Implementing Coordination Styles in Piccola

Student Project

by

Stefan Kneubuehl

Care: Franz Achermann
Software Composition Group

Institute for Computer Science (IAM)
University of Bern, Neubrückstrasse 10, Bern

Version 1.1
February 2001

Contents

1 Introduction **4**

1.1 Motivation and Aim 4

1.2 Definitions 4

1.2.1 Components 4

1.2.2 Connectors 5

1.2.3 Coordination Styles 6

1.3 Piccola Features 7

1.3.1 'Functional Forms' 7

1.3.2 Labels and Arguments 8

1.3.3 Importing Java Objects 8

1.3.4 Operators 9

2 A Framework for Implementing Styles **9**

2.1 Components 9

2.1.1 Component Instances 9

2.1.2 Component Abstractions 10

2.2 Provided and Required Services 10

2.3 Connectors 13

2.3.1 Instance Connectors 13

2.3.2 Component Connectors 13

2.3.3 Connectors as Operators 15

2.4 How to Use the Framework 17

3 Push-Flow Style **17**

3.1 Component Types 17

3.2 Signature 18

3.3 Connectors 19

3.3.1 Pipe 19

3.3.2 Duplex 20

3.3.3 Sequential 20

3.3.4 Parallel 21

3.4 Examples 22

4 Grouped Actor Style **24**
4.1 Example: Chat Application . . . 24
4.2 Component Types . . . 25
4.3 Signature . . . 26
4.4 Connectors . . . 26
4.4.1 Add . . . 26
4.4.2 Compose Reactive Actor . . . 27
4.4.3 Compose Handler . . . 28
4.5 Components and Wrappers . . . 28
4.5.1 EmptyGroup component . . . 28
4.5.2 EmptyRActor component . . . 29
4.5.3 makeActor wrapper . . . 30
4.5.4 makeMessage factory . . . 30
4.6 Example: Chat Application Revisited . . . 31

5 Regulated Coordination Style **33**
5.1 Example: Electronic Voting . . . 33
5.2 Component Types . . . 34
5.3 Signature . . . 34
5.4 Connectors . . . 35
5.4.1 Add Actor . . . 35
5.4.2 Add Rule . . . 36
5.5 Components and Wrappers . . . 36
5.5.1 EmptyPolicy . . . 37
5.5.2 Events and Rules . . . 37
5.6 Example: Electronic Voting Revisited . . . 38

6 Lessons Learned **39**
6.1 Strengths of Piccola . . . 39
6.2 Proposals for Improvements . . . 39
6.2.1 Indentation . . . 39
6.2.2 Labels as First Class Values . . . 39
6.2.3 Syntax Extensions . . . 41
6.2.4 Global Operators . . . 41

6.2.5 Type System 41

7 Future Work **42**

8 Conclusion **42**

1 Introduction

1.1 Motivation and Aim

Piccola is a small composition language currently being developed at the University of Bern [2][3]. The language uses 'forms', which are immutable, extensible records as first-class values. Forms are an excellent basis for implementing object models [10] or a simple component model, which will be done in this work. Being based on the pi-calculus, Piccola also provides a good formal basis for coordination.

Many different coordination models and languages have been proposed, each focussing on more or less specific coordination issues. Large software systems may have several coordination problems best approached with different models. It is not easy to bridge between coordination models, as they are often based on different technology or even use their own languages.

We propose to define different *coordination styles* [1] in one composition language, namely Piccola. This would not only lead to a smaller and thus simpler formal basis, but also to the possibiliy to easily bridge between different coordination styles.

In this work, we will implement a small style framework in Piccola. The framework will be used to embed some well-known coordination models in Piccola as 'coordination styles'. It will become apparent that a clear separation of computational and coordination code can be achieved.

This report is structured as follows: The remaining part of this section gives informal definitions for some important concepts and goes into various important Piccola features. In section 2, the style framework will be defined. Sections 3 to 5 show how three different coordination models can be implemented in Piccola using the framework. Section 6 summarizes the Piccola experience of the author and makes some proposals for improvement. The report will be concluded by section 7.

1.2 Definitions

1.2.1 Components

In the component oriented approach, a software system is composed of components that are plugged together [9][11] [12]. In an ideal world, software engineering would be much like playing with Lego-blocks: Different kinds (types) of stones (components) can be composed with certain other sorts of stones using their plugs, yielding bigger entities. These composite stones can be handled the same way as the basic ones (thus still being components). The final Lego construct, e.g. a spaceship (the software system) can still be considered a stone (component) with respect to plugging.

Coming back to the realm of software architecture, it can be summarized that components

- have plugs.
- can be attached to one or more component types, providing information about their composability.
- can be composed by connecting their plugs, yielding new components.
- are black-box entities, i.e. they hide implementation details behind an interface (a set of plugs).

There are two major views of a software system: the static and the dynamic structure. The static structure describes which elements of a software system require what other elements, how they are connected to each other, and how they perform tasks. The dynamic structure of the software shows which elements communicate with each other, and in which state of computation the different elements are.

As the elements (components) of the static and dynamic structure of a software differ in their properties, it seems appropriate that a clear distinction between the static and dynamic aspects of components is made:

- *Component abstractions* are the basic compositional elements of a software architecture. They *describe* how to perform computations, and what services are provided and required by the component. They are stateless.
- *Component instances* are the basic computational elements of a running software system. They *do* perform computations, and provide and require services. Thus they have a state and can be considered as primitive objects.

When a software is executed (or during its execution), the component abstractions making up its architecture are instantiated, i.e. component instances are created.

In this paper, the term *component* is used as a synonym for *component abstraction*, *instance* has the same meaning as *component instance*. Note that the term *abstraction* is used in this paper to refer to Piccola 'services'.

1.2.2 Connectors

Connectors are used to compose compatible components to a larger unit. They are composition operators that define how the components' instances interact with each other. When composition takes place at run-time, instances, rather than components, are composed. Thus, there are two kinds of connectors, instance and component connectors.

In order to illustrate the terms described above, consider the following Unix shell script:

```
less myfile.txt | sort
```

This script is a component composed of the `less` and the `sort` component using the pipe component connector. Note that `less` is parameterized with a file name. When executed, this script starts two Unix processes (instances) and creates a pipe connecting the two processes (instance connector).

The above example illustrates the behaviour of a composite component: When instantiated (executed), instances of the original components are created and then composed using an instance connector.

1.2.3 Coordination Styles

Many different coordination models have been proposed such as Linda [5], Actors [4] and Regulated Coordination [8]. No single model seems to fit for all coordination issues.

Typically, different parts of a software system have different coordination issues and therefore require different coordination models. These models, though, are hard to combine.

This problem is addressed by the introducting *coordination styles*, implementations of different models in a single coordination language. A coordination style consists of the three following parts:

- *Component types* describe the interface a component must provide to be able to participate in the coordination style.
- *Connectors* are the heart of a coordination style. They define the communication protocol for the composed components, i.e. they *coordinate* components.
- The *Signature* is a set of composition rules that define valid compositions in a style.

Following is a short consideration how the three parts of a coordination style can be defined. In parallel, the parts are illustrated on a well-known example.

> The pipe-and-filters style is a classic example of a coordination style.

A component type is given by the provided and required services a component of that type has. For a formal definition of the behaviour of composite components, requirements for the behaviour of these services is also needed. This issue is beyond the scope of this report.

> We choose a definition of filters and pipes where the single component type in the pipe-and-filters style is the *filter*. A filter provides a `put` service to enable the previous filter to push data towards it and also requires a `put` service to push data towards the next filter.

Connectors have to wire the provided and required services of the composed components and create the resulting component. They are given by an implementation in the coordination language.

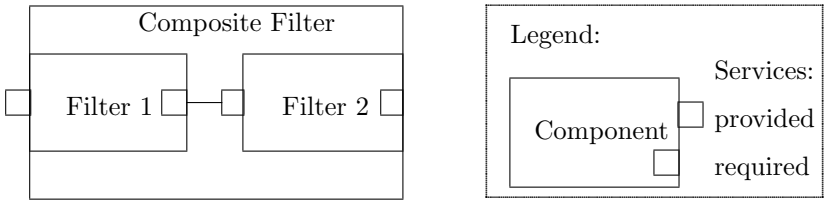

Figure 1: Connecting two filters

> The connector of our example style is the *pipe*. It connects two filters by binding the required `put` service of the left-hand filter to the provided `put` of the right-hand filter. This creates a composite component that is again a filter by making available the left-hand filter's provided `put` and the right-hand filter's required `put`. (See figure 1). A pipe may also provide buffering for the transmitted data (e.g. Unix).

The signature is given by a set of operators of a many-sorted algebra, where the connectors are the operators and the component types are the sorts. It's a formal specification how components may be legally composed in a style. The signature could be enforced by a type system in the coordination language.

> As there is only one component type and one connector, the signature of the pipe-and-filter style is very simple:
>
> $$filter \mid filter \rightarrow filter.$$
>
> This means that two filters may be composed using the | operator yielding another filter.

1.3 Piccola Features

Piccola is a small composition language currently being developed at the University of Bern. This section will not give a complete introduction to Piccola, but focus on some special features relevant to this paper. For more information on Piccola, please refer to [3][2]. This report refers to Piccola version 2.1 as implemented in JPiccola 2.1a for Java. Note that some language features have changed in Piccola 3 (which is available as SPiccola 3 for Squeak).

1.3.1 'Functional Forms'

One of Piccola's most important features is that everything is considered a form[1]. Therefore, a Piccola *abstraction*, is just a form with a dedicated label `call'`. Readers who are familiar with the C++ Standard Library can think of such a form being a kind of functional object. These 'functional forms' can be treated both as an abstraction or as a 'normal' form:

[1] A *form* is an immutable, extensible mapping from labels to values

```
f(arg): println(arg)      # definition of service f
form =                    # definition of form ...
  f                       # as service form f ...
  world = "World"         # extended by label world

form("Hello")             # usage a service, calls println("Hello")
println(form.world)       # usage a form, calls println("World")
```

1.3.2 Labels and Arguments

Checking whether a form has a specific label mapped to a value is something often needed when programming in Piccola. In Piccola 2.1, this can not yet be done with a built-in service, but the existing built-in service `isEmpty` takes us far. `isEmpty` takes a form as argument and returns `true`, if the argument is the empty form [2], and returns false in any other case.

A form that maps a certain label to the empty form is extended with the form to be checked for that label. The resulting form will map the label to a non-empty form only, if the form to be checked does so. The code below returns `true`, if `form` maps `label` to a non-empty form:

```
isEmpty(label = (), form).label).not()
```

A very similar construct can be used to provide default arguments for abstractions:

```
myPrint(args):
  prompt = (prompt = "> ", args).prompt # provide default argument
  println(args.prompt + args.value)

myPrint(value = "Hello")                # prints '> Hello'
myPrint(value = "World", prompt = "? ") # prints '? World'
```

1.3.3 Importing Java Objects

JPiccola 2.1a is implemented in Java and allows us to integrate Java objects by the `javaObject` service. This service takes a string denoting a Java class as argument and returns a Java object wrapped as a form.

Below is an example usage of a `java.util.Vector` object in Piccola:

```
vec = javaObject("java.util.Vector")
vec.addElement(val = "Hello")
println(vec.firstElement())
vec.removeElementAt(val = 0)
```

[2] The empty form is the form that has no bindings, denoted in Piccola with empty parantheses `()`.

1.3.4 Operators

Piccola interprets unary operators as calls to a specially named service of the operand. Binary operators are translated to a service call of the left-hand operand. The service name for a unary operator is the operator's charactes prefixed with an underscore (_) character. For binary operators, the prefix is two underscores. Below is an example:

```
def cout =
  _$: "cout"                        # unary operator
  __<<(x):                          # binary operator
    print(x)
    return cout

cout << "Hello World: " << $cout  # prints 'Hello World: out'
```

2 A Framework for Implementing Styles

This section shows how the elements of our considerations such as components or connectors can be implemented in Piccola. This gives a basis for a small framework for implementing styles.

2.1 Components

2.1.1 Component Instances

Component instances are *black-box* entities. This means they have an interface visible to clients, but a hidden implementation. In Piccola, the `return` keyword can be used to separate those two parts of an instance:

- The form before `return` is the instance's implementation and is not accessible outside of the instance's definition.
- After `return`, the instance's interface is defined by exporting the instance's services that are intended to be public.

Consider the following example:

```
hello =
  myMsg = "Hello"           # implementation
  print: print(myMsg)
  return
    print = print           # interface

hello.print()               # ok, prints 'Hello'
hello.myMsg                 # error, hello has no label 'myMsg'
```

`hello` refers to a form that contains a `print` label. The `myMsg` label is not accessible outside of the instance definition since it is not exported.

2.1.2 Component Abstractions

In Piccola, an abstraction is expressed simply by replacing the equal sign by a colon in the first line of an instance definition and thus turning the assignment into an abstraction. For example, the following `Store` component abstraction is a service that returns a form containing a `get` and a `set` label:

```
Store:
  myValue = newChannel()        # implementation
  myValue.send()
  get:
    value = myValue.receive()
    myValue.send(value)
    return value
  set(value):
    myValue.receive()
    myValue.send(value)
    return ()
  return
    get = get                   # interface
    set = set
```

Note that in contrast to the `hello` label in section 2.1.1 to which a form is assigned and thus refers to an instance interface, the `Store` label refers to a service that creates a new instance and returns its interface form on every invocation.

The behaviour of `Store` is the following: When invoked, a new channel to store the value is created and bound to the name `myValue`. Then, the two services `get` and `set` are defined, making use of the private `myValue`. Finally, the interface form containing the names `get` and `set` is defined and returned as result of the `Store` service.

Now the `Store` component can be used:

```
store = Store()             # instantiate the component
store.set(3)
println(store.get())        # prints '3'
```

2.2 Provided and Required Services

Provided services are implemented straightforward, as they correspond to Piccola services. For example, the `Store` component defined in section 2.1.2 provides the two services `get` and `set`.

Required services need more consideration. A required service says that the component needs a service provided by another component in order to function

correctly. The component will invoke the required service. Thus, it should provide call semantics. When composing components, provided services will be bound to required services. They must therefore provide a *bind* operation. The behaviour of the required service is to forward calls to the provided service bound to it.

Here arises a first question: Should it be possible to bind multiple provided services to a required service (just as it is possible to bind a single provided service to different required services)? Such a required service would be a kind of fork, forwarding calls to possibly more than one provided service. This behaviour can be desirable, but it has also the drawback, that such a required service cannot have a return value. (Which return value of the bound provided services should be preferred?)

The second question concerns the behaviour of an unbound required service: Should a call to such a service be ignored? Or should the execution block until the required service is bound? Probably, both behaviours can be desired.

The above considerations show that required services are a locus of *coordination*. To meet different needs, all of the above mentioned variations of required services are provided:

- `ReqOnceBlock` can be bound once and blocks if unbound.
- `ReqOnceIgnore` can be bound once, execution continues immediately if unbound.
- `ReqMultiBlock` can be bound multiple times and blocks if unbound.
- `ReqMultiIgnore` can be bound multiple times, execution continues immediately if unbound.

For sake of brevity, we only show the implementation of `ReqOnceBlock`.

A required service can be thought of as a slot for a service location. In that case, to bind a provided service to the required service is to put the provided service's location into the slot. A call to the required service means to retrieve the provided service's location from the slot and make a call to that location.

```
ReqOnceBlock:
  myService = newChannel()
  myBound = newChannel()
  myBound.send(false)
  bind(service):
    if (myBound.receive())
      else: myService.send(service)
    myBound.send(true)
    return ()
  call(args):
    service = myService.receive()
    myService.send(service)
    return service(args)
```

```
  return
    call
    bind = bind
```

The `ReqOnceBlock` abstraction returns a form representing a required service. The form uses two internal channels. `myService` is the location where the provided service will be stored. `myBound` acts as a boolean, containing `false` as long as the service is not bound yet.

The form has a `bind` service that binds the provided service `service` by sending its location to the `myService` channel. The enclosing code ensures that any further attempt to bind the service have no effect. The `call` service reads the location of the bound service from the channel and makes a call to that location with the provided arguments.

The last thing is to get the required service, which is a form, to behave just like a normal service, once it is bound. As shown in section 1.3, this is done by extending the returned form with the service form `call`.

Following is an example of how `ReqOnceBlock` required services can be used:

```
reqSet = ReqOnceBlock()
reqGet = ReqOnceBlock()

store = Store()                 # instantiate Store
reqSet.bind(store.set)          # bind reqSet
reqSet.bind(store.get)          # no effect, can be bound once
reqSet("Hello")                 # store.set("Hello") is called
x = reqGet()                    # unbound, blocks
```

With the required service abstractions, the fact that a component needs, i.e. requires a service can be neatly expressed. Consider for example a `FlagAdapter` component that acts like a store for a boolean value but delegates the actual storing mechanism to another component. It requires a `set` and a `get` service to access the actual store.

```
FlagAdapter:
  reqSet = ReqOnceBlock()
  reqGet = ReqOnceBlock()
  run(do: reqSet(false))       # store default value
  set: reqSet(true)
  clear: reqSet(false)
  isSet: reqGet()
  return
    reqSet = reqSet
    reqGet = reqGet
    set = set
    clear = clear
    isSet = isSet
```

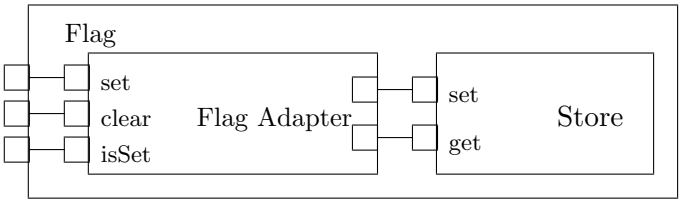

Figure 2: Composing a flag with the `makeFlag` connector

Of course, before a `FlagAdapter` instance can be used, its required services must be bound to the provided services of a storage instance.

2.3 Connectors

2.3.1 Instance Connectors

An instance connector combines two[3] instances to a larger component instance by binding the appropriate provided and required services. It also defines the behaviour of the composite instance. The instance connector thus specifies how the original instances communicate with each other and with the outside world, i.e. the connector *coordinates* the two instances.

The following abstraction implements a connector that composes a flag adapter with a store, yielding a flag (see figure 2).

```
makeFlag(flagAdapter)(store):
  flagAdapter.reqSet.bind(store.set)  # bind services
  flagAdapter.reqGet.bind(store.get)
  return                              # return composite instance
    set = flagAdapter.set
    clear = flagAdapter.clear
    isSet = flagAdapter.isSet
```

Now, the `makeFlag`, `FlagAdapter` and `Store` abstractions can be used to create a flag instance:

```
  fa = FlagAdapter()
  st = Store()
  flag = makeFlag(fa)(st)
```

2.3.2 Component Connectors

Component connectors compose two components to a larger component. Instances of the resulting component must have the following behaviour on creation:

[3]Note that higher-order connectors can be expressed in terms of binary connectors.

- Instantiate the two original components.
- Bind the appropriate services.
- Return the composite instance.

The connector has to return a component, i.e. an abstraction over the above behaviour. A component connector that connects a flag adapter and a store component to a flag component has following implementation:

```
MakeFlag(FlagAdapter)(Store):
  Flag:
    fa = FlagAdapter()          # instantiate left-hand argument
    st = Store()                # instantiate right-hand argument
    fa.reqSet.bind(st.set)      # bind services
    fa.reqGet.bind(st.get)
    return                      # composite instance
      set = fa.set
      clear = fa.clear
      isSet = fa.isSet
  return Flag                   # return Flag component
```

The behaviour of a component connector differs only in two points from the corresponding instance connector's:

- The arguments are instantiated.
- An abstraction over the resulting instance is returned.

Thus, given an instance connector, the corresponding component connector is easily obtained using the following abstraction. Note that component connectors are referred to as `cconnectors` as opposed to `iconnectors` for instance connectors.

```
cconnector(connector)(Left)(Right):
  Component: connector(Left())(Right())
  return Component
```

`cconnector` takes a connector and its two arguments and instantiates the arguments before passing them to the instance connector. It returns a component abstraction over the result of the instance connector.

Using the `cconnector` abstraction, the `MakeFlag` component connector can be defined in one line of code, given the instance connector:

```
MakeFlag = cconnector(makeFlag)
```

2.3.3 Connectors as Operators

Since connectors are thought of as operators in a many-sorted algebra (see section 1.2.3), they should be represented as operators in the code as well. Piccola 2.1 doesn't allow the definition of global operators, thus a workaround is needed: Connectors are attached to their left-hand operands by extending the forms that may act as left-hand operands with an operator abstraction (see section 1.3).

Wrappers are needed to perform the extension on both instance and component level. It is convenient to have an abstraction that takes care of all this work:

```
newComponent(args):
  empty: ()
  makeInstance = (instance = empty, args).instance
  iconnectors = (iconnectors = empty, args).iconnectors
  cconnectors = (cconnectors = empty, args).cconnectors
  def component =                 # define the component
    call(args):
      def instance =              # define the instance
        makeInstance(args)        # calls args.instance
        iconnectors(instance)     # extend instance with operators
      return instance
    return
      call                        # abstraction over instance
      cconnectors(component)      # extend component with operators
  return component
```

The `newComponent` wrapper takes following arguments:

- `instance`: a component, i.e. an abstraction over a form that represents an instance such as `Store`
- `iconnectors`: a mapping of operators to (globally defined) instance connectors.
- `cconnectors`: a mapping of operators to component connectors.

The first four lines of `newComponent` ensure that any argument may be omitted (see section 1.3). The abstraction defines the component as a service that returns an instance (`call`) extended by the mapping of operators to component connectors. The instance is obtained by a call to `args.instance` and extended by the mapping of operators to instance connectors (`args.iconnectors`).

A remaining problem is that the resulting component or instance of a connector must also be extended with the appropriate operators. A modified version of the `cconnector` abstraction takes care of that for component connectors. A new `iconnector` abstraction wraps the extension of the resulting instance of an instance connector:

```
cconnector(signature)(connector)(Left)(Right):
  newComponent
    instance: connector(Left())(Right())
    signature

iconnector(signature)(connector)(this)(right):
  empty: ()
  iconnectors = (iconnectors = empty, signature).iconnectors
  def instance =
    connector(this)(right)
    iconnectors(instance)
  return instance
```

The `cconnector` wrapper takes now as additional first argument the resulting component's *signature*, a form providing the `iconnectors` and `cconnectors` mappings (see above). `cconnector` uses the `newComponent` abstraction to create the resulting component.

The `iconnector` wrapper takes the same arguments as `cconnector`. It just calls the connector and extends it's result with the mappings provided by `iconnectors` of the singnature argument.

With the above defined wrappers, the `FlagAdapter` component can be extended with a >> operator that represents the `makeFlag` connector:

```
FlagSig = ()              # flag component has no connectors

FlagAdapterSig =
  iconnectors(this):      # instance connectors
    __>> = iconnector(FlagSig)(makeFlag)(this)
  cconnectors(this):      # component connectors
    __>> = cconnector(FlagSig)(makeFlag)(this)

FlagAdapter = newComponent
  instance:               # instance implementation
    ...
  FlagAdapterSig          # extend with signature
```

`FlagSig` defines the signature of flag components. It's the empty form since a flag has no connectors. The signature of the flag adapter provides the mappings from operators to wrapped connectors. The `makeFlag` connector is wrapped with `iconnector` and `cconnector`, which makes the connector behave in the expected way and extends the result of the connector with the signature of the flag (since the resulting component is a flag) provided by `FlagSig`.

The definition of the flag adapter component can largely remain the same as in section 2.2. The only necessary thing to do is (1) to put the instance definition in the `instance` abstraction, (2) to provide the signature (`FlagAdapterSig`) and (3) let the `newComponent` wrapper compose the final component abstraction out of the given instance and signature description.

Having defined the flag adapter component this way, compositions using operators can be performed both at instance and at component level:

```
Flag = FlagAdapter >> SimpleStore   # compose components
flag1 = Flag()                      # instantiate composite component

fa = FlagAdapter()                  # instantiate flag adapter
st = SimpleStore()                  # instantiate simple store
flag2 = fa >> st                    # compose instances
```

2.4 How to Use the Framework

This section has gone into major issues for implementing coordination styles in Piccola. It should be finished with a recipe for defining such styles:

You need Java SDK 1.2, Piccola 2.1a, a spoon of creativity and some spare time.

- Define component abstractions (e.g. FlagAdapter).
- Define connectors (e.g. makeFlag).
- Define component signatures (e.g. FlagSig).
- Put the above ingredients together using `newComponent`, `cconnector` and `iconnector`.

You are free to spice your style with fancy names.

3 Push-Flow Style

The push flow style is a simplification of the pipe and filter style, where each component can have at most one input and one output pipe connected to it. A component that accepts no input is called a *source*, a component that produces no output is a *sink*. A *filter* accepts input and produces output.

Thus, this style models a stream or flow of data that is generated by a source, possibly processed by filters and finally consumed by a sink. The *push* in the name of the style is chosen to emphasize that the upstream components push the data downstream to the next component. In contrast, the components of a pull-flow style would pull their data from upstream.

3.1 Component Types

As seen above, the push-flow style deals with three component types:

- *Sources* that produce data and push it downstream.

component type	provided services	required services
Source	-	put() close()
Sink	put() close()	-
Filter	put() close()	put() close()

Table 1: Component types of the push-flow style

- *Filters* the accept data from upstream, process the data and push it further downstream.
- *Sinks* that consume data pushed to them.

The signaling in this style is based on two services: A `put` service that accepts data from upstream and a `close` service that signals an end-of-data.

As sources and filters push data downstream, they both require a `put` and a `close` service. On the other hand, sinks and filters accept data from upstream and thus provide the two services. See Table 1.

As Piccola is dynamically typed, the determination of the types of components has to be done by checking for existing provided and required services at run-time. For example, if a component requires but does not provide a `put` service, it is a source. Based on that idea, type checking abstractions for all component types can be implemented:

```
isSource(F):
  isEmpty((put = (), F).put) &&
  isEmpty((close = (), F).close) &&
  isEmpty((reqPut = (), F).reqPut).not() &&
  isEmpty((reqClose = (), F).reqClose).not()
```

The above Piccola abstraction checks whether `F` is a source, i.e. whether the argument has a `put` and a `close` label and neither a `reqPut` nor a `reqClose` label. The construct to check for an existing label in a form using the `isEmpty` abstraction is explained in section 1.3. The implementations of the `isSink` and `isFilter` abstractions are analogous.

3.2 Signature

The simplest push-flow style architecture consists of two components: a source pushing data towards a sink (see Table 2). This is how every push-flow configuration looks like on the top-most level. Thus, the first composition rule of the signature is essential: it connects a source to a sink with the pipe connector. This composition yields the empty component [4] as the style does not define a

[4]The component that has no plugs, denoted with empty parentheses, the Piccola notation for an empty form.

Source | *Sink*	→	()	Connecting a source to a sink
Source | *Filter*	→	*Source*	Appending a filter to a source
Filter | *Filter*	→	*Filter*	Composition of filters
Filter | *Sink*	→	*Sink*	Prepending a filter to a sink
Source + *Source*	→	*Source*	Sequential composition of sources
Source & *Source*	→	*Source*	Parallel composition of sources
Sink + *Sink*	→	*Sink*	Multiplex flow to two sinks

Table 2: Signature of the Push Flow Style

way to interact with a whole data flow.

Now let's have a look at filters. Filters can be connected to any other component of this style. Composing a source with a filter yields a source, composing a filter with a sink yields a sink and connecting two filters results in another filter. These compositions are all performed by the pipe connector.

The last connectors introduced provide means to join and split a data flow. The sequential composition of two sources (denoted with the + operator) yields another source that first pushes the output produced by the left-hand source downstream while blocking the right-hand source. When the left-hand source closes the flow, the output produced by the right-hand instance is sent further downstream. The parallel composite of two sources sends the data produced by either of the sources immediately downstream. It closes when both original sources close. The duplex connector splits up a data-flow to two sinks.

Note that whilst data-flows can be joined and split using the above mentioned connectors, on a higher level there appears to be just one single data-flow.

3.3 Connectors

We now discuss the implementation of the connectors: the pipe (|), parallel (&) and sequential (+) composition and the multiplex (+) operators.

3.3.1 Pipe

The pipe connector composes two push-flow style components to a sequential partial data-flow. Actually, there are four different pipe connectors, two connecting a source to a sink or a filter and two connecting a filter to another filter or a sink. But the four connectors differ only in the types they require and return; otherwise, their behaviour is the same. All four connector have the following protocol:

- Connect the provided `put` of the right operand to the required `put` of the left operand.
- Connect the provided `close` of the right operand to the required `close` of the left operand.

Below are the implementations of the pipe connectors that connects a source to a filter, respectively a filter to another filter. The code for the other three pipe connectors differs only in the component instance to be returned.

```
pipeSourceFilter(left)(right):
  left.reqPut.bind(right.put)       # wiring services
  left.reqClose.bind(right.close)
  return                            # return composite source
    reqPut = right.reqPut
    reqClose = right.reqClose

pipeFilterFilter(left)(right):
  left.reqPut.bind(right.put)       # wiring services
  left.reqClose.bind(right.close)
  return                            # return composite filter
    put = left.put
    close = left.close
    reqPut = right.reqPut
    reqClose = right.reqClose
```

The pipe connector wires the provided `put` and `close` services of the right-hand instance to the appropriate services of the left-hand instance.

3.3.2 Duplex

The duplex connector composes two sinks. The resulting component is a sink that duplicates the incoming data flow and directs one flow into each operand. The code for this connector is straightforward:

```
duplex(left)(right):
  put(X):
    left.put(X)
    right.put(X)
    return ()
  close():
    left.close()
    right.close()
    return ()
  return
    put = put
    close = close
```

Calls to the `put` and `close` services of the resulting sink are forwarded to both operands.

3.3.3 Sequential

This connector combines two sources sequentially. The resulting component is a source that produces first all data produced by the left-hand operator, then

all data produced by the right-hand operator.

```
sequential(left)(right):
  reqPut = ReqOnceBlock()
  reqClose = ReqOnceBlock()
  myLeftClosed = newChannel()
  closeLeft():
    myLeftClosed.send()
  putRight(X):
    myLeftClosed.receive()
    myLeftClosed.send()
    reqPut(X)
  left.reqPut.bind(reqPut)
  left.reqClose.bind(closeLeft)
  right.reqPut.bind(putRight)
  right.reqClose.bind(reqClose)
  return
    reqPut = reqPut
    reqClose = reqClose
```

The resulting instance is a source. Thus, it requires both a `put` and a `close` service. The `myLeftClosed` channel signals when the first source closes (see `closeLeft`). The second source blocks until the first source has been closed (see `putRight`). The bindings of services are following in the source code. Finally, the resulting source is returned.

Another method of implementing this connector is to buffer the elements produced by the second source rather than block that source. Note that this implementation would result in a different run-time behaviour and should be considered as another connector. An enhanced push-flow style might thus provide both a `sequentialBlock` and a `squentialBuffer` connector.

3.3.4 Parallel

The parallel connector joins two sources into a resulting source. It can be considered as the reverse of the duplex connector: whereas `duplex` splits the data flow, `parallel` joins two flows.

```
parallel(left)(right):
  reqPut = ReqOnceBlock()
  reqClose = ReqOnceBlock()
  myLeftClosed = Store(false)
  myRightClosed = Store(false)
  leftClose:
    if (myRightClosed.get())
      then: reqClose()
      else: myLeftClosed.set(true)
  rightClose:
    if (myLeftClosed.get())
```

```
      then: reqClose()
      else: myRightClosed.set(true)
  left.reqPut.bind(reqPut)
  right.reqPut.bind(reqPut)
  left.reqClose.bind(leftClose)
  right.reqClose.bind(rightClose)
  return
    reqPut = reqPut
    reqClose = reqPut
```

Joining the data flows is simple: the required `put` services of the operands are bound to `reqPut` which represents the required `put` service of the resulting source but is internally a *provided* service. The tricky part is the closing signal. The resulting source may only close when both operands have been closed. Thus, the two stores `myLeftClosed` and `myRightClosed` are introduced, which contain `true` when the corresponding source has been closed. The required `close` services are bound to `leftClose` or `rightClose` respectively, which set the content of the proper store to true and close the resulting source only if the other operator has already been closed.

3.4 Examples

Below are some sample push-flow style components that give an idea how to use this style, how the above explained things actually work, and how easy and nice this form of software composition can be.

```
Echo = newSource
  instance(args):
    reqPut = ReqOnceBlock()
    reqClose = ReqOnceBlock()
    run
      do:
        reqPut(args)
        reqClose()
```

The `Echo` component is a source (requiring a `put` and a `close` service) that starts an agent at instantiation that pushes the instantiation argument (`args`) downstream and then closes the data-flow. Note that the `reqPut` service blocks until it is bound to the corresponding provided service of a filter or sink.

```
Itemize = newFilter
  instance:
    reqPut = ReqOnceBlock()
    reqClose = ReqOnceBlock()
    put(X):
      reqPut("[" + X + "]")
    close(): reqClose()
```

This filter modifies each data element passing it by converting it to a string representation and enclosing it in brackets.

```
Count = newFilter
  counter = Store(0)
  instance:
    reqPut = ReqOnceBlock()
    reqClose = ReqOnceBlock()
    put(X): counter.set(counter.get() + 1)
    close():
      reqPut("Counted " + counter.get() + " items.")
      reqClose()
```

The `Count` filter counts and consumes the data elements pushed towards it. When the flow is closed, it sends the number of elements counter further downstream.

```
Print = newSink
  instance:
    put(X):
      print(X)
      print(" ")
    close():
      println()
      println(">>> print closed")
```

This sink displays all data elements consumed on the screen, inserting a space between two elements. The closing of the flow is signaled by a screen output, too.

Having defined some push-flow style components, some examples can be implemented that show the usage of the different components and connectors:

```
  Echo("Sequential") + Echo("Composition") | Print()     # 1
  Echo("Parallel") & Echo("Composition") | Print()       # 2
  Echo("Multiplex") | (Print + Print)()                  # 3
  PrintAndCount = Itemize | ((Count | Print) + Print)    # 4
  Echo("Hello") + Echo("World") | PrintAndCount()        # 5
```

Some facts to be noted:

- As mentioned before, the `Echo` component can be instantiated with a parameter, the data element to be produced.
- Whereas the line 1 always produces the output 'Sequential Composition', the line 2 can produce either the text 'Parallel Composition' or 'Composition Parallel'.
- Both component and instance composition are used in the above example. Can you identify them?
- `PrintAndCount` is a composite sink component.

4 Grouped Actor Style

Actors constitute a foundational model of communicating agents. [4] Actors are autonomous entities that exchange messages asynchronously with each other. Messages sent to an actor are stored in a queue. The actor can read the next pending message, send messages to other actors, create new actors and become another actor, i.e. replace its behaviour.

Here, the focus lies not on the specification of actors, but on the definition of *groups* of actors that share a communication medium (the group). In such approaches, an actor can typically be member of multiple groups [6][7].

4.1 Example: Chat Application

The typical example of actors sending messages to each other is a chat application. Such a program deals (a) with GUI interaction and (b) with coordination. We want a clear seperation of the two aspects. Thus, GUI classes are implemented in Java, then imported into Piccola as external components and adapted to the grouped actor style. The style takes care of the coordination and lets us explicitly specify the architecture of the application.

The chat server is a simple Java class displaying a frame on the screen that allows to enter a name and then login with a button:

```
class ChatServer extends java.awt.Frame {
  public Button getLoginButton() { ... }
  public String getName() { ... }
  public void clearName() { ... }
  ...
}
```

The chat client is a class almost as simple that displays incoming messages in a frame on the screen. Messages can be typed in a seperate text input control and sent by clicking on the send button. An addressee can be specified in an other text input box. If the addressee is left empty, the message is broadcasted to all other clients. The class provides the following methods:

```
class ChatClient extends java.awt.Frame {
  public String getMessage() { ... }
  public String getAddressee() { ... }
  public Button getSendButton() { ... }
  public void clearMessage() { ... }
  public void addLine(String) { ... }
  ...
}
```

The server will be adapted to a group component of the style, the clients to actors.

component type	provided services	required services
Group	send()	receive()
Actor	receive() id()	send()
Message	isAddressee() type()	
RActor	receive() send() id()	send() handle()
Handler	handle()	send() id()
Service	()	

Table 3: Component types of the grouped actor style

Following, we will introduce the component types and connectors of the grouped actor style, then the Piccola wrappers for the chat server and client will be implemented.

4.2 Component Types

The obvious component type in this style is the *actor*. An actor can receive messages and must be identified. Thus, an actor component must provide a `receive` and an `id` service. The actor requires a `send` service to be able to send messages to the group.

A group provides a `send` service for its actors that relays a message sent by one of its actors and forwards it to the appropriate destination actor(s). To perform this task, the group requires a `receive` service.

A *message* can also be seen as a component, although messages are communicated between actors rather than composed. A message provides a `isAddressee` service that returns `true`, if the argument is an identifier of a valid adressee. The `type` service returns the type of themessage, i.e. it's component abstraction.

A special kind of actor is the *reactive* actor. In contrast to an active actor that can act (e.g. send a message) on it own, a reactive actor can only react to incoming messages. The default reactive actor just consumes messages. It delegates the actual message handling to handler components, thus requiring a `handle` service.

A *handler* is a message type/service pair. If the type of the message to be handled corresponds the message type of the handler, the service is executed. Thus, a *service* is just an abstraction providing call semantics (see table 3).

EmptyGroup	$\rightarrow$	*Group*	default empty actor group
Group + *Actor*	$\rightarrow$	*Group*	add actor to group
Actor + *Actor*	$\rightarrow$	*Group*	compose new group
RActor	$\rightarrow$	*Actor*	r-actor is an actor
EmptyRActor	$\rightarrow$	*RActor*	default r-actor
RActor | *Handler*	$\rightarrow$	*RActor*	extend with handler
Handler | *Handler*	$\rightarrow$	*RActor*	
Message –> *Service*	$\rightarrow$	*Handler*	

Table 4: Signature of the grouped actor style

4.3 Signature

In contrast to the push-flow style signature, which suggests a top-down view of an architecture, the signature of the grouped actor style (see table 4) implies a bottom-up view. The smallest possible group is the empty group represented by the default component `EmptyGroup`. An actor can be added to a group using the + operator. The following expression is just syntactic sugar:

$$ActorA + ActorB := EmptyGroup + ActorA + ActorB$$

The rule $RActor \rightarrow Actor$ expresses that a reactive actor can be used as an actor. `EmptyRActor` is the default reactive actor that just consumes messages. A handler can be attached to a reactive actor, yielding another reactive actor. The following expression is syntactic sugar:

$$HandlerA \mid HandlerB := EmptyRActor \mid HandlerA \mid HandlerB$$

The last rules of the signature specify how to compose handlers. A handler can be obtained by associating a message type with a service. It will execute the service when it handles a message of the appropriate type.

4.4 Connectors

The following, the connectors of this style are implemented: the add connector (+), and the reactive actor (|) and handler (–>) composition connectors.

4.4.1 Add

The add connectors adds an actor to a group, yielding another group. This connector has the following responsibilities: It must forward messages sent by the right-hand operand (the actor) to the group. This is done by binding the required `send` of the actor to the provided one of the group. The second responsibility is to deliver messages from the group to the actor if it is a valid addressee. The `receive` service, bound to the required `receive` of the group, takes care of this (see figue 3).

Whether the actor is a valid addressee is checked by calling the `isAddressee` service of the message with the `id` of the actor as argument.

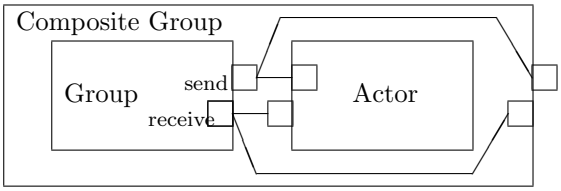

Figure 3: Adding an actor to a group

```
add(group)(actor):
  reqReceive = ReqOnceIgnore()
  receive(message):
    if (message.isAddressee(actor.id()))
      then: actor.receive(message)
    reqReceive(message)
  group.reqReceive.bind(receive)
  actor.reqSend.bind(group.send)
  return
    reqReceive = reqReceive
    send = group.send
```

The resulting component is another group. It therefore requires a `receive` of actors to be composed with the group yet. Messages are forwarded to the required receive in the `receive` service. As the group should function with an unbound `reqReceive`, it must ignore calls if unbound, and thus be of the required service type `ReqOnceIgnore`. The resulting group must also provide a `send` service. This is simply the original `send` of the group.

4.4.2 Compose Reactive Actor

This connector adds a handler to a reactive actor, producing a new reactive actor. It connects the provided `handle` service of the handler to the required one of the actor. Furthermore, the provided `send` and `id` services of the reactive actor are wired to the corresponding required services of the handler. The interface of the actor is returned, representing the composite interface.

```
composeRActorHandler(ractor)(handler):
  ractor.reqHandle.bind(handler.handle)
  handler.reqSend.bind(ractor.send)
  handler.reqId.bind(ractor.id)
  return
    reqSend = ractor.reqSend
    reqHandle = ractor.reqHandle
    receive = ractor.receive
    send = ractor.send
    id = ractor.id
```

4.4.3 Compose Handler

This connector composes a handler out of a message and a service. The resulting handler requires a `send` and an `id` service (1). These services may be called by the service component in order to react on a handled message. The provided `handle` service (2) has the following behaviour: If the type of the handled message (i.e. the component abstraction of the message) equals the type of the message component (3), the service component will be called with the message, and the `send` and `id` services as argument (4).

```
composeMessageService(message)(service):
  reqSend = ReqOnceBlock()                  # 1
  reqId = ReqOnceBlock()
  handle(msg):                              # 2
    if (message.type() == msg.type())       # 3
      args =
        msg = msg
        send(msg): reqSend(msg)
        id: reqId()
      service(args)                         # 4
  return
    handle = handle
    reqSend = reqSend
    reqId = reqId
```

The following requirements for the original components can be identified: The service component must provide call semantics. The message instances must provide a `type` service that returns their component abstraction (3). The message component abstractions must be comparable with the equality operator (`==`).

4.5 Components and Wrappers

As the grouped actor style uses a bottom-up architecture, some minimal default components (`EmptyGroup` and `EmptyRActor`) are needed. Additionally, the style provides a `makeActor` wrapper that takes care of much of the standard work when implementing an actor, and a `makeMessage` factory that produces message components.

4.5.1 EmptyGroup component

The empty group is the starting point for every actor group to be composed. It's functionality is straight forward. It receives messages from the chain of actors that may be attached to it through the provided `send` service and resends those messages back to the chain by calling a required `receive` service. This service is of type `ReqOnceIgnore` in order to let the empty group behave the same way

as a non-empty group [5] (see add handler in section 4.4).

```
EmptyGroup = newGroup
  instance:
    reqReceive = ReqOnceIgnore()
    send(msg):
      reqReceive(msg)
    return
      reqReceive = reqReceive
      send = send
```

4.5.2 EmptyRActor component

The empty reactive actor is a special actor component. It has an internal message queue (`myQueue`), where incoming messages are stored by the `receive` service. The actor `id` is obtained from the (`actorCounter`) counter and bound to `myId`. As any actor component, it requires a `send` service to send messages to the group. The `EmptyRActor` provides `send` and `handle` services to handler components that may be attached to it.

```
EmptyRActor = newRActor
  instance(args):
    myQueue = Queue()
    myId = actorCounter.next()
    reqSend = ReqMultiIgnore()
    reqHandle = ReqMultiIgnore()
    id: myId
    receive(message): myQueue.push(message)  # 1
    send(args):
      reqSend
        args.type
          content = args.content
          from = myId
          to = (to = (), args).to
    def loop() =
      message = myQueue.pull()              # 2
      reqHandle(message)                    # 3
      loop()
    run(do: loop())
    return
      reqSend = reqSend
      reqHandle = reqHandle
      receive = receive
      send = send
      id = id
```

[5] Actually, the unbound behaviour of `receive` doesn't matter, as it will never be called when unbound.

The run-time behaviour of this actor is the following: If a message is in the message queue (1), it is retreived (2) and forwarded to the required `handle` service (3).

4.5.3 makeActor wrapper

The `makeActor` wrapper takes care of the standard work of defining an actor. It provides an internal message queue (1), the `id` of the actor, the required `send` and the provided `receive` service. The user of this wrapper has just to provide an actor body, which is executed by the actor (2).

The body can use four services of the actor, which are passed as arguments (3): `nextMessage` (4) retreives and returns the next pending message in the message queue. It blocks, if the queue is empty. `hasMessage` (5) checks whether there is a pending message. `send` (6) takes a message body as argument and wraps it as a message instance. The `id` service (7), finally, returns the `id` of the actor.

```
makeActor(args): newActor
  instance:
    myQueue = Queue()                          # 1
    myId = actorCounter.next()
    reqSend = ReqMultiIgnore()
    receive(message): myQueue.push(message)
    nextMessage: myQueue.pull()                # 4
    hasMessage: myQueue.empty().not()          # 5
    send(args):                                # 6
      reqSend
        args.type
          content = args.content
          from = myId
          to = (to = (), args).to
    id: return myId                            # 7
    run
      do:
        args.do                                # 2
          nextMessage = nextMessage            # 3
          hasMessage = hasMessage
          send = send
          id = id
    return
      reqSend = reqSend
      receive = receive
      id = id
```

4.5.4 makeMessage factory

This factory service returns a message component. Different message components have a distinct `id` provided by the `messageCounter` counter (1). They can be compared using the `==` operator (2). Handlers make use of this feature.

A message has the following labels (3): `from` is the `id` of sending actor, `to` is the `id` of the destination actor. If `to` is empty, the message will be broadcasted. The content of the message is a form that will be bound to the `content` label. The `isAddressee` (4) service returns true, if the id passed as argument is the id of an addresse of the message. The `type` service returns the message component from which the message has been instantiated.

```
makeMessage:
  def Msg =
    component = newMessage
      instance(args):
        from = args.from                  # 3
        to = (to = (), args).to
        content = args.content
        isAddressee(id):                  # 4
          result = if (isEmpty(to))
            then: true
            else: to == id
          return result
        type: Msg
        return
          from = from
          to = to
          content = content
          isAddressee = isAddressee
          type = type
    myId = messageCounter.next()          # 1
    equals(right): myId == right.id()     # 2
    return
      component
      id: myId
      __== = equals                       # 2
  return Msg
```

4.6 Example: Chat Application Revisited

Given the implementation of the grouped actor style, we can write the Piccola wrappers for the chat application example.

In order to send chat messages around, a new message component is needed:

```
ChatMsg = makeMessage()
```

The chat client is adapted to an actor in the grouped actor style, as it can receive and send messages. The chat server plays the role of the group, as clients (actor) can be added to it. Naturally, the Java classes do not conform their intended component types' interfaces. Thus, two abstractions that wrap the Java classes as actor, or group respectively, are needed.

```
newChatClient = newActor
  instance(args):
    reqSend = ReqMultiIgnore()
    frame = intoAWTComponent(JavaChatClient.new())
    frame.setTitle(val = args)
    frame.show()
    receive(message):
      frame.addLine(val = message.from + ": " + message.content
    send = intoAWTComponent(frame.getSendButton())
    send ? Action
      do:
        to = frame.getAddressee()
        to = if (to = "")(then: (), else: to)
        msg = frame.getMessage()
        reqSend
          ChatMsg(from = args, content = msg, to = to)
        frame.clearMessage()
    return
      id: args
      reqSend = reqSend
      receive = receive
```

The `newChatClient` component abstraction has the following behaviour: On instantiation, a Java `ChatClient` object is generated. `receive` forwards incoming messages to the Java object. If the send button is pressed, a new message is generated and sent by calling the required `send` service.

The `newChatServer` component abstraction creates a store bound to `myGroup`, which will contain the actor group composite instance. A Java `ChatServer` object is created on instantiation. If the login button is pressed, a new chat client instance is created and added to the current group stored in `myGroup`. Thus, the chat actor group can be dynamically extended.

```
newChatServer:
  myGroup = Store()
  myGroup.set(EmptyGroup())
  frame = intoAWTComponent(ChatServer.new())
  frame.show()
  login = intoAWTComponent(frame.getLoginButton())
  login ? Action
    do:
      if (frame.getName() == "")
        else:
          myGroup.set(myGroup.get() + newChatClient(frame.getName()))
          frame.clearName()
  return
    myGroup.get()
```

Using the above component abstraction and the signature, a chat application can be intantiated very easily:

```
newChatServer() + newChatClient("Franz") +
  newChatClient("Oscar") + newChatClient("Stefan")
```

5 Regulated Coordination Style

Regulated coordination has been proposed by Minsky [8]: Agents communicate with each other with *law-governed* message passing, i.e. the exchange of message between agents is controlled by a set of rules.

A coordination policy P is a triple (M, G, L) where

- M is a set of messages
- G is a group of agents that are permitted to exchange messages in M
- L is a set of rules (a law) regulating the exchange of messages between the agents in G.

The grouped actor style models the concept of agents communicating by message passing. The regulated coordination style will be built on top of that style, with the actor group representing the policy and the actors being the agents.

The law will be enforced on the agents of a policy by attaching a `controller` to each agent participating in the policy. A controller maintains a link to the rules and a state, which can be accessed by the rules. The controller applies appropriate rules to every message received or sent by the agent.

5.1 Example: Electronic Voting

A good example for this policy is electronic voting. Such a vote must be fair, i.e. it must be ensured that every participant may not cast more than one vote. This is ensured by the law (i.e. the rules). The example is an adaption of the electronic voting policy introduced by Minky [8].

As in the previous example, the GUI functionality is encoded as Java class, whereas the coordination is provided by the regulated coordination style.

The `VoteClient` Java class is an adaption of the chat client class (see section 4.1). It has four buttons to initiate a vote, end a vote, and replying yes or no.

```
class VoteClient extends java.awt.Frame {
  public String getMessage() { ... }
  public Button getAskButton() { ... }
  public Button getEndButton() { ... }
  public Button getYesButton() { ... }
  public Button getNoButton() { ... }
  public void clearMessage() { ... }
  public void addLine(String) { ... }
  ...
}
```

component type	provided services	required services
Rule	apply()	
Policy	send() apply()	receive() apply()
Actor	receive() id()	send()

Table 5: Component types of the regulated coordination style

EmptyPolicy	$\rightarrow$	*Policy*	empty policy
Policy & *Rule*	$\rightarrow$	*Policy*	add rule to policy
Rule & *Rule*	$\rightarrow$	*Policy*	compose new policy
Policy + *Actor*	$\rightarrow$	*Policy*	add actor to policy
Event 'of' Message(*action* :)	$\rightarrow$	*Rule*	compose rule

Table 6: Signature of the regulated coordination style

The `VoteClient` class is adapted to an actor using a Piccola wrapper that is very similar to the one of the previous example.

The voting policy needs several types of messages: A `StartVoteMsg` message initiates a vote, an `EndVoteMsg` message terminates it. Both messages are triggered by the corresponding buttons. `CastVoteMsg` messages are used to transmit the replies back to the initiator.

The rules will be written using wrappers provided by the regulated coordination style.

5.2 Component Types

A *rule* is a simple component that provides just an `apply` service that applies the rule to the message and state passed as arguments.

A *policy* is an extension of an actor group that supports regulated coordination by providing an `apply` service that applies rules to sent or received messages. The `apply` service calls are forwarded to rules attached to the policy by calling the required `apply` service of the policy, which will be bound to the provided `apply` of rules composed with the policy.

The actor component of the grouped actor style is reused in this style.

5.3 Signature

The signature of the regulated coordination style is similar to that of the grouped actor style, also implying a bottom-up view of the architecture. The simplest component is the empty policy. It contains neither rules nor agents. A rule can be added to a policy with the & operator. The third rule of the signature (see

table 6) is just syntactic sugar:

$$RuleA \ \& \ RuleB := EmptyPolicy \ \& \ RuleA \ \& \ RuleB$$

An actor is added to a policy using the + connector, yielding another policty. This connector attaches controller functionality to every actor that joins the policy.

The last line of the signature table shows how to define a rule. This will be described in more detail below.

5.4 Connectors

This style uses the following two connectors: Add actor (+) adds an actor to a policy, add rule (&) adds a rule to a policy.

5.4.1 Add Actor

The first connector discussed is the heart of this style. The policy-actor composition connector provides the functionality of Minskys' controllers: It regulates the sending and receiving of messages of the actor being added to the policy.

First, an instance of a `Map` (a component that maps keys to values) is created and associatied with `myState` (1). This map will represent the state of the controller. Then, the requested `receive` service of the composite policy is defined (2). The internal `receive` service checks whether `actor` is a valid addressee of a message (3). If so, the apply service of the policy is called, which will apply matching rules (4). The types of rules to be considered are of type `ArrivedRule` (5). Furthermore, the rule may access the message (`msg`) and the state. The rule can also `deliver` the received or any other message to the actor (6). Finally, the service calls the required `receive` service (7), making the message available for actors that will be added later.

```
composePolicyActor(policy)(actor):
  myState = Map()                              # 1
  reqReceive = ReqOnceIgnore()
  receive(message):                            # 2
    if (message.isAddressee(actor.id()))       # 3
      then:
        policy.apply                           # 4
          ruleType = ArrivedRule               # 5
          msg = message
          state = myState
          deliver(msg):                        # 6
            if (isEmpty(msg))
              then: actor.receive(message)
              else: actor.receive(msg)
    reqReceive(message)                        # 7
  send(message):
```

```
    policy.apply                                # 8
      ruleType = SentRule
      msg = message
      state = myState
      forward(msg):                             # 9
        if (isEmpty(msg))
          then: policy.send(message)
          else: policy.send(msg)
  policy.reqReceive.bind(receive)               # 10
  actor.reqSend.bind(send)
  return
    reqApply = policy.reqApply
    reqReceive = reqReceive
    apply = policy.apply
    send = policy.send
```

The internal `send` service applies rules for messages (8) send by `actor`. The rules applied are of type `SentRule` and may `forward` the message to the policy (9). Note that no message will be delivered or forwarded, if a rule doesn't explicitly allows it.

The next two lines bind the internal services to the appropriate required services of the policy or the actor (10). Here, it's evident that the sending and receiving are intercepted by the connector (the controller).

5.4.2 Add Rule

This connector composes a policy and a rule by binding the required `apply` service of the policy to the provided one of the rule. The resulting component is, of course, another policy.

```
composePolicyRule(policy)(rule):
  policy.reqApply.bind(rule.apply)
  return
    reqApply = policy.reqApply
    reqReceive = policy.reqReceive
    apply = policy.apply
    send = policy.send
```

5.5 Components and Wrappers

The single default component of the regulated coordination style is the `Empty-Policy`. The style also provides the two wrappers `Sent` and `Received`, which allow us to write rules in a very Minsky-like style.

5.5.1 EmptyPolicy

The empty policy component is very similar to the empty group component of the grouped actor style (see section 4.5). It relays messages coming down the chain of provided `send` services and returns them back up the chain of required `receive` services. Additionally, the `EmptyPolicy` forwards calls to the provided `apply` service to the rules' apply services, that are bound to the required `apply`.

```
EmptyPolicy = newPolicy
  instance:
    reqApply = ReqMultiIgnore()
    reqReceive = ReqOnceIgnore()
    apply(args): reqApply(args)
    send(msg): reqReceive(msg)
    return
      reqApply = reqApply
      reqReceive = reqReceive
      apply = apply
      send = send
```

5.5.2 Events and Rules

In regulated coordination, rules are triggered by *events*. There are two kinds of events: *Sent(M)* occurs if an agent tries to send a message of type M, *Arrived(M)* takes place if a message of type M is about to be received be an agent.

A rule definition starts with an event description. The rule is triggered, if the specified event occurs. The rule can also define a condition (`cond`) and an `action`, which are both services. The action is executed if, and only if, the condition returns `true`.

Below is an example of a rule, that counts the number of messages of type MyMessage that are addressed to an actor with the id "minsky".

```
CountRule = Sent `of` MyMessage
  cond(args): args.msg.to == "minsky"
  action(args):
    state = args.state
    if (state.containsKey("count").not())
      then: state.put("count")(0)
    state.put("count")(state.get("count") + 1)
    args.forward(args.msg, count = state.get("count"))
```

The first line of code specified the event; the rule is triggered, if a message of type `MyMessage` is sent by an actor. The action is only executed, if the condition is `true`, i.e. if the addressee has the id 'minsky'. The body of the action makes sure that the state contains a 'count' key, increments the value associated with that key and finally forwards the message, extended by a `count` label, to the policy.

Another example is the `IgnoreRule`, that ignores all messages of type `msgType` from the actor with the specified `senderId`. Only if the message is *not* sent by the specified actor, the condition evaluates to `true` and the action will be executed, delivering the message unchanged to the actor. Note that `IgnoreRule` actually is a rule abstraction.

```
IgnoreRule(senderId)(msgType): Arrived ‘of‘ msgType
  cond(args): args.msg.from == senderId
  action(args): args.deliver()
```

The `Sent` and `Arrived` event wrappers provide supporting functionality as well as the syntax for defining rules. The code of `Sent` is shown below:

```
Sent =
  of(messageType)(rule) = newRule
    cond = (cond: true, rule).cond
    action = (action: (), rule).action
    instance:
      apply(args):
        if ((args.ruleType == SentRule) &&
            (args.msg.type() == messageType))
          then:
            if (cond(args))(then: action(args))
```

Note that the infix operator `of` used in the rule definitions translates to the `of` service provided by `Sent` or `Arrived` respectively.

5.6 Example: Electronic Voting Revisited

Now, we can complete the code for the electronic voting policy. The rules are implemented using the `Sent` and `Arrived` wrappers introduced above.

The electronic voting policy consists of six rules. The first is given as an example:

```
EV1 = (Sent ‘of‘ StartVoteMsg)
  cond(args):
    state = args.state
    return state.containsKey("askingVote").not()
  action(args):
    state = args.state
    msg = args.msg
    state.put("yes")(0)
    state.put("no")(0)
    state.put("askingVote")(msg.content)
    args.forward()
```

As in the previous example, the application code is very small, and the architecture is explicit:

```
VotingPolicy = EV1 & EV2 & EV3 & EV4 & EV5 & EV6

VotingPolicy() + newVoteClient("Franz") +
  newVoteClient("Stefan") + newVoteClient("Oscar")
```

6 Lessons Learned

In this section, we summarize our experience with JPiccola 2.1. We also make some proposals for the improvement of Piccola.

6.1 Strengths of Piccola

The mapping of composition concepts to Piccola as well as the implementation of coordination styles has been quite easy and straight-forward. This suggests that the Piccola is a good low-level basis for software composition as well as for coordination. Forms have proven to be a simple, flexible and usable programming concept in practice. Piccola is based on a simple formal model, the pi-calculus, but allows to model higher-level concepts inside the language. It is easy to import external components (e.g. Java classes and objects) into Piccola. We conclude that Piccola is a powerful, general composition and coordination language.

6.2 Proposals for Improvements

Below are some proposals for improvements of the Piccola language, which are based on the author's experience with Piccola 2.1. Note that some of these suggestions have already been integrated in Piccola 3.

6.2.1 Indentation

Piccola syntax has its niceties, but there are also some drawbacks. Experience has shown that the need for exact indentation often leads to syntax or run-time errors or even unexpected run-time behaviour. These errors can be caused by just a missing or overfluous white-space character, and their location is quite hard to detect. Therefore, we propose to use an indentation-independent parsing.

6.2.2 Labels as First Class Values

Labels are a central concept of Piccola. When programming in Piccola, there is a need to deal with labels. A programmer typically needs to do the following things in Piccola:

- Check, whether a form contains a specific label.

- Iterate over all labels of a form.
- Iterate over certain labels of a form.
- Restrict a form to a set of labels.
- Define and operate with sets of labels.
- Convert strings to labels and vice versa.

The things above can be done to a certain degree in Piccola. To check, for example, whether the form `form` contains the label `labels`, one can write the following piece of code:

```
isEmpty((label = (), form).label).not()
```

Of course, it would be convenient not to write this code over and over again, but use an abstraction:

```
contains(form)(label): isEmpty((label = (), form).label).not()
```

This requires, of course, that one can abstract over labels, i.e. that labels are first class values. As now, there is a limited support for this provided by the `forEachLabel` built-in service and the libaray services defined in `labels.picl`, which make use of `forEachLabel`. For example, it allows the implementation service that returns a set of all labels defined by a form:

```
labelsOf(form):
  set = Set()
  forEachLabel
    form = form
    do(label): set.add(label)
  return set
```

The form passed to the `do` service is a representation of a label, providing `name` to access a string representation of the label as well as a `project` service.

Still lacking is a convenient way to define label constants. Currently, this has to be done like this:

```
label = getLabel(myLabel = ())
```

Piccola 3 offers a `label` service that returns a form representing the label specified by the argument. Such a *first-class label* provides the following services:

```
lbl = label("a")     # represents label 'a' as form
lbl.restrict(F)      # returns F with the visibility
                     # of label 'a' restricted.
lbl.exists(F)        # returns isEmpty(a = (), F).a).not()
lbl.name()           # returns "a"
lbl.bind(F)          # returns (a = F)
lbl.project(F)       # returns F.a
```

6.2.3 Syntax Extensions

The coding experience with Piccola has shown that there seem to be some code patterns describing higher level language conceps. Special syntax for some of these patterns has already been introduced in SPiccola (Piccola 3).

Such a pattern that is occuring very often is the public interface. When writing a component, the internals are hidden by returning a form that represents the public interface (see 2.1.1):

```
Store:
  ...
  return
    get = get
    set = set
```

If the public interface is large, the code gets quite clumsy. A nice way to express the intention would be to specify only the labels that should be epxorted:

```
Store:
  ...
  !{get, set}
```

This code could be interpreted in the following way: `{get, set}` is a set of labels, which can be represented as a form, the form being the characteristic function of the set by mapping labels to 0 or 1. The hiding labels not in the set could be modeled by form restriction.

6.2.4 Global Operators

One of the most awkward things in this work has been the need to define signature wrappers that attach operators to the forms representing components. This need comes from the lacking possibility for defining global operators.

Thus, a mechanism that maps infix operators to global services rather than services of the left-hand operand is proposed. Such a mechanism would need a kind of type system in order to be able to select from different global services depending on the 'type' of the operand forms.

It is possible to define global operators in Piccola 3.

6.2.5 Type System

Currently, there is no way to detect misconfigurations in Piccola styles. A forbidden configuration, such as

```
Echo("Hello") | newChatClient("Stefan")
```

results in a error message like 'No label "put" in form "right".' in the code of the push-flow style's source-sink pipe connector. This error message is correct, but too low-level. What we'd expect is something like "Filter or Sink component expected after pipe operator".

Such high-level error detection mechansims and the use of global operators require some kind of type system, where Piccola forms have an associated type. Following are some possible starting points:

- The type of a form is the set of the form's bound labels. A type hierarchy with sub- and supertypes could be defined in terms of super- and subsets of labels. How should we treat labels bound to the empty form?
- A form provides a label `type` that returns a type description of the form (similar to the message component of the grouped actor style, see section 4.5).
- A syntax extension provides a type system that is transparent to Piccola itself.

7 Future Work

It has been shown that it is possible to factor out the coordination aspects of programming into connectors while leaving the computational code in the components. Examples are the join operators of the push-flow style or the policy-actor composition connector of the regulated coordination style.

An open issue is the definition of the behaviour of required services. It is clear that some unbound required services should block and others should not. Also, some services must be able to be bound more than once, wheras other required services should be bound to one provided service only. But the behaviour of required service is more a coordination than a computational aspect. Thus, it should be defined in connectors rather than in components. The obvious way to do is is to turn required services into "twin" slots where a connector does not only plug in a provided service, but also the behaviour mentioned above.

8 Conclusion

This paper has shown that higher-level concepts of software architecture such as components and connectors can be modeled in Piccola. Its abstraction power allows to implement a composition model based on plugging on top of wiring mechanisms that are again built on top of the native Piccola features (forms and channels). The experience made with Piccola has shown that it is a useful language to implement software composition.

We have introduced a meta framework for coordination styles in Piccola which clearly distinguishes between immutable components and their run-time instances. The frameworks supports a kind component composition that is defined in terms of instance composition.

This framework has been used to implement three different, well-known coordination models as styles. Two toy applications show that our intentions have been achieved: External components (e.g. Java classes) are imported to Piccola, adapted to a given style and plugged toghether using connectors of that style.

When looking at the implementation of the toy applications, we notice that

- there is a clear separation of computational code which resides in components, and coordination code provided by the style.
- the architecture of the applications can be clearly identified in the Piccola script.

We conclude that the above presented boot-strap approach (defining more and more complex and high level composition styles in Piccola) is promising both for coordination and component composition issues.

References

[1] Franz Achermann, Stefan Kneubuehl, and Oscar Nierstrasz. Scripting coordination styles. In António Porto and Gruia-Catalin Roman, editors, *Coordination Languages and Models*, LNCS 1906, pages 19–35, Limassol, Cyprus, September 2000. 1.1

[2] 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, an Object Oriented Approach*. Cambridge University Press., 2000. to appear. 1.1, 1.3

[3] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*. Kluwer, 2000. to appear. 1.1, 1.3

[4] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986. 1.2.3, 4

[5] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986. 1.2.3

[6] Fernanda Barbosa and José C. Cunha. A coordination language for collective agent based systems: GroupLog. In *Proceedings of SAC'00*, Como, Italy, March 2000. ACM. 4

[7] Juan-Carlos Cruz and Stéphane Ducasse. A group based approach for coordinating active objects. In *Proceedings of Coordination'99, LNCS 1594*, pages 355–371, 1999. 4

[8] Naftaly Minsky and Victoria Ungureanu. Regulated coordination in open distributed systems. In David Garlan and Daniel Le Mètayer, editors, *Proceedings COORDINATION'97*, LNCS 1282, pages 81–97, Berlin, Germany, September 1997. Springer-Verlag. 1.2.3, 5, 5.1

[9] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Langages for Concurrent Systems*, LNCS 924, pages 147–161. Springer-Verlag, 1995. 1.2.1

[10] 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. 1.1

[11] 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, 1999. 1.2.1

[12] Dennis Tsichritzis, Oscar Nierstrasz, and Simon Gibbs. Beyond objects: Objects. *IJICIS (International Journal of Intelligent & Cooperative Information Systems)*, 1(1):43–60, 1992. 1.2.1