Making Aspects Natural: Events and Composition

Christoph Bockisch, Somayeh Malakuti, Mehmet Akşit
Software Engineering group, University of Twente, 7500 AE Enschede, The Netherlands
{c.m.bockisch,malakutis,m.aksit}@cs.utwente.nl

Shmuel Katz
Department of Computer Science, The Technion
Haifa 32000
Israel
katz@cs.technion.ac.il

ABSTRACT

Language extensions are proposed to make aspects more natural for programmers. The extensions involve two main elements: (1) Completely separating the identification of events and locally accumulating information about them from any possible response to the events, and (2) composing both events and aspects into hierarchies that loosen the connection to code-level methods and field names. The combination of these extensions are shown (in preliminary experiments) to increase modularity, and facilitate using terminology natural for each concern. Extensions to AspectJ and Compose* are shown to illustrate how the concepts can be easily embodied in particular languages. The execution model of ALIA4J is used to present the concepts in a language-independent way, providing a prototype generic implementation of the extensions, that can be used to implement them for both AspectJ and Compose*. The extensions increase the flexibility of aspects, encourage reuse, and allow expressing events and responses to them in terms natural to the concern that an aspect treats.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, packages*

General Terms

Languages, Design

Keywords

Aspect-oriented programming, event declarations and detectors, aspect and event composition

1. INTRODUCTION

Two main directions are presented here in order to make the use of aspects (sometimes also called concerns or aspect beans) more natural for programmers. First, we fully separate the identification of relevant events from the response to those events by adding *event declarations* that make it easy to accumulate information over time. Second, we provide support so that both events and aspects can be easily composed into more complex events and aspects.

Aspects express both *when* they are to be applied, and *what* they are to do, and there has always been some attempt to separate these two facets [15], e.g., with AspectJ [14] pointcuts versus advice code. However, this separation has been imperfect, and in many cases, aspect advice is used to gather and accumulate information needed to determine when an aspect should be applied, as well as to show what must be done as a result of application. This resulting mixing ("tangling" in aspect terminology) of event identification and responding to events makes aspects hard to understand, and complex to specify and verify.

What is more, aspect languages today—including AspectJ, Composition Filters [3], CaesarJ [2], etc.—are overly dependent on the structure and organization of the code. The programmer is forced to express concerns in terms of code-level events, usually method calls, that may be very far from the natural expression of when the aspect is relevant.

It also is difficult to combine aspects into more complex aspects. When multiple aspects are woven into a system, even without composing them into a new aspect, it is not clear how they should interact. Consider a logging aspect L, and an aspect for performance evaluation P that counts operations. When both are woven to a system (or when the two are composed), should L log the operations of P in addition to those of the underlying system? Should P count the operations that L performs in addition to counting underlying system operations? Or should they both be applied to an underlying system, without influencing each other? All of these are possible, and the correct composition depends on the intention of the programmer.

Our extension proposes reestablishing the desired separation between *when* and *what*—which exists already in the early conceptual work, but never completely made it into actual languages—by doing all detection of events separately from responses (that are left for the aspect construct). In addition to defining events using declarative predicates (as in AspectJ's pointcuts), we propose *event declarations* as an explicit language construct that resemble aspect code but do not have external side-effects. This separation is especially natural when alternative responses are possible to the same complex event.

AOSD'11, March 21–25, 2011, Pernambuco, Brazil.

For the composition of multiple aspects, we provide a powerful notation that allows expressing if and how the composed aspects influence each other at a fine granularity. The convenient composition of events and aspects, proposed in this paper, allows defining building-blocks of named, simple events and responses to them, and then using the names in defining a hierarchy of more abstract events and aspects.

A common example of the flexibility needed can be seen in potential discount policies of (online) stores, such as Amazon. A discount could involve a particular kind of product, a particular manufacturer, the amount of the total purchases made by a client over a given period, the geographic location of the purchaser, special holiday periods, and so on. There could either be conflicting or complementary discounts, or complex criteria for a single discount based on information and events from completely different parts of the system. Moreover, the discount policies change frequently, and are not a fundamental part of the underlying system.

The events triggering a discount could be the accumulation of a series of other events, or even the absence of other events, and often involve terminology far from the method names in the system. Thus, aspects seem to be appropriate for such an application, but expressing them is far from easy in present aspect languages.

As a more system-level application, consider an integrated development environment such as Eclipse, where aspects could be used over the Eclipse implementation code to monitor or enforce desired software engineering development practices [18]. Relevant events for the aspects might include, "creating a new class", or "committing a modified version of code to the repository". However, the relevant code-level events of Eclipse could be (and often are) internal method calls with non-intuitive names, reacting to clicking a "finish" button on a Wizard that has particular parameters, with information boxes previously filled in an appropriate way. As will also be demonstrated, sequences of events on several levels are the most natural way to express both events and responses, and this is difficult in existing aspect languages.

Yet a third, network application could be for run-time monitoring and verification of message passing systems. In that case, the flow of messages that are related, e.g., in a sequence diagram, needs to be followed, and when abnormal behavior is detected, recovery actions need to be initiated. The abnormal behavior and the recovery actions may involve the lack of an expected message, or an illegal sequence of messages that need to be aborted or, as the case may be, reproduced along a different path.

Thus, for server-side software, application-level systems like development environments, and even for system monitoring and runtime verification, separating events from responses makes understanding and reasoning easier. A means of composition to form a hierarchy of events and aspects, as proposed in this paper, is required to naturally implement all the above examples because the terminology used at the code-level may be very distinct from the terms natural to an aspect. Traditional code-level aspects often make it difficult to express and understand the intention of the developer.

Rather than suggesting yet another aspect language that would embody the ideas, we minimally extend existing aspect languages, so that the concepts can be integrated with the experience and programming paradigm natural for each language. The generic language extensions should allow the ideas to be incorporated into a variety of languages.

Our extensions will facilitate the creation of reusable libraries of aspects and events, and of aspect-oriented frameworks, as well as making aspects easier to use and verify. It will help alleviate the well-known fragile pointcut problem [27], by making most of the aspects independent of the code in the system to which they are to be applied. Programmers can then create hierarchies of aspects and of events, and compose aspects and events from a library, independently of any underlying code. Only at the basic level, where a connection with a particular underlying code is created, reference will be made to specific methods and fields of the underlying system. Moreover, besides the improvement in classic AOP applications, our extensions can be seen as making AOP appropriate for Complex Event Processing [16, 12].

In our view, a language design encourages modularity when key conceptual units (e.g., events and responses) can be easily expressed as syntactic units. It enables natural expression when the terminology appropriate for each concern can be easily expressed in the syntactic program units that treat that concern. To summarize the goals and contributions of our work:

- We present minimal language extensions for aspect-oriented languages to completely separate responses (that make externally visible changes to the system or environment) from event definitions that describe and signal when an event has occurred, accumulate information over time, and use events to define other events.
- We show examples and describe case studies demonstrating that the extensions increase potential for modularity and enable a natural expression of concerns.
- We demonstrate the programming style that the extensions facilitate, and show how complex events and aspect compositions can be expressed.
- We show how the extensions can be effectively implemented and used for static analysis.

In Section 7, we further explain these goals and present preliminary evidence that our extensions achieve them. Several other works suggest language constructs or methodologies to increase the separation between events and responses, as well as improving compositionality. A detailed comparison with these works is delayed until after our extensions are presented, and appears in Section 6.

In Section 2 we introduce the new concepts in general terms exemplified by a preview of the AspectJ extension. In Section 3, the AspectJ version is elaborated and an additional application is discussed. This is followed by a possible Compose* realization of the extensions, demonstrating that the extensions are relevant to different AOP languages and discussing a third example application. In Section 5 an intermediate execution model is introduced and used to provide a semantics and a prototype implementation for the generic extensions. This is followed by the related work and evaluation sections. We conclude with some further discussion in Section 8.

2. TERMS AND BASIC CONCEPTS

Below we explain the new language extensions for defining and composing events and for composing aspects, using

language-independent terminology. We also use the following well-known terms: A *join point* is an occurrence (an execution) of a primitive operation (where the primitive operations are assumed to have a known semantics). A *join-point model* for an aspect language defines which join points are *visible*, i.e., can be considered in defining events. A *join-point shadow* is an instruction in the code of the system whose execution corresponds to a join point.

Event Declarations.

First, we introduce *event declarations* to allow defining events in an imperative way. Such declarations resemble an aspect declaration, but have no external side-effects: They only can collect information in locally defined fields, and return control to the point at which they are activated. As will be discussed in Section 7, using static analyses, it is usually possible to check that all operations in an event declaration are free of external side-effects. In addition, event declarations have a list of *event parameters*, i.e., values or references that are exposed by the event. They also can have instances of a special operation called **trigger**() that indicates the detection of the event being defined. The names of event declarations are used in the event detectors defined below.

Event Detectors.

Event detectors are boolean expressions that match join points during the program execution. They may have parameters that expose relevant parts of the system state at matched join points. An event detector can be:

1. a primitive expression matching a join point on a low abstraction level (like the built-in pointcut expression primitives in AspectJ),
2. an identifier bound to an event declaration or to an event detector, or
3. a logical combination of event detectors.

The boolean value of an event detector is *true* at a join point either when the associated expression is *true* or the associated declaration executes **trigger**(), and is *false* otherwise. When it is *true*, we informally intend that the event has been detected, and the event parameters must have appropriate values or references to enable a response.

An event detector is, conceptually, evaluated at each visible join point, i.e., at each operation execution supported by the language's join-point model. Of course, optimizations are possible so that event detectors are in practice only evaluated at join-point shadows where they might be *true*.

Event Compositions.

Because event detectors are boolean expressions, they can easily be composed into new events by simply using logical operations such as conjunction or disjunction. Event declarations can also define new events as a sequence of simpler events, keeping track of the needed hierarchy.

Consider the e-commerce discounts example in the introduction. We here illustrate event detectors and declarations using the AspectJ version of our extensions. We can define the event RelevantPurchase(Purchase purchase) in terms of method calls, using regular pointcut notation, to detect whenever a relevant purchase is being made. Based on this, we can define new events such as LowActivity to detect if the number of purchases in a given period is too low:

```
1  event LowActivity(P product){
2      int LOWER_BOUND = 100;
3      Info purchaseInfo = new Info();
4      after(Purchase purchase): RelevantPurchase(purchase) {
5          purchaseInfo.increase(purchase.product());
6      }
7      when(P product): call(P.timeDone()) && target(product) {
8          if (purchaseInfo.count(product) < LOWER_BOUND) {
9              trigger(product);
10         }
11         purchaseInfo.reset(product);
12     }
13 }
```

Note that this event declaration accumulates information, and has a conditional trigger based on the execution history. It actually detects that certain events did not occur sufficiently frequently during a given time period. The event declaration LowActivityPurchase then can react to the LowActivity event by storing the products which have low activity in a local field of type Set, and identify subsequent purchases where the purchased product is in this set, as in the listing below. Presumably, it also reacts to another event by removing products from the lowActivityProducts set, but we do not include this here.

```
1  event LowActivityPurchase(C cart) {
2      Set<P> lowActivityProducts = new Set<P>();
3      after(P product): LowActivity(product) {
4          lowActivityProducts.add(product);
5      }
6      when(Purchase purchase): RelevantPurchase(purchase) {
7          if (lowActivityProducts.contains(purchase.product())) {
8              trigger(purchase.cart());
9          }
10     }
11 }
```

Basic Units and Aspect Declarations.

Usual AspectJ aspect declarations are composed of pairs of pointcuts and advice, while Compose* filters have selectors and typed operations. The pointcuts and selectors correspond to event detectors, and the advice/typed operations to *responses*, which are sequences of operations. An (event detector, response) pair, plus information on when to respond relative to the event (e.g., before, after) is called a *basic unit*. It can be given a name, to help control how aspects are composed.

An *aspect declaration* is an identifier (called the *aspect name*), local declarations of fields or methods, and basic units. It also may include compositions of other aspects, as will be described below. Thus an aspect LowActivityDiscount, in AspectJ syntax, could respond to the LowActivityPurchase event, as in:

```
1  aspect LowActivityDiscount {
2      before(C cart): LowActivityPurchase(cart) {
3          cart.applydiscount(10);
4      }
5  }
```

Aspect Compositions.

A composition of aspects is defined by listing the aspects in the composition, with the meaning that the aspect is composed of the union of the component basic units. To deal with composition options, modifiers are provided to possibly remove or restrict the applicability of some basic units

in the composition. The modifiers can affect applicability of basic units in two ways.

First, the scope of the event detector in a basic unit or those in an entire aspect may be defined in terms of the composed aspects. In particular, event detectors relevant for basic units in one component aspect might not be applied to another component aspect's basic units. We say that *A ignores B*, where A and B could be names of basic units or of entire aspects.

Second, for basic units with events that share a join point, we can control which responses are applicable when the events are detected, i.e., one can be preferred with another then excluded (*A overrides B*), or an order can be declared (*A precedes B*). It could even be specified that no response is to be executed when several are possible, but we have not yet found examples where this seems reasonable.

For illustration, assume a FrequentCustomerDiscount aspect that responds to a FrequentCustomerPurchase event detecting when a customer has made enough purchases to justify special treatment and is now making another purchase. When a situation that justifies discounts both due to the product having a low number of purchases, and due to a frequent customer, only the frequent-customer discount should be applied:

```
1 aspect Discount composes FrequentCustomerDiscount,
2         LowActivityDiscount {
3     local declare overriding FrequentCustomerDiscount,
4         LowActivityDiscount;
5 }
```

Instantiation Strategy.

In most languages, including AspectJ and Compose*, aspects are also types and instances of these types exist. A response is then executed in the context of such an instance which it can access, for example, using the **this** keyword. Typically, aspect languages allow defining a so-called *instantiation strategy* that defines when to create and when to reuse an aspect instance. In a composed aspect, the responses of basic units taken from the components must also be executed in the context of an aspect instance, expected to be of the type of the component aspect from which they originate.

Different approaches to create and reuse the aspect instances for composed aspects are conceivable and should be chosen according to the programming style of the concretely extended language. One possibility is (1) to compose the types of the component aspects into the type of the composed aspect. But this can lead to problems similar to those of multiple inheritance [11], depending on, e.g., the type system of the aspect language. As another approach, (2) objects can be composed instead of types; this means that the composed aspect maintains separate instances of the component aspects' types which are used as the context when executing a response originating from a component aspect.

Evaluation Order.

Regarding the evaluation strategy, given a collection of event detectors and declarations, and a collection of aspects, there are two basic operational semantics possible when a new system operation is reached that is a join point:

1. All of the event detectors are evaluated *once* to determine which of them is detected. Then all responses from basic units are executed for which the corresponding event has been detected, where any user-provided restrictions on the partial order of basic units are considered.

2. First the order of potential basic units is determined in advance, and their event detectors are evaluated *individually* just before deciding whether to immediately execute the relevant response. This potentially leads to evaluating the same event detector multiple times if it is used by more than one basic unit.

Both approaches have their own trade-offs. In the first semantics the result of an event detector is the same for each basic unit applicable at the same join point. This improves the analyzability of the program because it is known which responses are executed together at the same join point without having to consider the effect of preceding response executions on the evaluation of an event detector. And the fact that the event detector is *true* in the system state when the join point occurs can mean that the user's intention is to execute the response, no matter what other responses do in the meantime. If the user wanted to execute a response only if some other response did or did not occur at the join point being considered, that could be expressed using the extended composition notation. The difficulty with this approach is that a response may change the state so that a previously detected event would no longer be detected (or would be detected with different values or references bound to the event parameters) by the time the associated response is considered.

In the second semantics, the evaluation of event detectors can also consider changes in the program state that are induced by preceding response executions. Thus, a program may have different observable states at the same join point, allowing basic units to refer to the program state more accurately. The downside of this approach is the reduced analyzability but also the increased difficulty in developing event declarations: It must be considered that event declarations are potentially evaluated multiple times at the same join point. Consider the declaration of the LowActivity event which contains a **when** basic unit that conditionally detects the event and resets the internal state purchaseInfo in line 11; this is allowed by our definition because this side-effect is internal. Nevertheless, the result is that only the first evaluation of the event declaration at a join point will correctly refer to the accumulated execution history. In the given example, this would lead to undesired behavior, although there may be examples where such behavior is intended. In general, if the second semantics is applied, developers of event declarations must be careful with **when** basic units that have internal side-effects.

In the remainder of this paper, we adopt the first semantics because it yields more analyzable programs and because the semantics of event declarations with internal side effects are clearer. Nevertheless, in Section 5.1 we also outline an implementation approach for the second semantics.

3. AN ASPECTJ EXTENSION

In this section, we define extended AspectJ by (1) extending the pointcut notation to encompass more general event detectors and adding event-declaration entities, and (2) providing facilities for composing both event declarations

and aspects themselves to more complex ones in a hierarchy, while dealing with the semantic issues seen in the previous section. The extension presented here is only one possibility, where the new notation is intended to resemble similar features already in the language, and default compositions have been determined based on our subjective assessment of the most common situation.

3.1 Event Detectors and Declarations

Some examples of event detectors and declarations in extended AspectJ have already been shown. Here we consider the extensions in more complete terms. Any legal AspectJ pointcut expression is also a legal event detector in the extension, and an event detector can be used wherever a pointcut can appear in regular AspectJ. We now allow "global" named event-detectors, not directly included in any aspect. An event detector can also include a name bound to an event declaration. An event declaration differs from an aspect in the following ways:

- In an event declaration, the **event** keyword appears at the beginning instead of **aspect** and the event's name is followed by a parenthesized list of formal parameters (consisting of a type and a name) denoting the context exposed by the event.
- Basic units in event declarations cannot change non-local fields, or redirect control.
- An event declaration can contain special basic units beginning with the **when** keyword (instead of **before**, **after** or **around**) which may use the new, built-in operation **trigger**() to announce the identification of an event. The signature of the **trigger**() operation corresponds to the formal parameters defined for the event itself.

A **when** basic unit allows a response that identifies the event being declared with *the same join point as it responds to*. When the **trigger**() operation is executed in this response (which may conditionally depend on, e.g., local fields in the event) the named event is detected and the event parameters are made available. The choice of the word "when" is supposed to emphasize this behavior of the corresponding response, i.e., affecting the join point as a whole rather than having an effect at the beginning or ending of the join point.

Introducing the **when** keyword also has another benefit. It allows syntactically identifying basic units that can use **trigger**() in their response. This is similar to the **proceed**() operation which may only be used in **around** advice.

3.2 Aspects and Compositions

Turning to aspect definitions in extended AspectJ, the main change is that an aspect can now also be defined as a combination of other aspects. To do this, component aspects can be listed in the head of the aspect and composition modifiers can optionally be specified in the aspect body. To facilitate expressing the modifiers on a fine-grained level, a basic unit can have an optional name before the regular AspectJ **before**, **after**, or **around** keyword of a basic unit. If these names are not defined, modifiers can only be specified for all basic units in a component aspect at once.

In the definition of an aspect, the **composes** clause can be added to the aspect header (similarly to **extends** or **implements** in standard AspectJ) listing all component aspects. All basic units from the component aspects are included in the composition. Additionally, a composed aspect can also define new basic units.

Without further configuration, the execution order at join points shared by multiple basic units is unspecified. All basic units whose event-detector matches a join point are executed, and event detectors from basic units are applied to the execution of the underlying system as well as to all responses in the composition. This default behavior is identical to that of AspectJ when all component aspects are active individually.

The syntax to modify this default is in the style of AspectJ's inter-type declarations using the **declare** keyword. However, we add the keyword **local** before the **declare** keyword to emphasize that the modification is in the context of a composition. This can be followed by **precedence**, **overriding**, or **ignoring** keywords that relate two aspects appearing in the list of composed aspects.

With the **local declare precedence** statement precedence relations for all possible combinations of basic units in a pair of aspects are defined at once. This default relation can optionally be overruled for single pairs of basic units from these aspects using the **except** keyword. This keyword is followed by a list of basic-unit-name pairs defining that the precedence between these basic units is the reverse of the precedence defined between their containing aspects. This two-staged definition of relations between the basic units in component aspects allows to define relations at a fine granularity in a space-saving way.

The **local declare overriding** statements have a similar form. Aspect A overrides aspect B means that if basic units from A and B are jointly applicable at a join point, only the responses of units from A should execute. Exceptions can be defined using the **except** keyword as above to reverse the overriding. Additionally the **nooverride** keyword is followed by a list of basic-unit-name pairs which do not override each other: At join points where a pair in this list are both applicable, both their responses will be executed.

The modifier **local declare ignoring** has a similar form. When aspect A ignores aspect B, responses of basic units taken from A are not executed when the relevant event detector is *true* because of a join point in the scope of B. The keyword **noignore** followed by the name of a basic unit in A declares that this basic unit *does* apply to events caused by basic units of B.

Since event declarations are very similar to aspect declarations, we can compose event declarations in the same way as described above. However, event declarations can only be composed using the **composes** clause when they have matching event parameters, i.e., the same number and types of parameters in all components and in the composed event declaration. Otherwise, the parameters of the resultant composition are not well-defined. In general, we prefer to compose events using logical combinations of event-detector expressions, or using a **when** basic unit with a response containing a trigger operation to define the exposed context in terms of values and references exposed by the component events.

As mentioned in Section 2, aspects are types in AspectJ—in fact, the AspectJ compiler even creates Java classes for aspects—and response code can use the keyword **this** to access fields and methods of an instance of this type. The strategy for creating and reusing aspect instances at a join point is determined by the instantiation strategy defined in the aspect header. In our AspectJ extension we follow the

approach of composing instances, presented in Section 2, in a way that considers the instantiation strategy of the component aspects. That means at a join point where a component basic unit is executed, the strategy defined by the original aspect's instantiation strategy is used to create or reuse an instance of the original aspect class.

When an aspect A is in effect individually and another aspect C is in effect that composes A with other aspects, two equivalent responses may be executed at a join point and both expect to execute in the context of an instance of A. In extended AspectJ, aspect instances are managed separately for individually active aspects and for aspects active in a composition. Thus in the example, both responses are executed in the context of a different aspect instance, but the strategy of when to create and when to reuse an aspect instance is the same. Because we require that instances of the component aspects exist, they must be concrete.

In the AspectJ programming model, every aspect which is compiled is also in effect. Nevertheless, with the so-called *Load-Time Weaving*[1] supported by the standard AspectJ tool suite, a configuration file can be provided that defines which aspects are supposed to be active. This configuration file can be used to enable and disable composed aspects and their components individually.

3.3 Development Practices Example

Consider a hierarchy of events for monitoring whether desired development practices are being followed. In particular, we wish to detect whether a code file, previously checked out for local development, has been properly tested before being committed to the source control repository. The event-declaration code below uses the lower-level events characterized in the following. CodeModified occurs whenever the specified code file is changed. CommitBegun occurs when an attempt is begun to check the code back in. The event TestSucceeds occurs when a test suite (e.g., in JUnit) is executed and the test suite finishes successfully, while TestFails occurs when a test suite finishes but not all tests in it succeed.

The first event declaration in the example below uses these events to detect whether the latest activation of the test suite succeeded or failed, using a Set to record this state per code file. It defines the CommitWithFailingTest event as occurring when the commit dialog for a file is just about to begin where the last test run for that file failed. The second declaration simply defines TestRun as either of the events TestSucceeds or TestFails. The third event uses the newly defined one to define CommitWithoutTest as the event denoting an attempt to commit without having run the test suite at all since the last modification of the code, again using local Set fields to record past events to be checked when a commit dialog is attempted. Finally, the different kinds of errors are combined into a higher-level CommitViolation event.

```
event CommitWithFailingTest(File code) {
    Set<File> failedTests = new Set<File>();
    after(File code): TestFails(code) {
        failedTests.add(code);
    }
    after(File code): TestSucceeds(code) {
        failedTests.remove(code);
    }
    when(File code): CommitBegun(code) {
        if (failedTests.contains(code))
            trigger(code);
    }
}

event TestRun(File code) :
  TestSucceeds(code) || TestFails(code);

event CommitWithoutTest(File code) {
    Set<File> testedFiles = new Set<File>();
    after(File code): CodeModified(code) {
        testedFiles.remove(code);
    }
    after(File code): TestRun(code) {
        testedFiles.add(code);
    }
    when(File code): CommitBegun(code) {
        if (!testedFiles.contains(code))
            trigger(code);
    }
}

event CommitViolation(File code):
    CommitWithoutTest(code) || CommitWithFailingTest(code);
```

These (and other) events can then be used in aspects that react to them in various ways. Below are a few examples. The Logger aspect defines a basic unit with the side-effect of writing (to a persistent log) a message containing the file name for which an attempt to commit has been detected. The Proactive aspect runs the test suite when there is a violation. It proceeds to the commit dialog when the problem is solved, and otherwise cancels the commit, while announcing to the user that code modification and retesting is needed.

```
aspect Logger {
    Log myLog;
    before(File code): CommitBegun(code) {
        myLog.info("Attempt to commit " + code);
    }
}

aspect Proactive {
    void around(File code): CommitViolation(code) {
        boolean testSucceeded =
            code.testSuite.run(code);
        if (testSucceeded)
            proceed(code);
        else
            print("Correct and retest before commit");
    }
}
```

In the example so far, it is not specified in which order responses are executed at a CommitBegun join point which can ultimately also be a CommitViolation join point. Since it is desired to log any attempt to commit code, the Logger aspect should precede Proactive when they are used together. Furthermore assume that the Logger aspect contains more basic units logging different operations relevant to a development process, such as test-suite execution. If only user-triggered operations should be logged, operations executed by the Proactive aspect must be ignored.

Finally, consider an aspect Prevention that contains basic units that respond to any violation of a development practice by preventing the underlying operation from executing. When this is used together with the Proactive aspect, executing the basic units from the latter should be favored over the basic units from the former. The aspect DevelopmentPractices

[1] `http://www.eclipse.org/aspectj/doc/released/devguide/ltw-configuration.html`

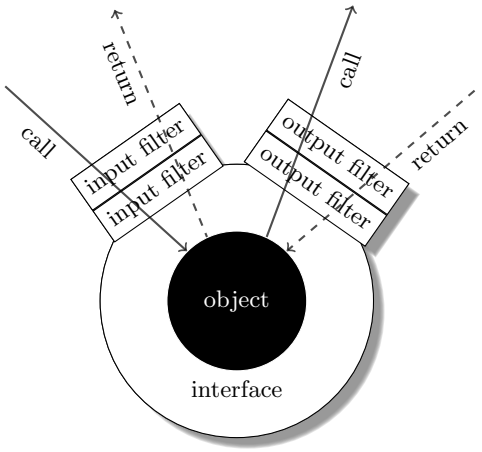

Figure 1: Diagram of an object with superimposed filters and how they participate in input and output messages.

shown below composes the three aspects Logger, Proactive, and Prevention into one aspect that fully specifies their relations.

```
1 aspect DevelopmentPractices
2       composes Logger, Proactive, Prevention {
3     local declare precedence Logger, Proactive;
4     local declare precedence Logger, Prevention;
5     local declare overriding Proactive, Prevention;
6     local declare ignoring Logger, Proactive;
7 }
```

4. EXTENSIONS TO COMPOSE*

Compose*[2] is a language and compiler for the Composition Filter Model (CFM). In the CFM, the messages that are exchanged between objects are processed by a group of filters superimposed on objects modifying their interface as illustrated in Figure 1. The messages and the objects that send or receive them form the basic elements of the Compose* join-point model. Two sorts of messages are distinguished: *input* (left-hand two arrows in the figure) and *output* (right-hand two arrows), which are, respectively, equivalent to the method invocations on and by an object. Each message has two flows, *call* and *return*, which represent points before and after the execution of methods for input messages, respectively, method invocations for output messages.

Filters define both when they are to be applied, and what they are to do. The former is expressed by specifying application messages that are accepted by the filter, while others are rejected. The functionality of a filter is implemented in the filter's type which processes the accepted and/or rejected messages. This implementation can, in general, also modify message properties, e.g. the message name, the target object, or the flow; the applicability of subsequent filters is determined considering the changed message properties. Developers can use built-in filter types and define new ones.

Filter modules group one or more filters and are the unit of superimposition, i.e., filter modules can be deployed individually. We extend Compose* with new constructs to facilitate the explicit declaration of events, separation of events from responses, and composing events and responses. In the following we provide a brief explanation of the extensions.

[2] `http://composestar.sf.net/`

4.1 Event and Response Declarations

In extended Compose* there are two new categories of filters called *events* and *responses*. In contrast to ordinary Compose* filter types, event types cannot modify message properties, so that external side-effects are prevented. Extended Compose* provides built-in event types, and also provides an API to define new dedicated event types. For example to compose events using temporal logic operators, a new event type must be defined that evaluates a temporal logic formula and triggers an event based on the result of evaluation. Custom event types are implemented in the same way as custom filter types but should additionally contain a *trigger* operation.

Each event declaration contains a set of *selectors* that specify events or application messages over which the event is declared. The selectors may be combined with boolean predicates to incorporate runtime values in the selection process. An event type may receive or expose information via its parameters. Responses are similar to ordinary Compose* filters, except that they can only select events.

4.2 Filter Modules and Compositions

As noted above, Compose* filters and other declarations are grouped in filter modules that are units of superimposition. Data fields can be declared within a filter module and the fields can be used by the events and filters within the filter module to maintain state information. Two types of data fields are distinguished: internals and externals. The internals are instantiated for each instance of the filter module, while externals are instantiated outside the filter module and can be shared among multiple filter module instances or application objects. Conditions can also be defined within filter modules, and they can be used in the declarations of events and filters to form more complex predicates in the selection of messages.

Filter modules are extended with event and response declaration blocks. To avoid external side-effects, events cannot modify the external data fields defined within filter modules. In our extension, a filter module can also be composed from other filter modules. In this case, the composite filter module can refer to the events declared within the component filter modules, for example, to define composed events or to define a response. The possible composition rules among the filter modules can be defined within the composite filter module.

4.3 Superimpositions

A *superimposition selector* chooses a set of classes using a query language and superimposes a specified filter module on their instances. Individual instances of a filter module are created for individual instances of classes on which the filter module is superimposed. As part of our extensions, it is possible to superimpose a filter module as a Singleton, so one single instance of the filter module will be shared among all instances of classes on which the filter module is superimposed.

In the superimposition of a composed filter module, the component filter modules are superimposed on the same classes as the composed one, unless specific superimpositions of the component filter modules are specified. At runtime

the composition relation is maintained between instances of composed and component filter modules.

When a filter module is superimposed on an object, all incoming/outgoing messages to/from the object are received by the event declaration blocks defined within the filter module, if any. If the message is selected by the event declaration block, the event type's action is executed to determine whether the event is detected. If so, the event is triggered. Composed events that select the triggered event are evaluated in the same way. After all events are evaluated, the responses defined for the events (i.e., filters that select the events) are executed.

If a composed filter module refers to an event defined within a component filter module, upon the detection of the event, all instances of the composed filter module are notified.

4.4 Runtime Verification Example

Runtime verification aims at monitoring the active execution trace of a program, checking it against the formally specified properties of the program, and possibly taking a recovery action if the properties are violated. In our terminology, monitors are mapped to event declarations, because they collect information about the state of a program over time. They detect events when the status of some condition becomes available, usually when a desired condition does not hold, and they must not have external side-effects on the program. One may argue that monitors may slow down the program, so they have side-effects on programs by influencing execution times. However, we only refer to side-effects on the program *state*. Recovery actions can change the state of the program, so they are mapped to responses.

To illustrate the influence of our extensions on the development of monitors and recovery strategies, in the following we provide an example and illustrate its implementation in extended Compose*. Assume that we want to log information about incorrect runtime accesses to a file, and to prevent such accesses. A correct access has two properties: (1) It is performed by an authenticated user; (2) it complies with the usage protocol defined for the file. If an access to a file does not satisfy the above properties, we would like to log its relevant information and prevent it. The filter modules declaring relevant events and filters are presented below.

The event representing un-authenticated accesses to a file is defined in the filter module AccessError. Here, we assume that the method isAuthenticated(), defined in the application class User, specifies whether the current user is authenticated. The attempts of an un-authenticated user to access a file is detected by the event eNotAuthenticated. The type BooleanDetector is a built-in event type in extended Compose*, which evaluates a boolean predicate expressed over application messages, and triggers the event if the result of evaluation is *true*. In our example, the event is triggered on the call flow of all input messages when the value of authenticated is *false*:

```
1 filtermodule AccessError {
2   conditions
3     authenticated: User.isAuthenticated();
4   inputmessages
5     eNotAuthenticated:BooleanDetector=
6      (!authenticated && selector ==[*.*]);
7 }
```

The incorrect accesses to a file are detected by the event eProtocolViolation defined in the filter module ProtocolError. The user-defined type RegularExpressionViolation receives a regular expression predicate as its input, and evaluates it against the selected messages. If the order of messages does not comply with the predicate, the violation is triggered as an event. A correct access to a file starts with the execution of the method open, followed by zero or more times read and/or write, and is finished by the execution of close:

```
1 filtermodule ProtocolError {
2   inputmessages
3     eProtocolViolation:RegularExpressionViolation=
4      (selector == ['read','write','open','close']) {
5        event.predicate =
6          "(open (read | write)* close)*";
7     }
8 }
```

Both events are composed in the filter module FileErrors, so when either event happens, the event eFileError is triggered:

```
1 filtermodule FileErrors composes AccessError, ProtocolError {
2   inputmessages
3     eFileError:BooleanDetector=
4       (selector==['eNotAuthenticated', 'eProtocolViolation']);
5 }
```

The filter module LogAccesses encapsulates a response to this event; therefore, it is composed with FileErrors to be able to refer to the event eFileError. The filter logger of the user-defined type Log receives the information to be logged as its parameter. Here, **inner** refers to the object on which the containing filter module is superimposed. The name property of this object is logged when the response is executed:

```
1 filtermodule LogAccesses composes FileErrors {
2   filters
3     logger:Log=
4      (selector == ['eFileError']){
5        filter.info = inner.name
6      }
7 }
```

The filter module PreventAccesses encapsulates another response to the event eFileError. The filter prevention of the user-defined type Prevent intercepts the call flow of messages and prevents their execution. The filter module AccessControl composes PreventAccesses and LogAccesses to define the execution order of the responses. Here, the information about incorrect accesses must be logged, before the accesses are prevented:

```
1 filtermodule PreventAccesses composes FileErrors {
2   filters
3     prevention:Prevent=
4      (selector == ['eFileError'])
5 }
6
7 filtermodule AccessControl
8     composes PreventAccesses, LogAccesses {
9   constraints
10    precede (LogAccesses, PreventAccesses);
11 }
```

The listing below depicts the superimposition of the above-defined filter modules on application classes. We assume that the classes, which implement the functionality to access files, are defined in the namespace Application and have

the suffix File. The superimposition of AccessControl implies that at runtime individual instances of AccessControl and its component filter modules are bound to individual instances of the selected classes.

```
1 superimposition{
2   selectors
3     fileSelector = {C| isClassWithName ('Application.*File')};
4   filtermodules
5     fileSelector <- AccessControl;
6 }
```

The above listings show the possibility to define separate monitors (events) and recovery strategies (responses) and compose them with each other. This helps to localize changes, and consequently improves the modularity, reusability and maintainability of different runtime-verification concerns. For example, if the access pattern of a file changes, a developer only needs to modify the filter *eProtocolViolation* defined in the filter module *ProtocolError*, and the other filters remain unchanged.

5. AN EXECUTION MODEL FOR EVENTS AND COMPOSITION

After we have presented two concrete language extensions, in this section we explain the extensions we have formulated in Section 2 in a language-independent way. This provides an implementation path for including the extensions in various AOP languages. Therefore we have extended previous work, the *Advanced-dispatching Language-Implementation Architecture for Java* (ALIA4J) [5, 4] which provides generic implementations of language concepts, including those in aspect languages. In the following subsection we present the building blocks of extended ALIA4J's execution model. In subsection 5.2 we present functions operating on the model that realize the requirements for composing and refining whole aspects naturally.

5.1 A Semantics for the Execution Model

The first major component of ALIA4J is the *Language-Independent Advanced-dispatching Meta-model* (LIAM) for expressing advanced-dispatching declarations and relationships among them. In the context of aspect-oriented languages, *dispatch* is the identification of a join point and execution of appropriate responses. Advanced-dispatching declarations correspond to declarations of basic units. Code of the program *not* using advanced-dispatching mechanisms is represented in its conventional Java bytecode. This is also true for responses written in the base language as is the case in, e.g., AspectJ.

The second component of ALIA4J is the *Framework for Implementing Advanced-dispatching Languages* (FIAL). A FIAL-enabled execution environment can deploy and undeploy basic-unit declarations and relations between them conforming to LIAM. The framework embodies the semantics of executing LIAM-based basic-unit declarations by defining an algorithm to derive an execution strategy per join-point shadow that considers all currently deployed declarations.

It has been demonstrated that ALIA4J is suitable to realize the languages AspectJ, Compose*, CaesarJ and JAsCo [6]. A mapping of these languages to ALIA4J is provided in an electronic appendix[3]; for AspectJ even a tool that facilitates an automatic creation of LIAM models from AspectJ source code is provided.

The extensions to ALIA4J presented in the following lay out the path to implementation of our extensions for languages supported by ALIA4J. Extending the AspectJ translator tool to support extended AspectJ source code or implementing a similar tool for Compose*, however, is still future work. The implementation of the extensions is contributed to the main development branch of ALIA4J[4].

5.1.1 Basic Unit Declarations

Meta-Model.

Figure 2 shows the meta-entities of LIAM for defining a basic unit. For better readability, we use names for the LIAM entities that correspond to the terminology introduced in Section 2 whenever possible. We capitalize terms when referring to the LIAM entity.

The *Basic Unit* entity consists of three components: *Response* defines the response functionality and a set of *Specializations* defines the event detector: When a join point matching one of the Specializations is detected, the response is executed with the system state exposed by that Specialization; *Schedule Information* specifies when it should execute relative to the join point, i.e., before, after or around the underlying operation.

A Specialization matches join points by means of a *Pattern* and a *Predicate*. The former is evaluated based on static properties of the join point's shadow instruction, the latter based on the dynamic state, in terms of *Atomic Predicates*, at the join point. *Contexts* model the exposed system state. Atomic Predicate and Context itself can also refer to Contexts meaning that they require certain context values to be evaluated themselves.

LIAM now has three concrete entities that require special treatment in the algorithm that derives the execution strategy from Basic Units: An *Event Triggering Response*, with a property specifying the name of an event, corresponds to a response that *can* perform the trigger operation proposed in Section 2. If the trigger operation is executed, the Event Triggering Response notifies FIAL of the detection of the named event, also providing the event-parameter values. An *Event Atomic Predicate* has a property specifying an event name; when the named event has been detected at a join point, the Event Atomic Predicate evaluates to *true*, otherwise it evaluates to *false*. Finally, an *Event Context* has two properties, i.e., the name of an event and the name of a parameter of this event; the Context evaluates to the parameter's value.

In Section 2 we defined two alternative semantics for evaluating event-trigger basic units. The choice of the supported semantics significantly impacts the execution model, which is why extended ALIA4J only supports one semantics. As discussed in the same section, in this paper we favor the semantics in which all event detectors are evaluated at once and then all responses of basic units referring to detected events are executed. This semantics elegantly integrates with ALIA4J's execution model [23] in which an Atomic Predicate (and, thus, an event detector) can only discriminate events by referring to values as they are before the first response execution. Nevertheless, we discuss the implica-

[3] http://www.alia4j.org/alia4j-languages/mappings.html

[4] http://www.alia4j.org

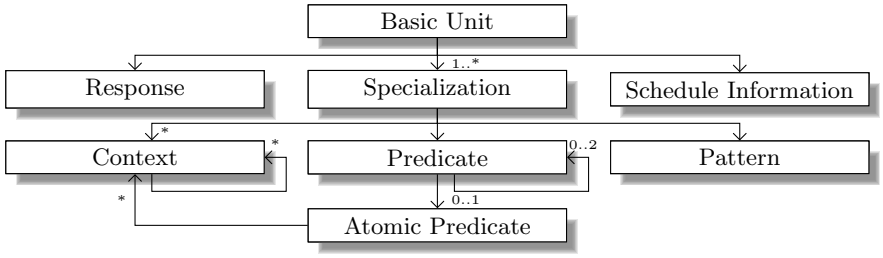

Figure 2: Entities of the Language-Independent Advanced-dispatching Meta-Model (LIAM) as UML class diagram.

tions of the second semantics, i.e., an *alternative execution strategy*, on the ALIA4J implementation below.

Execution Strategy.

For each join-point shadow in the executed program those Basic Units are determined that have a Specialization with a Pattern matching the join-point shadow; these are called the *matching Basic Units* and correspond to Basic Units with a Response that is potentially executed at a join point caused by this shadow (we also write *executing a Basic Unit* as an abbreviation of executing a Response referenced by the Basic Unit).

All Basic Units that have an Event Triggering Response—we refer to them as *Event Triggers*—are first removed from the set of matching Basic Units and treated specially. The *matching Event Triggers* must be executed first to ensure that all appropriate trigger operations have been performed before event names referred to in the evaluation of Specializations of other Basic Units.

Since Specializations of an Event Trigger F may themselves depend on events detected by another Event Trigger E, an order for evaluating the Event Triggers must be determined in which E is evaluated before F. For this purpose, an *Event Dependency Graph* is created which consists of the matching Event Triggers as nodes with a directed edge between two Event Triggers E and F iff F has an Event Atomic Predicate or an Event Context depending on the event detected by E. This graph is topologically sorted resulting in a legal execution sequence. This is not possible if the graph contains cycles, which is therefore forbidden.

Whenever a join-point shadow is executed, the Event Triggers are evaluated in the order determined as described above. When an Event Triggering Response is executed that performs the event-trigger operation, the event is detected and its parameter values are recorded. Event Atomic Predicates subsequently evaluated at the same join-point-shadow execution look up the event's detection while subsequent Event Contexts can retrieve needed parameter values.

Next, the execution algorithm of ALIA4J continues with the remaining matching Basic Units. The so-called dispatch function is created from all Predicates associated with the Specializations whose Pattern has matched. This is evaluated to the set of *applicable Basic Units*, i.e., Basic Units with a Specialization whose Pattern matches the join-point shadow and whose Predicate is satisfied. Being applicable does not necessarily mean that the Response associated with the Basic Unit is also executed, as is detailed in the following subsubsection.

Alternative Execution Strategy.

In the second semantics, Event Triggers are only executed when an Event Atomic Predicate refers to them. Therefore, all Event Triggers are removed from the matching Basic Units. The Event Dependency Graph is not needed. Each Predicate of a Basic Unit is brought into the form of a conjunction of two subpredicates, one with the non-Event Atomic Predicates and the other with the Event Atomic Predicates. The dispatch function is formed from only the non-Event Atomic Predicates of the matching Basic Units. Each Response is then associated with the relevant conjunction of Event Atomic Predicates, called the residual dispatch function, that is evaluated just before the Response is to be executed, which ultimately only happens when the evaluation is successful.

5.1.2 Relations Between Basic Units

Meta-Model.

Besides modeling the Basic Units themselves, LIAM also can model relations between Basic Units applicable at the same join point. Two kinds of relations are supported, namely composition rules and ordering relations.

A *Composition Rule* specifies requirements and restrictions for Basic Units executed together at the same join point and is defined by four sets of Basic Units: *Present*, *Absent*, *Required*, and *Forbidden*. A set of Basic Units S satisfies a Composition Rule if whenever S contains the Basic Units in Present, and does not contain the Basic Units in Absent, then S also contains the set Required, and does not contain any Basic Unit in the set Forbidden.

When the Required set is empty, a Composition Rule can *heal* sets of Basic Units that violate the rule. That is, when a set of applicable Basic Units S contains the set Present, and does not contain any Basic Units in Absent, but also has Basic Units in Forbidden, simply remove the latter ones from S to obtain a set that satisfies the rule. This behavior for a Composition Rule can be enabled by setting its boolean property *heal* to *true*.

Adding Basic Units from the Required set of a Composition Rule that were not applicable at a join point in the first place is not possible because there is no way to make them applicable: A Specialization of a Basic Unit forms a contract between the Response and the join point at which it is executed. It defines properties of the program state at these join points and it defines which context values are exposed to the Response. Since the needed properties do not hold at a join point where a Basic Unit was not applicable in the first place, and since it is unspecified which values to expose,

it is generally not possible to execute a Basic Unit without a matching Specialization. Therefore the heal property of a Composition Rule may only be enabled when the Required set is empty.

For ordering, *Precedence Rules* can be defined, which associate two Basic Units of which one has the *preceding* and the other the *preceded* role. Precedence Rules cannot be specified between a Basic Unit with a "before" Schedule Information and another with an "after" Schedule Information, furthermore, cyclic rules are prohibited.

Execution Strategy.

The set of applicable Basic Units is further refined according to the Composition Rules with *heal* set to *true*. Because excluding Basic Units from execution changes the combination of Basic Units, new Composition Rules may be applicable afterwards and therefore healing rules must be recursively applied until no more are applicable.

To enable the algorithm for healing rules, a graph is maintained with healing Composition Rules as nodes and directed edges between them. The edges reflect two possible relations between Composition Rules r_1 and r_2: r_1 *enables* r_2, when $r_1.Forbidden \cap r_2.Absent \neq \emptyset$ and r_1 *disables* r_2, when $r_1.Forbidden \cap r_2.Present \neq \emptyset$. The *enables* relationship only means that the rule r_2 should be reevaluated after applying the healing according to rule r_1. Whether r_2 actually will match the applicable Basic Units after r_1 is applied depends on the concrete set of applicable Basic Units. In contrast, the *disables* relationship means that r_2 cannot match after r_1 has been applied.

This graph must not contain cycles where at least one edge is labeled disables because at least for one Composition Rule the precondition of present and absent Basic Units does not hold in the end result. Therefore the rule should not have been applied in the first place, which is a conflict. Cycles where all edges are labeled enables are legal, but the above algorithm must keep track of rules already applied in order to avoid an infinite recursion in such a case.

After determining a subset A of the applicable Basic Units which satisfy all those rules, the Composition Rules that have the healing property set to *false* still have to be checked. The set of matching Composition Rules is constructed as $R = \{r | r.Present \subset A \wedge r.Absent \cap A = \emptyset\}$. If at least one rule $r \in R$ is not satisfied, i.e., $r.Required \nsubseteq A$ or $r.Forbidden \cap A \neq \emptyset$, the combination of Basic Units is illegal and an error is raised.

Using the Precedence Rules and the relative order defined by the Schedule Information of the Basic Units to execute, an ordered tree is determined defining the execution order. A node corresponding to an "around" Basic Unit can refer to nodes corresponding to other Basic Units which are executed when the former performs the "proceed" operation. More details on the execution order of Basic Units can be found in [5].

Since the active Basic Units only change at deployment time, the execution strategy for join-point shadows and auxiliary graphs are generated at deployment and cached in ALIA4J. This is done as an optimization, but it also allows reporting violations of Composition Rules at deployment time instead of at the execution of a join point. Due to lazy class loading in Java, new join-point shadows may be added to the running system dynamically. For join-point shadows in classes loaded at deployment, the execution strategy can be created. For join-point shadows added later, a new execution strategy is created. Thus violations of Composition Rules are reported at class-loading time.

Alternative Execution Strategy.

Composition Rules define which Responses may or may not be executed together. However, with the second semantics of event-declaration execution, it is not always known in advance which Responses will be executed at a join point. Consider a healing Composition Rule whose Present set contains the Basic Unit b_1 and whose Forbidden set contains the Basic Unit b_2. Furthermore consider that b_1 depends on an Event Atomic Predicate and that a Precedence Rule exists such that b_2 is executed before b_1. Then b_2 must only be executed if the event associated with b_1 is not detected, but this is not known at the time when b_2 is executed. Thus, with the alternative semantics, Composition Rules can, for instance, only be resolved when the Basic Units in the Forbidden set are ordered after those in the Present set. Similar considerations hold for healing Composition Rules with a non-empty *Absent* set or for non-healing Composition Rules.

5.2 Realizing High-Level Languages with the Execution Model

In AO source languages, generally the central unit is an aspect which is comprised of multiple basic units. Therefore operations like defining precedence are specified in terms of aspects rather than single basic units as is the case in the execution model. Similarly, our proposed composition operators for natural aspects should be able to compose aspects and not only basic units.

For those operations relevant in the context of this paper, here we present functions in the FIAL framework that convert aspect-level operations to operations in terms of the execution model. The conversion algorithms use an *Aspect* data structure that has the sets *BasicUnits*, *CompositionRules*, and *PrecedenceRules* representing the basic units, composition and precedence rules defined in a source-level aspect. Generally, an aspect has a corresponding class defining fields and methods that can be used by the aspect's responses. Therefore, a response requires access to an instance of this class, the so-called *aspect instance*. In the meta-model an aspect's *Instantiation Context* is used by a Basic Unit to specify how to retrieve the aspect instance.

There are other features of aspect languages that have no direct correspondence in ALIA4J and have to be realized by defining Basic Units in certain ways. Examples are abstract aspects in AspectJ or parameterized filter modules in Compose*. Examples of how to create Basic Units to realize such language features can be found in our electronic appendix.

Composing multiple aspects into a new one effectively creates new basic units. While these basic units are not explicit in the source code, we create Basic Unit entities for them in the execution model. Creating new Basic Units makes it possible to separately deploy the component aspects as well as the composed one. Besides, the copied Basic Units are slightly altered because the Instantiation Context of component Basic Units is usually different in the composition.

The algorithm for composing Aspects uses the set of Aspects $a_{i \in \{1..n\}}$ to be composed and a map defining how to alter the Instantiation Contexts of the component Aspects' Basic Units in the composition. This map associates a component Aspect with the new Instantiation Context. The

algorithm creates a new Aspect, a_{new}; for each a_i it iterates over $a_i.BasicUnits$ and from each such Basic Unit it creates a copy and adds it to $a_{new}.BasicUnits$. If the map contains an entry for a_i, in the copy all references to the old Instantiation Context are replaced with a reference to the new Instantiation Context associated in the map. Next, the Composition Rules and Precedence Rules defined by the component aspects are considered. Each such rule is copied and added to the corresponding set of a_{new}, referring now to the copies of Basic Units in the composition.

In order to treat the *ignores* modifier, e.g., for A ignores B, an additional input maps Aspects to scopes that should be ignored, specified as Predicates (e.g., an Atomic Predicate evaluating to *true* when the execution is in the control flow of a basic unit in B). For each Basic Unit copied from a component aspect, the map is used to look up whether a scope is defined for the Aspect containing this Basic Unit. If a scope that is to be ignored, say s, is defined for an aspect (A above), the Predicate p of each Specialization of a Basic Unit in A is replaced with a Predicate equivalent to $p \wedge \neg s$ in the copy.

Defining overriding or precedence between two Aspects in a composition means to create Composition Rules or Precedence Rules between the Aspects's individual Basic Units. Both algorithms build the cross product of both Basic Unit sets, and define precedence and overriding pair-wise. To declare overriding between two Basic Units one Composition Rule is created with the overriding element being the single element in the Present set and the overridden element being the single element in the Forbidden set. The sets Absent and Required are empty and the property heal is set to *true*. If an exception—defined by the keywords **except**, **noignore** and **nooverride** in the previous sections—is specified for a pair of basic units, rules for them are created according to the exception's semantics for this pair. The created rules are added to the corresponding sets of a_{new}.

6. RELATED WORK

In most aspect languages, the events are declared using a descriptive (declarative) notation, like AspectJ pointcuts or Compose* matching patterns, or even a logic notation, as Prolog used by Compose* selector predicates. The problem is that current languages do not allow gathering information over time in these declarations, so aspect code has been used to overcome the limitations.

Named and abstract pointcuts of AspectJ might seem to provide a solution to the abstraction problem. However, often several layers are needed to naturally express the terminology appropriate for the aspect or event separately from that of the underlying system. In that case abstract and concrete pointcuts are insufficiently expressive, since they can either be entirely undefined if declared abstract, or must be described using regular pointcut notation otherwise. They are local to an aspect and also do not allow collecting information over time. Java annotations and their incorporation into AspectJ do facilitate giving "relevant" names to arbitrary collections of points in the code, and then referring to those in pointcut definitions. However, they suffer from the same problems as above, and require modifying the underlying program when new annotations are added.

In [21] the Eos language is proposed, with "classpects" to unify classes and aspects. Join points are declared separately from responses, using AspectJ-like declarative predicates to expose information and identify points in code. Thus, even though the static definition is separate, when information must be accumulated to determine whether a complex event has occurred, it appears in the code of a classpect rather than in a join point binding.

In a similar vein, the JAsCo language [28] enforces a more syntactic separation of the *when* and the *what* of an aspect. The functionality, called a *Hook*, is always defined with an abstract pointcut; in order to activate it, a *Connector* must be implemented that concretizes the pointcut. JAsCo also allows to define custom combination strategies for aspects. In contrast to the work presented in this paper, it is not possible to define Connectors at multiple abstraction layers or events that accumulate runtime state programmatically. The composition strategies are defined per *shared join point*, instead of at the aspects level, and excluding aspects from the scope of others they are composed with is not possible.

In [26], implicit invocation with implicit announcement is investigated. The authors introduce the concept of *join-point types* which encapsulate pointcuts and define an interface of exposed values. This is similar to events in our work. But, as the authors point out, this does not treat the issue of scattering. Classes declare which join-point types they *exhibit*—similarly to the way methods declare exceptions they may throw. Each join-point type is thus defined locally for the class in which it might occur, and it seems difficult to treat complex events that crosscut classes. Furthermore, pointcuts cannot refer to join-point types which makes it impossible to compose events other than along a single-inheritance hierarchy.

Stateful aspects in JAsCo [29], tracematches [20] and tracechecks [7] do treat a sequence of states that culminate in an event, but emphasize pure detection, and do not support decomposing events into layers at different levels of abstraction. Accumulating state information and context that can be used in an aspect to respond to the detected event is partially supported in these notations, but is still inadequate for all needs. In contrast to those works, it is more powerful to use code to accumulate and evaluate relevant information rather than logical declarative constructs with limited expressive power. Defining events imperatively may also be more natural to programmers who otherwise also have to be familiar with the particular logical formalism. In these approaches event detectors and responses to detected events are defined together and event detectors cannot be reused with different responses or as part of other event detector definitions. The same is true of the more abstract approach in [9, 10] that allows accumulating information in an aspect state. Tracecuts [30] fall in the same category of language mechanisms but can be defined similarly to named pointcuts in AspectJ. They can be reused with different responses and in the definition of other tracecuts; but reuse is only possible within the defining syntactic unit. The above discussion about limited expressiveness and being unnatural to programmers also applies to tracecuts.

E-Chaser [17] is an extension to Compose* to treat sequences of messages as an event and form a hierarchy of events. The extensions proposed in this paper enhance the modularity of E-Chaser by separating responses from the declaration of an event, and by facilitating the composition of filter modules.

The desirability of composing aspects is seen in early work on a calculus of superimpositions [24, 25], relating aspects

to superimposition constructs developed for distributed systems. That work also shows some of the various ways in which aspects can be composed.

Observer and spectative aspects [8, 13] somewhat resemble events, but are not supported in syntax. In some versions, those kinds of aspects are allowed to output information gathered or save it to global variables, thus differing from pure event detectors. As described in the following section, tools to identify these kinds of aspects can be adapted to ensure that event declarations are legal.

The ideas of clearly separating event detection from responses, and defining events in a hierarchy based on lower-level events is seen in the event-based development approach [16, 12]. This is integrated with aspects in the HighspectJ framework [19], but as a methodological framework not extending AspectJ. In that work, interfaces for event aspects and for response aspects are defined, and a library of aspects and methods provides facilities to encourage usage that follows the ideas of an event hierarchy. As noted in the Introduction, the extensions here can be seen as making aspect languages more appropriate for complex event systems.

7. PRELIMINARY EVALUATION

Although a full evaluation of the extensions in this paper will need further investigation, some preliminary conclusions can be drawn. The goals of the work have been to (1) present language extensions for AO languages that completely separate event detection from responses, and aid in forming compositions, (2) show the increased separation from code-level operations, improved potential for modularity, and use of terminology natural to the concern of each aspect or event, (3) demonstrate the programming style the extensions facilitate, and (4) show how they can be effectively implemented and used for static analysis. The language extensions and their use referred to by our first three goals are discussed in Sections 2 – 4, while an implementation is described in Section 5. The degree to which our second and third goals are achieved is further considered in case studies below. Our fourth goal is discussed in terms of ease of implementation including the syntactic support provided, and the implications for correctness in Sections 7.2 and 7.3.

7.1 Case studies

7.1.1 Software process guidance in AspectJ

As a first case study, a collection of aspects for software process guidance from [19] and additional examples over the Eclipse software development environment were rewritten using the AspectJ version of the extensions presented here. Unfortunately, as we have not yet implemented a compiler for extended AspectJ, we presently cannot execute the results. The aspects deal with test-driven development, detecting usability problems of user interfaces, and enforcing a protocol for committing code to a source control repository. Some of the results are seen in the software-process example in Section 3.3. The low-level events directly connected to the Eclipse environment were generally expressed as event detector expressions using pointcut notation, while the aspects that actually defined complex events became event detectors expressed as code declarations based on other events.

In writing aspect code using the extensions, it became clear that there is a trade-off between defining separate event detectors, or defining more general ones differentiated by the data values returned in the event parameters. For example, it is a matter of taste whether there are separate SuccessfulTest and FailedTest events that combine into a Tested event, or just a Tested event with a parameter OK given a value of *true* when the test passed, and *false* otherwise. In our case study, a resolution similar to that seen in the examples was used, tending towards the first option above, especially when the treatment of those events differs.

For the code integration example, four basic events were defined, and six higher-level event declarations that only refer to other events, and thus are separate from the code. For the usability example, five basic events were defined, and eight higher-level ones, while for test-driven development, there are seven new basic events and six higher-level ones (that also use previously defined basic events), in two levels. That is, for an initial investigation of software-process support we defined 36 events overall, 16 connected directly to the code of the underlying system through pointcut-like syntax, and 20 higher-level ones that use other events.

The response aspects in the case study correspond to recording events in persistent logs, sending announcements of violations to users and/or managers, preventing continuation until a deviation is corrected, and, when feasible, using an aspect to automatically correct a user deviation from a desired practice. There are also aspects that combine the several responses. In our case study we defined 10 such responses, seven of which do not refer directly to code of the base system. Thus overall, 19 events or aspects are directly linked to the base system, while 27 are not. Although many parts of the Eclipse implementation have been stable over multiple versions, the way in which a JUnit test is recognized changed from JUnit version 3 to 4, and different version control systems have been added (e.g., CVS, SVN, GIT). Adjusting to these changes only requires adding or changing basic events, and otherwise does not influence the collection of events and aspects for software-process guidance.

Even though the original AspectJ code used for the case study was based on a methodology that encouraged separating events, the results using the syntactic extensions improved the separation and reuse potential of the event detectors, and defined new responses in aspects that combined simpler ones. Of course, future work will involve refactoring aspect systems not initially designed with a clean separation of events from responses, and needs to include a more detailed comparison with other alternatives.

7.1.2 Runtime monitoring using Compose*

The code fragments involving runtime verification in Section 4.4 represent a re-implementation—again, this is a dry run because we have not yet implemented a compiler for extended Compose*—of a much more extensive runtime verification module in Compose*, using our extension. Since the current version of Compose* does not support the definition of events nor the separation and composition of events and responses, in that work ordinary filter types were used to define monitors and to implement the correlated monitors and recovery strategies in one monolithic filter module. An implementation of our example in the current Compose* is shown in the listing below. Here, the filter module FileAccess groups four filters, notAuthenticated, protocolViolation, logger and prevention, which are executed in sequence. The data fields authorizationResult and protocolResult are defined to maintain the result of monitoring; and are passed to the filters

notAuthenticated and protocolViolation. The recovery filters logger and prevention check the value of these data fields to infer whether the properties are violated.

```
1  filtermodule FileAccesses {
2      internals
3          authorizationResult : java.lang.Boolean;
4          protocolResult : java.lang.Boolean;
5      conditions
6          authenticated: User.isAuthenticated();
7      inputfilters
8          notAuthenticated:BooleanDetectorFilter =
9            (!authenticated && selector == [*.*]) {
10               filter.output = authorizationResult
11           };
12
13     protocolViolation:RegularExpressionViolation =
14         (selector == ['read','write','open','close']) {
15             filter.predicate =
16                 "(open (read | write)* close)*";
17             filter.output = protocolResult;
18         };
19     logger:LogFilter =
20         (!(authorizationResult && protocolResult)
21                 && selector == [*.*]){
22             filter.info = inner.name
23         };
24     prevention:PreventFilter =
25         (!(authorizationResult && protocolResult)
26             && selector == [*.*])
27 }
```

Such an implementation of monitors and recovery strategies has several disadvantages compared to one using our extensions. First, there is no guarantee that monitors do not have external side-effects. Second, there is a fixed composition of monitors and recovery strategies, therefore it is not possible to reuse monitors in other contexts. Third, we are forced to keep the results of filters in data fields accessible to all filters in the filter module. Thus no information hiding is possible in the composition of monitors and recovery strategies. Fourth, there is no linguistic support for composing monitors to form more coarse-grained monitors. Finally, as the number of monitors and recovery strategies increases, the readability of a monolithic filter module decreases.

A complex program usually has several correlated properties to be verified, and properties may have different levels of granularity. For example, one may want to verify the usage protocols at the level of individual classes, components, processes, etc. Accordingly, several recovery actions may also be defined to heal the program. A complex program also frequently evolves over the time, and consequently its properties to be verified, monitors and recovery strategies must be adapted with the changes. Therefore to ease the use of runtime verification for developers, it must be possible for example to add or remove monitors, to compose a coarse-grained monitor from fine-grained monitors, to add or remove recovery strategies and connect them with monitors, etc.

In the example presented in the extended Compose* syntax, we have separated the verification and recovery task into six filter modules as opposed to just one filter module in the plain Compose* solution. Of the six filter modules, only two refer to code-level messages. The remaining four are higher-level filter modules.

7.2 Implementation and syntactic support

The particular extensions for AspectJ and for Compose* demonstrate that the ideas can be incorporated into different aspect languages in a way that conforms with the existing constructs and style of each language. The design of an implementation in ALIA4J in Section 5 shows that the extensions are compatible with existing implementations, and there is no indication that they would significantly increase the translation or runtime overhead of an AOP language to which they are added. As noted, an optimizer can determine which events are actually used (directly or indirectly) in an aspect that responds by changing the system or the environment, and detect only those events. The optimizer could also apply in-lining and other compilation techniques when appropriate.

Moreover, there are existing tools to detect spectators and observers [1, 8, 22, 31] that can be adapted to validate that event detectors are free of external side-effects. These are all based on data-flow to determine that the only assignments in relevant code are to local variables or parameters. Difficulties in such analyses arise due to aliasing, exception handling, and system libraries. However, the methods in system libraries can be analyzed in advance to determine whether they are side-effect-free, and partial solutions exist for the other problems. In case studies of the cited works, over 98% of the spectative aspects are accurately detected. Using these techniques for event validation should provide similar results, and aid in detecting improper event declarations as part of compilation. A variant of such tools can announce an error for provably illegal event declarations and let pass those provably free of external side-effects. For event declarations for which it is unable to show either way, a warning can be emitted.

7.3 Implications for correctness

Although not the subject of this paper, the extensions facilitate easier reasoning about and verification of aspect systems. Because event declarations are required to be free of external side-effects, in themselves they have no global influence. The specification of an event detector should only designate that the event is detected at desired join points and only at those points, and that the information provided in the parameters reflects the desired relation with the execution of the underlying program. These properties can be expressed in temporal logic. Only the responses in aspects need to be analyzed to show that their effect on the overall system is as desired, assuming that the events and other aspects each satisfy their own specifications. This means that each verification task is relatively small, and that assume-guarantee reasoning can be used to show an event or aspect correct relative to the correctness of its components. Future work will explore such modular verification techniques in depth.

8. CONCLUSION

This paper has presented proposals to completely separate responses from event detection and exposing needed context, and to define compositions of events and aspects that can form a hierarchy of terminology and actions natural to each concern. Although the changes to aspect languages have been kept to a necessary minimum, the programming style that the changes enable and encourage is significantly different from previous use. In our experience so far, the

extensions indeed do lead to improved modularity by easily allowing differing responses to the same events, increasing the potential for reuse, and alleviating the fragile pointcut problem by making most event detectors and aspects independent of the code in any particular system to which they can be woven.

Perhaps most importantly, the extensions allow a more natural expression of programmer intentions. These extensions should be especially valuable in building libraries or frameworks of reusable aspects and event detectors both for classic applications of aspects and for new areas such as cloud computing, mobile devices, and web-based computing.

9. ACKNOWLEDGMENTS

Partial support for this work was provided by NWO grant 040.11.140.

10. REFERENCES

[1] Y. Alperin-Tsimerman and S. Katz. Dataflow analysis for properties of aspect systems. In *5th Haifa Verification Conference (HVC)*, LNCS, 2009.

[2] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of CaesarJ. In *TAOSD*, LNCS, pages 135–173. 2006.

[3] L. M. J. Bergmans and M. Akşit. Principles and design rationale of composition filters. In R. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 63–96. 2004.

[4] C. Bockisch. *An Efficient and Flexible Implementation of Aspect-Oriented Languages*. PhD thesis, Technische Universität Darmstadt, 2009.

[5] C. Bockisch and M. Mezini. A flexible architecture for pointcut-advice language implementations. In *VMIL*, 2007.

[6] C. Bockisch, A. Sewe, M. Mezini, A. de Roo, W. Havinga, L. Bergmans, and K. de Schutter. Modeling of representative AO languages on top of the reference model. Technical Report AOSD-Europe-TUD-9, Technische Universität Darmstadt, 2008.

[7] E. Bodden and V. Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In W. Löwe and M. Südholt, editors, *Software Composition*, LNCS, pages 147–162. 2006.

[8] C. Clifton and G. Leavens. Observers and assistants: a proposal for modular aspect-oriented reasoning (also, revised as spectators and assistants). In *FOAL*, 2002.

[9] R. Douence, P. Fradet, and M. Sudholt. Composition, reuse, and interaction analysis of stateful aspects. In *AOSD*, pages 141–150, 2004.

[10] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.

[11] S. Ducasse, O. Nierstrasz, N. Schärli, and A. P. Black. Traits: A mechanism for fine-grained reuse. *TOPLAS*, 2006.

[12] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Press, 2010.

[13] S. Katz. Aspect categories and classes of temporal properties. In *TAOSD*, LNCS, pages 106–134. 2006.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[16] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Boston, USA, 2001.

[17] S. Malakuti, C. Bockisch, and M. Aksit. Applying the composition filter model for runtime verification of multiple-language software. In *ISSRE*, pages 31–40, 2009.

[18] O. Mishali and S. Katz. Using aspects to support the software process: XP over eclipse. In *AOSD*, pages 169–179, 2006.

[19] O. Mishali and S. Katz. The HighspectJ framework. In *ACP4IS*, pages 19–24, 2009.

[20] C. A. Pavel, C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. D. Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA*, pages 345–364, 2005.

[21] H. Rajan and K. Sullivan. Unifying aspect and object-oriented design. *TOSEM*, pages 3:1–3:41, 2008.

[22] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *FSE*, pages 147–158, 2004.

[23] A. Sewe, C. Bockisch, and M. Mezini. Redundancy-free residual dispatch. In *Proceedings of FOAL*, 2008.

[24] M. Sihman and S. Katz. A calculus of superimpositions for distributed systems. In *AOSD*, pages 28–41, 2002.

[25] M. Sihman and S. Katz. Superimposition and aspect-oriented programming. *BCS Computer Journal*, 46(5):529–541, 2003.

[26] F. Steimann, T. Pawlitzki, S. Apel, and C. Kastner. Types and modularity for implicit invocation with implicit announcement. *TOSEM*, 20(1), 2010.

[27] M. Störzer and C. Koppen. Pcdiff: Attacking the fragile pointcut problem, abstract. In *EIWAS*, 2004.

[28] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *AOSD*, pages 21–29, 2003.

[29] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. De Fraine. Stateful aspects in jasco. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Software Composition*, LNCS, pages 167–181. 2005.

[30] R. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *FSE*, pages 159–169, 2004.

[31] N. Weston, F. Taiani, and A. Rashid. Interaction analysis for fault-tolerance in aspect-oriented programming. In *MeMoT*, pages 95–102, 2007.