Supporting Compositional Styles for Software Evolution[1]

Oscar Nierstrasz, Franz Achermann
Software Composition Group, IAM, Universität Bern
Neubrückstrasse 10, CH-3012 Berne, Switzerland
Tel: +41 (31) 631.4618. Fax: +41 (31) 631.3965.
{oscar,acherman}@iam.unibe.ch www.iam.unibe.ch/~scg

Abstract

Software is not just difficult to develop, but it is even more difficult to maintain in the face of changing requirements. The complexity of software evolution can, however, be significantly reduced if we manage to separate the stable artifacts (the "components") from their configuration (the "scripts"). We have proposed a simple, unifying framework of forms, agents, and channels for modelling components and scripts, and we have developed an experimental composition language, called Piccola, based on this framework, that supports the specification of applications as flexible compositions of stable components. In this paper we show how Piccola can be used to reduce the complexity of software evolution through the specification and use of an appropriate compositional style, and we illustrate the approach through a non-trivial example of mixin layer composition.

1. In *Proceedings International Symposium on Principles of Software Evolution* (ISPSE 2000)}, Nov 1-2, 2000, Kanazawa, Japan, pp. 11-19.

1. Introduction

As software engineers, we separate concerns in order to reduce the complexity of designing, constructing and maintaining large software systems. Some complexity is inherent in the requirements of large software systems, but, we argue, *the worst forms of complexity arise from software evolution*. Tangled code poses a problem when that code needs to be modified or extended. This, however, will occur for any interesting piece of software! As a consequence, we suggest that the biggest gains are to be achieved by concentrating on *reducing the complexity of software change*.

In fact, we see that the tools and techniques that achieve the biggest gains in productivity address precisely this issue: very high-level languages, software components, and fourth generation development environments raise the level of abstraction by *separating what is stable from what is not*, and thereby make it easier to introduce changes. We propose, therefore:

Applications = Components + Scripts

as a guiding principle to achieve flexible software. *Components* wrap up provided (and required) services behind a standard interface, and generally represent the *stable* parts of applications. *Scripts* plug components together, and represent the *flexible* parts of applications.

The scriptable components paradigm additionally assumes that *components have been designed to be plugged together*, according to a particular "compositional style". A compositional style defines components and how they can be composed into larger units using connectors. Composition rules constrain valid compositions. (We use the term "compositional style" here rather than the more familiar term "architectural style" [24] for the simple reason that not all compositions are at the level of architecture.) Furthermore, scripts may make use of special connectors to plug components together, namely *coordination abstractions* to mediate the connections between components, and *glue abstractions* to mediate between components whose plugs do not match [23].

To support this paradigm, we have developed Piccola, a small language for specifying components, connectors and scripts [1][4] corresponding to different compositional styles. As such, the kinds of abstraction mechanisms required by Piccola are somewhat different from those needed for a general-purpose language. In order to have the simplest possible framework to define compositional styles, we choose a small set of primitives that unify various concepts:

- *Forms* embody *structure*. Forms are immutable records that can be polymorphically extended with additional bindings (yielding a new form). Forms unify *objects*, *keyword-based arguments* and *namespaces*.
- *Agents* embody *behaviour*. Agents are concurrent, communicating entities whose behaviour is specified by a script. Agents implement the connections between components. Agents unify *concurrency* and *composition*.
- *Channels* embody *state*. Channels are the mailboxes that agents use to communicate. They unify *synchronization* and *communication*.

In section 2, we review the nature of software evolution, and argue that software systems that evolve gracefully conform to well-defined compositional styles. Next, in section 3, we present Piccola, a composition language designed to support a range of compositional styles. In section 4 we present a more complex example from the literature, of *mixin layers* represented as a compositional style in Piccola. We conclude in sections 5 and 6 with remarks about ongoing and future work.

2. The Nature of Software Evolution

All successful software systems are doomed to either change [16] or quickly become obsolete. The reasons these systems are asked to change may vary, but the technical details of how changes are implemented tend to be based on a battery of well-known techniques. Furthermore, systems that evolve gracefully typically are designed according to a *compositional style* that cleanly separates the *stable* software entities (which we will call *components*) from the more *flexible* specification of their configuration (which we will call *scripts*). In such systems, evolution is ideally achieved by either *replacing* components (i.e., *intra*-component evolution), or by *reconfiguring* them (i.e. *inter*-component evolution).

Of course, things are not always so neat, so it would be helpful to have some language-based support that makes it easier to build software systems that adhere to this separation into components and scripts. In this section we first review some typical examples of evolving software and then establish some requirements for supporting compositional styles.

2.1 Evolution and separation of concerns

Let us consider some of the most common changes made to evolving software systems, and see what factors influence the ease with which the changes can be implemented:

- *Change in business logic.* New functionality may be added, existing functionality modified or reconfigured, or new policies may be put into place.
- *Change in platform.* The operating system, database, user interface, standard libraries or even programming language may change.
- *Change in clients.* The system may need to offer its services to new, or different kinds of clients, or may need to be integrated with other new or existing systems.

When applying such modifications to existing software, the change often cannot be localized. This leads to "tangled" (i.e., strongly coupled) or "scattered" (i.e., weakly cohesive) code. Evolving such code becomes more and more difficult. The solution, very broadly, is to factor out the tangled aspects in such a way that they can be easily re-composed in a more flexible way. Factoring out aspects can be done in a large number of different ways, however. Let us consider a few typical examples:

- *Multi-language support for user interfaces.* Successful software designed for a single target language may have to be adapted to support different user languages. The user dialogues are tangled with all of the user interface code.

  Two typical approaches are to either (i) *generate* the UI code, parameterized by the target language, or (ii) *delegate* all dialogues to language-specific dialogue managers. The former technique is static, whereas the latter supports dynamically swapping languages. In both cases stable functionality is encapsulated behind an interface that allows it to be easily reconfigured, whether statically or dynamically. A new language can be supported by providing a new language component.
- *GUI look and feel.* Multi-platform applications are often required to support the look and feel of the host platform. The look and feel can be considered a *policy* which must be separated from the GUI mechanisms delivered by an abstract interface. Depending on the implementation technique, the policy may be statically or dynamically specified.

  Factoring out the look and feel separates GUI elements (frame, menus, modal dialogues etc.) from their concrete representation for a window manager. Here, the GUI logic in term of frames, menus and dialogues is stable, but the concrete representation varies. The application uses context dependent services to create frames, add and specify menus items, or invoke user dialogues. A separation defines a user interface model that defines the core operations as platform services.
- *Compiler extensions.* Extensions to a programming language or to the tools that support the language often entail changes and extensions to the way in which syntactic entities are represented. These changes may *cross cut* different parts of the compiler (i.e., those that parse and generate symbolic representations, and those that process them). In such cases, inheritance and generics are of limited help because they are purely static techniques.

  Programming languages that most successfully support new extensions typically do so by providing a stable core implementation that can be extended by means of a *meta-object protocol* (MOP) [12]. Either static (compile-time) or run-time reflection allows the behaviour of the language to be adapted.
- *Concurrency.* A functioning sequential system cannot always be easily converted to work in a concurrent context. One solution is to wrap the whole system up as a monitor, but sometimes more fine-grained concurrency control is required.

  Most programming languages, unfortunately are not expressive enough to allow synchronization or coordination abstractions to be separately specified from the synchronized or coordinated entities. In Java, for example, synchronization code is always tangled with computational code. In order to wrap an existing Java class with, for example, a readers/writers mutual exclusion policy, one is forced to write a lot of boilerplate code. *Higher-order wrappers*, some form of *reflection* (introspection, as provided by Java's "reflection" package, is typically not enough), or a *meta-object protocol* are techniques that help to factor out such aspects.
- *Heterogeneous systems policies.* There is a long list of aspects that arise in heterogeneous and distributed software systems that must be factored out as *policies*: persistence, transactions and security are three common

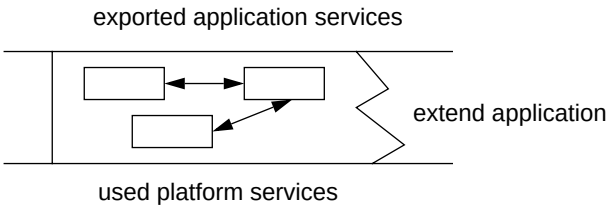

Figure 1 Hooks for changes

examples [13]. Depending on whether policies are reified as run-time entities or not, they may be either statically or dynamically reconfigured.

From these few examples, it should be clear that graceful software evolution depends on the ability to cleanly separate stable software components from the "scripts" that configure them, i.e., the ability to factor application software according to a particular compositional style. Aside from the methodological issue of how to decide what is a good style for a particular application domain, the ability to express such a factoring at all depends on having the right abstraction mechanisms available in the host language.

2.2 Requirements for supporting compositional styles

Let us establish some of the fundamental requirements necessary for expressing a range of different compositional styles. Consider figure 1, which illustrates the structure of a simplified application. The application consists of three components and two connections. It uses some services provided by the platform. In a layered system, we can draw these services into a layer below the application. The exported or provided services can be used by the layer above, which might be responsible for user interaction. The figure suggests three possible hooks where changes can come in: Changes in business logic correspond to extending the application. We can implement them by adding or replacing components or connections. Changes in platform can be expressed by inserting a new layer between the application and the used services. Changes in clients can be captured by adding a layer on top of the application.

It is important to keep in mind that a component itself may be a composition of (more low level) components.

We establish the following requirements:

1. *Platform dependencies must be explicit and configurable.* This means that during evolution we can insert a layer below the application and change the behaviour of the platform services. Observe that most programming languages provide primitives which cannot be adapted, e.g. the throws statement in Java or constructor calls in C++. A common way to achieve this requirement is to use factory methods instead of class constructors in class based languages like Java.
2. *Connections between components must be explicit and configurable.* Providing connectors as syntactic elements makes the configuration of an application (or, at a higher level, its architecture [9][19][24]) explicit and therefore scriptable.
3. *General, higher-order abstraction must be provided.* In order to define arbitrary connectors, it must be possible to abstract virtually everything, including interaction patterns. In most OO languages, on the contrary, one cannot abstract the set of methods without use of reflection. Scripting languages like Perl and Python provide an *eval* function to evaluate a string. This allows the programmer to do meta-programming, but normally at the cost of any type safety.
4. *Compositions of components can also be encapsulated as components.* In this way, we can develop a hierarchy of component types.
5. *The object model must be simple and open.* It must be possible to use and implement different compositional styles. A rich object model will lead to complex feature interaction between built-in features and those of the compositional styles (consider interaction of inheritance and synchronization [15]). Instead, we need to identify the "right" core concepts to model composition in different ways.

In the next section, we will show how forms, agents, and channels provide a clean and simple answer to these requirements.

3. Supporting Compositional Styles

Piccola is designed to be a *composition language*, rather than a general purpose programming language. Whereas many existing languages are very good at implementing software components, they tend not too offer very good abstractions for plugging components together, or even for describing the nature of the plugs (i.e., connectors). Piccola is built on top of a minimal computational model of *agents*, *channels* and *forms*. The language provides higher-level and higher-order abstractions on top of this foundation that make it relatively easy to define various compositional styles and scripts that conform to these styles. In this section we give an overview of the layered architecture of Piccola itself, we present a brief example of mixins in Piccola to serve as a introduction to the syntax and features of the language, and we give an overview how Piccola supports different compositional styles.

3.1 Piccola Layers

The Piccola framework is presented in table 1. At the lowest level, components can be *wired* together using the concepts of agents, channels and forms. An *agent* represents behaviour (thus a component). Interaction between components happens along *channels* (thus connectors). The values exchanged between agents must have a rich enough structure to express objects and component interfaces. *Forms* are extensible records, mapping from labels to other forms or channels. The interface for a component is expressed by a

form which binds its connection points. This model is formally described by the $\pi\mathcal{L}$-calculus [18]. Projection is used to look-up values bound by a label in a form. Form composition ensures that we can arbitrarily compose components and identify the sub-parts by projection.

Table 1: Piccola Layers

Applications	components + scripts
Compositional styles	streams, events, GUI composition, ...
Core libraries	basic coordination abstractions, basic object model
Piccola	services, operator syntax, name-spaces, built-in types
$\pi\mathcal{L}$-calculus	agents, channels, forms

The next level (see table 1) defines the Piccola language. Piccola hides the operators of the $\pi\mathcal{L}$-calculus and models everything in terms of *forms* and *services*. Services are modelled in the calculus by an encoding of call-by-value functions in the π-calculus [21]. In Piccola, *everything is a form*, including services. A service can be thought of as a form with single "call" label, just as a function object in C++ is an object with an `operator()` member function.

As we shall see, Piccola is a very small language, with a minimal syntax and very few built-in features. Piccola includes a set of core libraries to support basic control structures, exception handling and a simple object model.

On top of the core libraries, one may define various compositional styles that abstract away from the low-level wiring of the $\pi\mathcal{L}$-calculus, and provide instead higher level plugs, or connectors corresponding to a problem domain.

Finally, one may use these styles to plug together components.

Piccola addresses the requirements listed in the previous section as follows:

1. *Forms are used to represent the current context.* A form is a namespace holding the names of a set of services. Platform dependencies can thus be explicitly represented and manipulated or overridden. Piccola's own basic services are similarly available. For example, the operator to create a new channel is accessible as a service `newChannel` in the initial root context. Programmers can redefine `newChannel` if necessary.
2. *Connectors can be implemented by user defined infix operators.* For instance the form expression "`A ** B`", where `A` and `B` are forms, is interpreted as `A._**_(B)` when `A` contains such an operator. The projection `A._**_` is a service which is invoked with `B`. Otherwise, the definition of `**` is looked up in the root context, and the expression corresponds to **`root`**`._**_(A)(B)`. The `_**_` label binds a curried function in the root context.
3. *Everything is a form, including abstractions over forms.* Consequently Piccola offers a high level of abstraction power. It is important to note that also both the static and dynamic contexts are accessible as forms [3].
4. *The result of a form expression is also a form.* Since forms represent components, and form expressions compose components, the result of a composition is also a form. Within a suitably defined compositional style, this form will also be a component.
5. *Forms provide a primitive object model.* Richer models can be built on top of forms (such as that provided in the core library). User-defined operators encourage programmers to program in terms of plugging components.

3.2 Objects and Mixins in Piccola

The following example illustrates how objects and mixins can be implemented in Piccola using forms, and also serves as a brief introduction to the language and its features. The following service `newBlackboard` specifies a factory for blackboard objects:

```
newBlackboard():
  'channel = newChannel()# creates a local channel
  write = channel.send   # exported write service
  remove = channel.receive
  read():          # defines the read service:
    'X = remove()  # remove contents X
    'write(X)      # write them back on the channel
    X              # and return them
```

`newBlackboard` is an abstraction which, when evaluated, returns a form with bindings for `write`, `remove` and `read`. These will represent the services provided by the new blackboard. Note that indentation is used to indicate the nesting level of forms.

The binding for `channel` is only available locally within the definition of `newBlackboard` since it is quoted. Instead of being exported as part of the return value, the `channel` binding only extends the local **`root`** context. Observe that the binding `write = channel.send` is exported in addition to being used within the body of `read`.

Note that, whereas `newBlackboard`, `write`, `remove` and `read` are all services (i.e., abstractions), `channel` is an ordinary value.

We can factor out the definition of the `read` method and put it into a mixin. The `addRead` service *extends* an argument form `B` with a `read` service:

```
addRead(B):
  B              # return the form B ...
  read():        # ... extended with the read service
    'X = B.remove()
    'B.write(X)
    X
```

Observe that `read` uses the `remove` and `write` services provided by the form `B`. We can now apply the `addRead` mixin to *any* form that provides at least `remove` and `write` services:

```
ch = newChannel()
b = addRead(write = ch.send, remove = ch.receive)
```

The addRead mixin can also be applied to a form that provides more than just the required services:

```
eb = addRead              # invoke addRead
  write = channel.send
  remove = channel.receive
  asString(): "a Blackboard"
```

The final result will also provide the asString service. Note that, in this case, the argument to addRead is a form defined on multiple indented lines following the invocation.

3.3 Compositional Styles

We will now survey some of the typical compositional styles and techniques that are used to make software more flexible and adaptable, as we will see how Piccola supports them. Note that these styles are not orthogonal, it is for instance possible to see mixin layers as a kind of component algebra, as we will illustrate in section 4.

Component algebras. A *component algebra* is an algebra in which the objects are components, and the operators are connectors. Composing two or more components always yields a new component. A script, then, is just an expression that composes components, where each subexpression is also a component [1]. A component algebra is typically *many-sorted*, that is, several different kinds of components will be supported. The best-known example of a component algebra is pipes and filters. The components are sources, filters and sinks, and the principle operator is the pipe. A source composed with a filter yields a source, and a filter composed with a filter is again a filter. We claim that most styles can be expressed as component algebras [1][4][5][23].

In Piccola, component services are encapsulated as forms. Different sorts of components are represented as forms with different sets of labels. Connectors are user defined operators realized either as abstractions available in the local context, or as special services of a component. A script composes forms, and yields a new form.

Higher-order wrappers. As we saw in section 2, many kinds of extensions can be factored out as simple wrappers, adding "before and after" behaviour. Since all values including abstractions are forms in Piccola, abstractions are higher-order, making it easy to specify higher-order wrappers. Furthermore, abstractions are *monadic*, always taking a single form as an argument. This makes it possible to define *generic wrappers* that do not depend on the number of arguments. Since namespaces are represented explicitly as forms we can also modify the context for wrapped functions, for instance to change the policy (see implicit policies below).

In functional programming languages such as Haskell, the required services of a component are passed as arguments or they can be packed into monads [29]. Monads are used to encapsulate state within a purely functional program.

Glue abstractions. Many glue abstractions can be expressed as simple wrappers. Glue abstractions can wrap known services or add new ones while leaving other undisturbed [17][22].

Mixins and metaobjects. Higher-order wrappers make it possible to define mixins and other composition mechanisms for building objects. Piccola provides only forms as "primitive objects", but one can define a variety of other object models on top of forms [22]. One of Piccola's few keywords is **def**, used to define a fixpoint, but it is also possible to delay binding of self, which makes object models with explicit metaobjects very attractive. Metaobjects enable runtime reflection [12].

Mixin Layers. Whereas a single mixin changes the behaviour of a single class, a mixin layer changes the behaviour of several classes. In design, we often implement a role with a mixin. Thus a collaboration can be implemented by a mixin layer. Applying mixin layers to a set of classes then adds a collaboration to the application that consists of the classes [8]. We have already seen how mixins are implemented as higher-order wrappers in Piccola. However, in section 4, we will present mixin layers in more detail.

Coordination abstractions. Piccola provides primitives to instantiate concurrent agents or to explicitly create new channels within scripts. The formal semantics of Piccola is in terms of a process calculus, so concurrency is built-in, not added-on [17][22]. This makes it easy to define coordination abstractions as abstractions over scripts. Furthermore, coordination can be seen as a special case of scripting, and many coordination styles can be naturally expressed as component algebras [5].

Implicit policies. Forms are also used in Piccola to represent *namespaces* [3]. Whenever a script is evaluated, it has access to two special namespaces, representing respectively the *root* context and the *dynamic* context. The root context defines the global environment, but can be specialized to define a "sandbox" for an untrusted agent, or to override or extend global services (like print). The dynamic context is the environment provided by a client of an agent, and can be used to define implicit policies. This mechanism can be used, for example, to define an exception handling mechanism for Piccola [1][3] (the handler is always passed in the dynamic context). The same mechanisms are used more generally to optionally override any kind of default policy.

Aspect-Oriented Programming. Aspects cross cut a system and cannot be factored out using object-oriented design techniques [11]. They are implemented by associating behaviour with certain events, for instance sending a message to instances of different classes. Higher-order wrappers can express precisely this information. They can decorate classes or object with specific behaviour. For instance, adding a reader-writer policy to a non-synchronized object wraps the writer methods and the reader methods uniformly [2].

Default arguments. Since abstractions are monadic, taking a single form as an argument, and forms can be polymorphically extended, it is straightforward to define default arguments for services. (An agent just appends the received argument to the defaults.) This allows the programmer to extend services with new options without breaking existing clients.

4. An example: Mixin layer composition

In this section, we give a concrete example of mixin layer composition [27] implemented as a compositional style in Piccola. Mixin layers are (in our view) a less well known and non-trivial composition style. Implementing mixin layers requires an object-oriented language that supports nested classes and mixins. The language P++, for example, extends C++ to support static and type-safe mixin layer composition [25]. Implementing mixin layer composition in Piccola thus serves to validate that Piccola is expressive enough to tackle high-level composition abstractions. Finally, mixin layers are a good candidate to illustrate component algebras, as composed mixin layers are again mixin layers.

We present the graph traversal application proposed by Holland [10]. This application defines different operations on a undirected graph. *VertexNumbering* numbers the nodes in a depth-first order. *CycleChecking* determines whether the graph contains a cycle, and *ConnectedRegions* partitions the nodes of the graph into connected regions.

Holland implemented the application based on a framework. Later, Van Hilst et al. [28] reimplemented it using roles and mixins. Smaragdakis finally used mixin layers to implement the same application [26][27]. The three main implementation classes are `Graph`, `Vertex`, and `Workspace`. The graph class defines a container of vertices with the usual graph properties. The nodes are stored as instances of the class `Vertex`. The workspace class includes the specific part of a traversal. For instance, the workspace object plays the role *WorkspaceNumber* in the *VertexNumbering* application to associate numbers to the nodes. This role specifies a slot to store a current number and to assign and increment this number each time a new node is visited during depth-first traversal. We can implement such a role using a mixin. The mixin adds the specific members and operations to its superclass when composed.

Similarly, a mixin adds the number slot to a vertex class. In Piccola the mixin uses form extension to add a number slot to any vertex:

```
addNumberSlot(V):        # adds a slot to store the
  V                      # number during traversal
  number = newRefcell(0)
```

(A reference cell is essentially an initialized blackboard extended with a non-destructive `get` service.)

It is now important to apply the `addNumberSlot` mixin to a vertex class whenever we apply the `vertexNumbering` mixin to a graph class. Otherwise, the numbering traversal cannot store the numbers in the vertices.

Smaragdakis and Batory use the GenVoca model [8] to keep the different mixins applied to classes in sync. A GenVoca component is a mixin layer. In essence, a mixin layer encapsulates all the mixins necessary for a single collaboration. For instance the mixin layer *Number* to implement the *VertexNumbering* collaboration contains two mixins: one to add the vertex numbering during traversal and one to add the number to a vertex. The advantage of using mixin layers instead of isolated mixins is obvious. Design or change elements in the application are encapsulated and implemented in a single component.

In Piccola, nested forms can collect all the mixins of one layer. All we need is assign names to the individual mixins in order to compose them correctly. The `numberNodes` layer is:

```
numberNodes =
  asVertex(V):
    V
    number = newRefcell(0)
  asGraph(G):
    G
    visit():
      n = newCounter()
      G.each(do(V): V.number.set(n.inc()))
```

The two mixins `asVertex` (which was `addNumberSlot` above) and `asGraph` are expected to modify respectively a vertex and graph class to add the numbering operation. We do not need a special Workspace class, since we can, as in Smalltalk, pass first-class blocks to an iterator. In Piccola, we usually represent blocks as a form with a `do` service. The vertices are numbered when we call `visit` on a graph. Observe that `visit` iterates over the vertices and sets the number (`V.number`). However, it is guaranteed that `V.number` will be present, since the `asVertex` mixin establishes exactly this. The method `each` for the graph is defined in another layer (see below).

Next, we need to implement a composition operator `**` such that "`A ** B`" is a composite mixin layer, provided `A` and `B` are mixin layers. Using higher-order wrappers, we can implement `**` as follows:

```
'Defaults = (asVertex(X):X, asGraph(X): X)
_**_(A)(B):
  'A = (Defaults, A)      # set defaults
  'B = (Defaults, B)
  asVertex(X): B.asVertex(A.asVertex(X))
  asGraph(X): B.asGraph(A.asGraph(X))
```

The form `Defaults` defines default values for the `asVertex` and `asGraph` services. (The defaults are simply the identity function.) Next, we make use of polymorphic form extension to concatenate the default bindings and those provided by `A` or `B`, respectively. Note that only if `A` or `B` provide their own bindings for `asVertex` and `asGraph` will these override the default values. (Later bindings override earlier ones.) Observe that the result of `A ** B` will necessarily contain the mixins `asVertex` and `asGraph` as expected.

The `**` operator is hard-coded to compose only mixin layers that define `asVertex` and `asGraph` services. It is also possible to define a more generic connector that will iterate

over all the services defined in two mixin layers, or even to define a generator that will produce connector s to compose specific sets of services.

A concrete application may now require that we need numbering and a test to find cycles in our graphs. Thus we can script two services to create new vertices and corresponding graphs:

```
layers = graph ** dft ** numberNodes ** cycle
newGraph(): layers.asGraph()
asVertex(): layers.asVertex()
g = newGraph()
...
```

This application must define a graph `g` with methods `visit` (from the `numberNodes` layer) and `hasCycle` (from the layer cycle). The base layer `graph` defines basic graph operations to add vertices. The layer `dft` adds the method `each` to the graph to visit all nodes by depth-first-traversal. Note that the `numberNodes` layer depends on the `each` method in the graph to work correctly.

In summary, we have successfully separated the stable parts (the mixins) from the evolving part (the composing expression, i.e. the script). Let us consider some possible changes to our application:

We might want to apply a *synchronization policy* to the graph. One possibility is to make all the methods of the graph into a monitor. Since all the methods are services, we can iterate all of them and wrap the methods. This is expressed by the layer `exclusive`:

```
layers = graph ** dft ** numberNodes ** cycle ** exclusive
```

We can change the depth-first traversal to a breadth-first traversal by replacing a component:

```
layers = graph ** bft ** numberNodes ** cycle
```

Finally, we can adapt a layer which (by chance) does not follow the naming conventions by writing some *glue code:*

```
myLayer =
  asGraph = legacyLayer.addFancyFeatureToGraph
  asVertex = legacyLayer.addFancyFeatureToNode
```

In case there are many components that need to be adapted in the same way, we can abstract from the glue code to obtain a general-purpose *glue abstraction:*

```
legacyAdaptor(legacyLayer):
  asGraph = legacyLayer.addFancyFeatureToGraph
  asVertex = legacyLayer.addFancyFeatureToNode
myLayer = legacyAdaptor(myLegacyLayer)
```

5. Discussion

As explained elsewhere [4], Piccola lies somewhere between a *scripting language*, like Python or TCL, an *architectural description language* (ADL), like Wright [6] or Rapide [14], a *coordination language*, like Darwin [19] or Manifold [7], and a *glue language*, like Smalltalk or C.

Scripting languages and glue languages tend not to have any formal semantics, and do not especially address concurrency or coordination. Architectural description languages are typically more formal, in order to support reasoning about architectural styles, but they are mainly intended as specification languages, not to be used as run-time configuration languages. Coordination languages address concurrency and distribution, and they often have a formal basis as well, but they tend to be weak in abstraction, making it difficult to define higher-level coordination abstraction, or to deal with compositional issues not especially related to coordination.

Piccola has borrowed ideas from many different sources. Its innovation, we feel, lies mainly in the way it brings a few, simple concepts together that, in combination, support the paradigm, “Applications = Components + Scripts”.

A formal semantics in terms of process calculus exists, but this is too low-level to support reasoning. We would like to reason about applications at the level of their composition. That is, if we know that components conforming to a particular compositional style have certain properties, then valid compositions will also have certain interesting properties.

A type system has been developed for Piccola [17], but it too is at the level of the process calculus. We would like to reason about higher-level types in terms of components and their composition. Ideally, we might like a type system that can express not only required and provided services, but even some more detailed dependencies [20].

Sometimes it is necessary to migrate from one compositional style to another. An example may be the transition of one component implementing some business rules in a batch job. We now want to evolve the centralized batch processing to pipe architecture to enhance parallelism. Such a change will require to change all subcomponents to conform to the new style. However, we believe that it should be possible to automate this process, for instance by providing wrappers.

Piccola is still an experimental language. There currently exist implementations on three different platforms (Java, Squeak and Delphi), each with its own syntax. A preferred syntax is emerging, but there are still some open questions concerning usability and expressiveness.

Piccola is available from the SCG web site, either as an applet, or as downloadable Java or Squeak executables. The implementations are stable and efficient enough to run non-trivial examples (the JPiccola environment is scripted in Piccola itself), but no large experiments have been undertaken yet. Language bindings to Squeak and Java are currently ad hoc. We are now experimenting with using Piccola as a bean composition language, and expect this work to help in defining a standard mapping to Java and other languages.

Presently Piccola can communicate with remote machines only through the host language (Java or Squeak). We are beginning to work on distributed Piccola, which will support the specification of compositional styles for distributed applications.

6. Concluding remarks

In previous papers, we have presented the conceptual framework of *components*, *scripts* and *glue* [23], the formal underpinnings of Piccola in terms of the $\pi\mathcal{L}$-calculus [4] [18], and a tour of the Piccola language features [1]. We have demonstrated how Piccola forms can model different notions of *explicit namespaces* [3], we have shown how different forms of *coordination* can be expressed as compositional styles [5], and we have argued that aspect-oriented programming can be expressed as *feature mixins* in Piccola [2].

In this paper we have argued that software systems can evolve gracefully only if they are designed in such a way as to cleanly separate stable and flexible aspects into *components* and *scripts*. Furthermore, this separation can only be achieved if the right abstraction mechanisms are available in the host language. Piccola is designed to be a *composition language*, good at expressing different kinds of *compositional styles*, each of which may be suitable for composing components for different application domains.

We are still experimenting with applications of Piccola. Although we believe that Piccola provides the right abstractions needed to express applications as flexible compositions of software components, we still have to prove that these techniques can succeed in separating concerns for complex domains where other approaches have failed.

Acknowledgements. This work has been supported by the Swiss National Science Foundation under Projects #20-53711.98, “A framework approach to composing heterogeneous applications”, #20-61655.00, “Meta-models and Tools for Evolution Towards Component Systems”, and the Swiss Federal Office for Education and Science under Project BBW #96.00335-1, within the Esprit Working Group 24512: “COORDINA: Coordination Models and Languages.”

7. 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, “Language support for feature mixing,” *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000.

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

[4] 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.

[5] Franz Achermann, Stefan Kneubuehl and Oscar Nierstrasz, “Scripting Coordination Styles,” *Proceedings COORDINATION 2000*, to appear.

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

[7] Farhad Arbab, “The IWIM Model for Coordination of Concurrent Activites,” *Proceedings of COORDINATION’96*, Paolo Ciancarini and Chris Hankin (Eds.), Springer-Verlag, Cesena, Italy, 1996, pp. 34-55.

[8] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci and Marty Sirkin, “The GenVoca Model of Software-System Generators,” *IEEE Software*, September 1994, pp. 89-94.

[9] Stéphane Ducasse and Tamar Richner, “Executable Connectors: Towards Reusable Design Elements,” *Proceedings of ESEC/FSE’97*, LNCS 1301, 1997, pp. 483-500.

[10] Ian M. Holland, “Specifying Reusable Components Using Contracts,” *Proceedings ECOOP’92*, O. Lehrmann Madsen (Ed.), Utrecht, The Netherlands, June 1992, pp. 287-308.

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

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

[13] Doug Lea, “Design for Open Systems in Java,” *Proceedings COORDINATION’97*, David Garlan and Daniel Le Mètayer (Eds.), Springer-Verlag, Berlin, Germany, September 1997, pp. 32-45.

[14] David C. Luckham and James Vera, “An Event-Based Architecture Definition Language,” *IEEE Transactions on Software Engineering*, vol. 21, no. 9, September 1995, pp. 717-734.

[15] Satoshi Matsuoka, Ken Wakita and Akinori Yonezawa, “On Inheritance in Concurrent Object-Oriented Languages,” *Proceedings of 7th Annual Conference of Japan Society for Software Science and Technology (JSSST)*, LNCS 742, Tokyo, Japan, 1991, pp. 65-68.

[16] Lehman M. M. and Belady L., *Program Evolution - Processes of Software Change*, London Academic Press, 1985.

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

[18] 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.

[19] Jeff Magee, Naranker Dulay, Susan Eisenbach and Jeffrey Kramer, “Specifying Distributed Software Architectures,” *Proceedings ESEC ’95*, Springer-Verlag, 1995, pp. 137-153.

[20] Oscar Nierstrasz, Jean-Guy Schneider and Franz Achermann, “Agents Everywhere, All the Time,” *Joint ECOOP 2000 Workshops on Component-Oriented Programming and Pervasive Component Systems*.

[21] Davide Sangiorgi, “Interpreting functions as pi-calculus processes: a tutorial,” RR no. 3470, INRIA Sophia-Antipolis, France, February 1999.

[22] 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.

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

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

[25] Vivek P. Singhal, “A Programming Language for Writing Domain-Specific Software System Generators,” Ph.D. thesis, University of Texas at Austin, September 1996.

[26] Yannis Smaragdakis and Don Batory, “Implementing Reusable Object-Oriented Components,” *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

[27] Yannis Smaragdakis and Don Batory, “Implementing Layered Design with Mixin Layers,” *Proceedings ECOOP’98*, Eric Jul (Ed.), Brussels, Belgium, July 1998, pp. 550-570.

[28] Michael Van Hilst and David Notkin, “Using C++ Templates to Implement Role-Based Designs,” *JSSST International Symposium on Object Technologies for Advanced Software*, Springer Verlag, pp. 22-37.

[29] Philip Wadler, “Monads for functional programming,” *Advanced Functional Programming*, J. Jeuring and E. Meijer (Eds.), Springer, 1995.