Scripting Coordination Styles[1]

Franz Achermann, Stefan Kneubuehl, Oscar Nierstrasz

Software Composition Group, University of Bern[2]

Abstract. The fact that so many different kinds of coordination models and languages have been proposed suggests that no one single approach will be the best for all coordination problems. Different *coordination styles* exhibiting different properties may be more suitable for some problems than others. Like other architectural styles, coordination styles can be expressed in terms of components, connectors and composition rules. We propose an approach in which coordination styles are expressed as "component algebras": components of various sorts can be combined using operators that realize their coordination, yielding other sorts of components. We show how several coordination styles can be defined and applied using Piccola, a small language for composing software components. We furthermore show how *glue abstractions* can be used to bridge coordination styles when more than one style is needed for a single application.

1 Introduction

We are rapidly moving towards a world of spontaneously networked, multi-platform applications in which people, companies, web servers and mobile devices interact and exchange services with the help of software agents and components. Components will help to separate the stable from the evolving aspects of application domains, and to provide standardized interfaces and protocols for common services. Agents will help to represent both clients and service providers, negotiate terms of cooperation (both functional and non-functional aspects), manage aspects of concurrency (e.g., synchronization policies) and distribution (e.g., failure and recovery policies), and bridge differences in platform and protocol, in short, to *coordinate* the components.

Building such applications will not be trivial, even with the help of components, because too many low-level aspects come into play at once in the logic of the coordination code. There have been many proposals for high-level coordination models and languages, such as tuple spaces or the IWIM model [7], each of which proposes a way to separate coordination from computation. We believe that this is not enough, and take our cue from three other domains: *software architecture*, *scripting* and *object-orientation*. First, it is now well-established that different architectural styles exhibit different prop-

1. Proceedings Coordination '2000, António Porto and Gruia-Catalin Roman (Eds.), LNCS, vol. 1906, Springer-Verlag, Limassol, Cyprus, September 2000, pp. 19-35.

2. *Authors' address:* Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. *Tel:* +41 (31) 631.4618. *Fax:* +41 (31) 631.3965. *E-mail*: {acherman,kneubuhl,oscar}@iam.unibe.ch. http://www.iam.unibe.ch/~scg.

erties [10][27], and may be more or less well-suited to a given problem or problem domain. (For example, pipes and filters are great for text processing, but unsuitable for specifying GUI interaction.) Second, scripts and scripting languages can be very good at specifying how an application is constructed from external components and services according to a given architectural or compositional "style". Finally, object-oriented development encourages the programmer to develop his own model of the application and its domain using the abstraction facilities of object-oriented languages, rather than forcing the problem to fit a pre-packaged paradigm.

We propose an approach to composing and coordinating software components in which different high-level, algebraic *coordination styles* may be defined, and agents *script* components according to these styles. Furthermore, multiple styles may be required to address more complex problems and problem domains, since each style will exhibit different strengths and weaknesses. In this case, high-level *glue abstractions* may be needed to bridge the different styles.

Piccola is a small "composition language" designed to support this mode of software development. The core abstractions of Piccola are *forms* (immutable, extensible records), *agents* (communicating processes), and *channels* (locations where agents asynchronously exchange forms). The semantics of Piccola is given in terms of the πL-calculus, a variant of the π-calculus where agents exchange forms instead of tuples [20][21]. On top of this simple model, forms are used to build higher-level abstractions. Forms can be seen as "primitive objects", whose fields can store not only values but also abstractions, and they allow us to define styles of composition much in the same way that object-oriented languages are used to defined black-box frameworks of composable abstractions [18]. This encourages an *algebraic* view of styles, in which one defines different sorts of objects (forms) for a given style, each of which implements a given protocol, and algebraic operators (i.e., *connectors*, defined as fields of the forms) can be used to write expressions that compose objects and yield instances, possible of other sorts. For example, a stream (a kind of form) can be connected to a filter (another kind of form) by means of a pipe (a connector provided by the stream), yielding a new stream.

Basic services and components are written in conventional programming languages (presently Java, Squeak or Object Pascal), and appear to Piccola agents as native forms. Agents are *scripted* using forms that implement the coordination and composition abstractions of a particular style. Finally, styles may be combined if the appropriate glue abstractions have been defined in Piccola to bridge the styles.

In section 2 we introduce algebraic coordination styles in Piccola using the conventional example of streams and filters. In section 3 we show how Piccola can be used to develop very different coordination styles. We illustrate styles for event handling, grouped actors, and regulated coordination, each with their own compositional properties. In section 4 we show how glue abstractions can be used to bridge different styles. In section 5 we draw some observations from these experiments, and in section 6 we place our work in relation to others and outline future and ongoing work. Section 7 concludes the paper.

2 Component Algebras — Plugging vs Wiring

Software components are black-box abstractions that not only provide services, but may also require services in order to function correctly. Building an application from components, then, should be a *simple matter of wiring* these services together.

The problem is that wiring is the wrong paradigm for component-based development: wiring is an inherently low-level activity that can easily lead to configuration errors. Instead, it is more natural to *plug* components together. A *connector* captures a set of provided and required services that can be connected to a compatible socket in a single step. Furthermore, components that are plugged together hide their connected interfaces, *thus forming a larger component.*

We argue that this view of plugging components is the right way to think about coordination: components are computational elements that can be coordinated by plugging them into other components. The plugs and sockets represent not only services, but also the logic required to coordinate these services. Different kinds of components and connectors correspond to different coordination styles, and special adaptors will act as bridges between these styles.

We furthermore argue that coordination styles are most naturally implemented as *component algebras*, which define sets of components with similar plugs as the *sorts* of the algebra, and coordination abstractions as the *operators* of the algebra. A script, then, is an expression of the algebra that specifies how the components are plugged together. This immediately yields two important properties:

1. Scripts are *high-level specifications* that make the coordination of components explicit.
2. Scripts are *syntactically constrained* to generate only certain kinds of compositions, making it easier to reason about properties of the resulting configurations.

Note that we do not define an algebra in the mathematical sense, since the objects of our "algebra" may have mutable state. Thus, we don't necessarily have referential transparency and therefore cannot use the classical proof techniques. We mainly borrow the notation of signatures to achieve a declarative style of composition.

In the rest of this section we introduce the notion of expressing a coordination style as a component algebra with the familiar example of streams and filters. In section 3 we will see how this idea can be generalized to other coordination styles.

2.1 A Push-Flow Coordination Style

Let us consider a push-flow coordination style [19]. In this style, an individual component pushes data downstream to another component to which it is connected. There are three kinds of components: A source produces data and pushes it downstream. A filter allows an upstream component to push data towards it, process it, and pushes the result further downstream. Finally, a sink accepts data pushed towards it, and represents the end of the stream.

Sources, filters and sinks can be distinguished by the different basic services they provide and require (see Table 1). Basically, filters and sinks provide `put` and `close` services to upstream components which use them to push data and signal the end of the

	Provided services	**Required services**
Source		`put(X)`: write element downstream `close()`: signal end of stream
Filter	`put(X)`: accept a data element `close()`: close the input stream	`put(X)`: write element downstream `close()`: signal end of stream
Sink	`put(X)`: accept a data element `close()`: close input stream	

Table 1 provided and required services for stream style

stream. Sources and filters require these same services from downstream components to which they are connected.

Now, we can easily *wire* such components together by, for example, binding the provided services `put` and `close` of a filter to the corresponding required services of a source. Using a binding-oriented notation, as for instance in Darwin [13], this could be written as:

```
filter.put = source.put
```

There are two limitations to this approach: First it does not scale up, since we may only wire one connection at the time. Second, the composite is not a component.

Using an algebraic notation, on the other hand, we can define an operator to connect a filter to a data source and we can demand that the composite is again an instance with provided and required services. The composition rule *Source* | *Filter* → *Source* specifies:

- that the expression "`source | filter`" is again a *Source*.
- that the required `put` and `close` services of the source are bound by the provided services of the filter.

Following this approach, we can use the composite source as a first class value. It is important to notice that this is not only a question of syntax. Using the algebraic notation, connecting two components yields a composite instance, whereas the binding notation only binds services.

The set of composition rules of the stream style defines a *signature*, which is shown in Table 2. Connecting a source to a sink is the essential operation of this style. The data of the source is written to the sink and then the sink is closed. The *()* denotes the empty form, i.e. a component with no provided and required services.

The signature of a style is a level of abstraction above the notion of provided and required services. Instead of low-level wiring of provided and required services, we have defined a small language (an algebra) to work with. In this language we can compose streams without paying too much attention to the individual services of the components. The high level operators of the stream-language ensure that the services are bound correctly.

Source | *Sink*	→	*()*	Connect stream *s* to the sink
Source | *Filter*	→	*Source*	manipulate streams *s* using filter
Filter | *Filter*	→	*Filter*	composition two filters
Filter | *Sink*	→	*Sink*	build a new sink using the filter
Source + *Source*	→	*Source*	concatenate streams (sequential composition)
Source & *Source*	→	*Source*	merge streams (parallel composition)
Sink + *Sink*	→	*Sink*	multiplex a stream to two sinks

Table 2 Push stream signature

2.2 Implementing the Operators

How do we implement the operators in our stream style? In Piccola, forms represent interfaces to components. The component may be external to Piccola or it may be scripted entirely in Piccola. In the latter case the behaviour of a the component is implemented by a Piccola agent [1][20].

A *form* is an extensible record, given as a set of bindings. A binding maps a label to a value, which may be a nested form or a service. Thus a form is a kind of primitive object, providing public services. It is feasible to model advanced object-oriented features, such as inheritance using forms [25]. The required services are represented as *slots*, which are implemented as Piccola objects. Slots are analogous to futures: invoking a bound slot invokes the service it has been bound to; invoking an unbound slot delays the client until the slot is bound.

The following script defines a trivial example of a source:

```
mySource =
     reqPut = newSlot()          # required put service
     reqClose = newSlot()        # required close service
     run(do: (reqPut("Hello"), reqClose()))
```

A source does not provide any services. The service `run` is predefined in Piccola. It executes a block in a new agent and returns the empty form. Thus, the value of the form `mySource` contains two bindings, one called `reqPut` one called `reqClose`. These slots need to be bound by a client of the component. What is the behaviour of this component? The agent representing it is given as a `do` block and passed to `run`. It calls `reqPut` to invoke the required `put` service. But this service blocks, unless the slot has been filled. Thus, the agent representing the behaviour of `mySource` blocks until a sink or a filter is connected to it. Once connected, the agent writes “Hello” to the stream and closes the stream.

Wiring this source to a filter in Piccola means binding the provided services to the slots of the source:

```
mySource.reqPut.bind(filter.put)
mySource.reqClose.bind(filter.close)
```

The projection `filter.put` denotes the provided `put` service of the filter. Note that services are first class values in Piccola.

We would like to abstract from this low-level wiring by providing a high-level *connector* that treats the set of required or provided services as a plug. The extensibility of forms makes it possible to add such a connector to any source component. This connec-

tor will bind the services and return a form giving access to a new source component or the empty form. We can make an abstraction `asSource` that adds such a connector to any form:

```
asSource(S):
     S
     __|(Right):                          # define the | connector
          S.reqPut.bind(Right.put)
          S.reqClose.bind(Right.close)
          return asEmptyOrSource(Right)
```

The result of applying `asSource` to a form `S` is the form `S` extended with a `__|` service, representing the | infix operator. The right hand argument to the | operator may be either a sink or a filter, thus the form `Right` must provide the `put` and `close` plug. The connector binds these services and either returns the empty form if `Right` is a sink or it returns a new source with the required services of the filter (see Figure 1). The service `asSource` raises the level of abstraction by hiding the wiring behind connectors.

In order to plug `mySource` into a filter we apply `asSource` to it:

```
s = asSource(mySource)
s | filter | ...
```

The Piccola term `s | filter` is just syntactic sugar for `s.__|(filter)`, thus `s | filter` evaluates to a source, with the required services `put` and `close` of the filter.

The coordination between the two components, respectively the agents providing the behaviour for them is performed as procedure calls. Sinks and filters in the push flow style are always willing to accept data elements. An invocation of `put` on such instances is not allowed to block. In the case of the sequential composition of two streams (*Source* + *Source* → *Source*), the connector instantiates a pipe to buffer the data elements pushed by the second source. These elements are flushed when the first source calls `close`.

Connecting a source changes the state of the non-connected instance (with unbound slots) to a connected one. It is important to understand that individual *instances* are composed. Thus a source can only be connected to a single sink. The fact that we compose instances becomes essential when we glue components from different styles in section 4.

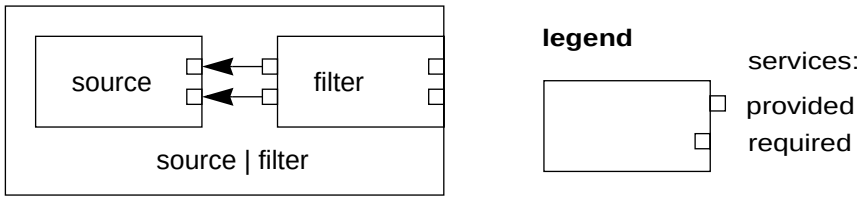

Figure 1 Connecting a source to a filter

3 Coordination Styles as Component Algebras

Streams and filters are perhaps the most obvious coordination style that can be naturally expressed as a component algebra. What is less obvious is that other, very different coordination styles may benefit from an algebraic notation.

In this section we intend to demonstrate, by a series of examples, that other coordination styles can indeed be expressed in this way. In each case, the main advantages gained are that (i) coordination is expressed at the higher level of *plugging*, which abstracts from the lower level of wiring, (ii) coordination is expressed as a declarative *composition* of components, thereby exposing the pertinent dependencies, and hiding irrelevant detail.

We do not claim that the styles we present are ideal! They are intended rather as a proof of concept and as an indication that much richer and more expressive styles could be formulated in a similar way.

3.1 Event Style

This style uses event notification [9] as its wiring mechanism. Event types are composed with Piccola services to yield event handlers. An event handler is attached to one or more event producers, which are components that can raise events. When an event occurs inside the component it notifies all its attached handlers. Table 3 contains the signature for the event style.

Event(do: Service)	→	*Handler*	compose event handlers
Producer ? Handler	→	*Producer*	attach handler to event producer

Table 3 Signature for the event style

Event styles are of particular interest in building graphical user interfaces. For instance, parts of the Java AWT framework can be wrapped so that they conform to the event style (see [1] for more details). The following example creates a button using the factory `newButton` and attaches an action event handler to this button:

```
myButton = newButton()
myButton.set(Label = "sayHello")
myButton ? Action(do: println("Hello"))
```

Contrast this code with the more typical event wiring, expressed directly in terms of an API. In Java, for example, an listener object must be explicitly instantiated and connected (i.e. wired) to a button using a particular binding method (`addActionListener`). The situation is pretty much the same using other frameworks, such as Tcl/Tk.

3.2 Grouped Actors (GA) Style

Actors constitute one of the foundational models of communicating agents [4][15]. Actors are autonomous entities that exchange asynchronous messages with each other. An actor has a queue of pending messages. It can accept the next message in the queue, create new actors, send messages to other actors, or *become* a new actor (i.e., replace its behaviour by a continuation).

	provided services	**required services**
Actor	`receive(M)`	`send(M)` `broadcast(M)`
Bus	`send(M)` `broadcast(M)`	`receive(M)`

Table 4 provided and required Services for GA Style

Here we are not interested in specifying the actors themselves, but in expressing *groups* of actors that can share communications that are broadcasted to the group. In such approaches, each actor can typically participate in multiple groups [8][12].

Groups exchange messages by means of a *software bus* [6]. The component types of the grouped style are given in Table 4. An actor can send a message to a single recipient (`send`) or broadcast it to all actors in the group (`broadcast`). Actors also provide a service to accept a message (`receive`). Accepted messages are kept in a queue. Observe that the provided and required services of the bus are matched by those of the actors. Figure 3 graphically illustrates the grouping of multiple actor into a set of actors and the connection with a bus, yielding a *group*. The signature for the grouped actors is given in Table 5. It is separated into four parts:

- Actors form sets of actors using the operator ^ to create a singleton from an actor and + to extend sets.
- A bus is parameterized by a set of message types.
- A bus is combined with a set of actors to yield a group. A group represents a configuration of actors connected through a message bus. A group provides services like `join` and `leave` to dynamically add and remove actors.
- A reactive actor is a specific case of an actor. A reactive actor reacts to a given message by executing a service. The scope of this reaction may be restricted by requiring that the message come from a specific actor. Multiple reactive actors can be composed.

^Actor	→	*Actors*	create a set containing one actor
Actor + Actor	→	*Actors*	combine two actors
Actors + Actor	→	*Actors*	extend the set with an actor
emptyActors	→	*Actors*	empty set
Msg + Msg	→	*Bus*	define messages of a message bus
Bus + Msg	→	*Bus*	extend bus with message
Bus || Actors	→	*Group*	combine bus with a set, yielding a group
Msg -> Service	→	*RActor*	run service S on behalf of message M
Msg / Actor -> Service	→	*RActor*	run S on behalf of message M from A
RActor & RActor	→	*RActor*	compose reactive actors
RActor	→	*Actor*	a reactive actor is also an actor

Table 5 Signature for grouped actors

```
StartVote = newMessageType                    # define message type
    contents(init): topic = init.topic
...
bus = StartVote + CastVote + EndVote + VoteResults

chair = newActor                              # define a chair actor
    do(self):
        StartVote.broadcast(topic = "Join EU?")
        sleep(10000)
        EndVote.broadcast(topic = "Join EU?")
        self.stop()

secretary = ...                               # collect and count votes

newEcoVoter():
    onStartVote(msg):
        CastVote.send
            destination = msg.sender
            vote = if (msg.topic.contains("car"))(then: "no", else: "yes")
    onVoteResults(msg):
        if (msg.result == "yes" && (msg.topic.contains("car"))
            then: demonstrate()
            else: smile()
    return                                    # return a reactive actor
        StartVote / chair -> onStartVote &
        VoteResults -> onVoteResults

members = chair + secretary + newEcoVoter() + ...
parliament = bus || members
```

Figure 2 Voting example

An example application to coordinate actors for a vote is given in Figure 2. It contains four parts:

1. First the message types are created and composed into a bus. The example contains the `StartVote` message type. A message of this type will contain a topic label. Several message types are connected to a bus component using the + operator.
2. Next, the `chair` actor is created using `newActor`. The behaviour of the chair agent is given in the `do` service. The chair initiates a vote by broadcasting a topic, waits some time, broadcasts the end of the vote, and then stops.
3. An `ecoVoter` is created as a reactive actor. Observe that this voter only reacts on startVote messages that come from the chair actor.
4. Finally, voting actors are collected into a member set and connected with the message bus to form the parliament group.

Note the expressive power of the || operator. It connects all the provided and required services of the actors with the bus (see Figure 3). Using an explicit binding notation, we

would have had to establish separately the bindings (`send`, `broadcast` and `receive`) for each actor in the group.

3.3 Regulated Coordination (RC) Style

Now we consider a style for expressing *regulated coordination* [23]. According to Minsky, 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 (law) regulating the exchange of messages between the agents of *G*.

The agents participating in a policy exchange messages with each other. This suggests that the actors of the previously described GA style can be used as our agents.

	provided service	**required service**
Actor	`receive(M)`	`send(M)` `broadcast(M)`
Law	`send(M)` `broadcast(M)`	`apply(M)` `receive(M)`
Rule	`apply(M)`	

Table 6 provided and required services for RC style

The basic component types in the this style are actors, messages, rules, and two predefined event types. Actors and messages have been described in the GA style. A law is a set of rules. Rules regulate the sending and receiving of messages. The event types *Sent* and *Arrived* are used to instantiate rules.

The signature for the RC style is given in Table 7. We reuse the signature to define and group actors from the GA style. A law regulates message passing, thus it subsumes the bus of the GA style. The connection of a law and a set of actors is a policy that enforces the law on all its actors.

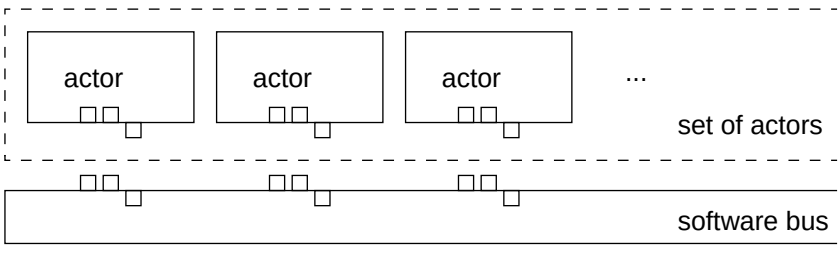

Figure 3 Connecting actors to a bus

^Rule	→	*Law*	create a law with one rule
Rule + Rule	→	*Law*	combine two rules
Law + Rule	→	*Law*	extend law with a rule
emptyLaw	→	*Law*	the empty law
Event 'of' Msg(Action)	→	*Rule*	compose rule
Event 'of' Msg(CondAction)	→	*Rule*	compose conditional rule
Law || Actors	→	*Policy*	bind actors to a law

Table 7 Signature for regulated coordination style

To enforce the law, a *controller* is placed between each actor and the communication network. The controller triggers the rules (i.e. calls `apply`) on sending and receiving messages. In the body of a rule, we can access and modify the state of the controller, deliver messages to the controlled actor, and forward messages to the network [23].

We illustrate how rules are scripted in Piccola using the voting example. A flaw of the example is that a malicious voter could cast multiple votes on the same topic. This is not the case if we use the regulated coordination style, since the RC style enforces rules on the actors. The following two rules ensure that a voter can cast at most one vote on each topic:

```
R1 = Arrived 'of' StartVote
     action(msg):
          state().put(key = msg.voteId)     # store identifier
          deliver()                         # and deliver msg
R2 = Sent 'of' CastVote
     cond(msg): state().containsKey(msg.voteId)
     action(msg):
          state().remove(msg.voteId)        # remove identifier
          forward()                         # forward message
```

The rule `R1` triggers when a `StartVote` message arrives at an actor. The body of the rule stores the vote identifier in the controller's state and delivers the message. Rule `R2` is triggered when an actor sends a `CastVote` message. The action is only executed provided the condition (`cond`) holds — that is only if the state contains the vote identifier. In that case the identifier is removed from the state, thus preventing further casts on the same vote.

The RC style is built on top of the GA style rather than implementing it from scratch. The controller is a special actor that guards every participating actor. The controller communicates with the actor over a local bus (unregulated communication) and with other controllers over the policy bus (regulated communication). Figure 4 shows the architecture of the RA style expressed in the GA style. The dotted components are hidden in the RC style.

Note that although the same actors can participate in an actor group as well as in a policy, it is impossible for an actor to avoid the enforcement of the law.

4 Combining Styles

So far, we only used a single coordination style within an application. However, there are at least two reasons why we need to be able to combine multiple styles, and therefore to bridge between styles:

- Different parts of an application are naturally expressed in different styles. Consider a 3-tier business application. The user interface part is composed using an event style, the business layer is expressed by a set of business rules, whereas the persistency layer is implemented in a database style using transactions. Such an application will contain components that participate in more than one style, e.g. in the event style as well as in the business rule style.
- Legacy components may be designed and developed for a different style than that required by the application in which we intend to deploy them. An example is the case where we deploy a pull-stream filter in a push-stream style [17].

Glue code is used to wrap components so that they can work in a context they have not been designed for. *Glue abstractions* provide generic glue to ease the generation of glue code. For example, a service may wrap a given component from one style to fit another.

Glue code can be tedious to write — for instance when we have to rename some services or add default arguments to functions calls. But bridging between styles is normally not only a question of renaming. Consider wrapping a pull-stream filter to a push-stream style. To resolve these kinds of compositional mismatch [28], we need to introduce glue code that bridges the gap between push and pull streams. The glue code consists of a coordinator and a pipe. The proceeding push stream then writes into the pipe where the data elements can be fetched by the pull stream filter. The responsibility of the coordinator is to terminate the pull-stream when the push stream gets closed [2].

4.1 Bridging Event and Actor Style

Assume we need a visual interface for our voting application. It would be a nightmare to entangle all the actors with statements to create and layout visual components and

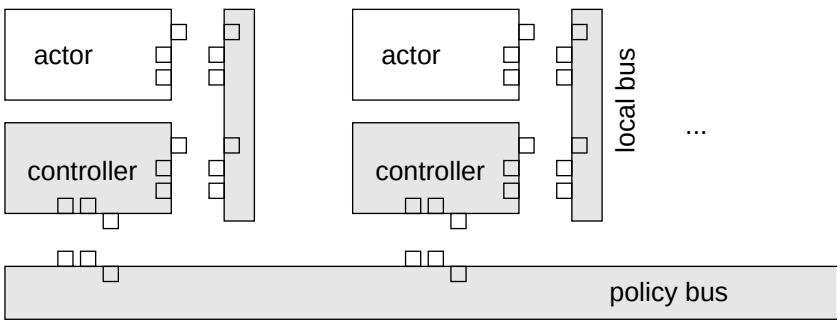

Figure 4 Implementation of the RC style on top of the GA style

add listeners to them. We prefer to wrap an existing event producer (e.g. a button) as an actor that broadcasts an action message. Here is a generic glue abstraction to do this:

```
eventToActor(GuiInst)(Event):
     actor = asActor         # wrap GuiInst as an actor
          GuiInst
          reqBroadcast = newMultiSlot()
          ...
     GuiInst ? Event
          do(event): actor.reqBroadcast(event)       # wire !
     return actor
```

This service wraps a GUI element as an actor and attaches to it an event handler. The handler is wired to the actor using the required `broadcast` service of the actor. Observe that the glue abstraction is responsible for both wrapping and wiring. The following code uses the glue abstraction to embed a button into the actor style:

```
yesButton = newButton(Label = "Say Yes")
...
GuiBus || eventToActor(yesButton)(Action) + ...
```

When the button is pressed, the wrapped actor broadcasts an action message on the `GuiBus`.

4.2 Multistyle Programming

Combining the actor style and the push-flow style, we wrap a source as an actor. This source pushes an element downstream each time it receives a message. The actor is used, for instance, as an archiver in the voting application, storing each vote together with its result. The resulting data flow is further processed by some filters and finally written to a log file. Using algebraic styles, we can directly express the architecture of the resulting application:

```
votingLaw || voter1 + sourceAsActor(archiver) + ...
archiver | filter | ... | logFile
```

The application is now easier to maintain because the architecture is clearly visible in the code. It is obvious how to change the code to achieve a different formatting or which component to replace in order to send the voting results directly to a printer. The low-level coordination aspect of such an application is hidden to the application developer. The chosen coordination styles ensure the correct wiring of the components.

5 Lessons Learned

What are the lessons learned from the experiment implementing coordination styles as many-sorted algebras? On one side, we made the experiments to validate the expressiveness of Piccola. On the other side, we want to see how coordination models should be implemented using existing languages. For instance, Jada is a object oriented framework providing tuple spaces in Java [11].

Validating Piccola, we conclude that the formal basis of Piccola of agents, forms, and channels turned out to be the right core abstractions. In particular the fact that everything is a form and that it is possible to abstract over arbitrary forms give Piccola high

expressive power. This is demonstrated by the fact that many higher level coordination abstractions can be reused. Coordination abstractions use various containers such as slots, blackboards, sets, buffers and queues. They also include generic synchronization policies. We have argued elsewhere how abstraction over forms is the key to implement, for instance, exception handling mechanisms [3].

Implementing coordination styles as algebras makes it possible to reuse the actor style for the regulated coordination style. In fact, the controllers are also actors and the actual actor together with its controller form a local group. Reusing a coordination style would be of particular interest in languages like Java. In Java, one often has to implement certain synchronization and coordination aspects at the lowest level the language offers: the final and native `wait` and `notify` methods and the keyword `synchronized`. This is not due to the chosen coordination primitives but to missing abstraction expressiveness. For instance it is not possible to abstract over the methods of an object, which would be necessary to implement certain generic coordination policies.

Representing a coordination style as an algebra considerably reduces the steep learning curve traditionally associated with object-oriented frameworks. For instance, using the actor style, it took one of the authors only half an hour to implement a bidding example. To a large extent, this is due to the compact representation of composition. In contrast to an object oriented framework which is normally presented as inheritance tree and API, the algebraic notation helps one to identify the components, the connectors, and the rules.

6 Related and future Work

This work is related to two distinct areas of research. The first area is the growing field of architectural description languages (ADLs) and tools to support their use (see for instance [26]). While ADLs are not yet in widespread use in industry, there have been several examples of their application to realistic case studies. ADLs support specification and reasoning about software at a very high level, but are not necessarily executable. Many ADLs provide a fixed set of predefined connectors to use and they do not support the definition of user-defined connectors at a higher level. They model architecture up to a certain abstraction level using predefined connectors. One of the few ADLs supporting user-defined connectors is Wright. It uses CSP as a formal basis to specify the roles of the connectors [5]. Studying the nature of connectors is an area of intense research. A recent paper by Mehta et al. [22] proposes a taxonomy of software connectors.

The other area that influenced our work is the algebraic specification approach (see for example [29]). In this approach, the behaviour of objects is specified using equational algebraic theories. We plan to further investigate how to formally analyse configurations using algebraic techniques. A challenging question is how to deal with mutable instances in a configuration. Is it possible to develop a type system to reason about immutable (i.e. algebraic) configurations? An interesting proposal deserving further investigation is the hidden order sorted approach of Goguen et al. [14].

We have only presented a limited number of coordination models as algebraic styles. Future work is needed to cover more coordination styles. We also want to investigate bridging between data and control driven coordination [24]. It is intriguing to ex-

plore whether we can use the formal foundation of Piccola to derive the operational semantics of coordination models. An earlier version of Piccola describes the mapping to the πL-calculus [21]. Implementing a model in Piccola gives us the denotational semantics in terms of the π-calculus almost for free. We further plan to develop a type system for the connectors and components in a style. A type checker might then be used to statically identify invalid configurations.

More work needs to be done in formally expressing the properties of components. The semantics of Piccola in terms of the πL-calculus serves as a starting point. Interesting work in that area is done by Issarny on the Aster environment. She uses pre and post predicates to describe the non-functional properties for connectors [16].

We are also working on a enhanced and distributed version of Piccola that will include syntactical elements for user defined collections. So far, we used infix operators to sum up sets and lists.

7 Conclusion

We have presented an approach to composing and coordinating software components in terms of high-level, algebraic coordination styles. User defined connectors hide the low-level wiring of provided and required services. Programming at the higher abstraction level leads to more flexible applications, since unnecessary coordination details are hidden. The connectors are responsible for plugging components. They ensure valid configurations and provide the coordination.

Using Piccola, a small composition language, we have demonstrated how connectors can be implemented, how an application can be scripted using a given style, and how styles are bridged. Since each style has its own strengths and weaknesses, glue abstraction help to combine styles and to get the best out of each style.

Acknowledgements

We thank the members of the SCG for stimulating discussions on this topic, in particular Jean-Guy Schneider and Sander Tichelaar for helpful comments on a draft of this paper. We also thank the anonymous reviewers for their constructive critic. This work has been funded by the Swiss National Science Foundation under Project No. 20-53711.98, “A framework approach to composing heterogeneous applications” and the ESPRIT working group “COORDINA” under BBW No. 96.0335-1.

References

[1] Franz Achermann and Oscar Nierstrasz, “Applications = Components + Scripts — A tour of Piccola,” *Software Architectures and Component Technology*, Mehmet Aksit (Ed.), Kluwer, 2000, to appear.

[2] 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. (Eds.), Cambridge University Press., 2000, to appear.

[3] Franz Achermann and Oscar Nierstrasz, “Explicit Namespaces”, *Proceedings of JMLC* 2000, to appear.

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

[5] Robert Allen and David Garlan, "The Wright Architectural Specification Language," Technical Report, September 1996, Technical Report CMU-CS-96-TB, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[6] Marcel Altherr, Martin Erzberger and Silvano Maffeis, "SoftWired iBus - Middleware for the Java Platform," *Java Report*, 4(12), December 1999.

[7] Farhad Arbab, "The IWIM Model for Coordination of Concurrent Activities," *Proceedings of COORDINATION'96*, P. Ciancarini and Chris Hankin (Eds.), LNCS 1061, Springer-Verlag, Cesena, Italy, 1996, pp. 34-55.

[8] Fernanda Barbosa and José C. Cunha, "A Coordination Language for Collective Agent Based Systems: GroupLog," *Proceedings of SAC'00*, ACM, Como, Italy, March 2000.

[9] 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.

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

[11] Paolo Ciancarini and Davide Rossi, "Jada: Coordination and Communication for Java Agents," *MOS'96: Towards the Programmable Internet*, LNCS 1222, Springer-Verlag, Linz, Austria, July 1996, pp. 213-228.

[12] Juan-Carlos Cruz and Stéphane Ducasse, "A Group Based Approach for Coordinating Active Objects," *Proceedings of Coordination'99, LNCS 1594*, 1999, pp. 355-371.

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

[14] Joseph Goguen, "Hidden Algebra for Software Engineering," *Proceedings Combinatorics, Computation and Logic*, 21(3), Springer Verlag, New Zealand, January 1999.

[15] Carl Hewitt, "Viewing Control Structures as Patterns of Passing Messages," *Artificial Intelligence*, 8(3), June 1977, pp. 323-364.

[16] Valérie Issarny, Christophe Bidan and Titos Saridakis, "Characterizing Coordination Architectures According to their Non-Functional Execution Properties," *Proceedings of the 31st Annual Hawaii International Conference on System Sciences*, 1998, pp. 275-283.

[17] Paola Inverardi, Alexander L. Wolf and Daniel Yankelevich, "Checking Assumptions in Component Dynamics at the Architectural Level," *Proceedings of COORDINATION'97*, LNCS 1282, Springer-Verlag, September 1997, pp. 46-63.

[18] Ralph E. Johnson and Brian Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, 1(2), 1988, pp. 22-35.

[19] Doug Lea, *Concurrent Programming in Java[tm], Second Edition: Design principles and Patterns* (2nd edition), Addison-Wesley, The Java Series, 1999.

[20] Markus Lumpe, "A Pi-Calculus Based Approach to Software Composition," Ph.D. thesis, University of Bern, January 1999.

[21] Markus Lumpe, Franz Achermann and Oscar Nierstrasz, "A Formal Language for Composition," *Foundations of Component Based Systems*, Gary Leavens and Murali Sitaraman (Eds.), pp. 69-90, Cambridge University Press, 2000.

[22] Nikunj R. Mehta, Nenad Medvidovic and Sandeep Phadke, "Towards a Taxonomy of Software Connectors," *Proceedings ICSE'00*, Limerick, Ireland, June 2000, pp. 178-187.

[23] Naftaly Minsky and Victoria Ungureanu, "Regulated Coordination in Open Distributed Systems," *Proceedings COORDINATION'97*, David Garlan and Daniel Le Mètayer (Eds.), LNCS 1282, Springer-Verlag, Berlin, Germany, September 1997, pp. 81-97.

[24] George A. Papadopoulos and Farhad Arbab, "Coordination Models and Languages," *The Engineering of Large Systems*, Academic Press, August 1998.

[25] Jean-Guy Schneider and Markus Lumpe, "A Metamodel for Concurrent, Object-based Programming," Proceedings of LMO'00, Québec, January 2000, pp. 149-165.

[26] Mary Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik, "Abstractions for software and architecture and tools to support them," *IEEE Transactionss on Software Engineering*, April 1995.

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

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

[29] Wolfgang Wechler, *Universal Algerbra for Computer Scientists*, Springer-Verlag, vol. 25, EATCS, 1991.