Electronic Notes in Theoretical Computer Science 68 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume68.html 20 pages

Coordinating Aspects and Objects

Timo Aaltonen [1], Joni Helin [2], Mika Katara [3], Pertti Kellomäki [4] and Tommi Mikkonen [5]

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

Abstract

Conventional software architectures emphasize individual software components and their interconnections. While offering many advantages this results in problems with concerns that cut across the component structure. The code addressing such concerns is scattered around the components and tangled with some other code inside the components. Approaches addressing such issues are emerging with new paradigms like *aspect-oriented programming*. However, aspects addressing cross-cutting concerns need to be incorporated in an object-oriented design without support from a higher-level design. We propose an approach where aspects and objects rely on coordination provided by a common high level specification. The common specification links the parts of specification that are refined independently and implemented using different techniques. We use the formal specification method DisCo, and demonstrate the approach by providing a specification of a simplified telecommunications system. In addition, we also sketch an AspectJ implementation built in the architectural style encouraged by the method.

1 Introduction

Software architectures emphasize individual components and their interconnections, usually called connectors. Dividing a system into components with well defined interfaces allows partitioning of work to separate teams that can proceed in parallel towards a more detailed design. The thesis of aspect-oriented software development [9,25] is that this modularity is not always natural, as addressing some concerns, i.e. matters of consideration, sometimes involves cooperation of several components. For example, many quality

[1] Email: tta@cs.tut.fi
[2] Email: jjhelin@cs.tut.fi
[3] Email: mika.katara@tut.fi
[4] Email: pk@cs.tut.fi
[5] Email: tjm@cs.tut.fi

attributes have system-wide effects. The code addressing such concerns is usually scattered around the components and tangled with some other code inside the components resulting in poor traceability and maintainability.

Programming languages dealing with cross-cutting concerns provide mechanisms for expressing the concerns in a modular way at the source code level. That is, a concern that would otherwise cut across several modules, classes for instance, is addressed in a separate module independently of other concerns. When the final executable is created from source code, code fragments relating to the cross-cutting concerns are inserted into the appropriate places.

Conflicting requirements for modularity are the result of multiple dimensions of concern [6]. For instance, on the one hand, modules should reflect the allocation of work in terms of modularity in programming languages. On the other hand, issues like error handling or logging should be uniform in all modules. Designing a software architecture that reflects all these issues is hard at best and impossible at worst.

In general, modularizing a specification or a program in a way that is natural for one concern may lead to a poor result with respect to the other concerns. The ultimate solution is to give each dimension of concern using a modularity deemed to be most appropriate, and merge the dimensions in a consistent way. However, merging the dimensions and ensuring that they are consistent is a difficult problem in the general case. Selecting one of the dimensions in a primary role and making the other dimensions subservient to it simplifies the problem, and brings it closer to current object-oriented practice. It also appears to work in many practical cases, as evidenced by numerous examples listed in [24], including logging, tracing and profiling.

Programming language AspectJ [24] is an example of this approach. In AspectJ programs, the primary dimension is a class structure, and any other dimensions are expressed relative to it using *aspects*. With aspects, a programmer can use a single syntactic unit to express functionality that crosses implementation module boundaries. [7]

A pragmatic way to deal with secondary dimensions is to require them to be increments that cannot alter the behavior indicated by the primary dimension. The design of the primary dimension is then needed before focusing to other dimensions of concern, which prevents their parallel design.

We propose an approach in which abstract coordinated behavior is first refined into more concrete behavior, and finally to a combination of aspect code and traditional object-oriented code. Separate concerns can be tackled in separate branches of specification, with the common high level specification serving as the interface between the different branches. Our proposal arises from experience with the DisCo specification language [11,26], which focuses

[6] A dimension of concern is a set of disjoint concerns. Concerns in the same dimension cannot overlap, but concerns in different dimensions can overlap [27].

[7] We use the term *aspect* strictly to refer to the syntactic units in AspectJ.

on early specification of collective behavior.

The rest of the paper is organized as follows. Section 2 discusses the proposed specification methodology, and Section 3 gives an example application of the approach. Related work is discussed in Section 4, and conclusions are drawn in Section 5.

2 Coordinating Aspects and Objects

High-level design of software systems requires delicate management of cooperating components. A major source of complexity is the coordination of the components, which should be tackled early on in a high level design. The following discussion provides an introduction to one particular approach that places special emphasis on these issues and provides facilities for coordinating the interactions of aspects and objects.

2.1 Problems in Reasoning about Aspect Programs

AspectJ [24] is an aspect-oriented extension of the programming language Java [10]. In AspectJ programs, the primary dimension is a class structure, and any other dimensions are expressed relative to it using *aspects*. With aspects, a programmer can use a single syntactic unit to express functionality that crosses implementation module boundaries. This scheme enables a modular introduction of cross-cutting concerns on top of a class-based architecture. In AspectJ *join points* indicate where aspect are woven into Java programs. Join points are typically associated with executions of method calls. *Pointcuts* are collections of join points for convenience and maintainability purposes.

AspectJ provides two ways to indicate flow of control between code segments given in different syntactic units. The traditional object-oriented way is method invocation, while *aspects* allow the programmer to give additional code to be executed at selected join points. This gives more freedom in where in the source code a particular functionality is addressed.

An AspectJ program is an object-oriented class hierarchy augmented with aspects that express the cross-cutting concerns. In order for the design of an AspectJ program to be manageable, aspects should not modify the behavior expressed by the class hierarchy in unexpected ways. Poor design may lead to systems that are hard to maintain and debug. Understanding where aspects do and do not interfere with the object design may require intimate knowledge on both the object design and the aspects. Furthermore, aspects may violate the architecture of the rest of the system by introducing dependencies that are not visible in the object-oriented design. The hidden dependencies can then introduce problems for e.g. distribution.

The fundamental property of any software program is behavior. Despite the assumption that aspects do not interfere, an introduction of aspects necessarily modifies behaviors of software systems they are woven into. Therefore,

code associated with an aspect can alter the behavior of a system as a whole. For instance, when an aspect and an object assign to the same variable, hard debugging problems may result. At very least, showing that harmful interference does not occur can cause a lot of reasoning, and may require intimate knowledge on both the object design and incorporated aspects.

In addition to the problems discussed above, there are also difficult issues related to the design and analysis of aspects in general. Behaviors related to aspects are only meaningful when analyzed in connection with a matching object-oriented design. It is only possible to analyze behavior of a system at the level where aspects and objects have been merged. If the order in which aspects are incorporated in a design is modified, unforeseen changes in the behavior may occur. This can lead to bugs that are hard to find.

While aspects designed by experts of aspect-oriented programming are easy to use and offer powerful programming support, the design of new aspects is much more difficult. The underlying general problem is that while aspects and classes are syntactic constructs and can therefore be separated at the level of programming language, the behavioral semantics of aspects and objects are interconnected only in the final executable. Handling this semantic interconnection adds complexity to the resulting design and can make reasoning about its behavioral properties difficult.

2.2 The Need for a Specification Methodology

Software design involves multiple levels of abstraction, from requirements down to source code. Ideally the levels should be related in an easily understandable fashion. In contrast, large gaps in the semantics of different levels make it hard to ensure consistency in the face of changes.

Using object-oriented design methods to specify aspect programs introduces such a gap. As pointed out above, aspects give a programmer the freedom to introduce additional functionality in almost arbitrary places in a program, whereas object-oriented design notations like e.g. UML [22] lack the ability to express the equivalent at the design level. Any uses of aspects at the implementation level are thus *ad hoc* without being traceable to the design at a higher level of abstraction. Without design level knowledge on how classes and aspects should be coordinated it would be all too easy to create unmaintainable systems that depend on non-obvious, undocumented interactions between aspects.

Aspects by definition address concerns that cross-cut several implementation level classes, so a matching specification method must be able to express such concerns as well. Traditional object-oriented design methods fail in incorporating aspects, because of their strong focus on classes and methods. A more abstract view of the collective behavior of objects is thus required.

We argue that a specification methodology is needed that guides implementors to structured use of objects and aspects. Our claim is that the points

where objects and aspects interact must be identifiable in a design, not only in the generated program code. Documenting these points also tells where such interactions do not take place. To achieve this, classes and aspects must be visible at design level. Conventional design approaches are not well suited to this, as they present each class (or object) in isolation. We propose starting at a higher level of abstraction, at the level of collective operations involving multiple objects. Whether to implement some functionality using aspects then becomes a design decision that can be based on a higher-level operational specification.

We view coordinating aspects and objects analogous to architecting a system where different technologies are used in different parts of the implementation. This requires a notation and a methodology where the technologies can be expressed at a more abstract level.

2.3 Abstracting the Interaction of Aspects and Objects

Our approach to coordinating aspects and objects is based on *joint actions*. Originally introduced by Back and Kurki-Suonio [2,3], they can be characterized as follows. A joint action represents a collective step in an execution, somewhat like a multi-object method. The step indicates which objects are involved in the step, but does not specify how they communicate.

Joint actions can be used to indicate join points. An illustration visualizing the difference of a method invocation, a join point, and a joint action is given in Figure 1. In a method call, the flow of control is explicit: each method needs to be explicitly invoked in order to be executed. Using aspects it is possible to give the behavior of *Object1* and *Object2* in the figure without referring to *Object3*. Functionality relating to *Object3* can be given separately in the aspect code. A joint action is an abstraction of both of the cases, it specifies the changes to the local states of the objects without indicating flow of control.

A sequence of executed joint actions models an execution of a system, where different objects and aspects collaborate. This simple execution model can be considered as an abstraction of AspectJ programs.

Joint actions are a mathematical construct for specification of collaborative systems, not a programming mechanism or an implementation method. When an AspectJ implementation of a joint action based specification is constructed, joint actions are implemented by method calls and execution of aspect code.

Joint actions offer a common notation for capturing both method calls and invocations of aspects. Properties of objects and aspects can be reasoned about in terms of joint actions, which allows their use as the common theoretical framework. When relying on this framework, structuring of aspects and objects becomes easier, as reasoning can be carried out at a level of abstraction above program code.

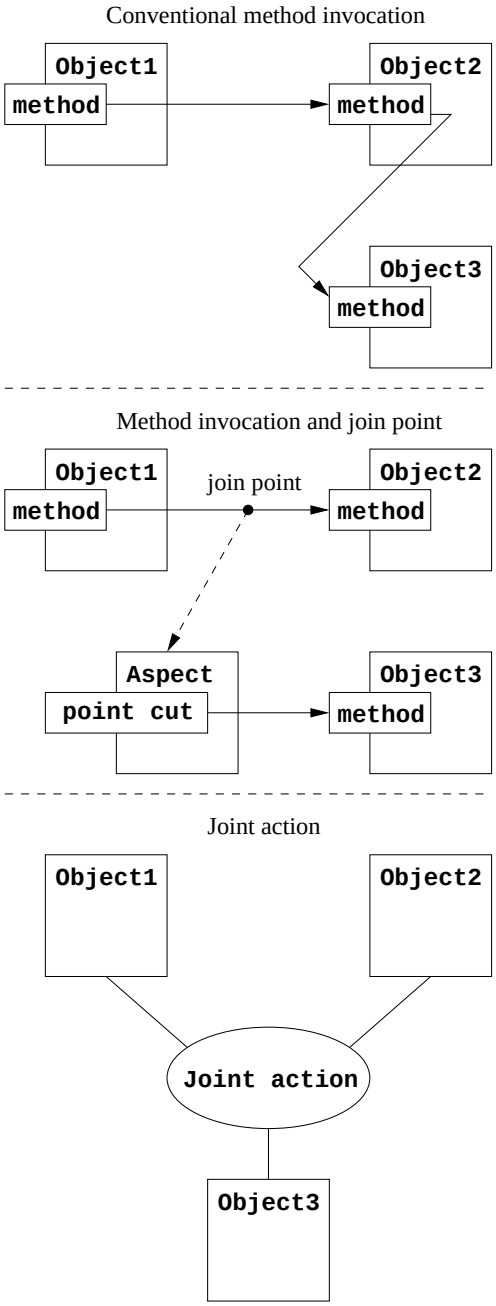

Fig. 1. Methods and Join points vs. Joint action

2.4 Generalized Form of Modularity

There is no fundamental form for software systems. In other words, we are not bound to introduce systems one implementation module at a time as often proposed in conventional programming setting. With joint actions, we have found it convenient to exercise our liberty as follows.

A joint action system describes the behavior of a system in terms of variables and related operations without committing to any particular partitioning

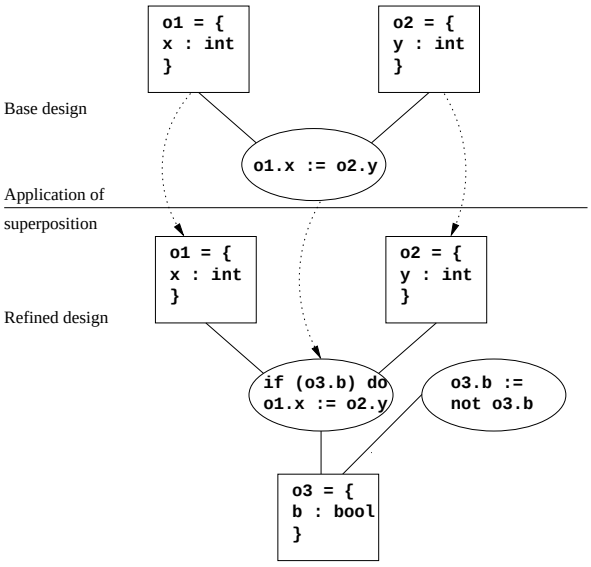

Fig. 2. A design step applying superposition

of functionality to implementation components. Instead of using implementation components as building blocks, we advocate using *superposition* for structuring specifications. Superposition is a well-known technique (e.g. [7,14,11]) that has a close connection with multi-dimensional separation of concerns, or aspect-orientation [15]. A superposition step introduces augmentations to an existing design which can cross-cut any existing module boundaries. There are different forms of superposition which differ on what restrictions are placed on the augmentations (see, e.g. [14]).

We apply superposition as follows. Each superposition step introduces a set of instance variables allocated to some classes, and related joint actions that may alter the values of variables. The newly introduced variables can be changed in completely new actions and in existing actions that have been augmented with assignments to the new variables. New assignments to previously introduced variables cannot be added. This scheme enforces the preservation of *safety* properties, i.e., properties of the form "something bad will never happen", expressible in the temporal logic used in conjunction with the methodology. Figure 2 provides a simple example of a design step applying superposition. In the figure, actions are denoted with ellipses, objects with rectangles, and superposition with dotted arrows.

2.5 Composing a Specification Architecture

On the one hand, an object-oriented design consists of classes, their relations, and related properties, reflecting the implementation mapping to a final executable. On the other hand, the structure formed by superposition steps and compositions captures the different dimensions of concern in a modular fashion. In the sequel, the latter will be referred to as *specification architecture*.

When composing a specification architecture, it is very common to intro-

duce only one or two instance variables at a time. Further, there tends to be a lot of *branching*, i.e., situations where one specification is referred to by several independent superposition steps that extend the specification with focus placed on different concerns. Branches can be merged in such a way that the safety properties all the branches hold for the result. Despite its simplicity, the possibility to introduce branches in designs has turned out to be extremely versatile. Logically independent parts of the system can then be handled separately, even if their implementations are co-located at the implementation level.

When two or more branches start from a specification, the specification provides a common ground for action refinements in the different branches. Usually this situation emerges when the common ancestor specification is a fundamental abstraction of the system. The different branches provide additional details that are associated with the abstraction, but not with each other. In particular, one branch can introduce the basic object-oriented design, and the rest of the branches may define aspects. Relying on the common specification only decouples aspects from the low-level object design. Operations where coordination may be necessary are thus restricted to those that reflect a higher level of abstraction. Changes local to one branch of the specification cannot affect the other branches.

A completed specification architecture is implementation-independent, i.e., design decisions are still necessary for mapping the specification to an implementation. What can be identified from the specification, however, are the different higher-level operations that need to be in the implementation. This provides an enhanced basis for partitioning different functions of the system to aspects and objects. If the aspect and object designs are derived from the specification architecture, the contents of different objects and aspects are well-defined. Similarly, aspect interaction can be identified from specification architecture. In fact, as the primary mechanism of modularity is superposition, the designer is guided to take overlapping of aspects into account in the specification architecture. If some aspects share common properties, they need to be given in a common specification. Then, different branches can be given that refine the common specification into two different aspects.

Ideally, the designer needs not consider different implementation techniques when composing a specification. However, practice has shown that some iteration is often needed for determining a convenient specification architecture. Further, like in any specification method, an anticipation of the eventual implementation technology may be helpful for an efficient implementation, i.e., a specification may give hints regarding the best possible implementation technique. For instance, if an aspect heavily influences the behavior of an object, problems may be inevitable in the long run. In addition to experiments using aspects and objects, we have previously gained similar experiences in joint action based specifications involving software and hardware elements [21,19].

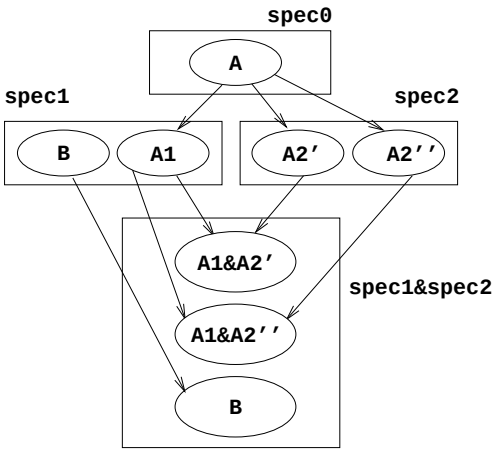

Fig. 3. Actions in the composition of specifications *spec*1 and *spec*2.

3 Example: Simple Switch

In this section, we give a specification and sketch an AspectJ implementation of a simple telephone switch. The focus of the section is on coordination of aspects and objects at a high level of abstraction, and on the architectural style a completed specification encourages for AspectJ programs.

3.1 DisCo Method

DisCo [26,11], based on theory of joint actions, is an experimental formal specification method for reactive and distributed systems. The method is based on superposition, as described in the previous section. When giving specifications, superposition can be applied as already described, but composition of different design branches requires more attention.

Composition of different design branches takes place as follows. If component specifications have a common ancestor in their refinement history, the common parts of component specifications only appear once. Classes and actions that have a common ancestor are merged. If new subclasses are introduced in component specifications merging becomes more complicated, but can still be automatic. Figure 3 depicts conjoining of actions in a composite specification. Action A is first introduced in specification *spec*0, then refined to $A1$ in specification *spec*1, and to $A2'$ and $A2''$ in *spec*2. Specification *spec*1 gives a new action B. In the composite specification *spec*1&*spec*2 action $A1$ must be merged with actions $A2'$ and $A2''$, because they share a common ancestor, A. Action B is taken as such to the composite specification.

The semantics of DisCo is given in the terms of Temporal Logic of Actions [18], enabling formal reasoning. Tools are available for analyzing DisCo specifications [1]. Syntactic and semantic correctness of specifications is verified by DisCo Compiler, which also takes care of automatic merging of classes and actions. The resulting specification can be animated with DisCo Animator, which shows objects and their internal variables on computer screen and lets

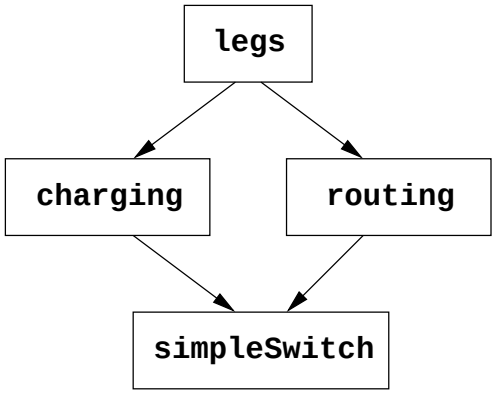

Fig. 4. Specification architecture.

a user execute actions.

3.2 DisCo Specification of the Switch

This subsection introduces a specification of a simple telephone switch. The discussion is structured in accordance to the specification architecture depicted in Figure 4. First, we give an abstract specification *legs*. Then, we add two different dimensions of concern (*routing* and *charging*) in parallel branches, and finally compose them into composite specification *simpleSwitch*.

Specification legs.
Concept *leg* is selected for a fundamental behavioral abstraction of the switch. Informally, a leg is a relation between two terminals modeling that the terminals are connected to each other. In the simplest case, the relation is between a caller and a callee, whereas more advanced call models like *call forwarding* allow several legs between terminals. Figure 5 depicts a case where a terminal A has called to B ($AB - leg$), but the call has been forwarded to terminal C ($BC - leg$).

Specification *legs* describes the behavior of the switch with classes $Terminal$ and Leg. The former has no attributes at this level of abstraction. The latter consists of two attributes a and b that model the two parties in the connection:

class $Terminal = \{\}$
class $Leg = \{a, b : \textbf{reference}\ Terminal\}$

Specification *legs* is the common high level specification for future specification branches. The specification introduces joint actions *connect* and *disconnect* that coordinate the creation and deletion of legs between terminals, respectively. Action *connect* synchronizes two terminals (a initiating the call and b modeling the terminal the call is being routed to) and a newly created leg l. Action *disconnect* simply deletes the participating leg object:

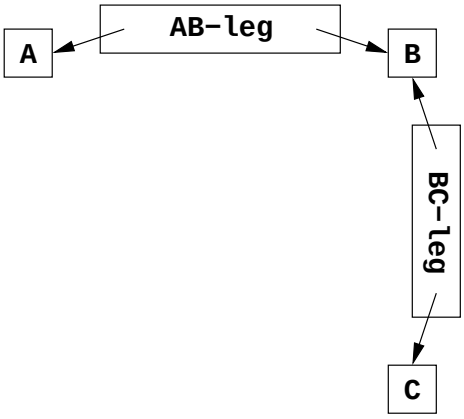

Fig. 5. Example: A has called B but the call has been forwarded to C.

connect$(a, b : Terminal;\ l :$ **new** $Leg)$ **is**
when *true* **do**
  $l.a := a \ ||\ l.b := b$
end

disconnect$(l : Leg)$ **is**
when *true* **do**
  delete l
end

Routing Dimension.
Specification *routing* defines how the actual connection between terminals is achieved in the switch. In the specification the abstraction *Leg* is implemented by processes *In* and *Out* controlling a connection to in-coming and out-going direction respectively. When routing, processes are started one after another, and they form a linked list. Figure 6 depicts the result of superimposing the processes on the specification in Figure 5, and indicates how processes $in1$ and $out1$ implement $AB - leg$, and $in2$ and $out2$ implement $BC - leg$.

Specification *routing* gives an abstract base class *Process* from which actual processes are derived:

abstract class $Process = \{$
  $b :$ **reference** $Terminal$
  $next,\ prev :$ **reference** $Process$
  $implementedLeg :$ **reference** $Leg\}$

class $In = Process\ +\ \{a :$ **reference** $Terminal\}$
class $Out = Process\ +\ \{\}$

Routing is modeled with actions *startInProcess*, *inStartOutProcess* and *outStartForwardingProcess*. Of these, *inStartOutProcess* is derived from *legs.connect*, and therefore it is combined with refinements of *legs.connect* from other branches of specification. The other two actions perform their

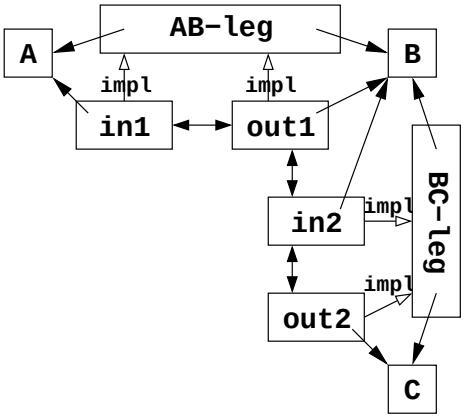

Fig. 6. Example: A has called B but the call has been forwarded to C.

obvious tasks, and can only affect variables introduced in the above classes. Therefore, we will only focus on action *inStartOutProcess*. The enabling condition of the action consists of the corresponding parts of the ancestor action plus two conjuncts, which bind the participant in role a and ensure that i is not yet linked. The modifications the action introduces to the state of the objects are the same as statements of *legs.connect* augmented with assignments to newly introduced variables. This is expressed as follows:

$inStartOut(a : Terminal;\ i : In;\ o :$ **new** $Out;$
$\qquad\qquad l :$ **new** $Leg)$
refines $legs.connect(a, i.b, l)$ **is**
when $a = i.a\ \wedge i.next = null$ **do**
$\quad o.b\ :=\ i.b\ ||\ i.next\ :=\ o\ ||$
$\quad o.implementedLeg\ :=\ l\ ||\cdots$
end

In addition to the above actions, the specification contains five joint actions (omitted here) to model tearing down a call.

Charging Dimension.
Charging is based on *leg*s. When a leg is created charging is started, and when a *leg* is terminated, charging is stopped. Specification *charging* does not specify any actual charging policy, but only shows how it could be added to the specification.

Singleton class $Charging$ is given to model the component controlling the billing:

class $Charging(1)\ =\ \{\}$

Actions *startCharging* and *stopCharging* are derived from actions *connect* and *disconnect* given in specification *legs*, respectively. The only thing the refinements do is the addition of a $Charging$ object to synchronize in the actions. For instance, action *startCharging* is given as follows:

startCharging(*a*, *b* : *Terminal*; *l* : *Leg*; *c* : *Charging*)
refines *legs.connect*(*a*, *b*, *l*) **is**
when *true* **do**
end

Composition of the Dimensions Charging and Routing.
Specification *simpleSwitch* composes the two dimensions of concern by importing specifications *charging* and *routing*. The composite specification is formed by merging the common parts of classes and merging actions that have a common ancestor in their refinement. For example, actions *routing.inStartOut* and *charging.startCharging* have both refined action *legs.connect*, therefore they must be merged. The signature of the created composite action *startO&C* contains roles of all component actions. The new enabling condition is the conjunction of enabling conditions of all component actions. Similarly, statements of the action are based on the assignments of the actions being composed. Thus, unfolded composite action *startO&C* of the example would given as follows:

startO&C(*a* : *Terminal*; *i* : *In*; *o* : **new** *Out*;
l : **new** *Leg*; *c* : *Charging*) **is**
when *true*
$\wedge\ a = i.a\ \wedge i.next = null$
$\wedge\ true$ **do**
$l.a := a \parallel l.b := b \parallel$
$o.b := i.b \parallel i.next := o \parallel$
$o.implementedLeg := l \parallel \cdots$
end

3.3 AspectJ Implementation

An AspectJ implementation was based on the described specification architecture. We considered *routing* as an active operation that has a fundamental role in implementing legs. Therefore, we decided to rely on an object implementation. This decision, in connection with the fact that the related specification is an expression of a specific dimension of concern, resulted in an aspect implementation for *charging*. The setting is depicted in Figure 7. As *simpleSwitch* is a composite of different design branches, it is a specification of an implementation produced by a Java compiler and an aspect weaver. In addition, we made design decisions regarding the allocation of responsibility for the execution of joint actions to methods in classes and advices in aspects. This process is not relevant for this paper and will only be addressed briefly.

The class structure of the routing implementation corresponds to the class structure given in the specification. Specification-derived classes *Terminal*, *Process*, *In*, *Out*, and *Leg* were introduced. When a call is initiated, a new *InProcess* managing terminal A end of a leg is formed. When routing reaches

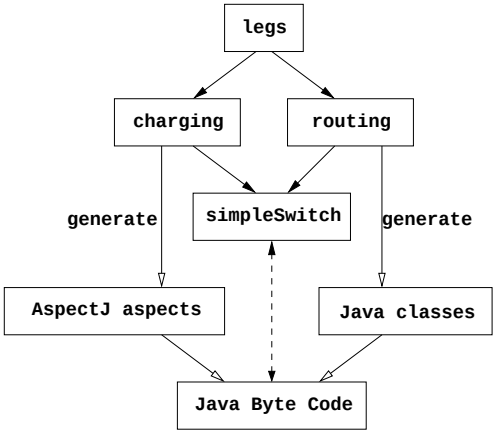

Fig. 7. Deriving implementation from specification.

terminal B, a new *OutProcess* is born, which implies an execution of the joint action *connect*. If terminal B has set up call forwarding to a terminal C, a new leg will be created as if terminal B had initiated a new call and the resulting *InProcess* is linked with the proceeding *OutProcess*.

Aspect implementation of the charging concern is straightforward. Only two advices are needed for starting and stopping charging for a leg. Points where the advices are to be woven can now be traced from the specification to the object implementation.

Similarly to the specification, the implementation uses *Leg* class as a contract between objects implementing it, and aspects interested in it. Instead of making join points in the charging aspect dependent on the implementation of routing object structure they can now be stated in terms of *Leg* class. Class *Leg* publishes methods *connect*(..) and *disconnect*(..) to enable the attachment of aspect-functionality.

As all data included in a *Leg* is redundant, any references to the internals of the class can be replaced with accesses to related *Process*es. Methods *connect*(..) and *disconnect*(..) contain no functionality, but they are still invoked as indicated by the high level design. As a result, all aspects that attach advice to pointcuts defined by these methods become independent of changes to the set of join points that make up the pointcuts. Class *Leg* could have been disposed of altogether, but in association with aspect technology its value as a specification-level join point collection becomes evident. Maintainability is improved, as multiple pointcut declarations do not need to be updated to reflect the change. This is visualized in Figure 8.

The selected approach maintains the independence of different specification branches in implementation. There is no need for aspects to define join points to the actual implementation constructs. Therefore, aspect implementations can be linked with object structures of which they need no detailed knowledge. While this exploits AspectJ for weaving aspect functionality in object code, AspectJ is used in a manner where join points serve a more declarative than

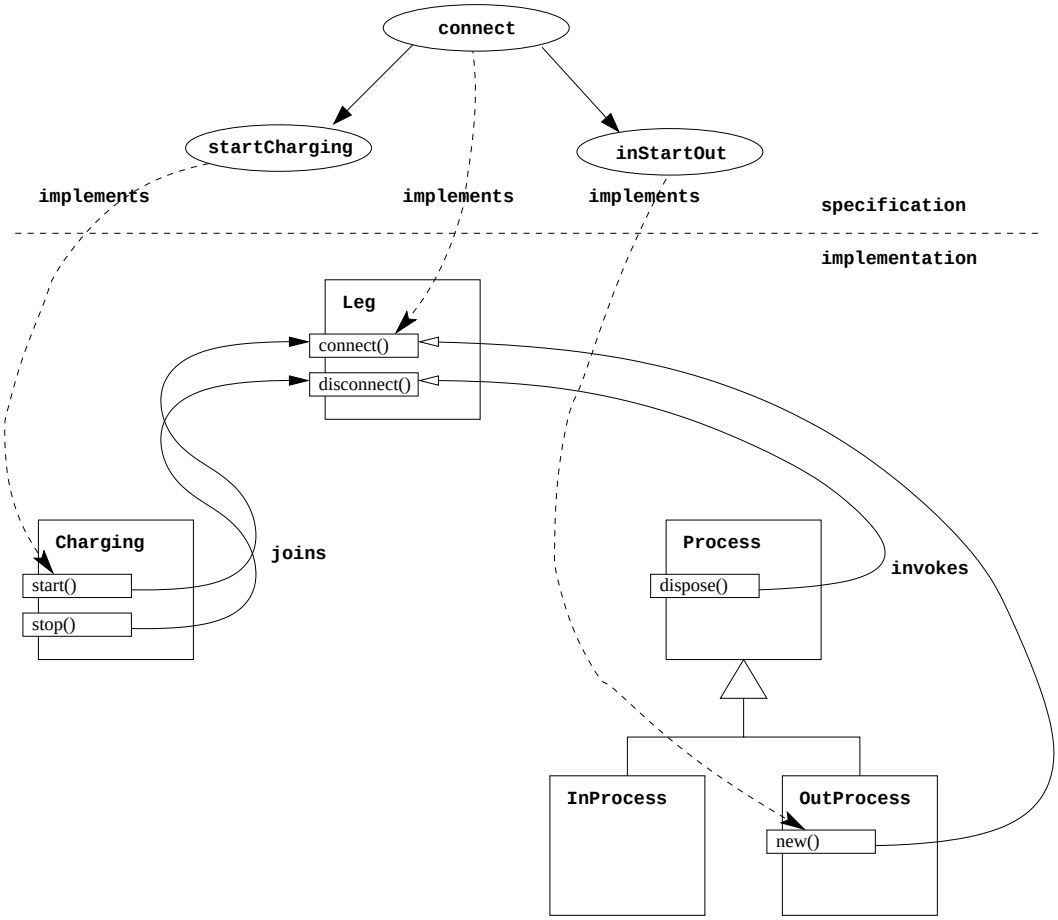

Fig. 8. Aspect joining advices at contracted points.

definitive purpose. By invoking a method derived from a common abstract specification coordination the aspects and objects, an object explicitly gives an opportunity for aspect code to be run.

In the above implementation, objects implementing the contract provide the actual places in control flow (join points) and clients of the contract are aspects delivering cross-cutting behavior. This places a constraint on the use of full expressive power of an aspect language for achieving more controlled, evolvable and maintainable implementation. The approach outlined above could be generalized as a language feature for declaring and managing interfaces between aspect and object code. Then, a compiler could optimize the architecture so that no execution overhead would result when invoking operations in class *Leg*.

We believe that the value of our approach lies in being able to address separation of concerns at the specification and design level, and in being able to coordinate where additional functionality may be inserted. This is distinct from lower-level development aspects (tracing, logging, profiling etc.) that can be dealt with using language facilities at the programming language level.

4 Related and Future Work

The foundations of multi-dimensional separation of concerns and its goals were laid in the context of software life cycle in [23]. That research has subsequently culminated to Hyperspaces [20,27], which is a more concrete model for achieving the goals. In this context, Aspect-Oriented Programming [17] with AspectJ [24] and Hyper/J [20] can been seen as concrete realizations of this technology at a code level using the Java language. The method described in this paper can also be seen as another realization at a design level, if we consider hyperslices to be formed from superposition steps. Hyperslices together with their relationships and rules for composing resulting specification branches would then correspond to hypermodules.

Recently, also other design methods supporting multi-dimensional separation of concerns have emerged, mainly as extensions of UML [22]. However, the interplay between different dimensions of concern at a high level of abstraction is usually neglected. For instance, Composition Patters [5,6] based on Subject-Oriented Design [4] provide mapping to AspectJ and Hyper/J. On the one hand, Subject-Oriented Design facilitates finding and managing the design subjects (concerns) of the system. One the other hand, Composition Patters define rules for composing (possibly overlapping) design subjects. In this paper we have placed special emphasis on coordinating the different subjects. Catalysis [8] aims at the design of reusable components in UML. Similarly to our approach, the focus is at a higher level of abstraction, and refinement is used as a vehicle for incremental design. The capturing of cross-cutting behavior takes place through joint actions. Our contribution takes a more formal approach to design and places a special emphasis on the specification architecture which makes explicit what the overlapping concerns are. Together with quite reasonable restrictions for superposition, this results in automatic composition, unlike in the previous approaches.

In [12] the joint actions combined with the superposition-based design were used for coordinating mobile agents moving in a two dimensional continuous space. Moreover, in [13] the relationship between DisCo and advanced separation of concerns were elaborated in the context of the behavioral specification of distributed real-time systems. Earlier, in [16], it was suggested to use the same ideas to support separation of concerns in the dimensions of product variance and domain concepts concerning software product lines.

5 Conclusions

In conventional software architectures a system is divided into components communicating using connectors. Usually, some support is provided for reasoning on concerns with system-wide effects. However, the code addressing such cross-cuttings concerns is scattered around the classes in object-oriented design and tangled with the code addressing other concerns inside the classes.

Aspect-oriented programming languages provide natural modularization also for code addressing cross-cutting concerns. However, the aspects have to be incorporated in the object-oriented design without support from a higher-level design. As the changes made for implementing cross-cutting concerns result in changes in the behavior of objects, the interplay between aspects and objects easily becomes troublesome. As an improvement, we have proposed an approach where aspects and objects rely on coordination provided by a common interface specification.

We have shown how to lift the level of abstraction to the joint behavior of objects and aspects. Instead of introducing these complex design decisions at code level, we suggest capturing object-aspect collaboration using joint actions at a higher level of abstraction. The resulting abstract specifications are then refined stepwise towards implementation. As a refinement mechanism we use superposition, which supports capturing overlapping dimensions of concern in the different branches of a specification. The branches can be composed automatically to form a specification of the total system. To illustrate our approach a simplified specification of a telecommunications application was given as an example. Moreover, an AspectJ implementation of the specification was outlined.

It has been suggested in [4] that a severe source of difficulty in building and maintaining software lies in the misalignment between requirements, design, and code. This calls for design methods supporting advanced separation of concerns. However, in order to have a real impact on simplifying software developed, these methods should based on theory rooted in a formal ground. Otherwise, there is a danger that instead of simplification, another layer of confusion is introduced on top of all unmastered complexity in building software.

Acknowledgement

Funding from Tampere Graduate School in Information Science and Engineering (TISE) and the Academy of Finland (project 100005) is gratefully acknowledged.

References

[1] Aaltonen, T., M. Katara and R. Pitkänen, *DisCo toolset – the new generation*, Journal of Universal Computer Science **7** (2001), pp. 3–18, URL: http://www.jucs.org.

[2] Back, R. J. R. and R. Kurki-Suonio, *Distributed cooperation with action systems*, ACM Transactions on Programming Languages and Systems **10** (1988), pp. 513–554.

[3] Back, R. J. R. and R. Kurki-Suonio, *Decentralization of process nets with centralized control*, Distributed Computing **3** (1989), pp. 73–87.

[4] Clarke, S., W. Harrison, H. Ossher and P. Tarr, *Subject-oriented design: towards improved alignment of requirements, design, and code*, ACM SIGPLAN Notices **34** (1999), pp. 325–339.

[5] Clarke, S. and R. J. Walker, *Composition patterns: An approach to designing reusable aspects*, in: *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Canada, 2001, pp. 5–14.

[6] Clarke, S. and R. J. Walker, *Separating crosscutting concerns across the lifecycle: From composition patterns to AspectJ and Hyper/J*, Technical Report TCD-CS-2001-15, Department of Computer Science, Trinity College, Dublin, Ireland (2001).

[7] Dijkstra, E. W. and C. S. Scholten, *Termination detection for diffusing computations*, Information Processing Letters **11** (1980), pp. 1–4.

[8] D'Souza, D. F. and A. C. Wills, "Objects, Components and Frameworks with UML: The Catalysis Approach," Addison-Wesley, 1998, 816 pp.

[9] Elrad, T., R. E. Filman and A. Bader, *Aspect-oriented programming*, Communications of the ACM **44** (2001), pp. 29–32.

[10] Gosling, J., B. Joy, G. Steele and G. Brancha, "$Java^{TM}$ Language Specification," Addison-Wesley, 2000, second edition.

[11] Järvinen, H.-M., R. Kurki-Suonio, M. Sakkinen and K. Systä, *Object-oriented specification of reactive systems*, in: *Proceedings of the 12th International Conference on Software Engineering* (1990), pp. 63–71.

[12] Katara, M., *Hybrid models for mobile computing*, in: A. Porto and G.-C. Roman, editors, *Coordination Languages and Models, 4th International Conference, COORDINATION 2000, Proceedings*, number 1906 in Lecture Notes in Computer Science, Springer–Verlag, 2000 pp. 216–231.

[13] Katara, M. and T. Mikkonen, *Aspect-oriented specification architectures for distributed real-time systems*, in: S. F. Andler, M. G. Hinchey and J. Offutt, editors, *Proceedings of the Seventh IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2001* (2001), pp. 180–190.

[14] Katz, S., *A superimposition control construct for distributed systems*, ACM Transactions on Programming Languages and Systems **15** (1993), pp. 337–356.

[15] Katz, S. and J. Gil, *Aspects and superimpositions*, Position paper in Aspect Oriented Programming workshop in ECOOP'99, Lisbon, Portugal (1999).

[16] Kellomäki, P. and T. Mikkonen, *Separating product variance and domain concepts in the specification of software product lines*, Position paper in ECOOP 2000 workshop on Aspects and Dimensions of Concerns, Sophia Antipolis and Cannes, France (2000).

[17] Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-oriented programming*, in: M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, Lecture Notes in Computer Science **1241**, Springer-Verlag, New York, NY, 1997 pp. 220–242.

[18] Lamport, L., *The temporal logic of actions*, ACM Transactions on Programming Languages and Systems **16** (1994), pp. 872–923.

[19] Mikkonen, T., *Codesign requires closed-system specifications*, in: H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), volume VI* (1999), pp. 2968–2974.

[20] Ossher, H. and P. Tarr, *Multi-dimensional separation of concerns and the Hyperspace approach*, in: M. Aksit, editor, *Software Architectures and Component Technology,* Kluwer, 2001.

[21] Pitkänen, R. and H. Klapuri, *Incremental cospecification using objects and joint actions*, in: H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), volume VI* (1999), pp. 2961–2967.

[22] Rumbaugh, J., I. Jacobson and G. Booch, "The Unified Modeling Language Reference Manual," Addison Wesley, 1998.

[23] Tarr, P., H. Ossher, W. Harrison and S. M. Sutton, Jr., *N degrees of separation: Multi-dimensional separation of concerns*, in: D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering* (1999), pp. 107–119.

[24] *AspectJ WWW site*, At `http://aspectj.org` on the World Wide Web.

[25] *Aspect-Oriented Software Development WWW site*, At `http://aosd.org` on the World Wide Web.

[26] *DisCo WWW site*, At `http://disco.cs.tut.fi` on the World Wide Web.

[27] *Hyperspace WWW site*, At `http://www.research.ibm.com/hyperspace/` on the World Wide Web.