# Combining Composition Styles in the Evolvable Language LAC

**Stephan Herrmann**
Technical University Berlin,
10587 Berlin, Germany
stephan@cs.tu-berlin.de

**Mira Mezini**
Darmstadt University of Technology,
64283 Darmstadt, Germany
mezini@informatik.tu-darmstadt.de

## 1 Introduction

Languages for Advanced Separation of Concerns (ASoC) have seen some rapid development during the last years. The current state is, however, in a way undefined. Some experience exists, where academic developers have applied ASoC technology to moderately sized systems. There is still more need for empirical studies in this field. None of the proposed languages really has yet proven its maturity for industrial style application.

The different language models pursued by different groups of researchers share the same intention of improving modularity in situations that where not support well by traditional languages. In spite of this common motivation, available ASoC languages are not necessarily compatible. Partly, they in fact address different sub–problems of broken modularity, partly, they utilize incompatible concepts for similar problems. The great merger has not happened.

While the two leading language implementations, AspectJ and Hyper/J are elaborated up-to a point, where moving towards each other seems to be impossible, a proposal dated 1999 [6] had not been implemented to date, which has the capability to combine some ideas from different groups of researchers. The model is called Aspectual Components. This paper presents a light–weight implementation of Aspectual Components, by which the design of this model is refined and open design decisions are cast into this discussion. We implemented LAC as a small extension to the interpreted language Lua [4]. Thus, LAC stands for *Lua Aspectual Components*.

In section 2, we will present the conceptual elements of Aspectual Components according to [6], supplemented by some considerations that arose while implementing LAC.

Section 3 presents the internal design of LAC, introducing concepts like facet/compound objects and run-time weaving as a special style of meta programming. This might lay the grounds for a larger family of languages.

Section 4 briefly shows, how the simplicity of Lua facilitated rapid development of LAC, which can easily be extended to other styles of defining and composing modules.

This work builds upon the experience, we gained when developing Pluggable Composite Adapters (PCA [7]) and Dynamic View Connectors (DVC [3]). Aside from similarities in the language model, DVCs also used some of the same technique, basically because it is implemented in the same language: Lua. There are two major differences between both developments: DVCs are embedded into a framework for software engineering environments, which puts a focus on many system properties, that are otherwise beyond the scope of a programming language. LAC does not carry this burden, but instead implements a language model that is much richer in terms of different styles of software composition.

## 2 Ingredients to Aspectual Components

Aspectual Components were in part inspired by the ideas of Adaptive Programming. The main similarity concerns the focus on graphs of classes, which both approaches share. A class graph is found to be the most appropriate unit when specifying collaborative behavior that involves not only a set of classes (resp. their objects) but also the structure of *relations* between classes (objects).

When considering class graphs as the components of a system, two kinds of components are distinguished: regular components ("base") can be identified with a concept like packages in Java. Fig. 1 introduces the syntax[1] of LAC by showing two (unrelated) classes, Point and FileLogger, that will serve as the base component for the following example. A base component is completely implemented or uses abstract methods that have to be

*Submission for ASoC Workshop at ICSE 2001.*

[1] This syntax is in a way non–standard, but note, that implementing LAC did not involve writing or modifying a parser. With only minor modifications, all examples in this paper follow the syntax of Lua.

```
Point = Class {
  attributes = {
    x : Integer,
    y : Integer
  },
  methods = {
    setX = method (x : Integer)
      self.x = x
    end,
    setY = method (y : Integer) . . .
    set = method (x : Integer, y : Integer)
      self.x = x
      self.y = y
    end
    swap = method (). . .
    tostring = method (). . .
  }
}
FileLogger = Class {
  attributes = {
    filename : String
  },
  methods = {
    log = method (msg : String)
      file_append(filename, msg)
    end
  }
}
```

Figure 1: Base component in LAC.

provided by subclasses of an abstract base class. The mechanism of subclassing is, however, not always suitable for filling in the missing details of a component. Aspectual Components add another kind of open spots in a collaboration, that are called *expected* methods. In the following the term *Collaboration* is used for a class graph that is abstract because it is implemented against an *expected interface*. The expected interface of a Collaboration is defined to be the set of all expected methods of its classes. These classes are called *Participants* of their respective Collaboration. Thus, a Collaboration is a partial implementation, that is, however, *declaratively complete*[2]. Fig. 2 shows a Collaboration that implements the publisher subscriber pattern, defining two roles: `Publisher` and `Subscriber`. `Publisher` provides the functionality of maintaining a list of subscribers (`subscribers`, `attach`, `detach`), while `Subscriber` declares a method `subUpdate`, by which a change may be notified to the subscriber. The open spots are expected methods `changeOp` and `subUpdate`.

```
PubSub = Collaboration {
  Publisher = Participant {
    expected = {
      changeOp = method () end
    },
    attributes = {
      subscribers : List
    },
    initialize = method ()
      self.subscribers = List:create()
    end,
    methods = {
      attach = method (sub: Subscriber)
        self.subscribers:append(sub)
      end,
      detach = method (sub: Subscriber) . . .
    },
    replacements = {
      changeOp = method ()
        expected ()
        self.subscribers:foreach(
            function (i: Integer, s: Subscriber)
              s:subUpdate(%self)
            end
        )
      end
    }
  },
  Subscriber = Participant {
    expected = {
      subUpdate = method (publ: Publisher) end
    }
  }
}
```

Figure 2: A Collaboration in LAC.

An expected method is *bound* to a real method during *deployment*, which connects the Participants of a Collaboration to classes of a base component. Expected methods differ from abstract methods, in that a Participant may *replace* an expected method. This might be compared to the hypothetical case of a superclass overriding the methods of its subclass. This reverse inheritance property allows to modify the behavior of a base by deploying a Collaboration. In our example, the expected method `changeOp` is replaced (wrapped) by a method that first passes control to the original version (`expected()`) and afterwards calls `subUpdate()` for each registered subscriber.

Deployment is defined by a module of its own that comprises a set of mappings between Participants and base classes, by which all expected methods are bound. Such a deployment module is called a *Connector*. Fig. 3 shows how the Collaboration `PubSub` is applied to classes

[2] This very usefull notion has been introduced in [8] as a property of hyperslices.

Point and FileLogger. At top level the role Subscriber is bound to class FileLogger and the role Publisher is bound to class Point. Each reference to a base class that is to play a given role is further specified by bindings of expected methods. So the player [FileLogger] in its role Subscriber implements subUpdate using its log() method (from FileLogger). It makes use of the fact, that in this context the corresponding Publisher must be a Point and invokes publ:tostring(). The player [Point] binds its expected method changeOp in a different way: The *Pattern* "set." refers to all methods whose names start with set followed by exactly one arbitrary character. In class Point this matches setX and setY, which means *both* methods play the role of changeOp, i.e., both methods are wrapped by the replacement changeOp from Publisher. If instead {"set.*", "swap"} were used as a pattern, also set and swap would be treated as changeOp.

```
LogSetOneCoordinate = Connector {
    Collaboration = PubSub,
    Subscriber = {
        [FileLogger] = {
            subUpdate = method (publ : Point)
                self:log("'set.'  on "..publ:tostring())
            end
        }
    },
    Publisher = {
        [Point] = {
            changeOp = "set."
        }
    }
}
```

Figure 3: A Connector in LAC.

**Composition Styles**

As we have just seen, Participants and expected methods can be bound in different ways. In its most simple form, a Collaboration mimics generic classes, and deployment binds its type parameters (=Participants) to actual classes. However, deployment will usually perform adaptations that would not be possible just by generic classes. Binding expected methods can be done declaratively or programmed.

- The declaration expected_name = provided_name binds one Participant method to one base method.
- expected_name = name_pattern binds one Participant method to a set of base methods with matching names.
- expected_name = method () ... end binds the Participant method to an implementation given here in the connector.

These declarations define which method(s) are *provided* to the Collaboration. The effect of binding is also determined by the absence or presence of a *replacement*. In the first case two methods are simply identified, while the second case introduces a modified (wrapped) version of the provided method.

Multiple binding may also occur on class level: If a class Line were added to the base component, the deployment could be changed to hold this mapping:

```
    Publisher = {
        [Point, Line] = {
            changeOp = "set."
        }
    }
```

This would add the Publisher functionality to both classes, Point and Line. Because this introduces common properties to classes that need not be related in any way within their component, this mimics the introduction of an anonymous ad-hoc supertype.

So far, we mainly followed the model given in [6]. The next section shows our design of the LAC interpreter.

## 3 Ingredients to the Design of LAC

When developing an interpreter for Aspectual Components, the first issue involved the operational semantics of a *Connector*. We found that, when using Lua as the basis for LAC, connectors are in fact meta programs, or more precisely: a connector definition is the input to a meta program, that also reads base classes and the Collaboration in order to produce a woven version that combines functionality from all three modules.

One step of this meta program is wrapping an expected method by its replacement. The generated wrapper has to ensure, that the replacement may invoke the original version by the syntax of expected(). This raises the question which arguments should be passed to that call, and how to proceed in the case of different signatures of expected and replacement. In Lua the answer is simple: the replacement may use any number of actual arguments or chose to ignore some or all of them (Lua automatically adjust the argument list). The call expected() is now implemented as a function *closure*, that is created by the wrapper such that it passes all original arguments to the original method. I.e., the replacement need not know about arguments, which are passed implicitly. Similar considerations regarding return values are currently being evaluated.

The next issue concerned the run-time structure that should be created by a Connector. Already [6] contains a few thoughts about different degrees of Connectors, which may be either static or dynamic. Anticipating any dynamic mapping, the classes generated by a LAC Connector are split into the base classes and additional *Facet* classes, from which also split objects are created

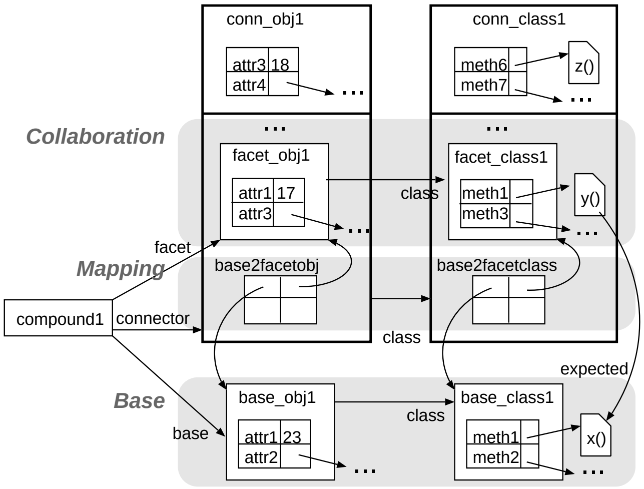

Figure 4: Run-time structures of the LAC interpreter.

at run-time, having a Facet part and a base part. So for any base object, a number of *CompoundObjects* may be created, that in addition to the base object designate a Facet and the enclosing Connector. Fig. 4 shows the structures that exist at run-time. Here we see, that a Connector also exists at run-time. It is the place where different mappings are kept. Connectors also have these properties of classes and objects: attributes, methods, inheritance.

Having this infrastructure of base, facet and connector — all at class and instance level — the behavior that had to be implemented is little more than just different styles of *delegation* between these different parts.

We then further elaborated the model of Aspectual Components by supporting three different kinds of Connectors: The structure given in Fig. 4 corresponds to a *DynamicConnector*, which needs to be instantiated before use. It allows the greatest level of flexibility because separate instances of the same Connector may define separate contexts, in which separate Facets of the same base objects may live. *DynamicConnectors* raise the issue of how to activate a Connector. Connector activation is responsible for entering the context which makes certain Facets visible to the program.

The second kind of Connectors is just a convenience for a restricted version of DynamicConnectors: a *SingletonConnector* has exactly one instance that is created implicitly. So, in terms of Fig. 4 `conn_class1` and `conn_obj1` can be merged to one structure.

*StaticConnectors* behave significantly different: these are the only Connectors, that actually modify the base. Only the replacements of a StaticConnector are woven directly into the corresponding base class. The original method is no longer available, except as the expected method of this replacement. Different situations have been identified, that require *implicit* activation of a Connector in order to find all Facets that may be needed within a method call.

Note, that two techniques allow to change the semantics of client programs which may be unaware of the deployment: A *StaticConnector* has effect on all objects of classes that were modified by the Connector, and this effect is on during the whole program. The other two kinds of Connectors allow an *explicit activation*, which temporarily changes the behavior of the interpreter such that the replacements of this Connector are also switched on. A client program that is aware of a Connector may also explicitly invoke a method within the context of a Connector, which is denoted by the special syntax `myconn[obj]:method()`.

## 4 Ingredients to the Implementation of LAC

Looking behind the scenes we can report, that Lua proved extremely convenient for this experiment. Its small imperative core provides one powerful data type: `table`, which is a hashtable of arbitrary key and value types. All runtime structures could easily be implemented as tables. The Connector meta program easily rearranges different class structures and benefits from the fact, that functions are values in Lua, and function closures can be created in order to store context information. The real speciality of Lua is the concept of *tagmethods* that allow to redefine the behavior for basic syntactical constructs. So, e.g., the evaluation of an expression `o.i` — i.e., lookup of field `i` in table `o` — can be redefined by a Lua function, that is installed as `gettable` tagmethod. These redefinitions happen just for a given set of values that are identified by a special *tag*. Thus, tags and their tagmethods range somewhere between subtypes of builtin types and freely programmable meta classes. By this mechanism all extensions to the interpreter as they were needed to implement LAC, could actually be performed in Lua itself. The details of this mechanism fall outside the scope of this paper, but the result is worth noting: the first version of the publisher subscriber pattern ran in an interpreter, that was extended by just 213 lines of Lua code. A more elaborated version, supporting all three kinds of Connectors uses two modules containing 88 resp. 522 net lines of Lua code. The first module implements standard classes and dynamic binding, the second module implements LAC.

## 5 Discussion and Related Work

The numbers given in the previous chapter underline, that LAC is a real light–weight language. Of course this is not an end by itself, but the small size should illustrate, how little effort it takes, to implement such an interpreter on top of Lua, because of the well designed hotspots by which the interpreter can be influenced.

By this technique we managed to combine different concepts that so far almost seemed incompatible. LAC allows modular software composition along these lines:

It allows *collaboration based development* like PCA[7], some techniques of generative programming [12] and Hyper/J [11].

Activating and deactivating Connectors provides for *dynamic, stateful contexts* as in PCA[7] and Context Relations [10].

Using *wildcard based weaving* allows modularization of code that would otherwise be scattered through many different places. In this, LAC resembles AspectJ [9].

*Composition* is performed by a separate module, the Connector, thus strictly decoupling Collaboration and base. Also Connectors are reusable software. This property it shares with PCA and Hyper/J.

By its *delegation* based implementation, LAC is certainly related to approaches like Composition Filters [1] and Darwin [5].

Consideration already go beyond this list of properties. While experimenting with a "Locking" Collaboration, we found that certain designs using Aspectual Components have the "Jumping Aspect" problem [2]. Adding just a slightly modified version of the `expected()` construct allows to explicitly invoke the base method *below* the Connector, i.e., without re-entering the replacement that is currently active. Thus the problem vanishes. This is just an example of well directed additions to the concept, that can be implemented by adding just a handful of lines to LAC.

Another example concerns the mechanism of selecting methods for replacement, which currently applies Lua's standard pattern matching function. This could easily be changed to the full power of regular expression, and even more striking: instead of a string pattern, a function could be allowed, that by introspective examinations collects methods for replacement. Such a function may rely on any meta information that can be made available at run-time. This illustrates, how at certain hotspots, the programmer might be allowed to directly provide pieces of meta programs, that determine the behavior of the weaver or similar mechanisms.

The experiment of implementing Aspectual Components in Lua has successfully shown, not only that Lua is suitable for rapid and evolutionary development of new programming languages. We have demonstrated how such a tool helps to elaborate programming models that have not yet been fully specified. We have settled some issues that had been left open in [6] and gave hints on further refinement of the model such as to give the power for solving problems of very different kinds.

Considerations about a type system for Aspectual Components have not been covered by this work. We believe that keeping execution and type checking of programs in separate tools is even a quite effective strategy. Issues of performance are only in part considered by the implementation. No measurements have been performed, yet. Of course, turn–around time when programming in LAC and when developing LAC itself is minimal, since Lua needs no compilation and starting the interpreter happens without noticable delay.

## REFERENCES

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition-filters. In *Object-Based Distributed Processing*, LNCS 791, pages 152–184, 1993. 5

[2] J. Brichau, W. De Meuter, and K. De Volder. Jumping aspects. position paper at the workshop "Aspects and Dimensions of Concerns", ECOOP 2000. 5

[3] S. Herrmann and M. Mezini. Pirol: A case study for multidimensional separation of concerns in software engineering environments. In *Proc. of OOPSLA 2000*. ACM. 1

[4] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996. 1

[5] Günter Kniesel. Type-safe delegation for run-time component adaptation. In *Proc. of ECOOP'99*, LNCS 1628, pages 351–366. 5

[6] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with aspectual components. In *Technical Report*, Northeastern University, April 1999. 1, 1, 3, 3, 5

[7] M. Mezini, L. Seiter, and K. Lieberherr. *Software Architecture and Component Technology: State of the Art in Research and Practice*, chapter Component Integration with Pluggable Composite Adapters. Kluwer Academic Publishers, 2000. 1, 5, 5

[8] H. Ossher and P. Tarr. *Software Architecture and Component Technology: State of the Art in Research and Practice*, chapter Multi-Dimensional Separation of Concerns and The Hyperspace Approach. Kluwer Academic Publishers, 2000. 2

[9] PARC Xerox, available from `http://aspectj.org`. *AspectJ Language Specification*, Aug 1999. 5

[10] L. Seiter, J. Palsberg, and K. Lieberherr. Evolution of Object Behavior using Context Relations. *IEEE Transactions on Software Engineering*, 24(1):79–92, Jan. 1998. 5

[11] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM Corporation, 2000. 5

[12] Michael VanHilst and David Notkin. Using role components to implement collaboration-based design. In *Proc. of OOPSLA'96*, volume 28(10) of *ACM SIGPLAN Notices*, 1996. 5