Application-Specific Models and Pointcuts using a Logic Metalanguage

Johan Brichau [a,d,1] Andy Kellens [b,2] Kris Gybels [b] Kim Mens [d]
Robert Hirschfeld [c] Theo D'Hondt [b]

[a] *Université des Sciences et Technologies de Lille, France*
[b] *Vrije Universiteit Brussel, Belgium*
[c] *Hasso-Plattner-Institut, Potsdam, Germany*
[d] *Université catholique de Louvain, Belgium*

Abstract

In contemporary aspect-oriented languages, pointcuts are usually specified directly in terms of the structure of the source code. The definition of such low-level pointcuts requires aspect developers to have a profound understanding of the entire application's implementation and often leads to complex, fragile, and hard to maintain pointcut definitions. To resolve these issues, we present an aspect-oriented programming system that features a logic-based pointcut language that is open such that it can be extended with application-specific pointcut predicates. These predicates define an application-specific model that serves as a contract that base-program developers provide and aspect developers can depend upon. As a result, pointcuts can be specified in terms of this more high-level model of the application which confines all intricate implementation details that are otherwise exposed in the pointcut definitions themselves.

Key words: aspect-oriented programming, logic metaprogramming, pointcut languages

Email addresses: johan.brichau@uclouvain.be (Johan Brichau), andy.kellens@vub.ac.be (Andy Kellens), kris.gybels@vub.ac.be (Kris Gybels), kim.mens@uclouvain.be (Kim Mens), hirschfeld@hpi.uni-potsdam.de (Robert Hirschfeld), tjdhondt@vub.ac.be (Theo D'Hondt).

[1] This work was partially supported by the European Network of Excellence AOSD-Europe

[2] Ph.D. scholarship funded by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

1 Introduction

Aspect-oriented Software Development (AOSD) is a recent, yet established development paradigm that enhances existing development paradigms with advanced encapsulation and modularisation capabilities [1,2]. In particular, aspect-oriented programming languages provide a new kind of abstraction, called *aspect*, that allows a developer to modularise the implementation of crosscutting concerns such as synchronisation, transaction management, exception handling, etc. Such concerns are traditionally spread across various modules in the implementation, causing tangled and scattered code [1]. The improved modularity and separation of concerns [3], that can be achieved using aspects, intends not only to aid initial development, but also to allow developers to better manage software complexity, evolution and reuse.

One of the most essential characteristics of an aspect-oriented programming language is that aspects are not *explicitly* invoked but instead, are *implicitly* invoked [4]. This has also been referred to as the 'obliviousness' property of aspect orientation [5]. It means that the *base program* (i.e., the program without the aspects) does not explicitly invoke the aspects because the aspects themselves specify when and where they need to be invoked by means of a *pointcut definition*. A pointcut essentially specifies a set of *join points*, which are specific points in the base program where the aspect will be invoked implicitly. Such a pointcut definition typically relies on structural and behavioral properties of the base program to express the intended join points. For example, if an aspect must be triggered at the instantiation of each new object of a particular class, its pointcut must capture those join points whose properties correspond to the execution of the constructor method. As a result, each time the constructor method is executed (i.e. an instance is created), the aspect is invoked. In most aspect languages, the invocation of an aspect corresponds to the execution of an *advice*, which is a sequence of instructions executed before, after or around the execution of the join point.

In the development of aspect-oriented programs, the definition and maintenance of appropriate pointcuts is often a complex activity. First of all, an aspect developer must carefully analyze and understand the structure of the entire application and the properties shared by all intended join points in particular. Some of these properties can be directly tied to abstractions that are available in the programming language but other properties are based on programming conventions such as naming schemes. 'Object instantiation' join points, for example, can be identified as the execution of constructor methods in languages such as Java. Accessing methods, however, can be identified only if the developers adhere to a particular naming scheme, such as through `put-` and `get-` prefixes in the method names. In contrast, a language such as C# again facilitates the identification of such accessor-method join points because

they are part of the language structure (through the C# 'properties' language feature). In essence, we can say that the more structure is available in the implementation, the more properties are available for the definition of pointcuts, effectively facilitating their definition. However, structure that originates from programming conventions rather than language structure is usually not explicitly tied to a property that is available for use in a pointcut definition. This is especially problematic in languages with very few structural elements such as Smalltalk. In such languages, application development typically relies heavily on the use of programming conventions for the implementation of particular concepts such as accessors, constructors and many more application-specific concepts. As a result, aspect developers are forced to explicitly encode these conventions in pointcut expressions, often resulting in complex, fragile, and hard to maintain pointcut expressions.

We propose to facilitate the definition and maintenance of pointcuts through an aspect-oriented programming language that features an *open*, *logic-based* pointcut mechanism. This allows us to reify the structural conventions as explicit properties available for use in pointcut definitions. Aspect developers can then define pointcuts in terms of *explicit* application-specific properties instead of *implicit* intricate structural details of the code. In turn, these intricate details are confined in the definition of the properties, which actually assert an application-specific model over the code. The application-specific model, as well as the pointcuts, are defined using a logic language (SOUL [6]) that exhibits some well-adapted characteristics for the definition and specialization of an application-specific model. This approach builds upon previous work on logic-based pointcut languages, where we have described how the essential language features of a logic language render it into an adequate pointcut definition language [7]. In this paper, we further exploit the full power of the logic programming language for the definition of application-specific properties. In particular, we have implemented our approach through an integration of the AspectS [8] and CARMA [7] aspect languages for Smalltalk.

In the following section, we discuss a number of pointcuts, implemented in AspectS, that rely on typical structural conventions that are adhered to by application developers in a Smalltalk environment. We explain how such pointcuts are complex, fragile, and hard to maintain. Next, section 3 describes how to deal with these issues by means of application-specific models and pointcuts, implemented in the integration of AspectS and CARMA, called AspectSOUL. Section 5 applies the approach to aspects that operate on the drag and drop infrastructure of the UI framework and the refactoring browser in the Smalltalk environment. We summarize related and future work in section 6 before concluding the paper.

2 Pointcuts based on Structural Conventions

When developing an application, developers often agree on particular programming conventions, design rules and patterns to structure their implementation. The intention of these structural implementation conventions is to render particular concepts more explicit in the implementation. For example, if all developers adhere to the same naming convention for all 'accessor' methods, we can more easily distinguish such accessors from any other method. In the context of aspects, the implementation structure that is introduced by these conventions is also often exploited in pointcut definitions.

In this section, we demonstrate this principle by studying the structural convention used to implement accessor and mutator methods, a simple but often-used pattern in Smalltalk. After a brief summary of the AspectS framework, we present a couple of aspects in AspectS whose pointcuts rely on these conventions to capture the execution of accessor methods. Finally, we discuss how pointcuts that implicitly capture the notion of an accessor method using the coding conventions, become more complex and easily suffer from the fragile pointcut problem.

2.1 AspectS

AspectS [8] is a Smalltalk extension for aspect-oriented programming. Unlike most other approaches to aspect-oriented programming, AspectS does not extend the Smalltalk programming language with new language constructs for writing down aspects and advice expressions. Instead, AspectS is a framework approach to AOP. Aspects are implemented as subclasses of the class `AsAspect`, pointcuts are written as Smalltalk expressions that return a collection of join point descriptors and advices can be implemented as methods whose name begins with `advice` and which return an instance of `AsAdvice`. Two of the subclasses of `AsAdvice` can be used to implement either an around advice or a before/after advice. An instance can be created by calling a method which takes as its arguments qualifiers, a block implementing the pointcut, and blocks to implement the before, after or around effects of the advice.

An example advice method is shown in Figure 1. It specifies that any invocation of an `eventDoubleClick:` method implemented by `WindowSensor` or any of its subclasses should be logged. The effect of the advice is implemented in the block passed to the `beforeBlock:` parameter. When one of the methods specified by the pointcut needs to be executed, this block is executed right before the execution of the method's body. The block is passed a few arguments: the receiver object in which the method is executed, the arguments passed to

```
adviceEventDoubleClick

 ^ AsBeforeAfterAdvice
    qualifier: (AsAdviceQualifier attributes: #(receiverInstanceSpecific))
    pointcut: [
      WindowSensor withAllSubclasses
        select: [:each |
           each includesSelector: #eventDoubleClick:]
        thenCollect: [:each |
           AsJoinPointDescriptor targetClass: each targetSelector: #eventDoubleClick:]]
    beforeBlock: [:receiver :arguments :aspect :client |
      self showHeader: '>>> EventDoubleClick >>>'
           receiver: receiver
           event: arguments first]
```

Fig. 1. Example advice definition in AspectS.

the method, the aspect and the client. In this example, the block simply logs some of its arguments to the transcript. Note that it calls a method on `self`, aspect classes can implement regular methods besides advice methods as well. The pointcut is implemented by the block passed to the `pointcut:` argument. It returns a collection of `AsJoinpointDescriptor` instances. This collection is computed using the Smalltalk meta-object protocol and collection enumeration messages: the collection of `WindowSensor` and all of its subclasses is filtered to only those that implement a method named `eventDoubleClick:`, an `AsJoinpointDescriptor` is then collected for each of these.

Advice qualifiers specify dynamic conditions that should hold if the advice is to be executed. These conditions are implemented as activation blocks: blocks that take as arguments an aspect object and a stack frame. The framework defines a number of activation blocks, that fall in two categories: checks done on the top of the stack, or on lower levels of the stack. The former are used for example to restrict advice execution to sender/receiver-specific activation: an advice on a method is only executed if the method is executed in a specific receiver object, or was invoked by a specific sender object, or is associated with a specific thread of control. The latter are used for control-flow related restrictions, such as only executing an advice on a method if the same method is not currently on the stack. The activation blocks have names, which are specified in the attributes of an `AsAdviceQualifier`. In the example advice, one activator block is specified: `receiverInstanceSpecific`.

Aspects can be woven into the Smalltalk image by sending an explicit `install` message to an aspect instance. The `install` method collects all advice objects in the class and executes their pointcut blocks to get the collection of join point descriptors. The methods designated by these descriptors are then decorated by wrappers [9], one for each advice affecting this particular method. The wrappers check the activation blocks specified in their advice, passing them the aspect and the top stack frame (accessed using the `thisContext` reflective feature of Smalltalk [10]). If an activation condition does not hold, the wrapper simply executes the next wrapper (if any), or the original method. If all activation conditions hold, the wrapper executes the advice's around,

before, and/or after block, and then proceeds to the next wrapper (if any) in the proper order, or the original method.

2.2 Pointcuts on Accessors and Mutators

We will now use the AspectS framework to implement an aspect that captures the execution of accessor and mutator methods in Smalltalk applications. In Smalltalk, clients are not allowed to directly access the instance variables of an object, and therefore they need to access them by means of dedicated methods. For each instance variable, a developer specifies an *accessor* method to retrieve the value of the variable, and a *mutator* method to change its value. Although these are regular Smalltalk methods, accessors and mutators are easily recognized since they are almost always implemented in an idiomatic way.

Most accessor and mutator methods are implemented according to the following structural convention:

- Both methods are classified in the `accessing` protocol;
- The selector of the *accessor* method corresponds with the name of the instance variable;
- The selector of the *mutator* method also corresponds with the name of the variable, however, this method takes one input parameter, namely the value to be assigned to the variable.

Moreover, the body of the accessor and mutator methods also follows a prototypical implementation. For example, suppose we have a `Person` class with an instance variable named `name`. The *accessor* and *mutator* methods for this variable are:

```
Person>>name
  ^name

Person>>name: anObject
  name := anObject
```

Since the join point models of current-day aspect languages do not explicitly reify these accessor and mutator methods as a separate kind of join points, aspect developers must exploit the structural conventions described above in order to capture the concept in a pointcut. For example, to capture all calls to accessor methods, the aspect developer can implement the following pointcut in AspectS:

```
1   [| all |
2    all := OrderedCollection new.
3    Root.Smalltalk allClasses do:
4       [:eachClass |
5         all addAll: (eachClass organization listAtCategoryNamed: #accessing)
6                       select: [:aSelector | eachClass allInstVarNames
7                                                 includes: aSelector asString]
8                       thenCollect:[:eachSelector |
9                                    AsJoinPointDescriptor targetClass: eachClass
10                                                        targetSelector: eachSelector])].
11   all]
```

The above pointcut makes the implicit assumption that accessor methods are rigorously implemented using the naming scheme in which the name of the method corresponds with the name of the instance variable. Lines 5 to 7 of the pointcut reflect the naming convention on which the pointcut is based. These lines select all messages corresponding to the name of an instance variable, and whose method is also classified in the `accessing` protocol.

As long as the developers of the base code adhere to the naming convention on which the pointcut relies, it will correctly capture all accessors. However, if a developer of the base program deviates from the naming convention, by for instance renaming the instance variable without also renaming the selector of the accessor, the pointcut no longer captures the correct set of join points. Instead of relying on naming conventions, a pointcut developer can also exploit the stereotypical implementation of accessor methods. This would result in the following pointcut:

```
1   [| all |
2    all := OrderedCollection new.
3    Root.Smalltalk allClasses do:
4       [:eachClass |
5           all addAll:(eachClass allSelectors
6                     select:[:eachSelector |
7                               eachClass selectorReturnsInstVar: eachSelector
8                             ]
9                     thenCollect:[:eachSelector |
10                                  AsJoinPointDescriptor targetClass: eachClass
11                                                        targetSelector: eachSelector])].
12   all]
```

Line 7 of the pointcut above invokes code that selects all methods which contain a return statement that directly returns the value of an instance variable. While this pointcut is not fragile with respect to changes in the names of instance variables, it still assumes that the base code developer rigorously followed the implementation idiom. However, often there exist slight variations on the programming idioms on which a pointcut is based. Consider for instance the following accessor method:

```
Person>>friends
  ^ friends isNil ifTrue:[friends := OrderedCollection new] ifFalse:[friends].
```

This method presents a variation on the often-used programming idiom for accessor methods. Instead of directly returning the value of the instance vari-

able, the method checks wether the variable has already been initialized, and if not, will set its value to an empty `OrderedCollection`. It is clear that this lazy-initialised version of accessor methods will not be captured by the pointcut which assumes that the accessor is implemented using a return statement that directly returns the value of the variable. In other words, the pattern that is expressed in the previous pointcut does not apply to this method, although it is an accessor method.

2.3 Complexity and Fragility

Although the example pointcuts described above rely on a rather simple structural implementation convention, their definition and maintenance is already a rather complex activity. First of all, an aspect developer needs to know and understand the intricate implementation details of the structural convention and implement a pointcut expression for it. The lazy-initialized accessor methods in the example above illustrate that there often exist a number of variations to the programming conventions used to implement a certain concept. Therefore, any pointcut that needs to capture the execution of an accessor method needs to capture all possible variations, which easily leads to complex and lengthy pointcut expressions. This is especially the case because the part of the pointcut which reasons about the join points and the part which expresses the structural convention are not clearly separated. In our example above, only a couple of lines of both pointcuts express the coding convention, while the other parts perform the actual selection of join points which are associated with the accessor methods. It is not instantly clear which part of the pointcut pertains to the coding convention, further complicating the reuse and maintenance of the pointcut expression.

Finally, the aspect developer must also carefully analyse the changes and additions to the base program in subsequent evolutions, which are possibly made by other developers. In essence, the definition of a pointcut that explicitly relies on structural conventions to capture an application-specific concept easily suffers from the fragile pointcut problem [11,12]. Due to the tight coupling between the pointcut and the implementation, seemingly safe modifications to the implementation may result in the pointcut no longer capturing the correct set of join points. For example, if the base program developers do not adhere to the coding conventions, or change the convention by for instance using the prefixes `put-` and `get-` to indicate a mutator or an accessor method respectively, the pointcut no longer captures the correct set of join points.

3 Application-specific Pointcuts and Models

We alleviate the problems outlined in the previous section through the definition of *application-specific pointcuts* that are expressed in terms of an *application-specific* model. Such an application-specific model is implemented as an extension to the pointcut mechanism and it identifies high-level, application-specific properties in the implementation and makes them available for use in pointcuts. Aspect developers can make use of these properties to define application-specific pointcuts, i.e. pointcuts that are no longer defined in terms of the low-level implementation details but, instead, are defined in terms of application-specific properties defined by the model. As a result, the intricate low-level details in the implementation remain confined to the implementation of the application-specific model, which is also the responsibility of the base program developers. The application-specific model effectively becomes an additional abstraction layer that is imposed over the implementation and it acts as a contract between the base program developers and the aspect developers.

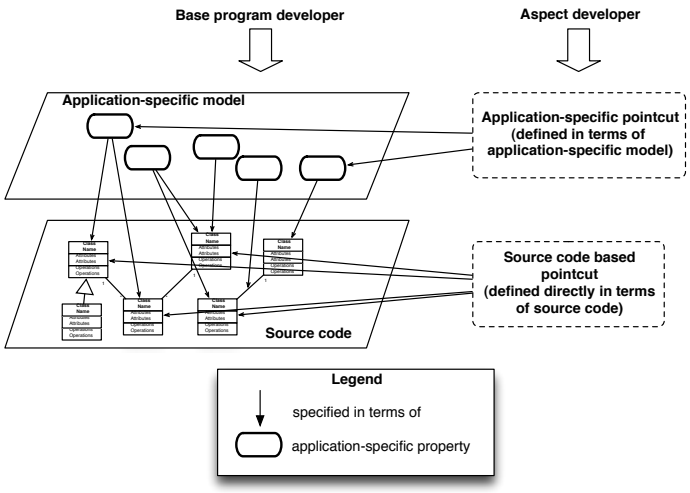

Fig. 2. Application-specific pointcuts are defined in terms of an application-specific model.

Figure 2 illustrates how application-specific pointcuts, implemented by the aspect developers, depend on the definition of the application-specific model that is certified by the base program developers. The application-specific pointcuts are defined in terms of the application-specific model which, in turn, is defined in terms of the implementation. This decoupling of the pointcuts from the intricate details of the implementation allows that base program developers define and maintain the application-specific model. In other words, the tight coupling to the implementation that is present in the source-code based pointcuts is effectively transferred to a more appropriate location, i.e. the definition of the application-specific model.

We have implemented this approach in AspectSOUL, an integration of the CARMA pointcut language [7] and AspectS [8]. In AspectSOUL, pointcuts are no longer written as Smalltalk expressions. Instead, pointcuts are written using the dedicated pointcut language of CARMA that is based on the logic programming language SOUL. Naturally, such a dedicated query language offers advantages for writing pointcuts, as pointcuts are essentially queries over a join point database. The integration of this logic-based pointcut language with AspectS further enforces the framework nature of AspectS by providing a full-fledged query-based pointcut language that can be extended with application-specific pointcut predicates. In essence, we combine the advantages of an extensible framework for defining advice expressions with the advantages of a dedicated and extensible pointcut language. In the remainder of this section, we first introduce CARMA and AspectSOUL and we subsequently focus on how the open, logic-based pointcut language provides developers with an adequate means to handle complex and hard-to-maintain pointcut expressions.

3.1 CARMA

CARMA is a pointcut language based on logic metaprogramming for reasoning about dynamic join points. Unlike pointcuts in AspectS, CARMA pointcuts do not express conditions on methods, its join points are representations of dynamic events in the execution of a Smalltalk program. CARMA defines a number of logic predicates for expressing conditions on these join points, and pointcuts are written as logic queries using these predicates. It is possible to express conditions on dynamic values associated with the join points. Furthermore, logic predicates are provided for querying the static structure of the Smalltalk program. These predicates are taken from the LiCoR library of logic predicates for logic metaprogramming [13]. The underlying language of this library and CARMA is the SOUL logic language [13,6].

The SOUL logic language is akin to Prolog [14], but has a few differences. Some of these are just syntactical, such as that variables are notated with question marks rather than capital letters, the "`:-`" symbol is written as `if`, and lists are written between angular (`<>`) instead of square brackets (`[]`). More importantly, SOUL is in linguistic symbiosis with the underlying Smalltalk language, allowing Smalltalk objects to be bound to logic variables and the execution of Smalltalk expressions as part of the logic program [15]. The symbiosis mechanism is what allows CARMA to express conditions on dynamic values associated with join points which are actual Smalltalk objects, such as the arguments of a message.

The advantage of building a pointcut language on the logic programming paradigm lies in the declarative nature of this paradigm. No explicit control

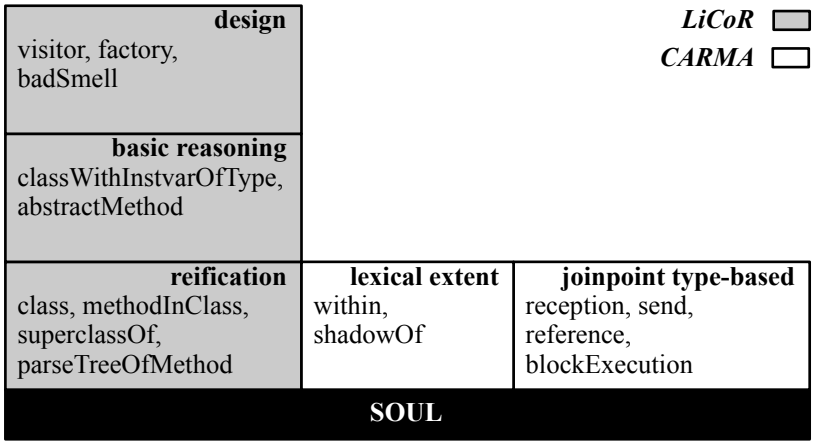

Fig. 3. Organization of, and example predicates in LiCoR and CARMA.

structures for looping over a set of classes or methods are necessary in pointcuts, as this is hidden in the logic language [16]. A pointcut simply states the conditions that a join point should meet in order to activate an advice, without specifying how those join points are computed. This makes declarative pointcuts, given some basic knowledge of logic programming of course, easier to read. A logic language also provides some advanced features such as unification that make it easier to write advanced pointcuts. A full discussion is outside the scope of this paper, but a more comprehensive analysis was given in earlier work [7]. In the next sections, we will however show how some of these features – particularly the ability to write multiple rules for the same predicate – are useful for writing model-based pointcuts.

The predicates in CARMA and LiCoR are organized into categories, as shown in Figure 3. The LiCoR predicates are organized hierarchically, with higher predicates defined in terms of the lower ones. The predicates in the "reification" category provide the fundamental access to the structure of a Smalltalk program: these predicates can be used to query the classes and methods in the program, and the fundamental relations between them such as which class is a superclass of which other class. The "basic reasoning" predicates define predicates that can be used to query more complex relations: which classes indirectly inherit from another class, which methods are abstract, which types an instance variable can possibly have etc. The "design" category contains predicates about design information in programs: there are for example predicates encoding design patterns [17] and refactoring "bad smells"[18].

The CARMA predicates access the dynamic structure of a Smalltalk program. There are two categories of predicates in CARMA, neither is defined in terms of each other, nor in terms of the LiCoR predicates. Nevertheless, the purpose of the "lexical extent" predicates is to link the dynamic and static structure, so that reasoning about both can be mixed in a pointcut. The `within` predicate for example can be used to express that a join point is the result of executing an

```
adviceEventDoubleClick

 ^ AsCARMAAroundAdvice
    qualifier: (AsAdviceQualifier attributes: #())
    pointcutQuery: 'reception(?jp, #eventDoubleClick:, ?args),
                    within(?jp, ?class, ?selector),
                    classInHierarchyOf(?class, [WindowSensor])'
    aroundBlock: [:receiver :arguments :aspect :client :clientMethod |
      self showHeader: '>>> EventDoubleClick >>>'
           receiver: receiver
           event: arguments first.
      clientMethod valueWithReceiver: receiver arguments: arguments]
```

Fig. 4. Example AspectS advice definition with a CARMA pointcut.

expression in a certain method. The "type-based" join point predicates are the basic predicates of CARMA, they express conditions on certain types of join points and basic data associated with those. An example is the `reception` predicate which is used to express that a join point should be of the type "message reception", which means it represents the execution of a message to an object. Besides the join point, the predicate has parameters for the basic associated data: the selector of the message and its arguments. There are also a few other predicates in CARMA (not shown in the figure), such as the `inObject` predicate which links a join point to the object in which it is executed. In the case of a reception join point, this is the receiver of the message.

A pointcut in CARMA is written as a logic query that results in join points. By convention, the variable to which these are bound is called "`?jp`". The join point representations should only be manipulated through the predicates provided by CARMA. An example pointcut is given in the next section.

3.2 CARMA Pointcuts in AspectS

AspectSOUL, the integration of CARMA with AspectS, is realized by subclassing the advice classes of AspectS so that a CARMA pointcut can be specified instead of a Smalltalk expression. The signature of the instance creation message for these subclasses is similar to the original. It takes as arguments a string with a CARMA pointcut, qualifiers and an around or before and/or after block. The message does a mapping to the instance creation message of the superclass. This is not a direct 1-on-1 mapping however, because CARMA pointcuts are about dynamic join points, in contrast with the more static join points of AspectS. Also, because AspectS does not support aspects that intercept block execution nor variable accesses or assignments, these features of CARMA are not adopted in AspectSOUL.

An example of an AspectS advice with a CARMA pointcut is shown in Figure 4. This is an around variant of the first example advice (of Figure 1), with a pointcut that has the same effect. The first condition in the pointcut speci-

```
reception(?jp, #eventDoubleClick:, <?event>),
objectTestHolds(?event, #isYellow)
```

Fig. 5. A CARMA pointcut with a condition on a dynamic value.

fies that `?jp` must be a message reception join point, where the selector of the message is `eventDoubleClick:`. The arguments of the message are bound to the variable `?args`. However, `?args` is not used any further in the pointcut which means that no conditions are put on the argument list. The second condition expresses that the join point must occur lexically in a method with name `?selector` in the class `?class`. For a message reception join point, this is effectively the method that is executed to handle the message. The final condition expresses that the class `?class` should be in the hierarchy of the class `WindowSensor`. The block has the same effect as in the first example, except that here it explicitly calls the next wrapper (if any) or original method.

Figure 5 gives an example of a CARMA pointcut which does express conditions on the arguments of a message reception. The first condition expresses that `?jp` must be a message reception join point of the message `eventDoubleClick:`, where the argument list unifies with the list `<?event>`. Thus the argument list has to have one argument, which is bound to the variable `?event`. The value of `?event` is the actual Smalltalk event object that is sent as the argument of `eventDoubleClick`. The second condition uses the `objectTestHolds` predicate, which uses the symbiosis mechanism of SOUL to express that the object in `?event` must respond `true` to the message `isYellow`. Thus, this pointcut captures join points when a message about a double click event of the yellow mouse button is sent to some object. Expressing the same in AspectS can only be done by defining an appropriate qualifier, or by including the dynamic condition in the around block of the advice. The CARMA approach means that what conceptually should go into a pointcut can be better separated from the effect of the advice: that we only want to intercept double click events of the yellow mouse button is part of the "when" of the advice, not of the "what effect" it has. All of the qualifiers of AspectS can be expressed in CARMA, except for the control-flow qualifiers because CARMA does not currently support a construct similar to the `cflow` pointcut of AspectJ [19].

For completeness, Figure 6 provides an implementation of the accessor pointcuts of section 2.2 in CARMA. Lines 1 to 6 implement the pointcut that is based on the naming convention and lines 8 to 13 present the pointcut that relies on the structure of the methodbody. In particular, lines 1 to 4 of the first pointcut reflect the naming convention and lines 5 and 6 will intercept all messages which correspond to the naming convention. Similarly, lines 8 to 11 of the second pointcut select all methods which contain a return statement that directly returns the value of an instance variable. As with the previous pointcut, lines 12 and 13 capture all occurrences of these methods. Both of these pointcuts are equally fragile as their AspectS-counterparts. We will now re-implement these pointcuts into application-specific pointcuts expressed in

```
1  class(?class),
2  methodWithNameInClass(?method,?accessor,?class),
3  instanceVariableInClassChain(?accessor,?class),
4  methodInProtocol(?method, accessing),
5  reception(?joinpoint,?accessor,?args),
6  withinClass(?joinpoint,?class)
7
8  class(?class),
9  methodWithNameInClass(?method,?selector,?class),
10 instanceVariableInClassChain(?var,?class),
11 methodWithReturnStatement(?method,variable(?var)),
12 reception(?joinpoint,?selector,?args),
13 withinClass(?joinpoint,?class)
```

Fig. 6. Accessor pointcuts in CARMA.

terms of an application-specific model.

3.3 Application-specific Models and Pointcuts using AspectSOUL

Both the application-specific pointcuts and the application-specific model are implemented using SOUL logic metaprograms. In essence, the application-specific model defines a set of logic predicates that reify application-specific properties of the implementation, based on the conventions that are adhered to by the developers. Because the application-specific model is built as an extension to the pointcut mechanism, aspect developers can straightforwardly use these predicates in the definition of application-specific pointcuts to access the application-specific properties. Furthermore, the essential features of a logic language also facilitate the use and extension of the application-specific model.

In the following subsection, we define application-specific models for the accessors convention that was described in the previous section. Subsequently, we use these models to redefine the pointcuts of the previous section into application-specific pointcuts.

3.3.1 Application-specific Model

An application-specific model defines a set of logic predicates that are available for use in an (application-specific) pointcut. These logic predicates are implemented using SOUL logic metaprograms. We illustrate the definition of an application-specific model by means of the accessors and mutators example.

The model that defines the accessor and mutator method properties consists of two predicates:

```
accessor(?class,?method,?var)
mutator(?class,?method,?var)
```

These predicates declare the accessor and mutator properties over methods named `?method` defined in `?class`. Furthermore, they also extract the name of the variable `?var` that is accessed or modified. The implementation of these predicates captures the coding convention that is followed by the developer of the application. For example, the following implementation of the `accessor` predicate combines the two separate conventions from Figure 6:

```
accessor(?class,?methodName,?varName) if
        class(?class),
        instanceVariableInClassChain(?varName,?class),
        methodWithNameInClass(?method,?methodName,?class),
        equals(?varName,?methodName),
        methodInProtocol(?method, accessing),
        accessorForm(?method,?varName).

accessorForm(?method,?var) if
        returnStatement(?method,variable(?var))
```

The logic program above consists of two rules that each implement a predicate: `accessor` and `accessorForm`. The first predicate is defined in terms of the second one and a variety of predicates that are available in LiCoR. The first rule captures the naming convention of accessor methods as well as their classification in the 'accessing' protocol, as we described earlier. The verification of the idiomatic implementation of the accessor method is located in the second rule. This rule verifies if the method's implementation consists of a single return statement that consists of a single expression: the variable. As a consequence, the above logic metaprogram classifies methods of the following form as accessor methods:

```
Person>>name
        ^name
```

3.3.2 Application-specific Pointcuts

Once the application-specific model is defined by the base program developers, the aspect developers can use it to define application-specific pointcuts. For example, the application-specific pointcut that captures the execution of accessor methods can now be written as follows:

```
reception(?joinpoint,?selector,?args),
accessor(?class,?selector,?var)
```

This application-specific pointcut no longer relies on a particular coding convention for accessor methods, as opposed to source-code based pointcuts. Instead, it relies on the application-specific property of an accessor method that is provided by the application-specific model. The base program developers ensure that this model is maintained such that all accessor methods are correctly identified. Furthermore, because the pointcut definition now explicitly states that it captures the execution of accessor methods, it is more readable and understandable to other developers. Of course, the above pointcut is a

rather simple use of a single application-specific property. However, a single application-specific property does not correspond to a single pointcut. For example, consider the following pointcut that is defined in terms of the accessor and mutator properties:

```
reception(?joinpoint,?selector,?args),
accessor(?class,?selector,?var),
mutator(?class,?otherSelector,?var)
```

This pointcut matches all accessor method execution join points for variables for which there also exists a mutator method. It can, for example, be used in a synchronisation aspect to execute a write lock advice.

4 Model Extension and Parameterization

In the previous section, we have constructed application-specific models and pointcuts using a logic metalanguage. We have also argued how the declarative style of a logic language facilitates the definition of pointcuts. Likewise, the definition of application-specific models is also facilitated through the use of a logic metalanguage. First of all, the definition of the model in a logic metalanguage can be extended in a straightforward manner. Such extensions are often required when a new structural coding convention (for a concept that is already covered by the model) is agreed upon by the developers. Secondly, a single logic predicate can be used in many different ways. Each of its arguments can be used both as a parameter and as a return value, i.e. they are multi-way parameters. Fortunately, the definition of a logic predicate does not have to explicitly deal with all possible ways, and their combinations. In the following subsections, we explain each of these advantages in more detail.

4.1 Extending the Model

A specific advantage of building the application-specific model using a logic metalanguage is that we can easily extend the model through the definition of alternative logic rules for existing predicates. For example, the application-specific model that we defined above does not classify all accessor methods correctly. There exist many more possible implementations of accessor methods, such as the lazy-initialisation presented in section 2.2. Because the coding convention is now explicitly defined in the application-specific model and because the application-specific model is restricted to the coding conventions only, the base program developers can easily extend it to accommodate additional accessor forms. This is in contrast to when the coding convention is implicitly used in a pointcut definition. More importantly, because the model

is defined as a logic metaprogram, additional accessor forms can be defined using alternative definitions for the `accessor` predicate. For example, we can extend the definition of this property to include lazy-initialised accessor methods by including the following logic rule:

```
accessorForm(?method,?var) if
        returnStatement(?method,send(?nilCheck,[#'ifTrue:ifFalse:' ],<?trueBlock,?falseBlock>)),
        nilCheckStatement(?nilCheck,?var),
        statementsOfBlock(<assign(?var,?varinit)>,?trueBlock),
        statementsOfBlock(<?var>,?falseBlock)
```

The above logic metaprogram provides an alternative definition for the `accessorForm` predicate. These alternatives are placed in a logical disjunction with the already existing alternatives and, as a result, our application-specific model also ties the accessor property to methods of the following form:

```
Person>>friends
  ^ friends isNil ifTrue:[friends := OrderedCollection new] ifFalse:[friends].
```

However, the following accessor method does not correspond to the coding convention:

```
Person>>phoneNumbers
  ^ phoneNumbers ifNil:[phoneNumbers := OrderedCollection new] ifNotNil:[phoneNumbers].
```

Therefore, we can again define an alternative logic rule that detects accessor methods of the above form:

```
accessorForm(?method,?var) if
        returnStatement(?method,send(?var,[#'ifNil:ifNotNil:' ],<?nilBlock,?notNilBlock>)),
        statementsOfBlock(<assign(?var,?varinit)>,?nilBlock),
        statementsOfBlock(<?var>,?notNilBlock)
```

Such extensions of the model are particularly useful if different developers implement different modules of the same base program. If all developers agree on a single application-specific model (i.e. a set of properties implemented by predicates), they can each follow their own programming convention to implement each property. For example, one set of developers might even agree on the use of `put` and `get` prefixes for all accessor methods while other developers can follow the common Smalltalk convention that we just explained. The first group of developers then needs to define an alternative logic rule that correctly detects methods prefixed with `put` and `get` and implemented in their part of the base program as accessor methods.

However, alternatives need to be added and used with care. There can be conflicting definitions and other developers might also not be aware of the conventions that were included as extensions. Therefore, in the approach we present in this paper, we assume that all developers are aware of the conventions that are relied upon by the model. We refer the interested reader to other work [12] that focuses on the verification of the conventions in the implementation of the application in a context of aspect-oriented programming.

The definition of an application-specific model using a logic metalanguage does not only allow developers to associate structural conventions to properties available for use in pointcuts. In addition, the properties can be parameterized *and* expose values associated to the property. For example, the accessor predicate does not only expose particular methods as accessor methods along with the actual variable that is accessed by the method [3], the predicate can also be used to retrieve all accessor methods that access a particular variable. This is because in a logic language, the parameters of rules can serve both to pass values to the rule and return values from it. There is no distinction between either, both are done by passing logic variables as arguments to rules. If the variable is unbound, the rule will return all values for that variable which make the predicate hold. If the variable is bound, meaning another rule already gave it a value, the rule simply checks if its predicate also holds for that value. Of course, literal values can also be passed as arguments to rules. Thus with the accessor predicate, the name of the instance variable can also be passed as an argument so that the rule is used to check whether there is an accessor for that variable. For example, in the following code excerpt, there are three example logic conditions in which the parameters of the `accessor` predicate (defined in section 3.3.1) are used in different ways:

```
1  accessor(?class,?selector,?var)
2  accessor([Array],#at:put:,?var)
3  accessor(?class,?selector,#name)
```

The first example will retrieve all accessor methods and expose their class, methodname and accessed variable. The second example checks if the `at:put:` method in the `Array` class is an accessor method and retrieves its accessed variable. Finally, the use of the `accessor` predicate in the last example retrieves all accessor methods that access a variable named `name`.

5 Application-specific Models in Practice

The accessors and mutators example is a valuable application-specific model but relies on very simple coding conventions. In the development of a Smalltalk application, there are many more conventions that can be used to expose application-specific properties valuable for use in a pointcut definition. We illustrate the use of two such conventions in the following subsections. In particular, we build a model that exposes properties based on structural conventions

[3] Mind that the method name can be different from the variable name, depending on the actual coding convention.

used in the *drag and drop framework* of the user interface and the *implementation of refactorings* in the refactoring browser in Visualworks Smalltalk.

5.1 Drag and Drop Application-specific Model

The drag and drop facilities in VisualWorks Smalltalk are implemented by means of a lightweight framework. This framework identifies a number of hooks that allow a developer to implement the drag and drop behaviour for his particular application. These hooks are:

- **Drag Ok**: a predicate to check wether the current widget may initiate a drag;
- **Start Drag**: actions which need to take place in order to start the drag (e.g. creating a drag and drop context, ...);
- **Enter Drag/Exit Drag**: these hooks are triggered whenever during a drag, the mouse pointer enters/exists the boundaries of a certain widget;
- **Over Drag**: actions which are executed when the pointer is hovering over a widget during a drag (e.g. change the cursor);
- **Drop**: actions which take place when an element is dropped on a widget.

A developer can add drag and drop functionality to an application by associating methods with the hooks specified above. This is done by means of the `windowSpec` system of the VisualWorks user interface framework. A `windowSpec` is a declarative specification of the different widgets which make up the user interface of an application. This specification is then used by the user interface framework to construct the actual interface. In the `windowSpec`, the developer can, for each widget, associate methods with the different hooks of the drag and drop framework. In order to access the data which is being dragged, the origin of the drag operation, etc. these methods pass around a `DragDropManager` object.

The structure of the framework described above can be used to define an application-specific model that associates methods to an explicit drag and drop property: i.e. for each of the hooks defined above, we define a separate predicate. For example, we define the `dragEnterMethod(?class,?sel,?comp)` predicate that classifies all methods that implement the 'drag enter' hook. Furthermore, this predicate exposes the name of the visual component in the interface that is dragged over. This predicate allows aspect developers to write application-specific pointcuts that capture a drag event as the execution of such a method:

```
reception(?jp,?selector,?args),
dragEnterMethod(?class,?selector,?component)
```

Furthermore, we also define the `draggedObject(?dragdropmanager,?object)` and `dragSource(?dragdropmanager,?source)` predicates that reify the object being dragged and the source component from where it is being dragged respectively. Both predicates extract this information from the `DragDropManager` instance that is being passed as an argument to the drag and drop methods. We can now further extend the pointcut such that it only captures drag events that originate from a particular source or drags of a particular object. For example, we complete the above pointcut with the following conditions to capture drags originating from a `FigureManager` (lines 2–3) and dragging a `Line` object (lines 4–5). The first line merely extracts the only argument being passed to the 'drag enter' method, which is the `DragDropManager` object.

```
1  equals(?args,<?dragdropmanager>),
2  dragSource(?dragdropmanager,?source),
3  instanceOf(?source,[FigureManager]),
4  draggedObject(?dragdropmanager,?object),
5  instanceOf(?object,Line)
```

This pointcut is particularly useful for the definition of an aspect that renders an icon in our user interface depending on the element that is being dragged. Without aspects, we would need to implement the visualisation of such an icon in the 'drag enter' method of every application model in our user interface, resulting in duplicated and scattered code. Furthermore, the application-specific model now also allows us to decouple the pointcut definition from the actual structural conventions used in the user interface framework and implement them in terms of the explicit application-specific properties associated to a user interface.

5.2 Refactorings

Refactorings are behaviour-preserving program transformations which can be used to improve the structure of the application [18]. A number of these refactorings can be automated up to a certain degree, which has resulted in the development of tool support for performing refactorings directly from the IDE. In VisualWorks, such tool support is integrated with the `Refactoring Browser`.

The `Refactoring Browser` makes use of a framework implementing these refactorings. In this framework, all refactorings are represented by a subclass of the abstract `Refactoring` class. Each subclass must implement a `preconditions` method, which specifies the preconditions that the source code to be refactored needs to adhere to in order to perform the refactoring, and a `transform` method, which performs the actual program transformation.

As an example of an aspect based on the refactoring framework, consider a software engineering tool (for instance a versioning system) which, each time

a refactoring is initiated, needs to be notified of the program entities which are possibly affected by the refactoring. Such information is hard to retrieve from the source code of the framework. However, by creating an application-specific model for the refactoring framework, we can explicitly document this additional information. The following pointcut retrieves all affected entities for the instantiation of a refactoring:

```
reception(?joinpoint,?message,?arguments),
inObject(?joinpoint,?receiver),
refactoringInstantiation(?receiver,?message,?arguments,?affectedentities)
```

The first two lines of the pointcut select all message receptions and their receiver; the last line restricts these message receptions to the ones which instantiate a refactoring. Also, the pointcut binds all affected entities, depending on the input and the type of the refactoring to the variable `?affectedentities`.

The `refactoringInstantiation` rule is defined as follows:

```
1  refactoringInstantiation(?refactoring,?message,?args,?affectedentity) if
2   refactoring(?refactoring),
3   methodWithNameInClass(?method,?message,?refactoring),
4   instanceCreationMethod(?method),
5   refactoringAffectedEntity(?refactoring,?refactoringclass,?args,?affectedentity)
```

The first line of this rule checks wether the receiver of the message is a refactoring (i.e. wether it is a subclass of the class `Refactoring`). The second and third line implement the selection of those messages (and their arguments) which create an instance of the refactoring. Finally, the last line calculates, based on the arguments of the message, the program entities which can be affected by the refactoring.

For each refactoring, the affected entities are explicitly documented by logic rules.

```
refactoringAffectedEntity(?refactoring,[PushUpMethodRefactoring],?input,?affectedentity) if
  originalClassOfPushUpMethod(?input,?affectedentity)

refactoringAffectedEntity(?refactoring,[PushUpMethodRefactoring],?input,?affectedentity) if
  originalClassOfPushUpMethod(?input,?class),
  superclassOf(?affectedentity,?class).
```

The above rules reflect this knowledge for the `Method Push Up`-refactoring. The first line of both rules extracts the class of the method which will be pushed up from the arguments of the message reception. For this refactoring, both the class from which the refactoring is initiated (the first rule), as well as its superclass are affected (the second rule).

6 Related and Future Work

In previous work [12], we have introduced the technique of *model-based pointcuts* that allows to define pointcuts in a similar way as the application-specific pointcuts presented in this paper. In fact, the approach presented in this paper is a first step towards an improved integration of model-based pointcuts and logic-based pointcut languages [7]. In essence, we further extended the technique of model-based pointcuts to exploit the full power of the logic programming language for the definition of application-specific properties. In [12], we merely extended the pointcut language with a single predicate that allows to query a conceptual model of the program, implemented using intensional views [20]. In this paper, the model consists of full logic predicates, resulting in an improved integration of the model and the pointcuts. In contrast, in [12], we have shown how model-based pointcuts are less fragile with respect to changes in the base program primarily due to tool support that enforces developers to adhere to the correct conventions such that the model remains valid. In this paper, we have focused on the adequate features of a logic language for the creation and extension of the model and we presented an improved integration of the model with the pointcut mechanism itself. We are currently working on how to reconcile the support for the detection of the fragile pointcut problem with the full power of the application-specific models presented in this paper. Furthermore, there are a number of related approaches or techniques that work towards the same goal:

6.1 Expressive Pointcut Languages

The work described in this paper further extends our work on the expressive pointcut language CARMA [7] through which we have previously highlighted the advantages of logic languages, such as user-defined pointcut predicates. Some other recent experimental aspect-oriented languages also propose more advanced pointcut languages. The Alpha aspect language, for example, also uses a logic programming language for the specification of pointcuts and enhances the expressiveness by providing diverse automatically-derived models of the program. These models and their associated predicates can, for example, reason over the entire state and execution history [21]. Similar to our work [7], Ostermann and Mezini use logic rules to write new pointcut predicates. The use of logic rules for writing model documentation was not considered. EAOP [22] and JAsCo [23] offer event-based or stateful pointcuts that allow to express the activation of an aspect based on a sequence of events during the program's execution.

6.2 Annotations

An alternative approach to application-specific pointcuts over application-specific models is to define pointcuts in terms of explicit annotations in the code [24,25]. Annotations classify source-code entities and thereby make explicit additional semantics that would otherwise be expressed through implicit programming conventions. This approach, however, does not benefit from the expressive power that is provided by the logic metalanguage.

6.3 Design Rules and XPI

Yet another alternative approach is to explicitly include the pointcut descriptions in the design and implementation of the software and to require developers to adhere to this design. Sullivan et al. [26] propose such an approach by interfacing base code and aspect code through *design rules*. These rules are documented in interface specifications that base code designers are constrained to 'implement', and that aspect designers are licensed to depend upon. Once the interfaces are defined (and respected), aspect and base code become symmetrically oblivious to each others' design decisions. More recently, the interfaces that are defined by the design rules can be implemented as *Explicit Pointcut Interfaces* (XPI's) using AspectJ [27]. Using XPIs, pointcuts are declared globally and some constraints can be verified on these pointcuts using other pointcuts. Our approach is different in the fact that we keep the pointcut description in the aspect, leaving more flexibility to the aspect developer. While XPIs fix all pointcut interfaces beforehand, our application-specific model only fixes the specific properties available for use in pointcut definitions.

7 Conclusion

AspectSOUL is an extension of the AspectS language framework with the open-ended logic-based pointcut language of CARMA. The resulting integrated aspect language allows developers to extend the pointcut language with an application-specific model. Such an application-specific model defines new pointcut predicates that reify implicit structural implementation conventions as explicit properties available for use in pointcut definitions. These *model-based pointcuts* are decoupled from the intricate structural implementation details of the base program, effectively reducing their complexity. The definition of the application-specific model confines all these technical details and serves as a contract between the base program developers and the aspect de-

velopers. Finally, the logic paradigm offers adequate language features for the definition and extension of the application-specific model.

Acknowledgements

The authors wish to thank the anonymous reviewers of a previous version of this article for their valuable and helpful comments.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), no. 1241 in LNCS, Springer-Verlag, 1997, pp. 220–242.

[2] R. E. Filman, T. Elrad, S. Clarke, M. Aksit, Aspect-Oriented Software Development, Addison-Wesley, 2004.

[3] Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM 15 (1972) 1053–1058.

[4] J. Xu, H. Rajan, K. Sullivan, Understanding aspects via implicit invocation, in: Automated Software Engineering (ASE), IEEE Computer Society Press, 2004, pp. 332–335.

[5] R. E. Filman, What is aspect-oriented programming, revisited, in: Workshop on Advanced Separation of Concerns, ECOOP 2001, Budapest, 2002.

[6] The Smalltalk Open Unification Language (SOUL), http://prog.vub.ac.be/SOUL.

[7] K. Gybels, J. Brichau, Arranging language features for more robust pattern-based crosscuts, in: Proceedings of the Second International Conference on Aspect-Oriented Software Development, ACM, 2003, pp. 60–69.

[8] R. Hirschfeld, Aspect-oriented programming with aspects, in: Lecture Notes in Computer Science: Objects, Components, Architectures, Services, and Applications for a NetworkedWorld: International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7–10, 2002. Revised Papers, Vol. 2591, Springer-Verlag, 2003, pp. 216 – 232.

[9] J. Brant, B. Foote, R. E. Johnson, D. Roberts, Wrappers to the rescue, Lecture Notes in Computer Science.

[10] F. Rivard, Smalltalk: a reflective language, in: Proceedings of the Reflection Conference 1996, 1996, pp. 21–38.

[11] C. Koppen, M. Stoerzer, Pcdiff: Attacking the fragile pointcut problem, in: First European Interactive Workshop on Aspects in Software (EIWAS), 2004.

[12] A. Kellens, K. Mens, J. Brichau, K. Gybels, Managing the evolution of aspect-oriented software with model-based pointcuts, in: D. Thomas (Ed.), Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Vol. 4067 of LNCS, Springer-Verlag, 2006, pp. 501–525.

[13] K. Mens, I. Michiels, R. Wuyts, Supporting software development through declaratively codified programming patterns, Journal on Expert Systems with Applications 23 (4) (2002) 405–413.

[14] U. Nilsson, J. M, Logic, Programming and Prolog, 2nd Edition, John Wiley & Sons, 1995.

[15] K. Gybels, R. Wuyts, S. Ducasse, M. D'Hondt, Inter-language reflection: A conceptual model and its implementation, Elsevier Journal on Computer Languages, Systems & Structures 32 (2006) 109 – 124.

[16] R. Kowalski, Algorithm = logic + control, Communications of the ACM 22 (7) (1979) 424–436.

[17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[18] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[19] C. Lopes, E. Hilsdale, J. Hugunin, M. Kersten, G. Kiczales, Illustrations of crosscutting, in: P. Tarr, M. D'Hondt, C. Lopes, L. Bergmans (Eds.), International Workshop on Aspects and Dimensional Computing at ECOOP, 2000.

[20] K. Mens, A. Kellens, F. Pluquet, R. Wuyts, Co-evolving code and design with intensional views - a case study, Computer Languages, Systems and Structures 32 (2-3) (2006) 140–156.

[21] K. Ostermann, C. Mezini, M. Bockisch, Expressive pointcuts for increased modularity, in: A. P. Black (Ed.), Proceedings of European Conference on Object-Oriented Programming (ECOOP), Vol. 3586 of LNCS, Springer-Verlag, 2005, pp. 214–240.

[22] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura, M. Südholt, An expressive aspect language for system applications with arachne, in: Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, 2005, pp. 27–38.

[23] W. Vanderperren, D. Suvee, M. A. Cibran, B. De Fraine, Stateful aspects in JAsCo, in: Software Composition (SC), Vol. 3628 of LNCS, Springer, 2005, pp. 167–181.

[24] W. Havinga, I. Nagy, L. Bergmans, Introduction and derivation of annotations in AOP: Applying expressive pointcut languages to introductions, in: First European Interactive Workshop on Aspects in Software, 2005.

[25] G. Kiczales, M. Mezini, Separation of concerns with procedures, annotations, advice and pointcuts, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), no. 3586 in LNCS, Springer, 2005, pp. 195–213.

[26] K. Sullivan, W. Griswold, Y. Song, Y. Chai, M. Shonle, N. Tewari, H. Rajan, On the criteria to be used in decomposing systems into aspects, in: Symposium on the Foundations of Software Engineering joint with the European Software Engineering Conference (ESEC/FSE 2005), 2005.

[27] W. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Teware, Y. Cai, Rajan.H., Modular software design with crosscutting interfaces, IEEE Software, Special Issue on Aspect-Oriented Programming.