

Language support for feature mixing

Franz Achermann
Software Composition Group, University of Berne[†]

April 27, 2000

Abstract

Object oriented languages cannot express certain composition abstractions due to restricted abstraction power. A number of approaches, like SOP or AOP overcome this restriction, thus giving the programmer more possibilities to get a higher degree of separation of concern. We propose *forms*, extensible mappings from labels to values, as vehicle to implement and reason about composition abstractions. Forms unify a variety of concepts such as interfaces, environments, and contexts. We are prototyping a composition language where forms are the only and ubiquitous first class value. Using forms, it is possible compose software artifacts focusing on a single concern and thus achieve a high degree of separation of concern. We believe that using forms it also possible to compare and reason about the different composition mechanisms proposed.

1 Introduction

It is well accepted that software should be developed as manageable pieces, ideally each implementing a single concern. These pieces are then composed together to achieve the desired behaviour of the application. However, this composition is far from trivial, since often the concerns overlap and interfere. Tarr et al. [10] argue that we cannot achieve this separation in a single paradigm language due to one dominant dimension of separation, typically given by the language. For instance, object-oriented programming separates everything into objects, whereas other concerns like persistence or synchronisation get tangled into several objects of the application. The fact that each concern cannot be factored out into a single abstraction leads to components incorporating several varying aspects which hinders their re-usability in other contexts.

There is a number of different formalisms and techniques available to enhance separation of concerns during program development such as AOP [7], Composition Filters [3], Role Models [4], and SOP [5]. These techniques have in common that composed components yield new components. The composite contains the services of the sub-components possibly wrapped or adapted. For instance, the "merge" operation in SOP unifies the set of methods of two subjects, multiplexing methods with equal names.

We are looking for a common formalism behind these composition techniques. This formalism cannot be expressed in a traditional programming language because the abstraction mechanisms offered by the language are not expressive enough. Assume an aspect specifying mutual exclusiveness. Combining this aspect with an object wraps all the objects methods such that they run in mutual exclusive mode. This aspect may be expressed by a function f taking an object X and returning a wrapped object $f(X)$ with the same interface as X but all methods of X are wrapped

[†] *Authors' address:* Institut für Informatik und Angewandte Mathematik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Bern, Switzerland. *E-mail:* acherman@iam.unibe.ch *WWW:* <http://www.iam.unibe.ch/~scg>.

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 Nr. 96.0335-1.

In Second Workshop on Multi-dimensional Separation of Concerns in Software Engineering (ICSE 2000).

such that they run in mutually exclusive mode. The behaviour of f is to associate X with a semaphore and to wrap each method such that the semaphore is locked before invocation of the original method and released afterwards. However, we cannot express f as a function or a method in an object-oriented language like Java. This is due to the fact that:

- we cannot abstract over the methods of an object, unless we use behavioural introspection,
- we cannot recombine the methods unless they are first class values.

We propose *forms* as a uniform and simple interface to components. A set of form *operators* allows us to re-compose forms in a flexible way. Piccola [1] is a composition language based on the notion of forms. In a nutshell, the expressive power of Piccola language comes from the fact that everything is represented as a form and we can abstract over forms.

In Section 2 we sketch the primitive form operators needed and show how forms are composed in Piccola. Section 3 presents an example of a higher order composition operator to combine reader-writer synchronisation policies with arbitrary objects. Section 4 concludes the paper.

2 Programming with Forms

A form is a extensible, finite mapping from labels to values. A value may be a service or another form. Forms unify a number of concepts found in programming languages such as interfaces, environments, packages, and keyword-based parameters.

The following are the operators over forms:

Projection on a label denotes the value bound by a label in a form. If the form does not contain a binding for this label, a special `null` value is returned upon projection. Two forms are equal when projection on all labels yields the same result.

Extension is the operation to build forms. Binding extension adds a single label-value binding to a form, possibly overriding existing bindings.

Restriction hides the visibility of bindings in forms.

Introspection An iterator over forms enumerates the labels of that form. Since forms are immutable, iterating over their labels is not complicated by aliasing forms. While visiting a label we use it to extend, restrict, and project on other forms.

The original definition of forms [8] does not provide introspection facilities. Instead, polymorphic extension is the built-in operation to concatenate forms. Form concatenation can be defined with iteration.

Piccola is a small composition language that incorporates the notion of forms to represent interfaces, environments, packages, and keyword-based arguments. Here is an example of a form written in Piccola:

```
aForm =
  aSubForm = ()           # empty form
  aService(X): X         # service definition
  r(count = 3)           # form expression
```

The form `aForm` contains the labels `aSubForm`, `aService`, and all the labels that are returned by invoking the service `r`. If `r()` returns a form with label `aSubForm` or `aService`, these bindings will hide the bindings that precede the invocation. The service `r` is invoked with the argument form `count = 3`. Observe that the last line is not only a statement to invoke the service `r()` but it also denotes a form to extend the previous bindings. This is due to the fact that we uniformly represent everything as a form and it is the key to implement highly generic abstractions.

Services in Piccola are abstractions over forms. The runtime model of Piccola is that of autonomous agents that communicate forms along shared channels. Predefined services are used

to instantiate channels and to create new agents. Here is the definition of the library service `wrapPrePost` in Piccola. This abstraction wraps all services of a passed form by adding pre- and post-methods to all of them. The implementation uses the built-in iterator `forEachLabel` to introspect the services of the passed form:

```

1 wrapPrePost(X):
2   res = newRefcell()           # Cell to contain the result
3   forEachLabel
4     form = X.form
5     do(Label):                 # for each service in X.form do:
6       service = Label.project(X.form) # unwrapped service
7       newResult = Label.extend    # extend previous result
8         form = res.get()
9         value(Args):              # with this service:
10          X.pre()                  # invoke passed pre-method
11          service(Args)           # invoke original service
12          X.post()                 # invoke passed post-method
13   res.set(newResult)
14   return res.get()

```

We do not explain the above code in detail, and focus only on the usage of the predefined iterator `forEachLabel`. It is invoked passing the form to iterate over (line 4) and a `do` service (line 5 - 13). The `do` service contains the code to be executed for each label. Within its body, we use the current label `Label` to `project` (line 6) and to `extend` (line 7) other forms.

3 A Composition Operator

Using form operators in Piccola, we can develop a library of composition operators. The definition of the generic wrapper `wrapPrePost` using the built-in iterator is a bit low level. But we now can use it to compose forms at higher level — for instance to synchronise arbitrary objects using the reader-writer policy. The policy states that multiple readers can be active within an object, provided no writer is active simultaneously. Writers themselves must be mutually exclusive. A reader-writer policy can be implemented as a component providing four services: two services `preR` and `preW` control entry for readers and writers and two services `postR` and `postW` signal that executing of the corresponding method has finished.

A *reader-writer-wiring* service `wireRWPolicy` wires a set of reader and writer methods with a given reader writer policy. For instance, a form `F` with two reader methods `r1` and `r2` and a writer method `w` is composed as:

```

synchronizedF = wireRWPolicy
  policy = newRWPolicy()           # create reader writer policy
  reader = (r1 = F.r1, r2 = F.r2) # r1 and r2 are reader methods
  writer = (w = F.w)              # w is a writer method

```

Then the form `synchronizedF` has service `r1`, `r2` and `w` with the reader writer policy incorporated. The service `wireRWPolicy` is invoked passing three nested forms: a policy, a form containing the readers, and form containing the writers. The implementation of the service `wireRWPolicy` simply wraps the readers and writers according the policy:

```

wireRWPolicy(X):
  wrapPrePost(form = X.reader, pre = X.policy.preR, post = X.policy.postR)
  wrapPrePost(form = X.writer, pre = X.policy.preW, post = X.policy.postW)

```

This example shows how to cleanly separate the implementation of the core functionality and its synchronisation. The composition of the two concerns is done by a user defined composition abstraction. The composition cannot be expressed by other object-oriented languages like Java, since these languages cannot abstract over sets of methods and recombine them.

We claim that it is feasible to implement in Piccola composition abstractions identified by the different approaches to separate concerns like SOP or AOP. This helps clarifying the differences between the approaches and helps developing and experimenting with new notions of composition. We also believe that the simple and compact notion of forms serves as a tool to give precise semantics for composition abstractions and will permit us to reason about them.

An interesting question is whether we can use semantics-based crosscutting [6] to derive the reader and writer methods in the above example instead of explicitly listing them.

4 Conclusion and Future Work

We have sketched how forms give languages better support for higher level composition abstractions. In Piccola we can program adapters and wrappers in a compact way without the need to write lots of boilerplate code. Other applications of forms include a user defined exception handling mechanism [2] or various kinds of inheritance [9].

The composition abstractions presented can also be implemented using meta programming, for instance in Lisp or CLOS. But form composition provides a more lightweight approach than meta programming. It is less expressive than meta programming in general as the primitive form operators like projection and extension cannot be changed.

In the original definition of forms [8], forms are extensible records. Primitive operators are projection and extension. In the form-calculus, Schneider [9] additionally defined restriction and label matching as operators on forms. In this paper, we propose forms with some simple reflective facilities which makes it possible to iterate over the labels of a form eliminating the need for polymorphic extension and restriction as well as label matching. We are currently missing a sophisticated type system for forms. Initial work in that area is done by Lumpe on the $\pi\mathcal{L}$ -calculus [8].

Our long term goal is to develop a framework which supports the definition of higher level composition operators and in which we can reason about properties of composite components. Piccola should serve as a platform to develop a composition environment hosting components and supporting the flexible scripting of components within user defined architectural styles.

References

- [1] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*. Kluwer, 2000. to appear.
- [2] Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In *Proceedings of JMLC 2000*, 2000. to appear.
- [3] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object Interactions Using Composition Filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, LNCS 791, pages 152–184. Springer-Verlag, 1994.
- [4] Egil P. Andersen and Trygve Reenskaug. System Design by Composing Structures of Interacting Objects. In O. Lehrmann Madsen, editor, *Proceedings ECOOP'92*, LNCS 615, pages 133–152, Utrecht, The Netherlands, June 1992. Springer-Verlag.
- [5] William Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 411–428, October 1993.
- [6] Gregor Kiczales, Jim Hugunin, Mik Kersten, John Lamping, Cristina Lopes, and William G. Griswold. Semantics-Based Crosscutting in AspectJ. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, 2000.
- [7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, LNCS 1241, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.
- [8] 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.
- [9] 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.
- [10] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of ICSE'99*, pages 107–119, Los Angeles CA, USA, 1999.