Application-Specific Models and Pointcuts using a Logic Meta Language

Johan Brichau[a,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-Platner Institut, 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 language that features a logic-based pointcut language that is open-ended such that it can be extended with application-specific pointcut predicates. As a result, pointcuts can be specified in terms of a more high-level model of the application that confines all intricate implementation details that are otherwise exposed in the pointcut definitions themselves. The application-specific model serves as a contract that base-program developers are licensed to provide and aspect developers can depend upon.

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

---

Email addresses: `johan.brichau@lifl.fr` (Johan Brichau), `andy.kellens@vub.ac.be` (Andy Kellens), `kris.gybels@vub.ac.be` (Kris Gybels), `kim.mens@uclouvain.be` (Kim Mens), `hirschfeld@acm.org` (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).

Preprint submitted to Elsevier Science *8 June 2006*

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 an *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 [3]. The improved modularity and separation of concerns [4], 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 [5]. This has also been referred to as the 'obliviousness' property of aspect orientation [6]. 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 behavioural 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 with 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.

Unfortunately, in many cases, defining and maintaining an appropriate pointcut is a rather complex activity. First of all, an aspect developer must carefully analyse 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 where 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.

The aspect-oriented programming language that is presented in this paper features an *open*, *logic-based* pointcut mechanism that allows to tie structural implementation conventions to explicit properties available for use in pointcut definitions. Our 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 [8]. In this paper, we further exploit the full power of the logic programming language for the definition of application-specific properties. In particular, we present an integration of the AspectS [9] and CARMA [10] aspect languages for Smalltalk. The result is an aspect-oriented programming language in which pointcuts can be defined in terms of an application-specific model that is asserted over the program. The application-specific model captures the structural conventions that are adhered to by the developers of the program and reifies them as explicit properties available for use in pointcut expressions. The model as well as the pointcuts are implemented using logic metaprograms in SOUL.

2 AspectSOUL

AspectSOUL is an integration of the CARMA pointcut language [10] and AspectS [9], 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 advices. AspectS is instead constructed as a framework. Pointcuts are written as Smalltalk expressions that return a collection of joinpoint descriptors. CARMA on the other hand, is a dedicated pointcut language based on logic programming. Such a dedicated query language naturally has some advantages for writing pointcuts, as pointcuts are essentially queries over a database of joinpoints. The integration of this logic-based pointcut language with AspectS further enforces the framework nature of AspectS by providing a fully 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 advices with the advantages of a dedicated and extensible pointcut language. In the remainder of this section, we introduce AspectS, CARMA and their integration: AspectSOUL. In subsequent sections, we focus on how the open-ended, logic-based pointcut language provides developers with an adequate means to overcome complex and hard-to-maintain pointcut expressions.

2.1 AspectS

Aspects are implemented in the AspectS framework as subclasses of the class `AsAspect`. Its advices can be implemented as methods whose name begins with `advice` and which return an instance of `AsAdvice`. There are two subclasses of `AsAdvice` that can be used to implement an around advice or a before/after advice. An instance can be created by calling a method which takes as 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 in a subclass of `WindowSensor` should be logged. The effect of the advice is implemented in the block passed to the `beforeBlock:` argument. When one of the methods specified by the pointcut needs to be executed, this block is executed right before the method's body. The block is passed a few arguments: the receiver object in which the method is executed, the arguments passed to 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 all subclasses of `WindowSensor` is filtered to only those that implement a method named `eventDoubleClick:`, an `AsJoinpointDescriptor` is then collected for each of these.

The qualifiers of an advice specify dynamic conditions that should hold if the advice is to be executed. These conditions are implemented as activator blocks: blocks that take as arguments an aspect object and a stack frame. The framework defines a number of activator 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. The latter are used for control-flow related restrictions, such as only executing

```
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.

an advice on a method if the same method is not currently on the stack. The activator 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 with the rest of the Smalltalk system by sending an explicit `install` message to the class. The `install` method collects all the advices in the class and executes their pointcut blocks to get the collection of joinpoint descriptors. The methods described by these descriptors are then wrapped in method wrappers [11], one for each advice affecting the method. The wrappers checks the activator blocks specified in their advice, passing them the aspect and the top stack frame (accessed using the `thisContext` reflective feature of Smalltalk [12]). If the activators do not hold, it simply executes the next wrapper, or the original method. If they all hold, the wrapper executes the advice's around, before and/or after block and the next wrapper in the proper order.

2.2 CARMA

CARMA is a pointcut language based on logic meta programming for reasoning about dynamic joinpoints. Unlike pointcuts in AspectS, CARMA pointcuts do not express conditions on methods, its joinpoints are representations of dynamic events in the execution of a Smalltalk program. CARMA defines a number of logic predicates for expressing conditions on these joinpoints, and pointcuts are written as logic queries using these predicates. It is possible to express conditions on dynamic values associated with the joinpoints. 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 meta programming [13], the underlying language of this library and CARMA is the SOUL logic language [13].

The SOUL logic language is based on Prolog [14], but has a few differences.

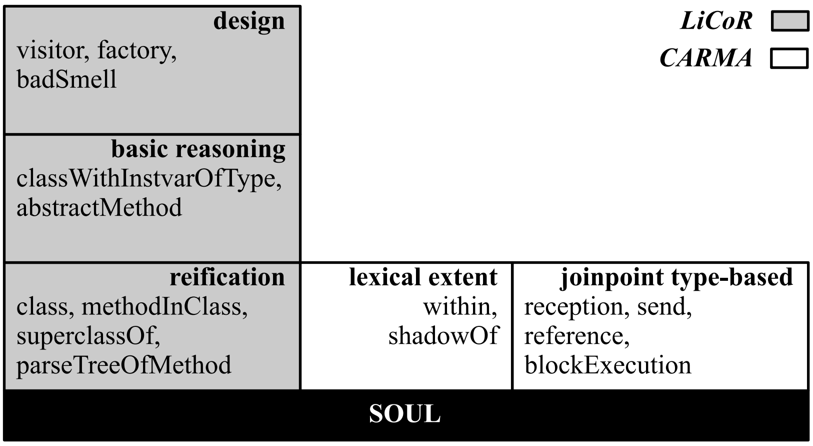

Fig. 2. Organization of, and example predicates in LiCoR and CARMA.

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 [15], allowing Smalltalk objects to be bound to logic variables and the execution of Smalltalk expressions as part of the logic program. The symbiosis mechanism is what allows CARMA to express conditions on dynamic values associated with joinpoints 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 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 joinpoint should meet in order to activate an advice, without specifying how those joinpoints 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 [10].

The predicates in CARMA and LiCoR are organized into categories, as shown in Figure 2. 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 the 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 and refactoring “bad smells”[17].

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. But 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 joinpoint is the result of executing an expression in a certain method. The “type-based” joinpoint predicates are the basic predicates of CARMA, they express conditions on certain types of joinpoints and basic data associated with those. An example is the `reception` predicate which is used to express that a joinpoint should be of the type “message reception”, which means it represents the execution of a message to an object. Besides the joinpoint, 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 joinpoint to the object in which it is executed. In the case of a reception joinpoint, this is the receiver of the message.

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

2.3 CARMA Pointcuts in AspectS

The AspectSOUL integration of CARMA in 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 joinpoints, in contrast with the more static joinpoints of AspectS. Also, because AspectS does not support aspects that intercept block execution nor variable accessing or assignation, these features of CARMA are not adopted in AspectSOUL.

An example of an AspectS advice with a CARMA pointcut is shown in Figure 3. This is an around variant of the first example advice, with a pointcut that has the same effect. The first condition in the pointcut specifies that `?jp`

```
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. 3. Example AspectS advice definition with a CARMA pointcut.

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

Fig. 4. A CARMA pointcut with a condition on a dynamic value.

should be a message reception joinpoint, where the selector of the message is `eventDoubleClick:`. The arguments of the message are bound to the variable `?args`, but this is not further used in the pointcut so this expresses that no conditions are put on the argument list. The second condition expresses that the joinpoint should lexically occur in a method with name `?selector` in the class `?class`. For a message reception joinpoint, 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 it explicitly calls the next wrapper or original method as it is an around advice.

Figure 4 gives an example of a CARMA pointcut which does express conditions on the arguments of a message reception. The first condition expresses that `?jp` should be a message reception joinpoint of the message `eventDoubleClick:`, where the argument list unifies with the list `<?event>`. The argument list should thus have on 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` should respond `true` to the message `isYellow`. Thus, this pointcut captures joinpoints 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 [18].

Two-phased weaving: The mapping done in the AspectSOUL advice subclasses to the original advice classes of AspectS involves the two-phase weaving model of CARMA. Because CARMA allows dynamic conditions and it is a Turing-complete language, it requires some advanced techniques to optimize weaving [10]. The mapping uses abstract interpretation [19] of the pointcuts to determine the methods which *may* lead to joinpoints matching the pointcut. For the pointcut of Figure 4, it determines that only executions of methods named `eventDoubleClick:` may match the pointcut. For these methods, `AsJoinpointDescriptor`s are generated and passed to the advice superclass. The effect block passed to the superclass is wrapped so that it at run-time executes the pointcut to check if the joinpoint actually matches it, only then does it execute the effect of the advice. As such, the mapping splits the static and dynamic parts of the pointcut as one would normally do in AspectS by specifying dynamic conditions as part of the advice's effect block. Currently, the pointcut is fully executed at run-time, including the static conditions, except if it doesn't include any dynamic conditions. The use of more advanced partial evaluation [19] to further optimize weaving has been demonstrated [20], but a full discussion of two-phase weaving and the use of partial evaluation is beyond the scope of this paper.

In the following sections, we discuss how pointcut definitions easily become rather complex to implement and maintain and how the logic-based pointcut language of AspectSOUL provides developers with a means to overcome these complexities.

3 Pointcuts based on Structural Conventions

In the development of 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. More importantly, 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. Next, we take a look at a couple of pointcuts which rely on these conventions to capture the execution of accessor methods. We demonstrate how, by implicitly capturing the notion of an accessor method using the coding conventions, the pointcut becomes more complex and easily suffers from the fragile pointcut problem.

In Smalltalk, clients are not allowed to directly access the instance variables of an object, but must rather 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 AspectSOUL:

```
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)
```

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 1 to 4 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. Lines 5 en 6 will intercept all messages which correspond to the naming convention.

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  class(?class),
2  methodWithNameInClass(?method,?selector,?class),
3  instanceVariableInClassChain(?var,?class),
4  returnStatement(?method,variable(?var)),
5  reception(?joinpoint,?selector,?args),
6  withinClass(?joinpoint,?class)
```

Lines 1 – 4 of the pointcut above select all methods which consist out of a single statement that returns the value of an instance variable. As with the previous pointcut, lines 5 and 6 capture all occurrences of these methods. 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 variable, 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 by a single statement returning the value of the variable.

3.2 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, the first four lines of both pointcuts express the coding convention, while the last two lines perform the actual selection of join points which are associated with the accessor methods. It is not imminently clear which part of the pointcut reflects 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 [21]. 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, are 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.

4 Application-specific Pointcuts and Models

We alleviate the problems associated with low-level pointcut definitions 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.

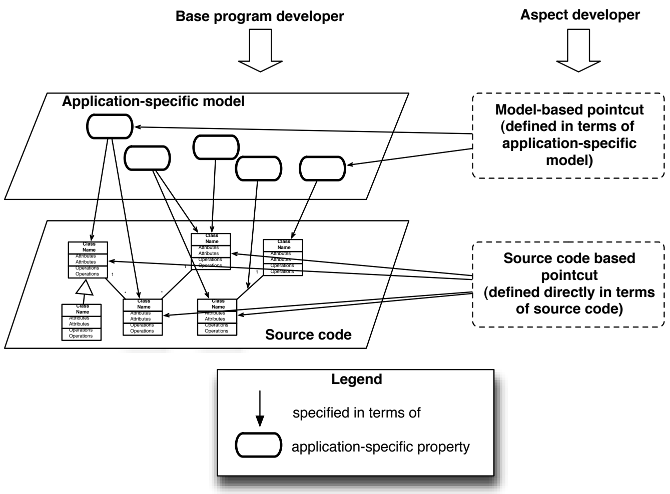

Fig. 5. Application-specific pointcuts are defined in terms of an application-specific model.

Figure 5 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.

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.

4.1 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 accessor predicate is implemented as follows:

```
accessor(?class,?varName,?varName) if
        class(?class),
        instanceVariableInClassChain(?varName,?class),
        methodWithNameInClass(?method,?varName,?class),
        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
```

4.2 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.

4.3 Model Specialisation

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 3.1. 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 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:[friends].
```

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:' ],<?trueBlock,?falseBlock>)),
        statementsOfBlock(<assign(?var,?varinit)>,?trueBlock),
        statementsOfBlock(<?var>,?falseBlock)
```

Such a model specialisation is 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 merely 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.

4.4 Property Parameters and Unification

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, it also exposes the actual variable that is accessed by the method [3]. More precisely, because a logic language does not make a distinction between arguments and return values, the variable that is accessed is also automatically a parameter of the accessor predicate. This also holds for all other parameters of the accessor predicate: they can act both as parameters as well as return values associated to the property. In essence, the logic language feature of 'unification' allows that we can automatically use the application-specific property that is defined by the accessor predicate in multiple ways, i.e. any argument of the predicate can be bound or unbound. A couple of examples are illustrated in the following code excerpt. Each line represents a separate use of the `accessor` predicate.

```
accessor(?class,?selector,?var)
accessor(Array,at:put:,?var)
accessor(?class,?selector,name)
```

The first line will retrieve all accessor methods and expose their class, methodname and accessed variable. The second line 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 on the last line 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 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:

---

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

- **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.

```
equals(?args,<?dragdropmanager>),
dragSource(?dragdropmanager,?source),
instanceOf(?source,FigureManager),
draggedObject(?dragdropmanager,?object),
instanceOf(?object,Line)
```

5.2 Refactorings

Refactorings are behaviour-preserving program transformations which can be used to remove bad smells from code by improving the structure of the application. 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:

```
refactoringInstantiation(?refactoring,?message,?args,?affectedentity) if
 refactoring(refactoring),
 methodWithNameInClass(?method,?message,?refactoring),
 instanceCreationMethod(?method),
 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 [7], 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 [8]. 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 [7], we merely extended the pointcut language with a single predicate that allows to query a conceptual model of the program, implemented using intensional views [?]. In this paper, the model consists of full logic predicates, resulting in an improved integration of the model and the pointcuts. In contrast, in [7], 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 effect:

6.1 Expressive pointcut languages

Some 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 [22]. In particular, Ostermann and Mezini have also identified how to build user-defined pointcut predicates using a logic language. EAOP [23] and JAsCo [24] 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 model-based pointcuts over application-specific models is to define pointcuts in terms of explicit annotations in the code [25,26]. 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. [27] 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 [28]. 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 developers. Finally, the logic paradigm offers adequate language features for the definition and extension of the application-specific model.

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.

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

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtoir, J. Irwin, Aspect-oriented programming, in: European Conference on Object-Oriented Programming (ECOOP), LNCS, Springer Verlag, 1997, pp. 220–242. URL `citeseer.ist.psu.edu/kiczales97aspectoriented.html`

[4] D. L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM 15 (12) (1972) 1053–1058.

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

[6] R. Filman, D. Friedman, Aspect-oriented programming is quantification and obliviousness, workshop on Advanced Separation of Concerns (OOPSLA) (2000).
URL `citeseer.ist.psu.edu/filman00aspectoriented.html`

[7] A. Kellens, K. Mens, J. Brichau, K. Gybels, Managing the evolution of aspect-oriented software with model-based pointcuts, in: To be published in Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Spring-Verlag, 2006.

[8] K. Gybels, J. Brichau, Arranging language features for more robust pattern-based crosscuts, in: Aspect-Oriented Software Development (AOSD), 2003.

[9] 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, 2003.

[10] K. Gybels, J. Brichau, Arranging language features for more robust pattern-based crosscuts, in: Proceedings of the Second International Conference of Aspect-Oriented Software Development, 2003.

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

[12] F. Rivard, Smalltalk: a reflective language, in: Proceedings of the Reflection Conference 1996, 1996.

[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, second edition Edition, John Wiley & Sons, 1995.
URL `http://www.ida.liu.se/ ulfni/lpp/copyright.html`

[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.
URL
`http://prog.vub.ac.be/Publications/2005/vub-prog-tr-05-13.pdf`

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

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

[18] 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.

[19] N. D. Jones, C. K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice Hall International, 1993.

[20] H. Masuhara, G. Kiczales, C. Dutchyn, Compilation semantics of aspect-oriented programs, in: G. T. Leavens, R. Cytron (Eds.), Foundations of Aspect-Oriented Languages Workshop at AOSD 2002, no. 02-06 in Tech Report, Department of Computer Science, Iowa State University, 2002, pp. 17–26. URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR02-06/TR.pdf`

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

[22] K. Ostermann, C. Mezini, M. Bockisch, Expressive pointcuts for increased modularity, in: European Conference on Object-Oriented Programming (ECOOP), 2005.

[23] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura, M. Südholt, An expressive aspect language for system applications with arachne, in: Aspect-Oriented Software Development (AOSD), 2005.

[24] W. Vanderperren, D. Suvee, M. A. Cibran, B. De Fraine, Stateful aspects in JAsCo, in: Software Composition (SC), LNCS, 2005.

[25] 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.

[26] G. Kiczales, M. Mezini, Separation of concerns with procedures, annotations, advice and pointcuts, in: European Conference on Object-Oriented Programming (ECOOP), LNCS, Springer Verlag, 2005.

[27] 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), ACM Press, 2005.

[28] 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.