

On Objects, Aspects, and Specifications Addressing their Collaboration

Tommi Mikkonen

Tampere University of Technology, P.O.Box 553, FIN-33101 Tampere, Finland

tjm@cs.tut.fi

ABSTRACT

Aspect-oriented programming enables addressing of cross-cutting concerns in a modular fashion. However, for utilizing that type of modularity effectively, we need to design architectures that enable well-considered use of aspects. This calls for new types of methodologies that enable the treatment of both aspects and objects as first-class citizens, instead of currently advocated approaches like UML that primarily place the focus on object-oriented architecture only, leaving aspects only a role as add-on facilities. In this paper, we sketch an approach where the relation of aspects and objects is defined already in the specification phase, thus allowing balancing between their role in the design. The approach is derived from experiences gained with the specification language DisCo, which was originally targeted as a formal specification method for reactive systems.

1. INTRODUCTION

Aspect-oriented programming enables addressing of cross-cutting concerns in a modular fashion [4]. However, for expressing them in AspectJ language [14], for instance, one needs to compose a completed architecture for the primary design goals first, because aspects are attached to an object-oriented design as a secondary dimension of concerns. The primary design goals should be satisfied with conventional specification notations like UML [16], on top of which aspects are attached. In contrast, hyperslices [11] first require an existing architecture used as a reference for composing different aspects, which can then be given separately.

A consequence of the above approaches is that architecting of systems consisting of collaborating aspects and objects is hard. Moreover, applying iterative and incremental development approaches (see e.g. [6]) that are currently considered favorable is hard. Such practices implicitly assume that software systems can be designed one set of features at a time, whereas managing the features in codesigns of objects and aspects is not supported. While the use of aspects as such enables a closer relation between features and code than conventional approaches, designing a new increment that does not interfere with already existing software is not eased with aspects in the general case. In fact, mastering the interaction of cross-cutting

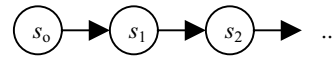


Figure 1. Execution as a state sequence

concerns and conventional code can be harder than if only traditional approaches had been used. To overcome the above problems, we need facilities that enable enhanced facilities for specifying the relation of aspects and conventional objects.

The rest of this paper is structured as follows. Section 2 discusses specification of software architectures in general, and provides an introduction on the different dimensions of software architectures that address the use of both objects and aspects. Section 3 introduces the specification method DisCo [15], and lists practical experiences on aspect-oriented modeling and specification gained with it. Section 4 finally concludes the paper with some final remarks.

2. SPECIFYING AN ARCHITECTURE

Software architecture is the first thing to be fixed when developing a software system [13]. Describing an architecture means construction of an abstract model that exhibits certain kinds of intended properties. Such a model is *operational*, if it formalizes executions as state sequences, as illustrated in Figure 1. In the figure, all the variables in the model have unique values in each state s_i .

2.1 Conventional Architecture Modeling

Fundamentally, the purpose of architecting is to partition the intended system into smaller pieces that can be attacked separately. In conventional approaches, the core is to define interfaces that encapsulate internals of modules. With the interfaces remaining unchanged, the underlying implementation can be changed relatively easily to use a more memory-efficient data structure, for instance. The design of behaviors can then be based on scenarios or use cases that denote the sequences needed for executions in terms of method names.

The use of interfaces and scenarios as the semantics of behaviors relies on the expectation that the semantics of an execution can be composed from the semantics of the

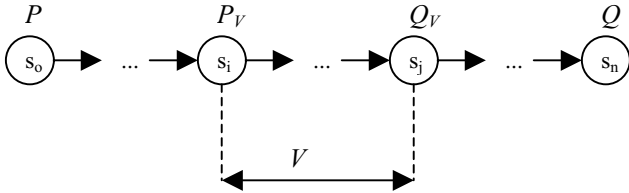


Figure 2. Subsequence in a behavior

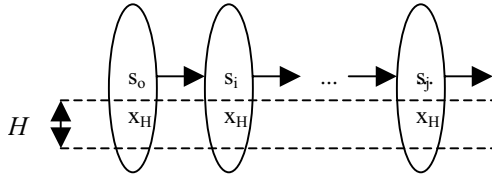


Figure 3. Horizontal projection in a behavior

methods attached to the components in a conventional architecture. More generally, an architecture that consists of conventional units can be seen to impose a structure of nested subsequences on each state sequence. Operations available for addressing the behavior of the system with architectural units are then *sequential composition*, i.e., concatenation of operations of one component, and *invocation*, i.e., embedding of a sequence produced by a component in a longer sequence. For both primitive operations, the resulting state sequences have subsequences for which the components are responsible, as illustrated in Figure 2.

In this paper, we will refer to the architectural dimension represented by this kind of modularity as *vertical*. Vertical architectures provide a natural basis for structuring the responsibility for implementation of different components in conventional systems.

2.2 Addressing cross-cutting concerns

As an alternative for focusing on invoked interface operations, the architecture can be modeled by focusing on how the variables of the system behave in an execution, similarly to program slicing [12]. However, unlike in conventional slicing, we are interested in composing new systems with such projections, not on decomposing existing ones. Such an architecture imposes a slicing or a projection of state sequences, where each unit is responsible for some subset of variables in all phases of the execution. We will refer to this architecture dimension as *horizontal*. The situation is illustrated in Figure 3.

The use of horizontal units makes the generation of state sequences fundamentally different from sequences relying on vertical architectures. The two basic operations between horizontal units are *parallel composition* that merges state sequences generated by the component units, and *superposition* that utilizes some sequences generated by a component unit, embedding them in sequences that

involve a larger set of variables. With both primitive operations, the resulting state sequences define projections for which the horizontal components are responsible.

The two dimensions of an architecture characterized above contrast each other. From the viewpoint of a vertical architecture, the behaviors generated by horizontal units represent crosscutting concerns, and from the horizontal viewpoint vertical units emerge incrementally as horizontal units are put together and embedded in larger units.

Unfortunately, we have no universally accepted approach for composing systems out of horizontal projections. The fundamental problem is that with projections, objects “grow” when more and more projections are introduced. Handling this growing at the level of programming abstraction is difficult, because the mapping of behavioral increments to a system with control flows becomes increasingly complex. Approaches that have an established reputation include aspect-oriented programming as introduced in AspectJ, where the developer tells places for join points in an existing system, frameworks, where the developer is responsible for the correct use of inheritance, and design patterns, where the designer should basically only reuse interfaces and ideas of collaboration. In hyperslices, an existing architecture is used to guide the composition.

We argue that the use of the horizontal architectural dimension is essential for improved understanding of software, as already demonstrated by the use of aspects and program slices. However, in order to architect with the horizontal dimension, we must overcome problems of composition, interference, and control flows. To accomplish this, the level of abstraction must be raised. The price is that while the systems remain executable, there may not be a direct mapping to program code.

3. USING HORIZONTAL DIMENSION

Our experiences on architecting with horizontal units of architecture arise from the design and use of the specification method DisCo [15]. In the following, we will provide a short introduction to the method.

3.1 Introduction of the DisCo method

The two basic items of every DisCo specification are objects and actions. Objects are instances of classes. For example, the following class definition introduces a class that contains one instance variable x of type integer.

```
class myClass = { x : integer };
-- Instances of this class contain instance
-- variable x.
```

Actions are used to mutate the states of objects involved in an execution. Each action has a signature that contains a list of objects needed for an execution, an enabling guard, and a list of assignments to the variables of involved objects. Thus, actions can be taken as multi-object

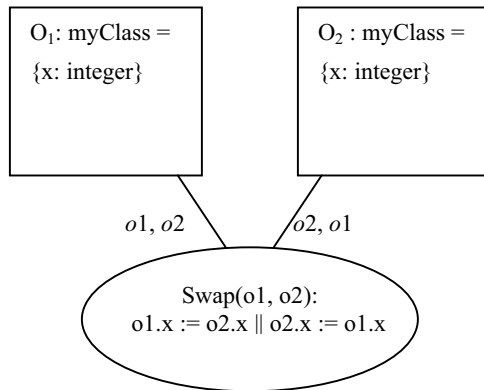


Figure 4. Simple action and two objects

methods. The following excerpt introduces a simple action that swaps the values of two instances of class *myClass*. In the action, Ada-style comments are used for explaining its structure.

```

action Swap(o1, o2 : myClass) is
-- Signature (participating objects,
--           their types and roles)
when true do
-- Enabling condition
    o1.x := o2.x || o2.x := o1.x;
-- Changes in states of participants
end Swap;
  
```

Figure 4 gives an illustration of two simple objects (boxes) and one simple action with two parallel assignments (ellipse) given above. Participant roles in the action are also included in the figure. The execution of actions is atomic, i.e., once started, executions are bound to be finished without interference. Executions take place in an interleaving fashion. The action to be executed next is selected nondeterministically from those whose enabling condition is true. Therefore, problems related to control flows need not be considered except when such restrictions are explicitly included in an action-based model.

Specifications given with actions and objects, i.e., instantiated classes, are used for generating operational interpretations. This has allowed us to introduce tool support for executing completed specifications [1].

3.2 Incrementality

For modularity, the main mechanism adopted in the DisCo approach is superposition. As such, superposition is a well-known technique whose usage was first reported in [3]. However, its close relation to aspect-orientation was pointed out only more recently by Katz and Gil [8].

In DisCo, superposition is used as follows. Each module of the specification is given as a *layer* that adds variables and

related operations to existing systems. Layers are horizontal. In other words, they are allowed to add details to multiple classes and actions, which make objects given in the specification grow.

When using superposition in connection with a joint action, new participants, restrictions on executions, and new assignments can be added, but they can never violate the original action. Logically, this means that any action must imply all the actions it has been derived from with superposition. In practice, the only restriction for new or modified actions is that it is only possible to modify the values of variables given in the same layer. For instance, extending the previously given class with a variable denoting whether or not the system is turned on or not would yield the following extensions. In the listing, three dots (...) are used to refer to the old parts of actions.

```

class myClass =
    myClass + {power: Boolean};
-- Adds a new variable to the class

refined Swap is
-- Adds new issues to an existing action
when ... o1.power and o2.power do
-- Makes the enabling condition stronger
    ...
-- No new state changes
end Swap;
  
```

In addition, an action is needed for turning the power on and off,

```

action OnOff(o:myClass) is -- New action
when true do
    o.power := not o.power;
-- Only new variables are assigned to
end OnOff;
  
```

Figure 5 extends the previous figure with the above additions. The contents of the new layer are included in the figure with italics, indicating the additions of the new layer in a complete system.

3.3 Relation to aspects and hyperslices

In the above example, it is essential that while the effect of the new layer is cross-cutting, its syntactic representation remains modular. Superposition steps are the building blocks for specifications, and, vice versa, specifications can be projected to individual layers where superposition has been used. Therefore, with each layer applying

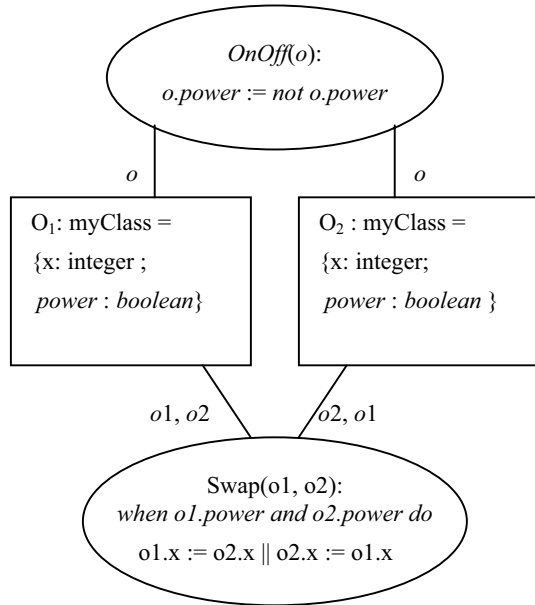


Figure 5. Introducing a new layer

superposition, it is possible to compose separately given layers into a more complex specification. Practice has shown that a common way to give specifications is to have one common ancestor specification that gives the most general specification for a system. Then, different branches introduce new properties in an aspect-oriented fashion. In the end, all the branches are composed into a complete specification, if all the layers introduce disjoint sets of new variables. This results in a true aspect-oriented specification style, where different aspects are given separately with superposition acting as the guarantee for later composability. Effectively, this approach provides facilities that are similar to hyperslices [11] for specification, with each branch representing a different hyperslice, with enforced composability.

Figure 6 gives an example of this approach. The figure illustrates the specification architecture of a telecom exchange modeled with DisCo. Each superposition and composition operations and resulting specifications have been explicitly depicted. Each of the different branches (or hyperslice) has an effect on e.g. on how call control takes place, which finally becomes formalized completely in specification *Completed Connections*. Being able to give the different concerns separately makes it easier to understand their eventual relation in the implementation. Figure 7 lists the attributes of class *Connection* derived in the example mentioned above. The contribution of the different layers is denoted with comments in more detail. In addition to the variables in the class, the DisCo specification also gives a definition for the behavior in a modular fashion.

Published results have confirmed that the DisCo approach enables formal specification of combinations of patterns

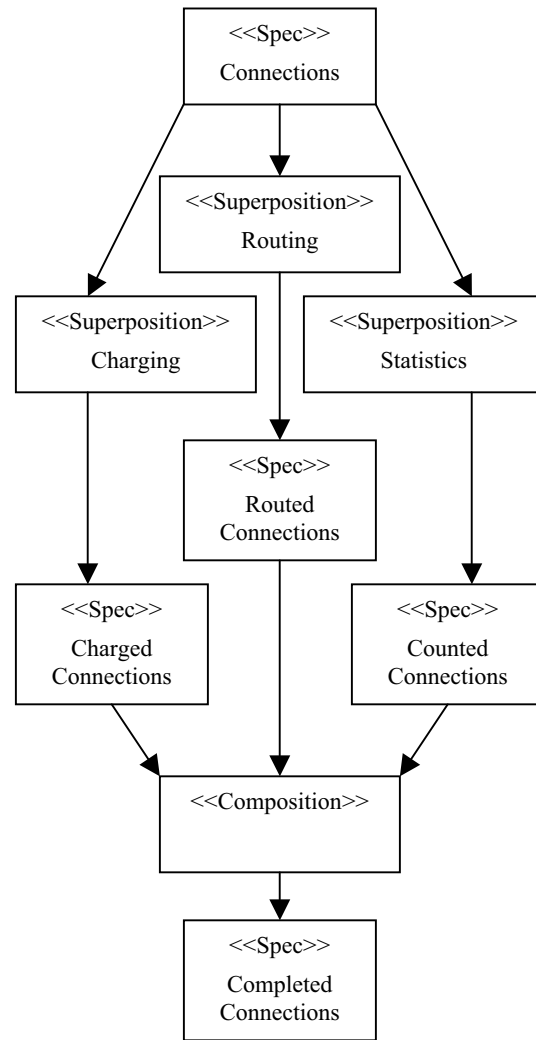


Figure 6. Specification architecture of a telephone

[9]. As an example, we derived a specification of a system utilizing both Mediator and Observer patterns of [5], with formal justification provided for the interaction of the patterns. Moreover, the approach has also been used in aspect-oriented specification of a real-time system in [7]. In that context, real time was treated as a separate aspect at the specification level. The approach has also been used for modeling and specification of increments delivered in different releases [10] and for the management of software evolution [2], which both benefit from advanced separation of concerns. Notice however that the level of DisCo specifications is abstract, and additional design decisions are needed for a practical implementation. Of course, as the specification contains both objects and aspects as separable entities, it is easier to partition the specification into objects and aspects for implementation.

4. CONCLUSIONS

In order to use both aspects and objects already in the specification phase, we need a methodology that considers both aspects and objects as first-class citizens. As long as

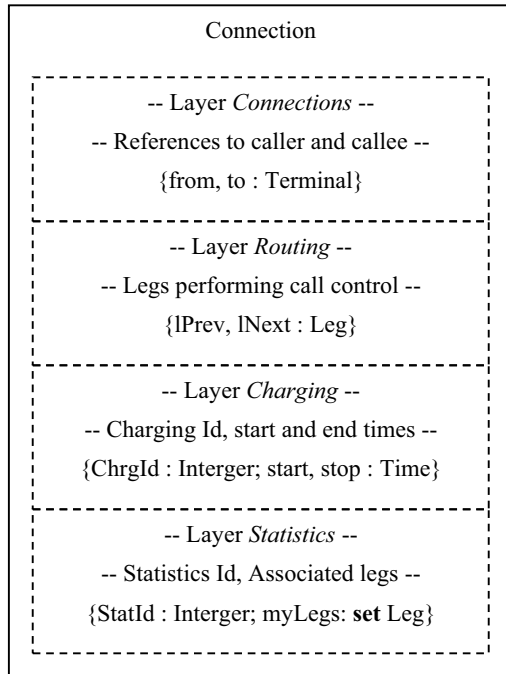


Figure 7. Class *Connection* in the specification

we consider aspects as add-on features to already complete systems, we do not intentionally design systems so that a certain efficient aspect implementation would be an option. Therefore, we are bound to use aspects inefficiently, in ad-hoc fashion, and for secondary features only.

In this paper, we sketched an approach that is capable of handling the relation between objects and aspects at an abstract level. The driving force behind the introduced specification approach is the use of superposition as a basis for modularity instead of conventionally used invocation-based relations. This allows specifications where both objects and aspects are addressed in a collaborative fashion. Moreover, the focus can be placed on both objects and aspects at the same time.

In connection with conventionally accepted aspect-oriented approaches, the proposed method relates as follows. The way we separate between logically different issues reminds hyperslices in the sense that the parts that are relevant for some concern are addressed in a modular fashion. At the same time, individual modules that the specifier gives remind aspects of AspectJ. We argue that this combination, in connection with raised level of abstraction and associated tools, provides a natural basis for specifications addressing objects, aspects, and their collaboration.

REFERENCES

- [1] Aaltonen, T., Katara, M. and Pitkänen, R. DisCo toolset – the new generation. *Journal of Universal Computer Science*, 7(1):3-18, 2001.
- [2] Aaltonen, T. and Mikkonen, T. Managing software evolution with a formalized abstraction hierarchy. *Proc. International Workshop on Formal Foundations of Software Evolution*, Lisbon, Portugal, March 2001.
- [3] Dijkstra, E. W. and Scholten, C. S. Termination detection for diffusing computations. *Information Processing Letters*, 11(1), 1-4, 1980.
- [4] Elrad T., Filman, R.E. and Bader, E. Aspect-oriented programming. *Communications of the ACM*, 11(1): 29-32, October 2001.
- [5] Gamma, E., Helm, R., Johnson, R., and Vlissides J. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
- [6] Jacobson, I. Booch, G. and Rumbaugh, J. *The Unified Software Development Process*. Addison Wesley, Reading, MA, 1999.
- [7] Katara M. and Mikkonen, T. Aspect-oriented specification architectures for distributed real-time systems. 180-190, *Proceedings of the Seventh IEEE International Conference on Engineering of Complex Systems (Eds. S.F. Andler, M.G. Hinchey, and J. Offutt)*, IEEE Computer Society Press, 2001.
- [8] Katz, S. and Gil, J. Aspects and superimpositions. *Position paper in Aspect Oriented programming workshop in ECOOP'99*, Lisbon, Portugal, June 1999.
- [9] Mikkonen, T. Formalizing design patterns. 115-124, *Proceedings of the 1998 International Conference on Software Engineering*, IEEE Computer Society, 1998.
- [10] Mikkonen, T. and Järvinen, H.-M. Specifying for releases. *International Workshop on Principles of Software Evolution*, 118-122, April 20-21, Kyoto, Japan, 1998.
- [11] Ossher, H. and Tarr, P. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 11(1): 43-50, October 2001.
- [12] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering*, 352-357, 10(4), 1984.
- [13] IEEE recommended practice for architecture description. *IEEE std 1471*, 2000.
- [14] AspectJ homepage. At URL <http://aspectj.org>.
- [15] DisCo project homepage. At URL <http://disco.cs.tut.fi>.
- [16] UML homepage. At URL <http://www.rational.com/uml>