Softw Syst Model
DOI 10.1007/s10270-006-0032-x

REGULAR PAPER

A concern architecture view for aspect-oriented software design

Mika Katara · Shmuel Katz

Received: 8 November 2005 / Revised: 7 March 2006 / Accepted: 21 August 2006

Abstract Although aspect-oriented programming is becoming popular, support for the independent description of aspect designs and for the incremental design of aspects themselves has been neglected. A conceptual framework for the design of aspects is presented, where aspects are viewed as augmentations that map an existing design into a new one with changes or additions. The principles of a Concern Architecture model are defined both to group designs of aspects, and to make explicit their dependencies and potential interferences in the design of a system with multiple aspects. The aspects are described generically, where any design element can be either *required* or *provided*. The required elements resemble formal parameters, and their binding to an existing design shows the context in which the provided parts are to modify that design. Overlap and a partial order among aspects and concerns are visualized in a Concern Architecture Diagram. An instantiation of the ideas as a UML profile is outlined, and the design of a digital sound recorder is used to demonstrate the utility of the approach.

Keywords Aspect orientation · Design concepts · UML

Communicated by Prof. Bernhard Rumpe and Mohamed Mancona Kandé.

M. Katara (✉)
Institute of Software Systems,
Tampere University of Technology, Tampere, Finland
e-mail: mika.katara@tut.fi

S. Katz
Department of Computer Science, The Technion, Haifa, Israel
e-mail: katz@cs.technion.ac.il

1 Introduction

Aspect-oriented programming (e.g., [33]) isolates code dealing with a concern of a system into modules. Without aspects, the code would cut across usual class or process hierarchies. In our view, on the design level an aspect provides *augmentations* to existing designs by mapping an existing design into a new one with more details or modifications. However, although some current design approaches enable a limited modular treatment of augmentations or modifications to existing artifacts, hardly any support is provided for the incremental design of aspects themselves, or for treating combinations of aspects that can either cooperate or interfere with one another.

As in conventional software architectures, which emphasize relationships among components constituting the software, the relationships among aspects of the system need to be made explicit. This is generally difficult because it cannot be assumed that aspects are always orthogonal, i.e., independent of each other. In order to do this, aspects will be grouped according to the names of *concerns* from the requirements definition activity of software development. Although the term 'aspect' is sometimes used at the Requirements, Design, and Programming stages, here we distinguish between *concerns* that originate as requirements of a system, and *aspects* at the Architectural and Design stages (while the implementation in programming languages is not treated here). A concern relates to any "conceptual matter of interest" and is simply a name indicating a grouping of requirements, whether from a separate Requirements Definition stage, or that arise later during system development. This name will be reused in the architecture description to group the

aspects treating that concern. On the other hand, an *aspect* is the design of a module to be implemented containing treatment of a concern.

Consider an aspect design for treating the concern of overflow of integer data values and another that encodes values as part of treating a security concern. These aspects can both involve the same methods or fields of an underlying object-oriented design, and may even overlap in the modifications applied. Such overlap in the aspect solutions arises even when the concerns at the requirements level seem disjoint. Such overlap between different aspects introduces a new type of problem, not seen in conventional languages, where it is clear to which class or module each language segment belongs. It is also a major source of complexity when composing and maintaining the aspects.

In this paper, based on an earlier conference paper [20], we define a *concern architecture* that groups aspect designs and can be seen as a software architecture *viewpoint*. Often, a viewpoint might correspond to a particular concern. Here, the organization of the aspects treating system concerns are themselves the focus of the viewpoint. In a concern architecture the aspects serve as building blocks that can be assembled incrementally to form more complex aspects from simpler ones. Each concern is addressed in a modular way by a collection of aspects, which can be composed when needed. The overlapping parts treating the different concerns are given explicitly by the aspects common to them.

Ideally, each aspect describes only a small increment. This allows on-demand creation or remodularization of aspects by composing different collections of aspects to match whatever concern is needed. In the case of overlapping concerns, less reconciliation is needed because the explicitly defined overlapping facilitates automatic composition.

In addition to the concern architecture description, a general approach to describing the designs of the aspects themselves is presented. The aspect designs are generic and reusable, and have *required* parts that correspond to assumptions about other entities with which they can be combined. Rules are shown for combining aspect designs according to the concern architecture.

One possible approach for aspect-oriented design is to introduce new constructs to model specific Aspect-Oriented Programming constructs. For example, aspects and join points as defined by AspectJ [33], the most popular aspect-oriented programming language, could be modeled. However, in our opinion, the mechanisms needed at the implementation level and those appropriate for the design levels are different. In particular, at the design level there should be no need for a reflective join point mechanism to capture the behavior of the base program that triggers the aspect code. Instead, we advocate an approach where aspects are seen as mappings between original and augmented design artifacts. Just as programming level pointcuts describe where advice should be activated and also expose context to be used by the advice, on the design level, (as will be shown) any element of the modeling language can be used to define the context and applicability conditions of an aspect. This decouples the design from intricacies of a particular programming language and even allows implementing the system in a conventional non-aspect language if needed.

There are basically two main categories of aspect-oriented design and programming languages [15]. Asymmetric approaches, particularly those influenced by AspectJ, make a separation between a conventionally structured base, e.g., classes with inheritance and other relations, and aspects that cut across its units. On the other hand, symmetric approaches, such as Hyper/J [26], consider systems to be composed of aspects, or slices, that potentially cut across each other. The approach here can either be used as a layer of aspects added on to a regular object-oriented design, or in a symmetric context where everything is an aspect. Examples of both types will be shown.

Because the concern architecture model is conceptual, it must be instantiated for some design language, similarly to the Hyperspace model [26,32]. A UML profile is sketched to illustrate the instantiation. The rest of the paper is structured as follows. In Sect. 2 the principles of the conceptual model are elaborated, showing how to model an aspect design, rules for combining such designs, and the concern architecture diagram for visualizing dependencies and overlap among aspects treating concerns. Sect. 3 discusses the UML instantiation of each of those elements. In Sect. 4 the approach is demonstrated with an example showing an embedded system design that includes feature interactions. Sect. 5 discusses the approach in the light of related work and Section 6 draws conclusions.

2 Principles of aspect modeling

The first principle of our approach, already discussed above, advocates integrating aspect modeling into existing modeling languages whenever possible. This is done by providing minimal syntactic extensions to model the aspects themselves, and expressing the relationships among aspects and concerns in a new diagram. Thus the expressive power for modeling aspects is inherited from the underlying language. Since languages vary in their semantics and mechanisms, some customization may be

needed. We will demonstrate the instantiation for UML in the following section.

As noted in the Introduction, we emphasize the distinction between concerns and software or other design artifacts implementing them (as suggested in [31]). Concerns are conceptual matters of interest from the requirements definition, such as data security. They can be treated by one or more aspects, perhaps using different techniques for different parts of the system, and modifying various values in distinct ways. We use the term *aspect* for a module that is potentially able to encapsulate software or design artifacts treating an otherwise cross-cutting concern. Aspects are often shared by more than one concern but can be composed when needed to form a composite aspect matching the treatment of a single concern.

2.1 Aspects and overlapping

Although current aspect-oriented approaches provide a modular way to augment or modify existing artifacts, there is only some preliminary support for designing aspects by composing simpler aspects [30]. However, aspects themselves can be complex, and must be adapted over time to changing requirements, and thus need to be designed incrementally. Moreover, collections of aspects have interrelations and cooperate to treat various concerns of the system, and these relations need to be explicit.

As will be demonstrated, the overlapping parts of different composed aspects may have been composed from aspects interesting in their own right. Moreover, such aspects may sometimes address concerns not initially identified but which become important during the software life cycle. This often justifies a remodularization where the common parts and the specialized parts are treated in separate aspects.

For example, consider again a system implemented in a conventional non-aspect language with two overlapping system-wide concerns: providing security of key variables and preventing overflow of variables. The security concern is addressed by encrypting and decrypting some values at sensitive points and the overflow by checking in advance that the needed data manipulations stay within a fixed range of values. However, the artifacts addressing these two concerns are closely intertwined, in that the encryption can assume values within those provided by the code implementing the overflow concern, and needs to provide decrypted values within those limits.

Initially there may be no separately identified concern dealing with either security or overflow. A remodularization could make the locations and tasks dealing with these concerns identifiable and separable from the rest of the system. When treatment of the two concerns is isolated into aspects to ease maintenance, it becomes clear that the aspects must overlap and cannot be completely orthogonal. Interest then turns to this overlap which again can be isolated into an aspect included in both security and overflow concerns. Identification of the new aspect can be vital if, for instance, the security scheme is changed and seamless co-operation with the overflow prevention must be guaranteed.

2.2 Composing aspects

Below we show in greater detail how to compose individual aspect designs into more complex designs.

2.2.1 Required, provided, and hidden parts

An aspect describes an increment to existing designs that potentially cuts across elements of the design, and formally defines a mapping from an existing design to an augmented one. For example, consider a design artifact in an object-oriented language, consisting of classes C and D. A simple aspect A might add classes E and F, subclasses of C, and introduce a method m to D. This in effect maps the artifact to an augmented one consisting of D including the method m, and C together with its subclasses E and F. Similarly, more cross-cutting and complex increments could be given.

As used here, aspects are inherently generic, i.e., they are parametric, and can be bound and instantiated multiple times. To support reuse, they are split into two disjoint parts, as also seen in [23]. The *required* part describes the "join points", to which the aspect is to be applied as well as their wider context, and can be seen as parametric elements, but of design artifacts rather than just variables, methods, or other program elements. The *provided* part introduces the augmentations. For instance, in the case of a generic version of the aspect A, above, the *required* part could consist of classes C and D, and the *provided* part of subclasses E and F, and method m. A could be instantiated for any artifact containing two classes, by binding C and D to the classes and renaming E, F, and m if needed to avoid name clashes.

Moreover, some elements provided by an aspect can be labeled as *hidden* from other aspects. This means that other aspects cannot bind elements to the hidden ones. In effect, *provided* elements not hidden from other aspects define an *interface* of the aspect. Similarly to interfaces in conventional languages, this permits internal modification of the *hidden* part without affecting the architecture composed of aspects. It should be noted that hiding an element also hides the elements inside its

"namespace". For instance, hiding a class hides all the methods and variables of the class.

Besides describing the points to attach new structure or behavior, the *required* part can be used to restrict the context in which the aspect is applicable. For instance, consider an aspect encapsulating an algorithm which detects termination of a computation in a distributed system, e.g. [8,21]. The *required* part can state the structural and behavioral assumptions about the underlying system which guarantee that the aspect is applicable, i.e., that termination can be detected. In the example above, because C and D are separate classes inside a single aspect, they cannot be bound to the same class. However, this would be possible if the classes were given in separate aspects. Note also that an aspect can be instantiated and utilized several times in one system, each time augmenting different classes or other design elements.

2.2.2 Regular binding

Composition of aspects is based on *superimposition*, which is an operation in which one aspect augments another one. A superimposition where aspect B augments A is denoted by B/A. Applying B/A to an underlying system S is equivalent to first applying A to S, and then applying B to the result. If there is no binding between the elements, then $B/A = A/B$. However, if some element in B is bound to an element in A, B may *depend* on A, thus breaking symmetry so that B may have to augment A, but not vice versa.

For example, if an aspect for monitoring a system defines new counter variables to record how often a variable is referenced, and an aspect treating overflow needs to consider *all* numerical variables, clearly the overflow aspect augments (and is applied after) the one for monitoring, even though this means that any additions in the aspect for treating overflow will not be monitored. On the other hand, if the monitoring aspect only prints values occasionally, and has no new variables, then the two aspects can be applied in either order. The independence of the aspects in this case is easy to see if, say, x is monitored, while y is treated for overflow. In fact, it holds even if both treat the same field, but then depends on the fact that monitoring does not modify the field.

Generally, when B augments A, the elements in the *required* part of B can be bound to the elements in the *required* and *provided* parts of A. This is called regular composition binding and is illustrated on the left side of Fig. 1. For instance, for Overflow/Monitoring, the *required* elements of Overflow include the numerical variables or methods to be treated for overflow. These can be bound to the elements of the aspect for Monitoring, including the new counters it provides, and the numerical variables which it may use from the underlying system to which it is applied. The *required* part of the composite aspect consists of the *required* part of A and unbound elements of the *required* part of B. The elements provided by B using the bound elements (e.g., methods inside bound classes), must be added to the corresponding elements in A. The unbound elements of B are added to A as such.

As defined above, a *required* element of an aspect can serve as an *abstraction* of a complex structure provided by some other aspect. In this case, the simple element is bound to the complex structure when the corresponding aspects are composed.

Whether bound or not, a *provided* element is an element in its own right, not necessarily a generic entity (for instance, a generic class that should be instantiated). However, from the binding point of view, *required* elements can be seen to serve the role of placeholders.

2.2.3 Replacement and unification binding

To provide greater flexibility and accommodate design maintenance, in addition to the regular binding described above, we allow a special type of binding among the elements *provided* by the aspects in B/A. An element a provided by A can be *replaced* by elements $b_1, .., b_n$ provided by B, if it is shown that the properties of A (presumably from its specification) that need to be preserved are indeed preserved. Note that this can also include replacing a single relation (e.g., an arrow representing a transition) by a substructure (e.g., an encoding algorithm). Using the binding to replace a provided element with a structure consisting of several elements corresponds to a sort of refinement. This is illustrated in the middle part of Fig. 1.

When an element name appears both in A and in B, or renaming is used to make previously distinct names identical, they are *unified* by the replacement binding. This is illustrated on the right side of Fig. 1. Renaming can also be used to prevent unifying elements that in A and B just happen to have the same name but are unrelated. Note that unification is simply a particular form of replacement, and is merely syntactic sugar added for convenience.

2.3 Concern architecture

In concern architectures the aspects are used as building blocks. Each concern is addressed in a modular way by a collection of aspects, partially ordered by the *depend* relation. As already mentioned, a binding between elements of two aspects indicates a potential dependency between the aspects. In fact, as was seen in the Overflow/

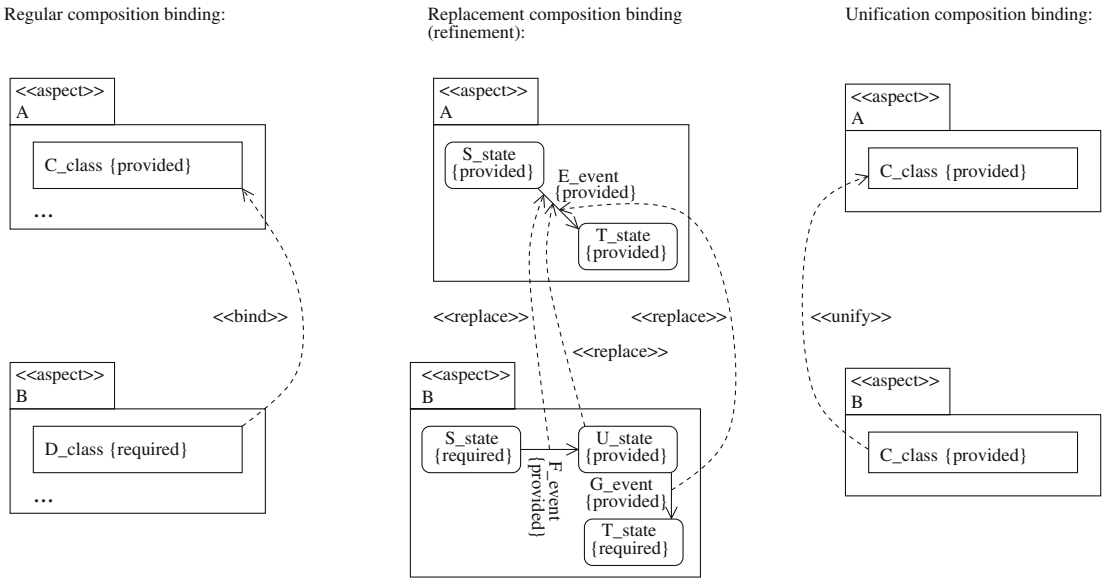


Fig. 1 Binding mechanisms illustrated

Monitoring example, if the aspects use the bound variable in complementary ways, there may not be a real conflict. However, here we conservatively define aspects as dependent if they cannot be shown independent. In addition, there might be some other reasons for a dependency. A more detailed analysis of possible restrictions on orderings can be found in [29].

To deal with a concern, the aspects within it can either be applied to a given underlying system one after the other in any order that satisfies the dependencies, or the aspects can be combined first among themselves into a complex aspect, that in turn is applied to an underlying system. Note that an aspect without any required elements is simply a regular design. Thus, if desired, any underlying system design can be viewed as an aspect, supporting the symmetric view that everything is an aspect. On the other hand, if a clear distinction between classes and aspects is natural for design and implementation, the concern architecture can be seen as a layer of aspects to be added to an existing object design.

Consider our earlier example with security and overflow concerns. In the absence of other concerns, the architecture could contain only three aspects: an aspect C with the parts addressing both concerns, and two others, S, with elements addressing exclusively security, and O, addressing overflow. The situation is illustrated in Fig. 2. For instance, if there is a common class with a method addressing security and another addressing overflow, the class itself would be *provided* in C and the methods addressing security and overflow in S and O, respectively. In the *required* parts of S and O there would be a class corresponding to the one *provided* in C. Other elements addressing both concerns would be *provided* in C.

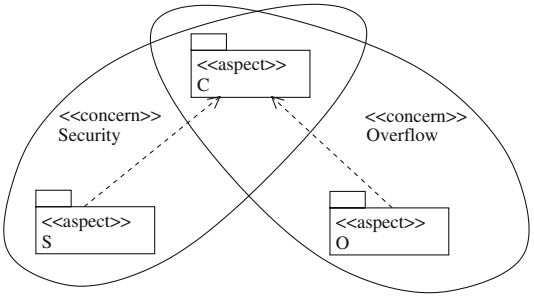


Fig. 2 Concern architecture

By treating the overlapping parts of the different concerns explicitly in the aspects common to them, it becomes easier to reason about the effects on one aspect caused by the changes in some other aspect of the same system. In the example, the treatment of the security and overflow concerns correspond to the collections $\{C, S\}$ and $\{C, O\}$, respectively.

The only restriction on the order in which the aspects are composed are the dependencies between them. In the example, the classes in the *required* parts of S and O must be bound to the one in the *provided* part of C. A design addressing both concerns is obtained by composing the aspects in either order, $S/O/C$ or $O/S/C$.

2.4 Aspects in the software development process

Although we believe that there is an optimal concern architecture for each system so that the common parts

of the different aspects are addressed explicitly and only once, it should be acknowledged that finding one may not be the first goal when developing the system from scratch. In fact, this is a reason for allowing unification binding, defined above, between elements in composition.

In iterative software processes, which have replaced traditional waterfall models, some aspects are designed incrementally from scratch while others involve reusing archived ones from previous projects. During early iterations some aspects treating different concerns may be overlapping in the sense that they provide common elements. However, in the course of the iteration, the common elements could be identified and moved into aspects shared by the concerns. The (changed) original aspects would then depend on the new aspects and require the moved elements. Naturally, whenever a new aspect is added to the system the parts of it *provided* by some existing aspects should be identified and the architecture remodularized accordingly.

Concerning the maintenance phase, if each aspect describes only a small detail of the system we could compose a collection of aspects to match whatever concern arose whenever needed. In this case the concern architecture would define a space for aspects created on demand by a remodularization.

2.5 Relations to software architecture description

In recent years the importance of multiple, concurrent "views" of software design and architecture has been recognized [6]. Various stakeholders at different stages of the software life cycle need to view the system from different perspectives. UML as well as the IEEE recommended practice for architectural descriptions (IEEE-Std-1471 [36]), documenting current architectural practices and research, are based on this paradigm.

In software architecture, concerns from the requirements stage are treated as first-class entities used to drive the conceptualization of an architecture. Concerns are first identified from the system stakeholders. These concerns form the basis for selecting architectural *viewpoints* with which to model the systems architecture. A viewpoint specifies the types of elements and relationships which can be used to describe a software architecture from a particular perspective. Each concern is allocated to one or more viewpoints and the resulting views address those concerns.

In our viewpoint, the concerns to be treated by aspects are considered. The concern architecture model can be seen as an aspect analysis viewpoint for analyzing the organization, overlap, and dependencies among aspects.

Obviously, it is not suggested to replace any previously existing viewpoints. In particular, it is valuable sometimes to view all effects of a method call, or of a class with the aspects and concerns woven in. Such slices avoid the problem of scattering parts of the effect of a system action that aspects can create while solving other scattering and tangling problems. If the organization of the aspects were our only viewpoint, the system design could also unnecessarily reflect the history of application of aspects, even when this is irrelevant to the present design.

2.6 Instantiation

Because the model described above is language independent, it has to be instantiated for some design language before it can be applied. More specifically, in order to support the model, the following well-formedness definition should hold for any design language L that defines well-formedness rules for artifacts given in that language:

Definition 1 (*Well-formedness of an aspect*) *An aspect is well formed if and only if the corresponding artifact in language L obtained by ignoring the required/provided/hidden tags is well formed.*

The definition implies that any ill-formed artifact in a particular language is also an ill-formed aspect. By basing the well-formedness definition for aspects on the well-formedness definition for the artifacts in the particular language, existing techniques and tools for checking artifacts in the language can be exploited.

Because composing two well-formed aspects does not necessarily result in a well-formed composite aspect, the following definition for composability of two aspects is given:

Definition 2 (*Composability*) *A well-formed aspect B can augment another well-formed aspect A with given bindings if and only if the resulting composite aspect B/A is well formed.*

Additionally, rules should be given to ensure that the binding preserves the structure of superimposed elements (similarly to [28]). For example, if a class C is bound to another class E, a method in C (either provided or required) can either be left unbound or be bound only to a method in E. Similarly, rules for conflict resolution could be given, for instance in the case where the base language does not support multiple inheritance. Such rules could be used to compose well-formed aspects that otherwise would not result in a well-formed composite aspect.

Obtaining greater automation in applying aspects often requires restricting their generality. For instance, the DisCo method [18,34] can be seen as an instance of the model. The composition operation, which is defined somewhat differently than here, is fully automatic. However, a *layer*, which corresponds to an aspect, can contain assignments only to variables introduced in the same layer, which limits incremental development of an aspect. The Open Module extension to OPM/Web seen in [28] also has elements of aspects.

3 Instantiation for UML

In the sequel, an instantiation of the concern architecture model for UML is sketched in the form of a UML profile. Profiles are UML's built-in extension mechanism consisting of stereotypes, tag definitions, and constraints. Compared to direct extension to the UML metamodel (for instance in [4]), profiles should preserve the integrity of the UML semantics by construction, and provide better interoperability between different extensions.[1]

Among other things, the instantiation introduces new syntactic elements to UML diagrams that are called an aspect and a concern. For brevity, we refer to the collection of aspects treating a concern simply as a concern, ignoring the underlying requirements that actually define the concern.

3.1 Views consisting of aspects and concerns

The fundamental decisions concerning a instantiation for UML are how to represent individual aspects (including how to denote *provided*, *hidden*, and *required* elements), how to represent collections of aspects treating concerns, and how to represent aspect and concern interrelationships. Following lines similar to [4], for individual aspects we introduce a new stereotype "≪aspect≫" of package, but with a new more general format for all UML diagram types rather than emphasizing the (most common) class diagrams. Packages are a general grouping mechanism and thus allow aspects to contain all diagram types. However, nesting of aspects should not be necessary because aspects can contain regular packages which can be nested.

A concern treated by a collection of aspects is modeled as a stereotyped package diagram, combining aspects. Unlike aspects, concerns can be nested. Instead of using the standard icon for packages, concerns are depicted as irregularly shaped encircling lines surrounding the corresponding collections of aspects. The dependency between aspects is denoted by the standard UML dependency relationship.

The interrelationships among aspects and concerns are captured by expressing multiple, overlapping concerns in the new *concern architecture diagram*. A diagram for concerns was originally proposed in [14], extending UML's component diagrams. By contrast, in our approach diagrams consist of aspects, dependencies between them, and collections of them matching concerns. For example, the concern architecture depicted in Fig. 2 illustrates the overlapping concerns in the case of the example system in Sect. 2.3. Furthermore, because of the overlapping, concerns use aspects but do not *own* them as model elements. Instead, the contents of a concern are always imported, meaning that there should be a repository common to all concerns that actually owns the contents.

Because for every concern name, there is always some aspect (usually composite) which treats the concern, we allow the clients or suppliers of dependencies in a concern architecture diagram to be concerns rather than aspects. So, if a client of a dependency is a concern, there is at least one aspect included in the concern that could be used as a client instead. Similar considerations hold for the supplier concerns. To avoid cluttering, the overall concern architecture of a non-trivial system is illustrated using several diagrams, which display only selected subsets of the aspects, dependencies and concerns.

Composition of aspects corresponds to "zooming-out" in a concern architecture diagram. When two aspects are composed they and any dependency between them are replaced by a composite aspect. A previous dependency from a non-component client to an aspect that is part of a composition is replaced by a dependency from the original client to the composite aspect, and similarly for a dependency from a component aspect to a non-component supplier.

3.2 Sequential structures

When composing sequential structures, e.g., sequence diagrams, the component sequences are considered to be partial orderings that the composite sequence must satisfy. Special care is needed to ensure that the result of composition is a well-formed structure and that the selected interleavings are consistent with the rest of the

[1] Our instantiation conforms to the UML standard version 1.5 [25]. In the forthcoming UML 2.0, the extension mechanism has been defined more formally. However, from the point of view of the instantiation, the changes should be merely syntactic. UML 2.0 also introduces new diagram types that should be addressed by the profile when defining a full instantiation, but this lies outside the scope of this paper.

Table 1 Stereotype definitions for aspects and concerns

Stereotype	Base class	Description	Constraints	Icon
≪aspect≫	Package	Aspect describes a potentially cross-cutting augmentation	Aspects cannot be nested	Package
≪concern≫	Package	Concern encapsulates the aspects addressing some conceptual matter of interest	Concern can only include imported concerns, aspects and dependency relationships between aspects	Encircling lines surrounding the imported aspects

Table 2 Stereotypes for bindings between elements of aspects in composition

Stereotype	Base class	Parent	Description	Constraints	Icon
Composition binding	Dependency	none	Binds elements between two aspects	Can only be used between elements of aspects	Dependency
Regular composition binding ≪bind≫	Dependency	Composition binding	In B/A, binds an element of B to an element in A	Client must be a *required* element of B	Dependency
Replacement composition binding ≪replace≫	Dependency	Composition binding	In B/A, binds an element of B to an element in A	Both client and supplier must be *provided* elements	Dependency
Unification composition binding ≪unify≫	Dependency	Replacement composition binding	Binds elements representing the same concept	See parent	Dependency

model. As already mentioned, existing analysis techniques can be utilized in detecting such cases. For example, in describing procedural flow of control between a non-recursive call/return pair, the caller is suspended after a call until control returns to it. In practice, this affects what orderings are possible in combining sequence diagrams, so that a suspended caller cannot receive a new call until the previous call has returned. Also, for instance, it is easy to produce sequences of "become flows" that are semantically inconsistent.

3.3 Stereotypes and tag definitions

As discussed above, aspects and concerns are modeled using stereotyped packages. The actual stereotype definitions are listed in Table 1.

Similarly, the binding between elements of aspects in composition utilizes stereotypes of the UML dependency relationship, as defined in Table 2. To accommodate the three kinds of bindings described in Sect. 2.2.2 and 2.2.3, we define a common parent stereotype "composition binding" of the dependency relationship. The stereotype is constrained so that such a dependency can only be used between elements inside aspects. The children of the parent stereotype are regular composition binding denoted by "≪bind≫" and replacement composition binding denoted by "≪replace≫". Furthermore, replacement is used as a parent stereotype for unification composition binding. If one element needs to be bound to a set of elements, this is modeled as a set of dependency arrows with a common client or supplier element.

Elements *required* by an aspect are distinguished from *provided* ones by a tag "{required}". All elements not tagged as "{required}", are assumed to be *provided*. However, a tag "{provided}" can be attached to a model element for legibility.

Additionally, a tag "{hidden}" is used for those *provided* elements that should not be included in the aspect's interface. To avoid cluttering, hiding an element implicitly hides also other elements not meaningful to remain in the interface, as discussed in Sect. 2.2.1.

Previously, we have used different colors to indicate *required* and *provided* elements [20]. Tags can only be associated with model elements so, multiplicities, for instance, cannot be marked as *required*. However, compared to colors, tags conform to the standard notation of UML and produce better results in black and white

printing. In practice, a CASE tool recognizing the tags can be used to highlight the corresponding model elements using colors.

3.4 Well-formedness rules

In addition to the constraints given in English prose in conjunction with the stereotype definitions, some of the most important well-formedness rules are stated formally below in OCL, a constraint language defined by the UML standard:

Concerns can only include aspects, concerns, and their dependencies:

```
context Package inv:
   self.isStereotyped('concern') implies
   self.contents->forAll(c |
      (c.oclIsTypeOf(Package) and (c.isStereotyped('aspect') or
         c.isStereotyped('concern'))) or
      c.oclIsTypeOf('Dependency'))
```

Concerns do not own elements:

```
context Package inv:
   self.isStereotyped('concern') implies
   self.allImportedElements = self.contents
```

The client of a regular binding dependency must be a *required* element:

```
context Dependency inv:
   self.isStereotyped('bind') implies
   self.client.taggedValue->exists( t | t.type.name = 'required')
```

Replacement binding must not involve *required* elements:

```
context Dependency inv:
  self.isStereotyped('replace') implies not
  (self.supplier.taggedValue->exists( t | t.type.name = 'required') or
     self.client.taggedValue->exists( t | t.type.name = 'required'))
```

4 Example

As an example of a concern architecture and the UML instantiation, a simplified design of a digital sound recorder is presented. The example has been adapted from [27], where the design of the system is shown in standard UML. The digital sound recorder is a small embedded device, a dictating machine, with a digital memory capable of recording, playing, and deleting messages. The device can also be used as an alarm clock. The hardware, described and encapsulated using wrapper classes, consists of six standard components: microphone, speaker, display, simple keyboard, timer, and battery.

The modularity of the original design is based on subsystems, where scattering and tangling of important concerns is evident. Thus we will decompose this system into aspects grouped into concerns. To show the ideas, in addition to concern architecture diagrams, we concentrate on class diagrams to describe structural extensions or changes, sequence diagrams for inter-object dynamic relations, and statecharts for internal object behavior.

As discussed above, we could view the system as composed of aspects only. However, we have chosen to model some parts in a regular object-oriented design to illustrate the flexibility of the approach. There are many ways to decompose the system and it is basically a design decision to choose which entities are modeled as aspects and which as objects. Moreover, some elements inside aspects could be placed in other aspects instead and extra *required* elements could be added to increase context sensitivity of an aspect.

4.1 Basic features

Considering the system bottom-up, each of the hardware components is modeled as a black box, i.e., as a class denoted as *required*, which is a wrapper for the hardware component, and a statechart (also denoted *required*) specifying its minimal expected behavior. In this way, the new design can facilitate hardware/software codesign by using aspects. The aspects we consider do not need further design of these components, which are naturally treated as classes. In a pure

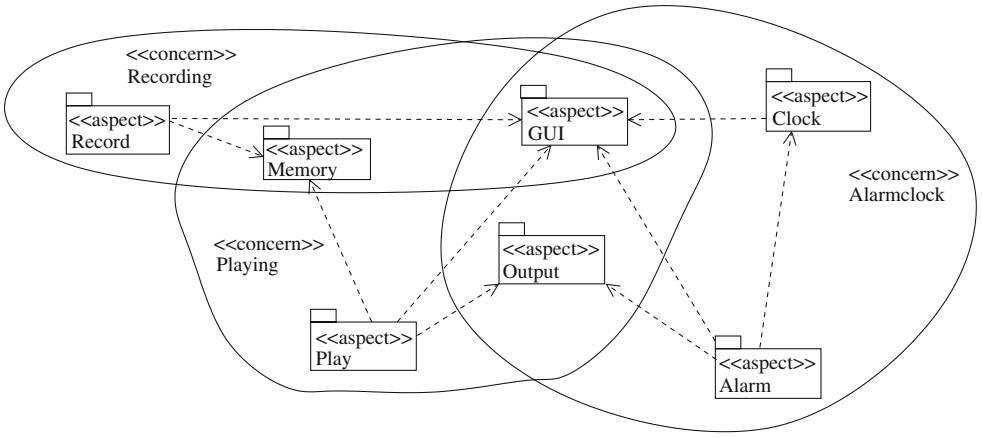


Fig. 3 Concern architecture: main features

symmetric approach they would be refined as aspects themselves.

Similarly to hardware components, some fundamental classes can be identified describing the low-level software, e.g., UserInterface, AudioController and AudioBlock. Each of these singleton classes is denoted as *required*.

Turning to a top-down analysis, the concerns of the system, from the requirements, can be divided into Recording, Playing, and Alarmclock functionality. We consider these concerns as collections of aspects, especially since they cut across several of the hardware or low-level software classes already identified. In Fig. 3, one possible concern architecture diagram consisting of the main features is depicted. In the diagram, the simplest aspects have been left out. The concerns are all overlapping because they all include the GUI aspect. Also, the Memory and Output aspects are both shared by two different concerns. The Recording and Playing concerns both include one aspect exclusive to the concern (Record and Play, respectively), while the Alarmclock concern includes two such aspects (Clock and Alarm).

Another kind of modularization in an asymmetric style would have been achieved by treating Recording, Playing, and Clock as basic functionality designed using plain object-orientation, with the needed parts of the GUI, Memory, and Output built into the appropriate classes. Then Alarm could have been seen as the only aspect cutting across many classes of the base design.

Now the details of the component aspects are considered. The main functionality of the graphical user interface is captured in one aspect, called GUI. The class diagram of the aspect in Fig. 4 requires a class UserInterface (which could be empty), a class Keyboard containing a public operation getLastKey and a class Display containing public operations on and off.[2] The aspect provides six new classes and several operations and relations. For instance, a public setUserMode operation is added to the UserInterface class. Additionally, behavioral diagrams could be given specifying the behavior of instances of both required and provided classes.

The common parts treating recording and playing are captured by a Memory aspect describing how messages are represented in the memory. The Memory aspect illustrated in Fig. 5 requires the AudioController class, and introduces three classes and four relations.

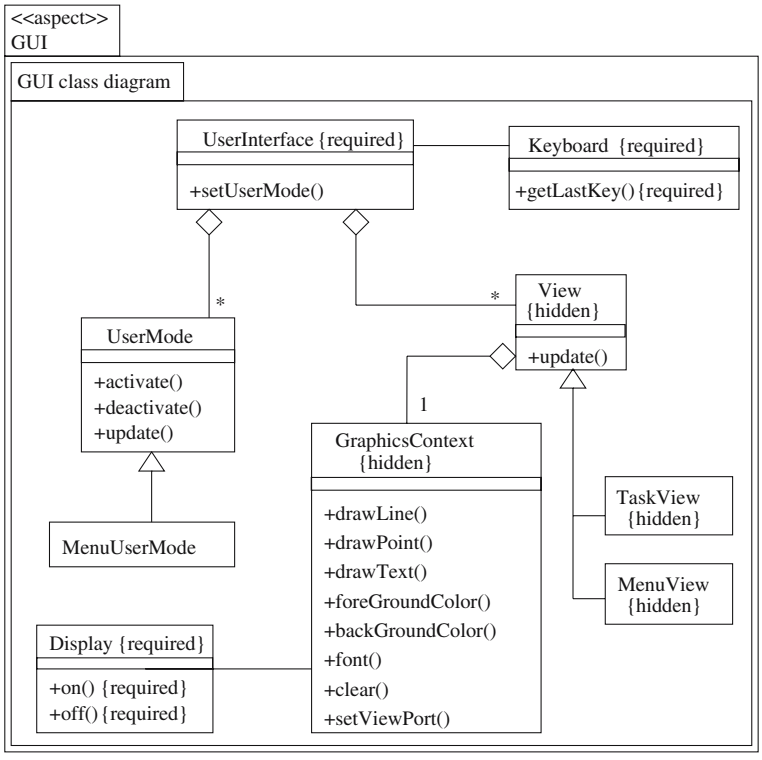
The Record aspect, in Fig. 6, provides the AudioInput class and adds new operations to the AudioController, Message (from the Memory aspect), and AudioBlock classes which are needed to implement the recording feature. The required association between the AudioController and Message classes serves as an abstraction of a more complex relationship provided in the Memory aspect. The aspect provides new relations between classes and adds a new state Record-on (as well as relevant transitions) to the statechart of class AudioController. Furthermore, the aspect introduces a new subclass RecordingUserMode for the UserMode class introduced in the GUI aspect. The sequence diagram included in the aspect illustrates recording of a message.

The Output aspect in Fig. 7 captures the common parts of playing and alarm clock features related to outputting sound. The aspect introduces the AudioOutput and Synthesizer classes.

The Play aspect, in Fig. 8, adds new operations to the AudioController, Message, and AudioBlock classes

[2] Technically, because visibility is not a model element, we cannot tag it as *required* in UML 1.5. However, in the composition of aspects, a regular binding between operations having different visibility would result in an ill-formed composite aspect.

Fig. 4 GUI aspect



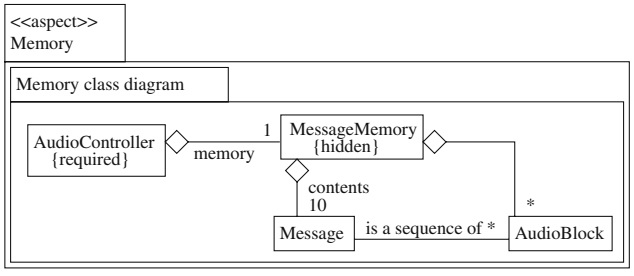


Fig. 5 Memory aspect

which are needed to implement the feature. It also provides a new state Play-on in the statechart of the AudioController class. Furthermore, the aspect introduces a new subclass PlayingUserMode for the class UserMode provided by the GUI aspect. Finally, the sequence diagram illustrates playing a message.

We have divided the design of the Alarmclock concern into two aspects, in addition to the GUI and the Output. The Clock aspect describes the clock using the hardware timer. The aspect is depicted in Fig. 9. It provides Time and Date classes to encapsulate the representation of time. Additionally, two new subclasses SettingTimeUserMode and SettingDateUserMode of UserMode are introduced for inputting the corresponding information.

The Alarm aspect (Fig. 10) requires the Time and Date classes introduced in the Clock aspect. Among other things, it provides the AlarmClock class with operations to set the alarm and to query the time, date, and state of the alarm.

4.2 Feature interactions

After depicting the aspects comprising the prime concerns of the system, the concerns and aspects defining the interactions of the concerns are given next. Because of the physical characteristics of the device, playing and recording features cannot interfere with each other. However, the alarm clock feature interferes with both of them. It must be decided what should happen if the alarm occurs while the user is playing or recording a message.

Each of the problematic feature combinations is resolved in a concern treated by a single new aspect in addition to the interacting concerns. This may often prove to be a valuable design strategy. In the case of playing and sounding the alarm, the alarm is given priority. This is indicated in the statechart of the AudioController class by adding a transition from state Play-on to state Alarm-on, triggered by a playAlarm event. However, we do not add a transition in the opposite direction, which means that the system does not automatically resume playing after an alarm. The aspect, depicted in Fig. 11, includes a sequence diagram illustrating the scenario when the alarm occurs while playing a message, while

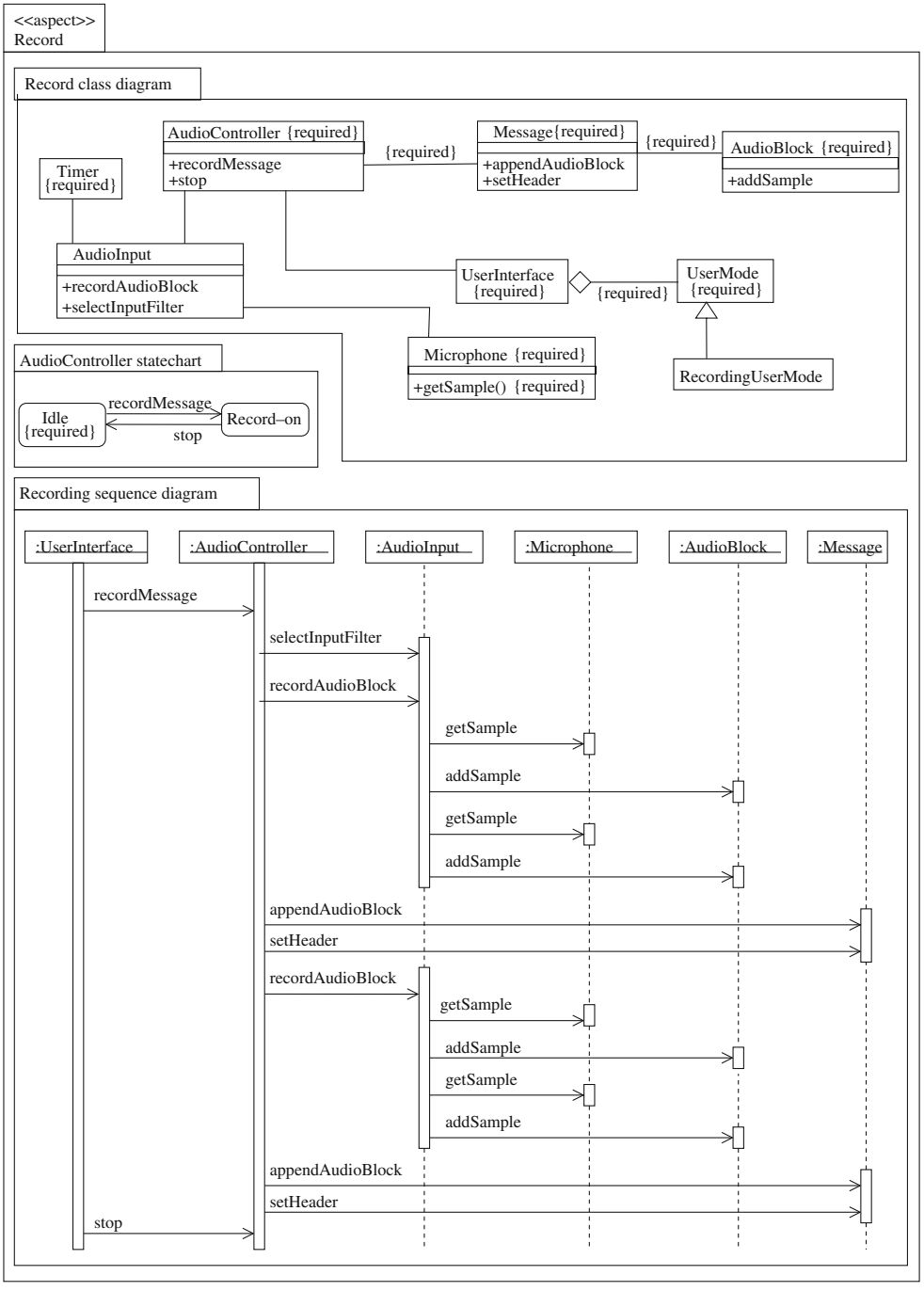


Fig. 6 Record aspect

a corresponding concern architecture diagram is shown in Fig. 12.

Similarly, an aspect (Fig. 13) and a concern (Fig. 14) are given to specify what should happen if the alarm occurs while recording a message. The situation here is somewhat different from alarm/playing interference. Because there are separate sound channels for inputting and outputting audio, it might be feasible to just let the alarm sound while recording. However, to ensure that a recording is not ruined because of the alarm, we have decided only to display the alarm indicator on the display and not to output sound. When the user stops recording, the AudioController goes to state Alarm-on and starts to play the alarm sound. (To stop the alarm, the user has to push the stop button again.) This behavior is modeled by adding two substates to the statechart of AudioController. Substate AlarmInRecord-on indicates displaying the alarm

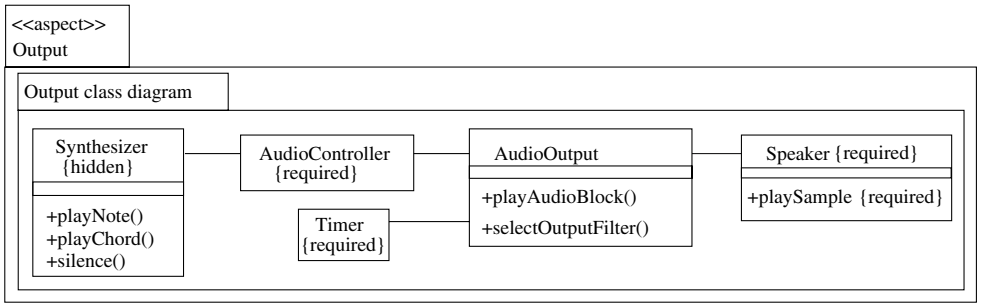


Fig. 7 Output aspect

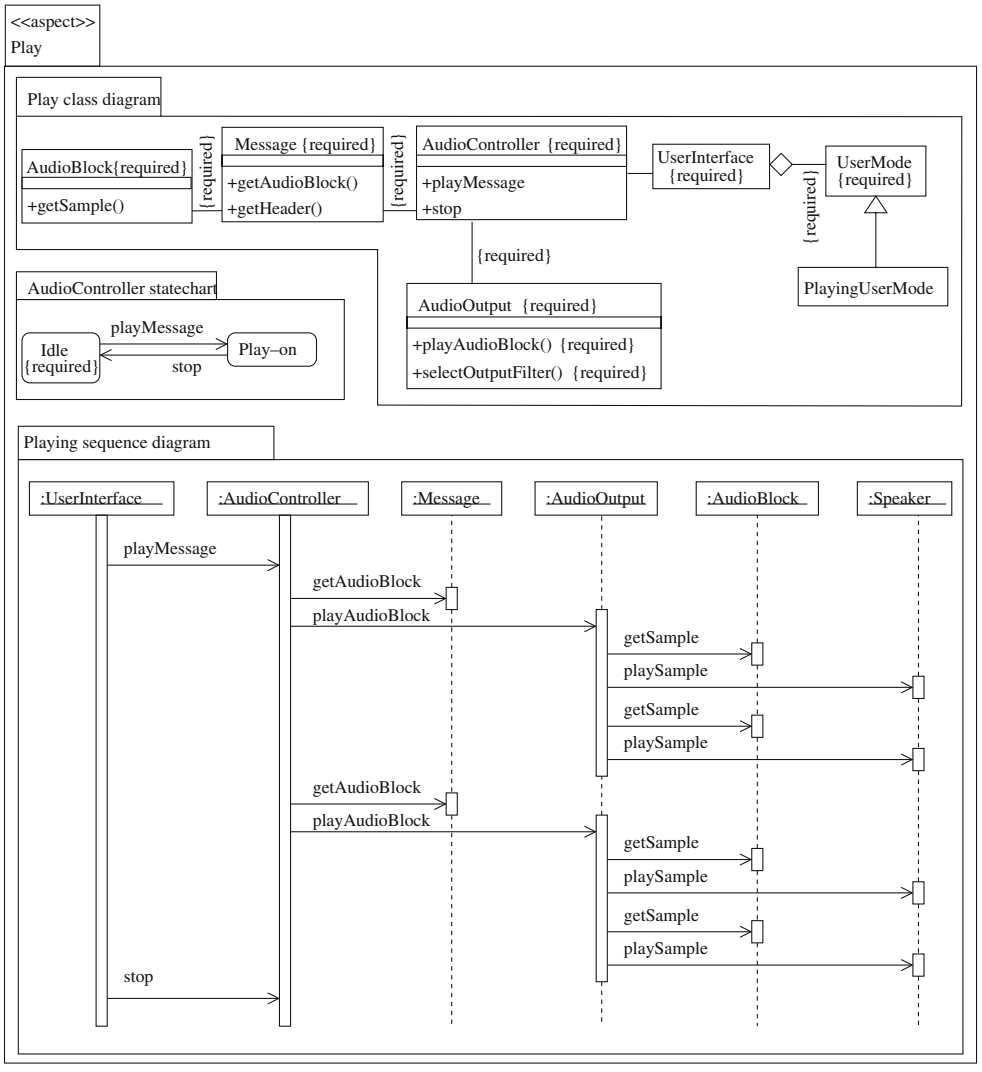


Fig. 8 Play aspect

indicator. If the alarm does not occur while recording, the stop event (of recording) triggers the transition to the Idle state (where the transition now has the new guard condition "IN Normal", denoted explicitly as *provided*). However, if the alarm has been activated, the stop event of recording (with the guard condition "IN AlarmInRecord-on") triggers the transition to the Alarm-on state.

The design of the complete system is obtained by composing all the aspects addressing the concerns. Figure 15 illustrates the composed AudioController statechart, and can be seen as part of a different

Fig. 9 Clock aspect

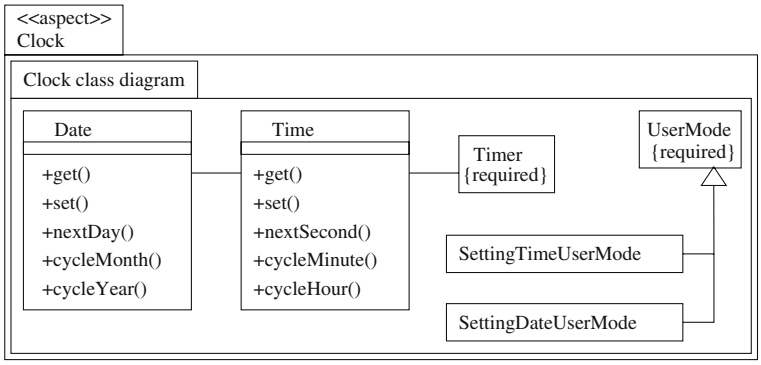


Fig. 10 Alarm aspect

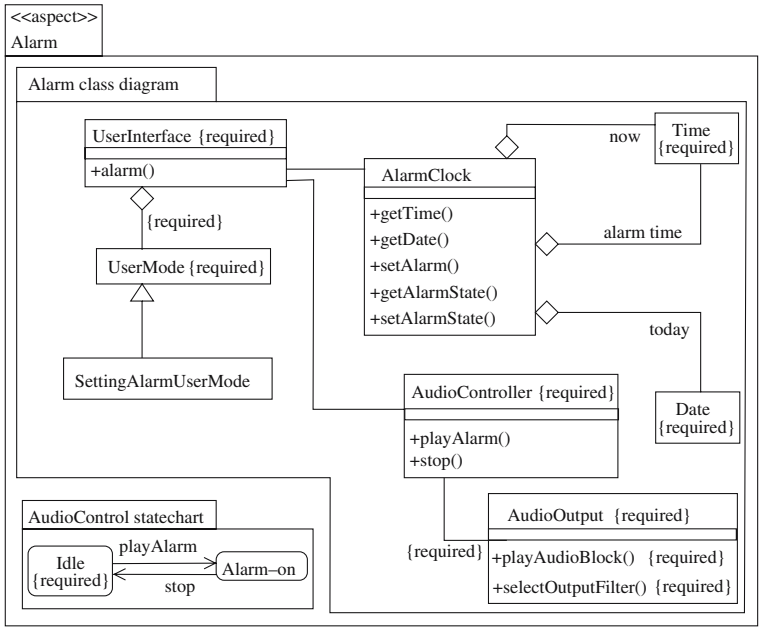
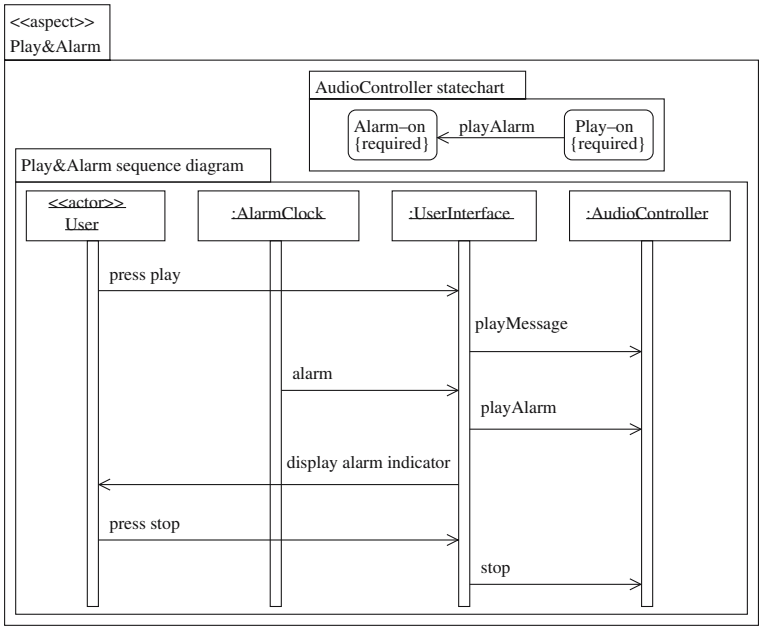


Fig. 11 Play & alarm aspect



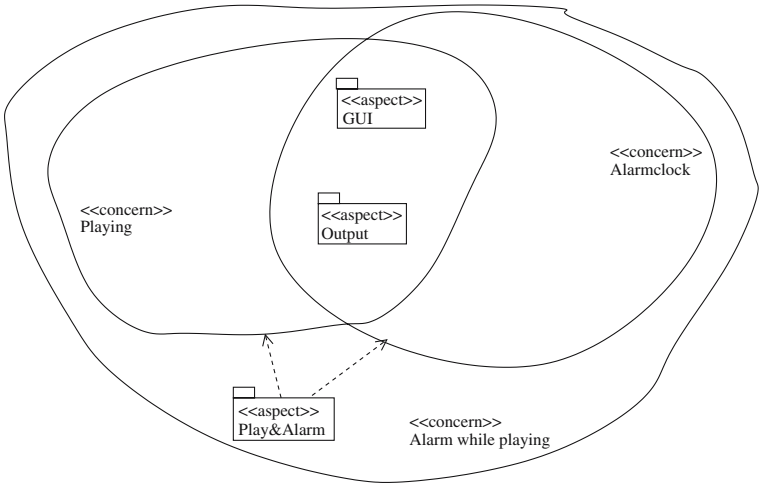


Fig. 12 Alarm while playing concern

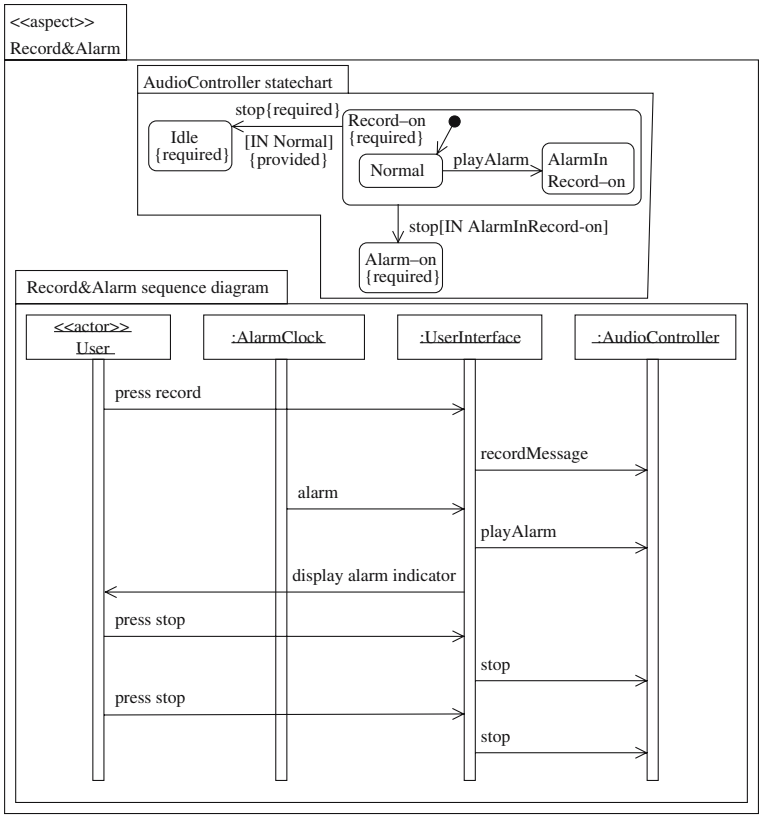


Fig. 13 Record & alarm aspect

viewpoint, described in regular UML. It contains all the states, transitions, triggering events, and guard conditions added in various aspects. If some element has been provided by more than one aspect, unification binding is used in their composition. This applies, for instance, to the stop operation, corresponding to the stop event, in class AudioController. Alternatively, the operation could have been introduced in a separate aspect and included in the *required* parts of the aspects defining recordMessage, playMessage, and playAlarm operations.

Additional observations about the example will be given in the conclusions (Sect. 6).

5 Related work

Recent years have witnessed emergence of a number of new approaches to aspect-oriented design and architecture. Below, we relate our work to the most relevant

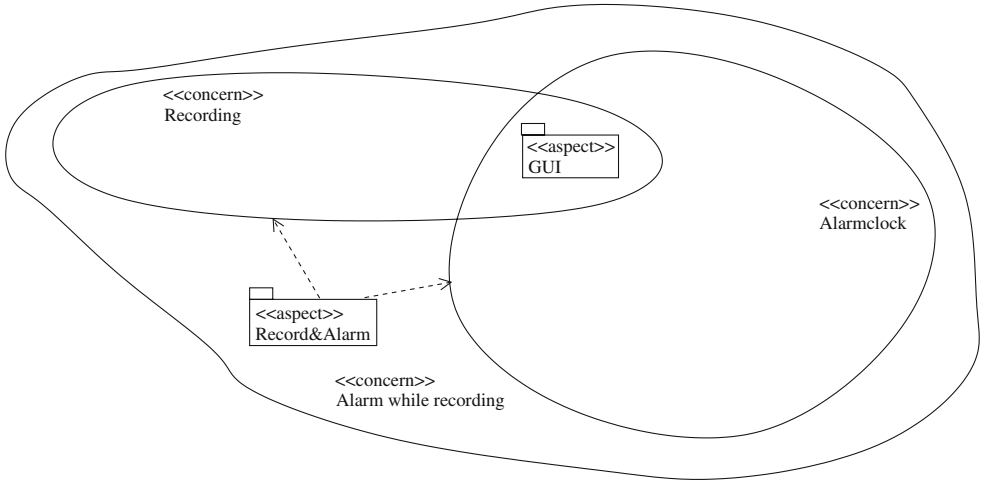
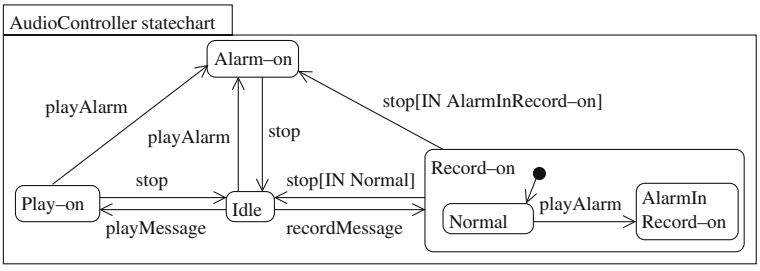


Fig. 14 Alarm while recording concern

Fig. 15 Composed AudioController statechart



and closely matching ones. Further analysis can be found from extensive surveys in [2,7].

The Hyperspace approach [32,26] has a similar goal of aspect orientation at a higher level of abstraction. In the Hyperspace conceptual model (where aspects correspond to hyperslices), slices can be composed (recursively) to hypermodules which contain composition rules for the component slices. Hyper/J [26] is an instantiation of the approach in the Java language. However, in contrast to our approach, that work does not define an architecture diagram that clarifies relationships among hyperslices in terms of the concerns treated, and does not clearly separate assumptions from provided elements.

The concern architecture model is based on ideas of superimposition [3,8,21] in distributed systems, which has a close connection with aspect orientation [22,29]. In particular, [29] describes somewhat related ideas of dependency and interference in the context of a programming language and proofs of properties. Superimposition, like other methodologies based on stepwise refinement, provides support for traceability of crosscutting concerns across the intermediate design descriptions obtained by applying the steps (aspects).

In addition to DisCo and OPM/Web, the main aspect-oriented development approach with explicitly defined aspect relationships is Aspect-Oriented Component Engineering (AOCE) [9,10] which supports aspect orientation throughout the life-cycle of software components. Aspects are used to describe systemic properties, like distribution, security and persistence, that components provide or require from other components [10]. The *provided* and *required* parts in our approach correspond to the provided/required services method in AOCE and grouping of aspects provided by AOCE can be used to capture concerns. However, instead of utilizing these techniques only for components, our approach defines a more generally applicable model which can be instantiated for different domains, and provides support for concern architectures.

Just as Theme [4,5] can be seen as an instance of the Hyperspace model, we have outlined an instantiation of our conceptual model for UML. Compared to template parameters used in Theme, the *required* elements approach is more expressive for describing the context in which aspects are applicable. Also, we do not define "composition patterns" as separate entities. Instead, behavioral inter-aspect relationships in composition can

be captured as aspects and new concerns in the design, as was demonstrated in the example. Furthermore, Theme does not provide architectural support, other than that of standard UML, for describing interactions and possible conflicts or cooperation among aspects.

Kandé [19] suggests a concern-oriented view of software architecture using multidimensional artifacts and encapsulation of concerns. Towards this end, an architectural framework called Perspectival concern-spaces (PCS) is defined. That framework can also be seen as a kind of instantiation of the Hyperspace model in UML, focusing on architecture and conceptual issues. Additionally, the approach conforms to the IEEE recommended practice for architectural descriptions (IEEE-Std-1471 [36]). That work attempts to cover a much wider scope of architecture and variety of viewpoints than our approach which elaborates one particular viewpoint built around a general purpose composition operation for aspects, based on superimposition. However, the applicability of our approach would similarly benefit from explicit compliance with IEEE-Std-1471, which remains as future work.

6 Conclusions

The concern architecture model provides an aspect-oriented perspective on software design. The model can also be seen as an aspect analysis viewpoint for analyzing impacts of changes or trade-offs in concerns to be addressed by aspects. Other viewpoints can be used to show all changes that will occur when a method is called, or everything within a class, with functionality added by aspects implanted in the various UML diagrams. This overcomes the common objection to aspect orientation, that it becomes difficult to track all changes from the various aspects. When, for example, all actions taken in reaction to a method call are of interest, aspects themselves cause scattering. But by changing the point of view to a relevant slice with all the aspects incorporated, the scattering is eliminated. Moreover, as demonstrated with the example in the UML context, aspects can contain several UML diagram types thus modularizing multiple views.

Using the concern architecture view it is easy to trace requirements in the design, because the collection of aspects addressing each concern is made visible, as was shown in the example. Moreover, to support reusability, the *required* part can express the context in which the aspect is applicable. For instance, in a symmetric approach, the GUI and Memory aspects could be easily reused in a totally different design.

To minimize change propagation, the overlapping parts of the different cross-cutting concerns are stated explicitly. Thus, if a cross-cutting concern needs to be changed, disjoint concerns should be unaffected. Moreover, the effects on the overlapping concerns can be examined using the concern architecture diagram. If the changes are made to an aspect that is not shared between concerns, for instance the Play aspect in the example, it reduces to the case with disjoint concerns. However, if a change is made to a shared aspect, for instance the GUI aspect in the example, the effects on the concerns sharing the aspect must be dealt with. In [1], change management utilizing a similar architecture in the DisCo setting is discussed in depth. Moreover, combined with patterns, the concern architecture approach has been used to document maintenance activities in [12].

"Mix-and-match" of features to suit different members of a software family is achieved by composing different collections of the aspects constituting the family (similarly to [24]). For instance, a lower-end version of the digital sound recorder, not including the alarm clock feature, can be obtained by excluding the Alarm-clock concern and any aspects unique to it (here, the Alarm and Clock aspects) from the composition of the aspects. Alternatively, a high-end product containing hierarchical memory supporting compressed messages could be modeled by replacing the Memory aspect with one describing the new design and conforming to the interface defined by the original aspect.

Furthermore, a concern architecture model enhances parallel development. It can be used to coordinate teams working on different concerns. Only the common aspects of overlapping concerns need to be decided jointly by the teams, and thus those aspects effectively define an interface between the concerns. As with conventional interfaces, it must be anticipated that some iteration is needed before the common aspects are established.

The aspects at the design level can and should be reflected in the programs, by using programming level constructs for aspects. Otherwise, most of the power of aspect orientation is lost. Towards this end, we envision mapping our design aspects to AspectJ (possibly with extensions such as superimposition steps of SuperJ [30]), or to modules of MixJuice [17].

Concerning tool support, the MADE tool [11,13] combines our approach with patterns to enhance the separation of concerns in pattern-based development. Because patterns are inherently cross-cutting, techniques and tools for instantiating them can be reused for aspects as well. In composition, the *required* elements are seen as pattern roles that can be bound to elements in a base design semi-automatically. The MADE tool

has been implemented on the Eclipse platform [35], and Rational Rose [16] is used for viewing composed UML diagrams.

In summary, aspect-oriented software development provides new possibilities for composing and decomposing software artifacts. However, while providing solutions for some fundamental issues concerning modularity, new types of problems are introduced, such as expressing aspects as design artifacts, detecting and describing dependencies among aspects, and treating overlapping concerns. In this paper we have addressed these problems at the design level through the concern architecture model. The applicability of the approach was illustrated with an example utilizing an instantiation outlined for UML.

References

1. Aaltonen, T., Mikkonen, T.: Managing software evolution with a formalized abstraction hierarchy. In: Proceedings of 8th IEEE International Conference on Engineering of Complex Computer Systems, Greenbelt, MD, USA. IEEE Computer Society Press (2002)
2. AOSD-Europe: Survey of aspect-oriented analysis and design approaches. Available at http://www.aosd-europe.net (2005)
3. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading (1988)
4. Clarke, S.: Extending standard UML with model composition semantics. Sci. Comput. Program. **44**(1), 71–100 (2002)
5. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design. The Theme Approach. Addison–Wesley, Reading (2005)
6. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, Reading (2002)
7. Cuesta, C.E., del Pilar Romay, M., de la Fuente, P., Barrio-Solórzano, M.: Architectural aspects of architectural aspects. In: Morrison, R., Oquendo, F., eds. Software Architecture – Proceedings of the 2nd European Workshop on Software Architecture, number 3527 in Lecture Notes in Computer Science, pp. 247–262. Springer, Berlin Heidelberg New York (2005)
8. Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Inf. Process. Lett. **11**(4), 1–4 (1980)
9. Grundy, J.: Multi-perspective specification, design and implementation of software components using aspects. Int. J. Soft. Eng. Knowl. Eng. **10**(6), 713–734 (2000)
10. Grundy, J., Patel, R.: Developing software components with the UML, Enterprise Java Beans and aspects. In: Proceedings of 2001 Australian Software Engineering Conference, pp. 127–136. IEEE Computer Society Press, Canberra (2001)
11. Hammouda, I., Hakala, M., Pussinen, M., Katara, M., Mikkonen, T.: Concern-based development of pattern systems. In: Morrison, R., Oquendo, F., (eds.) Software Architecture – Proceedings of the 2nd European Workshop on Software Architecture, number 3527 in Lecture Notes in Computer Science, pp. 113–129. Springer, Berlin Heidlberg New York (2005)
12. Hammouda, I., Harsu, M.: Documenting maintenance tasks via maintenance patterns. In: Proceedings of 8th European Conference on Software Maintenance and Reengineering. (2004) IEEE Computer Society Press, Tampere (2004)
13. Hammouda, I., Koskinen, J., Pussinen, M., Katara, M., Mikkonen, T.: Adaptable concern-based framework specialization in UML. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering. IEEE Computer Society Press (2004)
14. Harrison, W., Tarr, P., Ossher, H.: A position on considerations in UML design of aspects. Position paper in Workshop on Aspect-Oriented Modelling with UML in conjunction with AOSD 2002, Enschede, The Netherlands. Available at http://lglwww.epfl.ch/workshops/aosd-uml/ (2002)
15. Harrison, W., Ossher, H., Tarr, P.: Asymmerically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Thomas J. Watson Research Center (2002)
16. IBM Rational Software. Rational Rose WWW site: Available at http://www.ibm.com/software/rational/
17. Ichisugi, Y., Tanaka, A.: Difference-based modules: a class-independent module mechanism. In: ECOOP 2002 – Object-Oriented Programming 16th European Conference, vol. 2374 of Lecture Notes in Computer Science, pp. 62–88. Springer, Berlin Heidelberg New York (2002)
18. Järvinen, H.-M., Kurki-Suonio, R., Sakkinen, M., Systä, K.: Object-oriented specification of reactive systems. In: Proceedings of 12th International Conference on Software Engineering, pp. 63–71. IEEE Computer Society Press (1990)
19. Kandé, M.M.: A Concern-Oriented Approach to Software Architecture. PhD Thesis, École Polytechnique Fédérale de Lausanne (2003)
20. Katara, M., Katz, S.: Architectural views of aspects. In: Proceedings of 2nd International Conference on Aspect-Oriented Software Development, pp. 1–10, ACM Press, Boston (2003)
21. Katz, S.: A superimposition control construct for distributed systems. ACM Trans. Program. Lang. Syst. **15**(2), 337–356 (1993)
22. Katz, S., Gil, J.: Aspects and superimpositions. Position paper in ECOOP 1999 workshop on Aspect-Oriented Programming, Lisbon, Portugal (1999)
23. Kellomäki, P.: A structural embedding of Ocsid in PVS. In: Boulton R.J., Jackson, P.B. (eds.) Theorem Proving in Higher Order Logics, TPHOLS2001, vol. 2152 in Lecture Notes in Computer Science, pp. 281–296. Springer, Berlin Heidelberg New York (2001)
24. Kellomäki, P., Mikkonen, T.: 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)
25. Object Management Group: OMG Unified Modeling Language Specification, version 1.5. Available at http://www.omg.org (2003)
26. Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the Hyperspace approach. In: Akşit, M., (ed.) Software Architectures and Component Technology. Kluwer, Dordrecht (2001)
27. Paltor, I.P., Lilius, J.: Digital sound recorder: a case study on designing embedded systems using the UML notation. Technical Report 234, TUCS - Turku Centre for Computer Science. Available at http://www.tucs.fi/ (1999)
28. Reinhartz-Berger, I., Dori, D., Katz, S.: Open reuse of component designs in OPM/Web. In: Proceedings of COMPSAC 2002, pp. 19–24. IEEE Computer Society Press, Oxford (2002)
29. Sihman, M., Katz, S. A calculus of superimpositions for distributed systems. In: Proceedings of 1st International

Conference on Aspect-Oriented Software Development, pp. 28–40. ACM Press, Enschede, The Netherlands (2002)
30. Sihman, M., Katz, S.: Superimpositions and aspect-oriented programming. Comput. J. **46**(5), 529–541 (2003)
31. Sutton, S.M. Jr, Tarr, P.: Aspect-oriented design needs concern modeling. Position paper in AOSD 2002 workshop on aspect-oriented design, Enschede, The Netherlands (2002)
32. Tarr, P., Ossher, H., Harrison, W., M. Sutton, Jr, S. *N* degrees of separation: Multi-dimensional separation of concerns. In: Proceedings of 21st International Conference on Software Engineering, pp. 107–119. ACM Press, Los Angeles (1999)
33. The AspectJ Team. AspectJ WWW site: Available at http://www.eclipse.org/aspectj
34. The DisCo Project. DisCo WWW site: Available at http://disco.cs.tut.fi
35. The Eclipse Consortium. Eclipse WWW site: At http://www.eclipse.org on the World Wide Web
36. The Institute of Electrical and Electronics Engineers (IEEE): IEEE recommended practice for architectural description of software-intensive systems, IEEE-Std-1471-2000 (2000)

Author's biography

Mika Katara obtained M.Sc. (Eng.) and Doctor of Technology degrees from Tampere University of Technology (TUT), Finland in 1996 and 2001. In 2002 he visited the Computer Science Department of the Technion – Israel Institute of Technology as a Postdoctoral Fellow. He currently holds a Senior Researcher position at TUT where he is in charge of the software testing training, heads an industrial research project on model-based GUI testing, and organizes testing seminars for students and the local industry. Katara's current research interests include model-based testing and aspect-oriented software development.

Shmuel Katz received a B.A. from UCLA, and an M.Sc. and Ph.D. from the Weizmann Institute of Science (1976). He has been on the faculty of the Computer Science Department of the Technion – Israel Institute of Technology since 1981. He has written over 70 journal and conference papers on program verification, specification, and methodology. His research interests include aspect-oriented programming and software development, program verification, partial order reductions in verification, and translations among verification and modeling tools. He is the head of the Formal Methods Lab of the AOSD-Europe Network of Excellence, coordinating work on formal methods and semantics for aspects.