```

Since there are many different types of event in the AWT framework, we use a generic glue abstraction, `newEventType`, to instantiate event types for our style:

```
Action = newEventType
    genericListenerClass =
            javaClass("pi.piccola.bridge.GenericActionListener")
    listenerMethod(service) = (actionPerformed = service.do)
    addListener(Component) = Component.addActionListener
```

The argument to `newEventType` is a form with the following labels:

- genericListenerClass is a factory service to instantiate Java listener objects. These objects will be created using new() with arguments val0 for the dynamic context and val1 for the delegate form.
- listenerMethod is a service that returns the delegate form used to instantiate the generic listener class.
- addListener(Component) is a (curried) service encapsulating the method to add listener instances.

Here is the implementation of newEventType. Note that it is a curried service — the event type it returns (e.g., Action) is itself a service that will return a listener. A listener provides a register functionality that will be used by GUI components:

```
newEventType(P)(Response) =
    register(Component) =
        ConstructorArgs =
            val = dynamic
            val1 = P.listenerMethod
                do(E):
                    Response.do(Informer = Component, Event = E)
        listener = P.genericListenerClass.new(ConstructorArgs)
        P.addListener(Component)(val = listener)
```

The listener object is instantiated using the new service of the (passed) generic listener class. As expected, the argument form for new() is the current dynamic context and a form with the delegate services, e.g. a binding actionPerformed for the action event type. Finally the listener registers itself on a Component by delegating registration requests to addListener().

The glue abstraction awtComponent instantiates AWT objects and extends them with the ? operator. This operation uses double dispatch to register the listener L:

```
awtComponent(ClassName) =
    object = javaObject(ClassName)
    def self =
        object
        java = object
        set(P) = ...                    # set properties P
        __?(L) = L.register(self)       # pass the component
    return self
```

The Java class is instantiated, and the Piccola representing it is extended with services needed to support the event style. In addition, the original base object is still available by a projection on the label java.

The implementation of this style may seem somewhat convoluted, but this is largely a side-effect of the fact we are adapting an object-oriented interface to a more compositional style. Keep in mind that the code presented here needs to be written only once. It can then be exploited by any number of scripts. Furthermore, advanced features like **`dynamic`** contexts are typically used only to implement abstractions to support a particular style, and do not normally appear in top-level scripts.

7. Scripting Classes

Although Piccola has no predefined object model, it is possible to implement different object models on top of it, much in the same way that CLOS is defined on top of Common Lisp [8]. In this section, we use one such model to script classes and mixins [3]. This particular model is implemented by a `Class` abstraction and a initial class `Object`, from which all classes inherit. The following code loads the object model and creates a `Point` class:

```
root = (root, load("classes"))     # get Class, Object

Point = Class
    name = "Point"
    super = Object
    instanceVars: (x=newRefcell(), y=newRefcell())

    delta(P):
        asString() = "x = " + P.self.x.get() +
                        ", y = " + P.self.y.get()

        rep() =
            println(P.self.class.name + ".new(" +
                P.self.asString() + ")")

        initialize(Init) =
            P.self.x.set(Init.x)
            P.self.y.set(Init.y)
```

We use the abstraction `Class` to create a new class. Individual classes are parameterized by the following bindings:

- The `name` of the class.
- The `super` or parent class from which this class is derived. The model described here only supports single inheritance.

- A service `instanceVars()` that creates the additional instance variables for instances of this class. Each instance variable is represented by a reference cell with `set` and `get` accessor services. The service `instanceVars` is optional. The default binding for this parameter assumes that there are no new instance variables to be added.
- The `delta(P)` abstraction defines the differences of the new class with respect to its super class. The formal parameter `P` contains the nested forms `self` and `super` for self sends and super calls. The `Point` class defines three methods: `rep()`, `asString()` and `initialize()`. The initialize method is special: whenever we override this method, a call to the overridden `initialize()` is inserted before the overriding method. We can omit a call to `super.initialize()`. This behaviour is implemented in the `Class` abstraction.

The abstraction `Class` creates forms with a service `new()` to create and initialize new objects. For instance, a point is created by:

```
aPoint = Point.new
    x = 1
    y = 2
```

Calling `aPoint.rep()` prints out the string: `Point.new(x = 1, y = 2)`, as expected.

Whenever a new instance is created, `delta()` and `instanceVars()` of all subclasses in the inheritance chain starting from `Object` are called. The assembling is done within a scope definition for `self`. That way we pass `self` and the intermediate objects as `super` to each call to `delta()`. Once the object is built `initialize()` gets called to establish the invariant of the object.

Having the instance variables created by `instanceVars` is not a restriction of the object model. In fact, we could also create the instance variables directly in `delta()`:

```
Point = Class
    name = "Point"
    super = Object
    delta(P):

        x = newRefcell()
        y = newRefcell()
        ...
```

but keeping them by in separate intention-revealing parameter for classes makes the code more self-documenting. In addition, clients that stick to `instanceVars()` for creating instance variables can implement generic operations for cloning objects or inspecting facilities.

`ColoredPoint` is a subclass of `Point` with an additional `color` field and overridden method `asString()`:

```
ColoredPoint = Class
    name = "ColoredPoint"
    super = Point
    instanceVars: color=newRefcell()
    delta(P):
        asString() =
            P.super.asString() + ", color = " + P.self.color.get()
        initialize(Init) =
            P.self.color.set((color = "Black", Init).color)
```

The method `asString()` overrides `asString` of the `point` class and appends a representation for the color of a point. Note how form extension is used to initialize the `color` slot with a default value.

Mixins are classes with a free `super`. Mixin-composition composes two mixins to a new one. Applying a mixin to a class yields a new class. A `color` mixin may look as:

```
ColorMixin = Mixin
    name = "Colored"
    instanceVars: color=newRefcell()
    delta(P) = ...                  # as above
```

This mixin adds a color part to any class it is applied to. Note that the parent class is not specified here. Now, we can apply the mixin to our previous class:

```
myClass = ColorMixin * PointClass

point = myClass.new
    x = 1
    y = 1
    color = "Yellow"
```

Note that we use the flexibility gained from the keyword-based argument to initialize the reference cells. We just pass a form as initializer, each `initialize()` method needs only its specific arguments. The `Mixin` abstraction builds a class name by prefixing the name of the mixin (e.g. `"Colored"`) to the name of the parent class (e.g. `"Point"`). Another mixin may add a `move()` method to change `x` and `y` coordinates of a given point:

```
MoveMixin = Mixin
    name = "Moveable"
    delta(P) =
        move(Diff) =
            P.self.x.set(Diff.x + P.self.x.get())
            P.self.y.set(Diff.y + P.self.y.get())

moveablePoint = ColorMixin * MoveMixin * PointClass
```

Observe that `ColorMixin * MoveMixin` is also a mixin. We summarize the classes and mixin style :

Components:	Class, Mixin	
Connectors:	*	*mixin operator*
Rules:	Mixin * Class → Class Mixin * Mixin → Mixin	*Mixin application* *Mixin composition*

TABLE 5. **Classes and Mixins**

The `Class` and `Mixin` abstractions shown in this section are implemented by approximately 80 lines of Piccola code. This illustrates that it is possible to encode a useful inheritance composition mechanism with feasible effort. Schneider [25] has shown how to encode other forms of inheritance composition, like Beta-style [10] composition.

8. Related Work

In the past years, there has been considerable work on the foundations of concurrency, and much of this on process algebras and process calculi. The π-calculus [15] has proven to be successful for modelling concurrent objects [22][23][29]. The πL-calculus [13] replaces tuple communication of the polyadic π-calculus with monadic form communication.

Pict [20] is a language that builds on the polyadic asynchronous π-calculus. Pict's language constructs are provided as syntactic sugar on top of the core calculus. We have used Pict to run extensive experiments with different object models [11][23] and synchronization policies [28] as examples for composition mechanisms. These experiments led us to conclude that form-based communication would be a better basis for modelling composition than tuple-based communication, and led us to develop the πL-calculus [12][13]. Pict was developed to study the relation of types and concurrent programming, whereas Piccola is used to experiment with composition abstractions.

We have been experimenting with different variants of Piccola. The version described here is Piccola 2.0. It completely hides the πL-primitives of the underlying process calculus as services, whereas these operators are visible in other versions. Piccola 2.0 can be compared to functional languages, where concurrency primitives where added, like this is done in CML [21]. In another variant, Piccola(T), we experiment with a type system for the πL-calculus [13]. Piccola(T) reflects the πL-operators as language primitives as in Pict. The type system is sound and complete, but lacks parametric polymorphism, which would be needed to type generic abstractions. We have also worked on extending the πL-calculus to the Form-calculus, which supports additional operators to hide labels. Piccola(F) offers these restriction operators as primitives [25].

In a much earlier paper with a similar title, we have explored visual composition of objects using scripts [18]. The present work provides a concrete textual syntax and a formal semantics for scripts.

The syntax of Piccola deliberately resembles that of Python [14][30]. Python is an object-oriented scripting language that provides a simple integration of functions and objects. Python models objects and classes in terms of dictionaries (which resemble forms). Methods and functions can be called either with positional parameters (i.e., tuples) or with keyword arguments (i.e., à la forms). Python provides operator overloading, and can also be used to implement architectural styles much in the way described in this report. It provides limited support for reflection, and it is possible to change the underlying object model to a certain degree (though Python does not have a meta-reflective architecture like Smalltalk). Python is not inherently concurrent, though there is a Posix-dependent threads library, and some researchers have experimented with active object models for Python [19].

In Perl [30], procedures may specify the visibility of their local variables in its declaration. To the best of our knowledge, Piccola is the first language that offers both static scoping and the possibility of dynamic scoping on demand, within a formal framework.

Aspect-oriented Programming [9] is an approach to separating certain aspects of programs that cannot be easily specified as software abstractions. AspectJ is a language used to specify aspects which can be weaved into Java source code. Initial experiments have shown that certain aspects can be nicely expressed in Piccola. For example, Readers and Writers synchronization policies cannot be factored out as software abstractions in Java whereas this is relatively straightforward in Piccola. Whether aspects in general can be addressed by Piccola's compositional paradigm of agents and forms is an open question.

Coplien uses C++ as multi-paradigm language [4]. He uses C++ built-in paradigms like OO-inheritance or templates to match different component models and styles as they evolve from domain analysis.

9. Future Work and Conclusion

We have described how Piccola supports the paradigm that Applications = Components + Scripts. We show how components conforming to a style are scripted and how different styles can be implemented within Piccola. This leads to a layered approach, where the abstractions provided by one layer connect components of the next level in a more declarative way.

We use forms to represent components, scripts, services, arguments to services, glue and coordination abstractions, and static and dynamic contexts. For an open component approach, however, it is clear that we must be able to cope with components obtained at run-time, possibly through network middleware. In this case Piccola must provide some reflective capabilities. It is not yet clear what capabilities precisely are needed to inspect forms. Should labels be first class values or is it enough to check for the existence of a given binding in a form? We are currently investigating lightweight approaches, like providing built-in abstractions to iterate over all labels of a form. This allows us to define more generic wrappers for forms, but forbids introducing *new* labels.

Another issue related to open systems is distribution. It is not yet clear whether the notion of locality should go into the channels, (as for example in Klaim [17]) or whether it should be handled by providing dynamic services.

A flexible type system is needed to cope both with statically known components as well as dynamically introduced ones. Should the type system be defined at the level of the πL-calculus (as is the case in Piccola(T)) or at the Piccola language level? Can we develop a type system that captures whether a service returns, may raise an exception, or block? Instead of a type system, could we augment Piccola with an *assertion language* that would allow us to express and reason about the *contracts* that components require and ensure, and correspondingly about the properties guaranteed by an architectural style? Other important non-functional properties include safety and security, real-time properties and reachability. For example, what services are needed by a composition environment such that we can safely install, upgrade, and de-install components without breaking other parts of the system?

Piccola is available from `www.iam.unibe.ch/~scg/Research/Piccola/`

Acknowledgements

The authors thank the members of the SCG, especially Jean-Guy Schneider, Serge Demeyer, Sander Tichelaar, and Markus Lumpe for helpful comments on improving this paper.

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

References

[1] Franz Achermann, Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz, “Piccola - a Small Composition Language,” *Formal Methods for Distributed Processing, an Object Oriented Approach*, Howard Bowman and John Derrick. (Ed.), Cambridge University Press., 1999, to appear.

[2] Daniel J. Barrett , Lori A. Clarke, Peri L. Tarr and Alexander Wise, “A Framework for Event-Based Software Integration ” *IEEE Transactions on Software Engineering*, vol. 5(4) , October 1996 , pp. 378-421 .

[3] Gilad Bracha and William Cook, “Mixin-based Inheritance,” *Proceedings OOPSLA/ ECOOP’90, ACM SIGPLAN Notices*, vol. 25, no. 10, Oct. 1990, pp. 303-311.

[4] James O. Coplien, Multi-Paradigm Design for C++, Addison-Wesley, Reading, Mass., 1999.

[5] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.

[6] David Garlan, Robert Allen and John Ockerbloom, “Architectural Mismatch: Why Reuse Is So Hard,” *IEEE Software*, vol. 12, no. 6, Nov. 1995, pp. 17-26.

[7] James Gosling, Frank Yelling and The Java Team, *The Java Application Programming Interface Volume 2*, Addison Wesley, 1996.

[8] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin, “Aspect-Oriented Programming,” *Proceedings ECOOP’97*, Mehmet Aksit and Satoshi Matsuoka (Ed.), LNCS 1241, Springer-Verlag, Jyvaskyla, Finland, June 1997, pp. 220-242.

[10] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard, “The BETA Programming Language,” *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner (Ed.), MIT Press, Cambridge, Mass., 1987, pp. 7-48.

[11] Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz, “Using Metaobjects to Model Concurrent Objects with PICT,” *Proceedings of Langages et Modèles à Objets*, Leysin, October 1996, pp. 1-12.

[12] Markus Lumpe, Franz Achermann and Oscar Nierstrasz, “A Formal Language for Composition,” Foundations of Component Based System, Gary Leavens and Murali Sitaraman (Ed.), Cambridge University Press., 1999, to appear .

[13] Markus Lumpe, “A Pi-Calculus Based Approach to Software Composition,” Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[14] Mark Lutz, *Programming Python*, O’Reilly, 1996.

[15] Robin Milner, “The Polyadic pi Calculus: a tutorial,” ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, Oct. 1991.

[16] Michael Morrison, *Presenting Java Beans*, Sams net, 1997.

[17] Rocco de Nicola, Gian Luigi Ferrari and R. Pugliese, “Klaim: a Kernel Language for Agents Interaction and Mobility ,” IEEE Transactions on Software Engineering (Special Issue on Mobility and Network Aware Computing), Catalin Roman and Ghezzi (Ed.), 1998 .

[18] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey and Marc Stadelmann, “Objects + Scripts = Applications,” Proceedings, Esprit 1991 Conference, Kluwer Academic Publishers, Dordrecht, NL, 1991, pp. 534-552.

[19] Michael Papathomas, “ATOM: An Active object model for enhancing reuse in the development of concurrent software,” RR 963-I-LSR-2, IMAG-LSR, Grenoble-France, November 1996.

[20] Benjamin C. Pierce and David N. Turner, “Pict: A Programming Language based on the Pi-Calculus,” Technical Report, no. CSCI 476, Computer Science Department, Indiana University, March 1997.

[21] John H. Reppy, “CML: A Higher-Order Concurrent Language,” *ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, vol. 26, no. 6, Toronto, June 26-28, 1991, pp. 293-305.

[22] Davide Sangiorgi, “An interpretation of Typed Objects into Typed Pi-calculus,” RR 3000, INRIA Sophia-Antipolis, September 1996.

[23] Jean-Guy Schneider and Markus Lumpe, “Synchronizing Concurrent Objects in the Pi-Calculus,” *Proceedings of Langages et Modèles à Objets ’97*, Roland Ducournau and Serge Garlatti (Ed.), Hermes, Roscoff, October 1997, pp. 61-76.

[24] Jean-Guy Schneider and Oscar Nierstrasz, “Components, Scripts and Glue,” *Software Architectures — Advances and Applications*, Leonor Barroca, Jon Hall and Patrick Hall (Ed.), Springer, 1999, pp. 13-25.

[25] Jean-Guy Schneider, “Components, Scripts, and Glue: A conceptual framework for software composition,” Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

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

[27] Clemens A. Szyperski, *Component Software*, Addison-Wesley, 1998.

[28] Patrick Varone, “Implementation of ‘Generic Synchronization Policies’ in Pict,” Technical Report, no. IAM-96-005, University of Bern, Institute of Computer Science and Applied Mathematics, February 1996.

[29] David Walker, “Objects in the pi-calculus,” *Information and Computing*, vol. 116, no. 2, 1995, pp. 253-271.

[30] Larry Wall and Randal L. Schwartz, *Programming Perl 2nd Edition*, O’Reilly & Associates, Inc., 1990.

[31] Aaron Watters, Guido van Rossum and James C. Ahlstrom, *Internet Programming with Python*, M&T Books, 1996.

Piccola Syntax

Form ::=	‘**dynamic**’ | ‘**root**’ | *Label* | *Literal*	
	‘\’ *Abstraction*	anonymous Abstraction
	Form ‘.’ *Label*	Projection
	Form ‘(‘ *Expressions* ‘)’	Invocation
	Form op *Form*	Infix Invocation
	op *Form*	Prefix Invocation
	‘(‘ *Expressions* ‘)’	
Abstraction ::=	*Pattern* { ‘=’ | ‘:’ } *Expression*	
Pattern ::=	‘(‘ [*Label*] ‘)’ [*Pattern*]	
Expression ::=	[*Expressions* ‘,’] ‘**return**’ *Form*	local declarations
	Expressions	
Expressions ::=	*Statement* [‘,’ *Expressions*]	
	Binding [‘,’ *Expressions*]	
Statement ::=	‘**root**’ ‘=’ *Form*	change root context
	‘**dynamic**’ ‘=’ *Form*	change dynamic context
Binding ::=	[‘**def**’] *Label Abstraction*	define service
	[‘**def**’] *Label* ‘=’ *Form*	assign form
	Label ‘:’ *Form*	define service without arguments
	Form	evaluate Form / add Bindings