Arranging Language Features for More Robust Pattern-based Crosscuts

Kris Gybels and Johan Brichau*
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussel, Belgium
{kris.gybels | johan.brichau}@vub.ac.be

ABSTRACT

A crosscut language is used to describe at which points an aspect crosscuts a program. An important issue is how these points can be captured using the crosscut language without introducing tight coupling between the aspect and the program. Such tight coupling harms the evolvability of the program and the reusability of the aspect. Current pattern-based capturing already offers a certain decoupling between aspects and the program but it may still suffer from what we call the *arranged pattern* problem. In this paper, we discuss this problem and present a logic-based crosscut language from which we distill what language features are beneficial to avoid this problem.

1. INTRODUCTION

The goal of Aspect-Oriented Programming (AOP) [16] is to modularize crosscutting concerns. Code that would otherwise have been spread throughout other modules in the program can now be encapsulated in an aspect. This is typically done by letting an aspect specify two things: how it influences other modules and exactly where or at what points it exerts its influence. Such points are generally referred to as join points and specialized *crosscut languages* are used to describe these points. Well known examples of such languages are AspectJ's pointcut language [13], HyperJ's composition language [26] and DJ's traversal strategies [25].

It is important that crosscut languages are expressive enough to accomplish the goals of AOP. The use of AOP should lead to better software because the localization of crosscutting concerns makes the program easier to read, maintain and evolve. However, simple crosscut languages often introduce tight coupling between the aspect and the points in the program it crosscuts. Such tight coupling hurts the evolvability of an aspect-oriented program and should thus be avoided.

*Both authors are research assistants of the Fund for Scientific Research, Flanders, Belgium (F.W.O.)

Especially enumeration-based crosscut languages lead to programs susceptible to evolutionary problems but the problem may also affect pattern-based crosscut languages. In some of the earliest AOP languages, such as COOL [19] and the first versions of AspectJ, crosscuts were formed by explicitly enumerating join points by name. This clearly introduces tight coupling between the aspect and the modules it crosscuts [29]: changing the program requires a review of the crosscut enumerations which conflicts with the idea of programs being oblivious [5] to the aspects applied to them. Pattern-based crosscutting, which can be found in most current aspect languages, attempts to solve some of the problems associated with enumeration-based crosscutting by expressing crosscuts by stating properties of join points instead of selecting them by name.

However a more subtle problem may occur in the pattern approach that also affects the evolvability of aspect-oriented programs: the use of naming conventions or other ways of structuring code in patterns so that these can be captured by crosscuts. Problems with evolution and obliviousness occur in this case as well because programmers need to keep these conventions in mind. We will refer to this problem as the *arranged patterns* problem. Note that this problem can also exist if the crosscut definition is separated from the aspect code (such as with AspectJ's abstract pointcuts or HyperJ's separate composition rules) which only decouples the crosscut definition from the aspect. To really avoid the situation in which aspect programmers rely on arranged patterns, crosscut languages need to be made expressive enough. More expressive crosscut languages allow to write more complex patterns. This allows to avoid the arranged pattern problem, leading to less coupling between aspects and the base program.

In this paper we take a look at a number of language features that can be beneficial for expressing better pattern-based crosscuts. We distill these features from a crosscut language that is based on a logic meta-language approach [33, 35, 23] with reification of static and dynamic join point properties as well as program structure and state.

In the next section we take a closer look at the different approaches to capturing join points and what problems can occur. In particular we discuss an example of implementing part of the observer design pattern as an aspect. We then discuss our crosscut language in full by breaking it down

into a number of distinct features. After this, we show how it can be used to implement a more evolution-robust version of such an observer aspect. We then discuss related work and our conclusions.

2. CAPTURING JOIN POINTS

In this section we consider some approaches to capturing the join points of an aspect which are used by most current AOP languages. An important criterium to evaluate the different approaches is how tight crosscuts are coupled to a specific base program. In the case of tight coupling, the aspect can be syntactically well separated from the base program, but changes to the base program can immediately require changes to the crosscut definition. This harms the evolvability of the program and is in direct contrast with the general goal of separation of concerns and AOP, which is to make programs easier to evolve.

To illustrate the different approaches we use the example of implementing an Observer [7] on `Buffer` objects as an aspect. Instances of `Buffer` are simple data structures supporting only operations to retrieve (`get`) and insert (`put`) elements, but we will consider what happens to the Observer aspect when new operations are introduced in the `Buffer` class. We will focus on the crosscut that expresses when changes happen in a Buffer object and thus observers need to be notified.

2.1 Enumeration

The first approach we consider is enumeration of the join points. This approach is used in the earliest incarnations of some aspect languages: COOL [19] and the earliest versions of AspectJ [14] are examples. In general, join points are referred to by name and a crosscut definition boils down to an enumeration of join points. As an example, consider the following AspectJ pointcut definition that enumerates the state changing methods on instances of `Buffer`:

```
pointcut changesState() :
 execution( void Buffer.put(int) ) ||
 execution( void Buffer.get() )
```

This pointcut would be used in an advice in the Observer aspect where the advice would notify all observers of the change.

Consider now what happens when the `Buffer` class is evolved. A method `putAll` might be added for example. This would unfortunately break the Observer aspect, requiring its `changesState` pointcut to be modified to also cover this new state changing method.

The problem with enumeration is that it tightly couples the aspect to a specific version of the `Buffer` class, thus making it not very robust to changes in that class: even simple changes can require a change to the crosscut definition. Another problem with this approach is that it does not scale for use in large software systems where overly long enumerations might be needed.

2.2 Pattern Matching

A better way to capture join points than using enumeration is by making use of patterns[1] which better describe the intended semantics of a crosscut. The problem with enumeration is that the aspect weaver is unable to figure out that new joinpoints that arise after the evolution of a program also fall under the intended semantics of a crosscut. In the observer example the intended semantics of the changesState crosscut is "all joinpoints where the state of an object is changed". While we cannot simply implement this using such a natural language description, we can aim at implementing it using an executable specification in the form of a pattern. Such a pattern would simply describe what is common to all the joinpoints that should match the crosscut. New crosscuts that fall under the intended semantics of the crosscut should thus be automatically included as well. This approach is found in most current AOP approaches such as AspectJ and JAC [27].

The term pattern-based crosscutting is closely related to the term property-based crosscutting [15] though there is an important difference. Property-based crosscutting replaces simple name-based enumerative crosscutting by associating a richer set of properties with each joinpoint. A crosscut then expresses the natural language semantics of a crosscut by describing through conditions the underlying pattern found in the properties of all the joinpoints matching it. While a lot of the flexibility of this approach depends on which properties are available it depends even more on the linguistic means available in the crosscut language to express the conditions. Hence we put forward the term pattern-based crosscutting to take the focus away from the properties in themselves and encompass the linguistic means of the crosccut language in the term as well.

As an example of pattern-based crosscuts, consider a rewrite of the `changesState` pointcut using AspectJ's pattern matching mechanisms:

```
pointcut changesState() :
  within(Buffer) &&
  (execution(* put*(*) ) ||
   execution(* get*(*) )
```

This version is somewhat more robust which is achieved by using a simple language feature: the wildcard. Using this definition, the new `putAll(int[])` method will be automatically captured by the pointcut. The use of the wildcard was already described by Kersten and Murphy as leading to better decoupling of aspects and programs [12].

But let's consider a few other possible changes. What if a method `collectFirstElements`, returning a collection with a specified number of elements from the buffer, or a method `wipe`, simply deleting all the elements in the buffer, is added? Again the pointcut does not capture any new execution joinpoints introduced by calling these new methods.

The problem with this pointcut is that it still does not really

[1]Note that the term pattern as used in this paper has nothing to do with design patterns [7]. We use the full term "design pattern" to refer to those kinds of patterns to avoid confusion.

describe what it means for a method to be "state changing", rather it relies on a naming convention. Furthermore this naming convention only needs to be followed to allow the pointcut to work: we could rename `collectFirstElements` to `getFirstElements` and `wipe` to `putWipe` but we would only be doing it for the sake of the Observer aspect and not because the name better expresses the intent of the method. Essentially the `Buffer` class would be telling the aspect where to crosscut and the aspect's pointcut would only be saying "do it wherever the Buffer class tells us to do it". We might then as well implement that with regular method calling instead of a crosscut [34].

Besides naming conventions there might be other ways in which methods could be tagged, such as using a specially named parameter or number of parameters, thereby essentially arranging a pattern in the code which is easy to write a pattern-based crosscut for. This situation is to be avoided because programmers will always need to remember to follow this convention when evolving the program. In general, we can speak of the *arranged pattern problem*.

2.3 Arranged Patterns

The general goal of our work is to allow for better decoupling of aspects and programs by implementing crosscuts as patterns using more flexible linguistic means in the crosscut language, a specific subgoal related to this is avoiding the arranged patterns problem. In general, *arranged patterns* occur because programmers purposefully structure their code so that afterwards this pattern can be captured by a crosscut expression, as happened in the observer example with the programming conventions. Even worse, programmers may start to refactor existing code simply to get to a pattern that can be picked out by an aspect. This syntactic (re)-arrangement violates the obliviousness property of aspect languages because this means that the syntactic structure of the base program implicitly determines where a particular aspect will crosscut the base code.

Arranged patterns cannot be avoided in all cases. However, crosscut languages should offer the capability to express as much patterns as possible. In an ideal crosscut language, any kind of pattern that can be identified by a programmer (perhaps in natural language) should be definable in the crosscut language. In literature, some illustrations and descriptions of people avoiding this arranged pattern problem can be found. For example, Lippert and Lopes [17] have identified this problem in a previous version of AspectJ when they used aspects to separate exception handling from the core program. In some particular cases, they lacked sufficient expressiveness to define the necessary crosscuts and had to resort to a simple enumeration of join points. Also, Hugunin [10] has expressed the desire for more expressive pointcut definitions in which we can describe crosscuts by the properties in which we are interested. The `changesState` crosscut should then be expressed as 'all method executions that change the state of the `Buffer` object'. Douence et al. [4] also expressed their need for more expressive crosscut languages, because they had to pollute aspect code with crosscut 'bookkeeping' code. In their work on applying AOP to an OS kernel implementation, Coady et al. [3] describe that they had to refactor some code into new functions to allow the definition of a crosscut that matched the necessary code.

We have argued that purposefully structuring the base code as an indicator for aspects to 'crosscut here' is to be avoided. This does not mean that a crosscut definition should not be allowed to use programming conventions. On the contrary, many conventions which can be considered a natural part of the base program's development can be very useful to define crosscuts. As such, we do think it is useful for crosscut programmers to be able to use such (static) information.

3. BETTER CROSSCUTS WITH BETTER LANGUAGE FEATURES

A pattern-based crosscut language should allow an AOP programmer to write a crosscut as an executable specification in the form of patterns as close to the intent of the crosscut as possible to make the crosscut robust to program evolutions.

The occurence of the arranged pattern problem seems to suggest the need to further enhance pattern-based crosscut languages. As discussed, pattern-based crosscut languages are founded on two ideas and thus their expressiveness also hinges on these two: the joinpoints and associated properties on one hand and on the other hand, the linguistic means for describing a pattern as conditions on the properties. We will focus on the second in this paper and make an attempt at providing a list of language features necessary, though maybe not sufficient, for improving the expressiveness of crosscut languages. We do this by considering a crosscut language we developed and distill the language features that seem most interesting for writing better pattern-based crosscut expressions.

4. THE CROSSCUT LANGUAGE

In this section we introduce our own crosscut language. The language is in essence a logic programming language, based on Prolog [6], in which predicates are provided which allow the writing of crosscut expressions. The reader need not be familiar with logic programming as the purpose of this paper is exactly to explain what features of our crosscut predicates and logic programming itself are interesting for writing pattern-based crosscut expressions.

The crosscut model of our language is a dynamic one, based very much on that of AspectJ, in which the join points are related to key events in the execution of an object-oriented program. The object-oriented language we have used as a base language is Smalltalk [8]. Thus there are five types of join points: message receptions by an object, message sends by an object, the accessing and updating of an object's state and the execution of code blocks. For each type of join point there are different types of properties associated with each join point. Many of these are dynamic properties.

For each type of join point there is a predicate in the language, these predicates are shown in figure 1. Each predicate takes at least one argument for which we always use the variable **?jp**. The value in this variable will be bound to a representation of the join point. Each predicate further takes some arguments for some basic properties associated with the join point.

- reception(?jp, ?selector, ?arguments)
  Used to express that **?jp** is a message reception join point, where the message with selector **?selector** is received with the arguments in the list **?arguments**.
- send(?jp, ?selector, ?arguments)
  The join point **?jp** is a message send join point where the message with selector **?selector** is sent and passed the arguments in the list **?arguments**.
- reference(?jp, ?varName, ?value)
  The join point **?jp** is a reference join point where the variable with name **?varName** is referenced at the time it has the value **?value**.
- assignment(?jp, ?varName, ?oldValue, ?newValue)
  The predicate used for assignment join points, where **?varName** is the name of the variable being assigned, **?oldValue** is the value of the variable before the assignment and ?newValue is the value of the variable after the assignment.
- blockExecution(?jp, ?args)
  The join point **?jp** is a Smalltalk block execution join point, where **?args** is a list of arguments that were passed to the block.

Figure 1: List of basic predicates for capturing crosscuts

Crosscut expressions are written as logic queries about join points. The query expresses the conditions a join point must meet in order to match the crosscut expression. Though in this paper we focus on the crosscut language of our aspect language, we show an example of how crosscut expressions are incorporated into advices:

```
before
  ?jp matching reception(?jp, test, ?arg)
do
  Transcript show: 'executing test'
```

This advice expresses how and when to log a statement to the transcript. The body or the "how" part of the advice is implemented in our aspect language as Smaltalk code. The crosscut or "when" part of the advice is the logic query reception(?jp, test, ?arg), which captures all receptions of the message test by any object.

In the next sections we take a closer look at some of the features of our crosscut language. While there is not much difference between our basic crosscut predicates (figure 1) and the equivalents from AspectJ at first sight, much of the extra flexibility comes from the fact that they are logic predicates in a logic language. Some of the features we will discuss next thus come from logic programming: unification, rules and recursion. Others are made available through other predicates in the language: the ability to reason about more than just joinpoints themselves, but also their properties, the objects in the program and, through shadow join points, the static structure of the program. In a later section we will apply all these features to the Observer example.

Feature: Unification

Logic programming languages offer a simple, yet powerful basic pattern matching mechanism in the form of unification. Unification is the process of making two structures equal by filling in the corresponding "holes" in either one with the values in the other.

A simple use of unification is as a wildcard pattern-matching mechanism, as illustrated in the following crosscut expression, note that variables are names beginning with a question mark:

```
?jp matching
  reception(?jp, ?methodName, <?firstArgument, 5>)
```

This crosscut expression would match any reception join point, regardless of the name of the message as long as it has two arguments of which the first can have any value and the second should be the number 5.

The real value of unification however lies not just in its emulation of more simple pattern matching mechanisms but in its effect of binding variables, in the next section this will allow us to put more complex conditions on join points. Unlike wildcard matching there are variables involved in unification which get bound to values. For example in the case above the **?methodName** variable will be bound to the name of the message sent which is one of the properties associated with the join point held in the variable **?jp**. Also, because of the unification of the actual argument list with the pattern <?firstArgument, 5> the variable **?firstArgument** will be bound to the first actual argument sent with the message.

Feature: Reasoning about Properties

A crosscut language should allow one to put more complex conditions on join point properties than simply requiring the property to have a specific value. In our language, unification already provides the possibility to match properties to specific values, for example the '5' value for the message argument in the previous crosscut, but it also allows these properties to be bound to variables. Because these variables can be used as arguments to other predicates in the crosscut we can put more complex conditions on properties.

Simply being able to pass around variables is not enough of course, additional predicates are also needed to actually express conditions on the properties. Some of these predicates in the language are simply standard Prolog predicates for reasoning about numbers, lists, strings, etc. We have also introduced predicates which reify the objects of the crosscut program and their state.

There are three object reifying predicates. One is inObject which provides access to the "context object" property of a join point. The "context object" is the object in whose context a join point originates. Another predicate is objectVariable providing direct access to the state of objects. The last predicate is objectResponse which can be used to express that an object should respond to a certain message with a certain response.

A simple example of using the same variable twice in one

crosscut expression is one that would match all receptions of messages with two arguments that are the same:

```
?jp matching
  reception(?jp, ?selector, <?arg, ?arg>)
```

As an example of using a standard Prolog predicate we use the member predicate in a variant of our earlier example:

```
?jp matching
  reception(?jp, ?selector, <?firstArgument, 5>),
  member(?selector, <getProperty, saveToDatabase>)
```

The changed crosscut expression would now match only a subset of the join points it matched earlier, namely those that have as message property either **getProperty** or **saveToDatabase**. The member predicate is a standard Prolog predicate expressing that its first argument should be an element in the list in its second argument.

As an example of using both the object reifying predicates and some standard Prolog predicates for dealing with numbers we consider the capturing of imminent bankrupties on account objects. The crosscut below captures receptions of the message **withdraw:** with the amount to withdraw as an argument and checks whether the current balance of the account object minus that amount would drop the balance below zero:

```
?jp matching
  reception(?jp, withdraw:, <?amount>),
  inObject(?jp, ?obj),
  objectVariable(?obj, balance, ?balance),
  difference(?balance, ?amount, ?afterWithdrawal),
  below(?afterWithdrawal, 0)
```

Feature: Link to Shadows

While we prefer a dynamic crosscut model because it can lead to the inclusion of dynamic values in join point properties, a crosscut language should still also provide a more static model of a program to aspects. The reason is the same: to get as much properties to match on as possible. As discussed earlier, programming conventions are harmful to the evolvability of AOP programs if they are arranged, but naturally occurring programming conventions are useful to AOP. The static model is needed to be able to express crosscuts making use of these conventions.

In our language we provide the static model as a link from the dynamic model to *join point shadows*. Join point shadows [21] are expressions in the base program that on execution lead to the dynamic join points. An example of this is a message send statement that every time it is executed results in a different message send join point, but all of these have that statement as its shadow. A predicate shadow can be used to link a join point to its shadow and vice-versa.

Join point shadows also have properties associated with them, some of which can be other join point shadows. The properties are typically the constituents of the statement or expression. For example a message join point shadow has as properties the expressions that are used as its arguments. We stress that here lies the difference with the dynamic message send join points, the properties are not the actual values passed around as arguments dynamically but the code that describes these values. It is also important to note that these properties are expressions and can thus be the shadows of other join points.

We now show an example of how join point shadows can be used to write crosscuts based on a programming convention in the code. We use the example of a programming convention well known to Smalltalk programmers: exception handling. In Smalltalk, exception handling is not a primitive construct of the language but is instead done by using regular message sending. To catch exceptions occurring during the execution of a code section, one sends the message **on:do:** to that section of code. A code section that understands messages is a feature of Smalltalk known therein as blocks. The **on:do:** message takes two arguments: the first is the class of the exception that should be caught and the other is another block that should be executed when an exception is caught. The second block is thus the exception handler.

We now want to capture the execution of exception handlers in our crosscut language:

```
?jp matching
  blockExecution(?jp, <?exception>),
  shadow(?jp, ?bs),
  enclosingShadow(?bs, ?ms),
  messageShadow(?ms, on:do:)
```

The above crosscut expresses the following: **?jp** should be a join point which is the execution of a block with one argument. The shadow **?bs** of that join point should have as its enclosing join point shadow **?ms** and that shadow should be a message send expression with **on:do:** as message. Some new concepts were introduced here: the shadow of a block join point is the expression that created that block and the enclosing shadow of a shadow is in this case simply the expression to which that block shadow was used as an argument.

Besides enclosingShadow there is another predicate for relating join point shadow to specific static contexts: the shadowIn predicate can be used to specify that a join point shadow should lie in the context of a method of a class.

Additionally other predicates are provided which reify the static structure of the base program. Some of the more important are: class, method, subclass and hierarchy. The latter two are used to to specify the interrelationship between classes. An example demonstrating some of the possibilities this offers:

```
?jp matching
  shadow(?jp, ?sp),
  shadowIn(?class, ?name, ?sp),
  hierarchy(?class, Collection),
  hierarchy(Dictionary, ?class)
```

The above crosscut captures all join points which lie in the context of any method which in the class hierarchy is below the Collection class but above the Dictionary class.

Feature: Reusable Parameterized Rules

Once we have the capability of writing complex patterns in crosscuts we need to be able to turn these into reusable elements of the language. The previously shown crosscut which matches exception handler executions for example is one that could be generally useful in different aspects. Reusability of course requires a form of parameterization to make the definition of the pattern applicable to different situations. In the exception handler example we would want to parameterize the exception class, the actual exception object etc. Logic languages provide rules for this purpose, rules define new predicates in the language.

There are some particularities about how logic rules work as opposed to the parameterized reusability mechanisms of other types of languages, such as procedures in procedural languages, that make logic rules especially suited to pattern-based crosscutting:

in/out parameters: a logic rule does not make a distinction between arguments and return values, instead parameters are bound through unification.

multiple solutions: like a crosscut expression, a rule expresses a set of elements that meet certain conditions. Thus rules match better with crosscuts than procedures etc. would.

multiple implementations: for each predicate there can be multiple implementations. This is mostly useful for expressing variations on a pattern.

In the application section on the observer aspect we will demonstrate the use of rules and in particular how multiple implementations for a single rule can be useful.

Feature: Recursion

A final feature to consider is the use of recursion in the language. This fully turns it into a computationally complete language, which is further discussed in a later section. Recursion is useful for capturing patterns that are defined recursively. We've found this to be mostly useful in combination with join point shadows to describe programming patterns which depend on detecting sequences of method calling chains. We've used this as a solution to the observer example of problems with pattern robustness we've discussed earlier. The solution is presented in the next section as an application of our crosscut language.

5. EXAMPLE: A MORE ROBUST CROSSCUT FOR OBSERVERS

The observer pattern problem was to capture in a crosscut when observers are to be notified of changes to an object. It would be simple to write a crosscut that captures state updates using the assignment predicate, but this is not a good solution. The problem is that methods can do multiple state updates with the object being in an inconsistent state until they are all completed. Observers should thus only be notified at the end of methods, which is typically what is found in non-AOP implementations of the observer design pattern: the updates are sent at the end of methods. Another point to take into account, previously discussed by Brichau et al. [2], is that when a method recursively calls other methods on the object itself which also perform state updates it is preferable to do the notification only at the end of the first method to avoid unnecessary overhead from observers.

What is needed is a way to detect the recursive pattern of methods changing state. This is quite easily done in our language by defining a simple rule over join point shadows:

```
Rule changesState(?class, ?methodName) if
  shadowIn(?class, ?methodName, ?sp),
  assignmentShadow(?sp, ?variable)

Rule changesState(?class, ?methodName) if
  shadowIn(?class, ?methodName, ?sp),
  messageShadow(?sp, ?rcvr, ?msg),
  selfReceiver(?rcvr),
  changesState(?class, ?msg)
```

Two rules for the predicate changesState are defined. This shows how rules are used, as well as the use of shadow join points in both rules and recursion in the second. The first rule determines that a method is a state changing method if it possibly does an assignment and the second determines that a method is also a state updating method if it does a self call to a method that is in turn a state updating method. Using this predicate an advice for notifying observers is easily defined:

```
after ?jp matching
  reception(?jp, ?msg, ?args),
  inObject(?jp, ?obj),
  objectClass(?obj, ?class),
  changesState(?class, ?msg),
  not(caller(?jp, ?obj))
do
  observers notify
```

The first four conditions in the crosscut capture all the message reception points which result in the invocation of an updating method. The last condition further restricts this to only those messages not sent by the object itself so that notification is only sent after the first message that was sent from outside to the object. The advice body simply notifies all observers registered with the aspect in which the advice would normally be defined. Note also that the first condition uses a basic joinpoint predicate, the second and third use "reasoning about properties" predicates, the fourth uses the changesState predicate for which we defined rules above and in the last condition logic negation is used along with a non-basic property predicate.

This version of the crosscut is close to the intended state updating semantics and is thus robust towards additions of

new methods in a Buffer class such as those described in section 2. Furthermore, the crosscut is actually fully decoupled from the Buffer class and works for other classes as well.

The changesState predicate and the above advice only deal with local changes to objects, we leave dealing with changes to subobjects to another part of the Observer aspect. In the Buffer example we have so far assumed the Buffer is implemented as an array and elements are stored into this array by assigning to it directly. If instead the Buffer's data element keeping is implemented using a more advanced type of container where storing elements is done by sending a message to it the above advice alone is not sufficient. To also handle this case we let an aspect instance of the Observer aspect associated with one object receive notifications from the aspect instances associated with that object's subobjects and forward these to its own observers. To avoid the sending of too many notification messages we have also attempted more elaborate versions of the Observer aspect where observers can explicitely specify which data querying methods they use on an observer and the aspect will then only monitor those changes which can have an effect on the result of those data querying methods. Interested readers can find a more elaborate discussion in the first author's licentiate's dissertation [9].

6. LANGUAGE IMPLEMENTATION

One issue left to address is how a powerful crosscut language can be efficiently supported by an aspect weaver. A naive weaver implementation straightforwardly derived from the semantics of our crosscut language would produce programs with unacceptable performance. To overcome this problem we used some techniques and ideas found in abstract interpretation and partial evaluation [11]. We explain how our current weaver works and what further optimizations we can still introduce.

6.1 Current Status

A naive weaver implementation would be one that operates fully at runtime. Runtime weaving is necessary because of the use of dynamic join point properties. The weaving process would be based on the simple semantics of a crosscut language: at every join point, or key execution step in an OO program, check whether any crosscut expression matches the point. But this would clearly bring program execution to a crawl.

We've optimized the naive weaver implementation by taking into account that many crosscuts can be easily eliminated based solely on the conditions they put on static properties. Our weaver thus operates in two phases: a compilation phase and a run-time phase. In the compilation phase it acts as a source transforming weaver by using each join point shadow in the program as a partial join point description with which it evaluates all crosscut expressions. The crosscut evaluation can either result in a definitive match, a definitive mismatch or a possible match meaning the crosscut depends on dynamic properties. In the match or possible match cases the source is transformed to include a call to the run-time phase of the weaver. In the possible match case the weaver will re-evaluate the crosscut at runtime using a full join point description to check the matching of crosscuts. The process is described in more detail in the first author's licentiate's dissertation [9].

6.2 Future Work

Besides eliminating crosscuts the remaining ones can also be reduced to the conditions that depend on dynamic properties. While currently not implemented in our weaver, this can be done by using program specialization [11]. In essence this would lead to an aspect weaver that performs as well as one for an aspect language with a purely static crosscut model where advices can still constrain their applicability based on dynamic program properties, we'll further discuss this comparison in the context of AspectJ in the related work section.

A final intricacy of our language is that it allows a form of dynamic weaving [28]. It is simple for example to write a crosscut expression capturing message points where the exact name of the message depends on the value of an object's instance variable. Partial evaluation and specialization are of no help here but we expect to find efficient solutions for this type of crosscut in the dynamic weaving research. So far we have not yet experimented with this feature of our crosscut language and thus did not need to implement such optimizations.

7. DISCUSSION

7.1 Crosscutting: Computing or Describing?

In this paper we have broken down our crosscut language into a number of features which may be interesting for other crosscut language designers to pick up but let us take a look now at what our language as a whole is. All features combined our language is *a computationally complete logic language targeted at describing crosscuts and making use of full static and dynamic program reification.*

This raises a few points to consider, the most important of which we consider why we used a computationally complete logic language. A related important question to consider is whether crosscutting should be computed or described. The reason for using a computationally complete language is that we did not want to limit the expressiveness of our language from its conception. By further combining this computational completeness with extensive reification of static and dynamic program properties we've created a flexible crosscut language. One may wonder whether in doing so we have not reintroduced the complexities of full meta programming into AOP. We however argue that most of the difficulties with using full meta programming stem from using meta programming in an imperative type of language where applying program transformations becomes a job of juggling them so that a correct result is achieved, the improvement provided by (low-level) AOP is its more specification-like approach: most of the transformation work is absorbed into the weaver with the AOP programmer specifying what and where to apply a transformation and leaving the how to do it to the weaver. Further clarity and understandability of aspects is provided by also clearly splitting the what and where and by possibly using specialized languages for specifying either. The need for an understandable specification of a crosscut is also the motivation for the use of a declarative language.

7.2 Related Work

In this section we will discuss some other crosscut languages and other work done on improving the expressiveness of crosscut languages.

AspectJ's crosscut language has gone through a number of evolutionary steps. The earliest versions used a simple enumeration based scheme for capturing crosscuts: the enumeration of the names of methods whose invocation should be captured by the aspect [20]. Later versions of AspectJ evolved to the use of a dynamic crosscut model and a pattern-based language [13]. Until recently many of the features we've discussed or equivalents seemed to be missing from AspectJ. Especially variable binding was troublesome: while it was possible to bind values to variables, these only served to expose pointcut values to advices and could not be used within a pointcut. This is important as variables are a key feature in our language to support most of the other features such as parameterized rules and conditioning of properties. The recently introduced "if" construct in the language has however put an interesting twist on matters. The "if" construct in AspectJ's crosscut language was introduced as a way of conditioning join point properties as it allows one to use boolean Java expressions in a pointcut and these can make use of any variables bound in the pointcut. It is still not possible however to define pointcuts that take parameters instead of exposing them nor can this be done with the primitive pointcuts and other features we discussed are lacking as well.

Reverse Graphics [22] was one of the earliest aspect language but surprisingly it used an expressive language for picking out join points. This is not to say it had an expressive crosscut language. It is difficult to say exactly what the crosscut language of RG is as it did not yet make clearly the now more common distinction between the when or where and how parts of an aspect's influence. Rather the RG weaver called aspects which were implemented as procedures taking join points as arguments. The aspect procedures could then decide whether or not to manipulate these join points and how to manipulate them if so. Thus when/where and how where implemented in a single procedure in the same language. This clearly allowed for flexible ways to decide where to crosscut a program, but exactly why the different features of the language were useful for expressing this was not yet studied.

Another example of using an existing programming language as the basis for a crosscut language can be found in the work of Dounce et al. [4] who used a functional language. They also discussed the pattern matching basis of a crosscut language, though they used this term mostly to refer to the pattern in a sequence of join points rather than as the pattern underlying a collection of join points as we did in this text. Nevertheless, we found similarities between our use of logic queries to describe a set of join points and their use of functions to select the join points belonging to a sequence. A dynamic crosscut model implicating dynamic join point properties was also used in combination with this computational expressiveness, unfortunately little demonstration was given of the possibilities offered by this combination. The additional use of including a reification of a static crosscut model as join point properties was also not considered though in earlier work the use of a functional transformation language with a purely static crosscut model was explored [30]. Whether the functional or logic paradigm is better suited as the crosscut language basis can be left to debate, though traditionally logic languages have been used more for pattern descriptions and functional languages for transformations.

An important example of such a use of logic languages can be found in Minsky's works on law-based systems [24]. Logic rules are used to describe 'laws' that enforce global, and crosscutting, properties of a software system. The laws can be described in terms of program-execution events (such as message-sends), which makes this system remarkably similar to our logic crosscut language. Indeed, Minsky's law-based systems can not only enforce the application of laws in the system, the laws themselves can trigger additional behaviour.

The use of partial evaluation as a weaver implementation technique to make crosscuts with dependencies on dynamic properties feasible was also discussed by Masuhara et al. [21]. However the crosscut language used was taken from an AspectJ version before the "if" construct was introduced so the same remarks about the flexibility of the language apply. Though a trick often applied by AspectJ programmers to gain some of the flexibility of our crosscut language was also made feasible by partially evaluating advices: the use of the computationally complete advice language to express part of a crosscut. This remains however a trick and not a desirable solution as it detracts from readability by mixing the crosscut (where/when) and the advice's body (what) again. For example, a crosscut specification such as our rules for changesState that is beyond the abilities of a simple crosscut language would have to be implemented in the body of an advice and is no longer reusable in different crosscuts.

7.3 Aspect-Oriented Logic Meta Programming

Our work is founded on De Volder's original proposal for using Logic Meta Programming as a basis for Aspect-Oriented Programming [33]. This work concentrated mostly on the use of LMP as a framework for writing source transformation based weavers. Our work thus extends this use of LMP by showing that it can also be a good basis for crosscut languages. We note however that our weaver for this language is not itself implemented in LMP. We intend to combine these two uses of LMP by offering our language as one of the aspect languages in Brichau's work on an open weaver in which he explores the rapid construction of weavers for user-defined composable aspect languages [1].

8. SUMMARY AND CONCLUSION

Crosscut definitions should avoid tight coupling of an aspect to the base program. Even pattern-based languages can suffer from coupling through the arranged pattern problem. Advanced crosscut languages should weaken the coupling of the aspect to the base program and hence, provide crosscuts that are more robust towards evolution. Avoiding this problem requires an expressive crosscut language that offers a powerful mechanism to describe the underlying pattern of the points crosscut by an aspect.

In this paper we distilled some key features of a logic-programming-

based crosscut language that allow the writing of more advanced pattern-based crosscuts. We can summarize this paper as a set of "lessons learned" for three interested parties. We advise aspect programmers to avoid using arranged patterns and make patterns more robust. This requires assistance from crosscut language designers who can add certain language features to crosscut languages to allow the writing of better pattern-based crosscuts. And finally all depends on weaver implementers to use some more advanced techniques to make the powerful crosscut languages possible.

9. REFERENCES

[1] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages using logic metaprogramming. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of GPCE Conference*, LNCS, pages 110–127. Springer-Verlag, 2002.

[2] Johan Brichau, Wolfgang De Meuter, and Kris De Volder. Jumping aspects. In Tarr et al. [32].

[3] Yvonne Coady and Gregor Kiczales. Exploring an Aspect-Oriented approach to OS code. In Tarr et al. [31].

[4] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. Technical Report 01/3/INFO, Ecole des Mines de Nantes, 2001.

[5] Robert E. Filman. Aspect-oriented programming is quantification and obliviousness. In Tarr et al. [31].

[6] Peter Flach. *Simply Logical*. John Wiley & Sons, 1994.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable Object-Oriented software*. Addison-Wesley, 1995.

[8] Adele Goldberg and Dave Robson. *Smalltalk-80: the language*. Addison-Wesley, 1983.

[9] Kris Gybels. Aspect-Oriented Programming using a Logic Meta Programming language to express cross-cutting through a dynamic joinpoint structure. Licentiate's thesis, Vrije Universiteit Brussel, 2001.

[10] Jim Hugunin. The next steps for aspect-oriented programming languages (in java). In Workshop on New Visions for Software Design and Productivity: Research and Applications, 2001.

[11] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[12] Mik Kersten and Gail C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In *Proceedings of the ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 340–352. ACM, 1999.

[13] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP 2001*, LNCS. Springer-Verlag, 2001.

[15] Gregor Kiczales, Jim Hugunin, Mik Kersten, John Lamping, Cristina Lopes, and William G. Griswold. Semantics-based crosscutting in aspectj. In Peri Tarr, Anthony Finkelstein, William Harrison, Bashar Nuseibeh, Harold Ossher, and Dewayne Perry, editors, *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering at ICSE 2000*, 2000.

[16] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European conference on Object-Oriented Programming*. Springer-Verlag, jun 1997.

[17] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering*. ACM Press, 2000.

[18] Cristina Lopes, Gregor Kiczales, Bedir Tekinerdogan, and Wolfgang de Meuter, editors. *International Workshop on Aspect-Oriented Programming at ECOOP*, 1998.

[19] Cristina Videira Lopes. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Xerox PARC, 1997.

[20] Cristina Videira Lopes. Recent developments in AspectJ. In Lopes et al. [18].

[21] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In Gary T. Leavens and Ron Cytron, editors, *Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, number 02-06 in Tech Report, pages 17–26. Department of Computer Science, Iowa State University, 2002.

[22] Anurag Mendhekar, Gregor Kiczales, and John Lamping. RG: A case study for Aspect-Oriented Programming. Technical Report SPL97-009 P9710044, Xerox Palo Alto Research Center, 1997.

[23] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. In *Proceedings of the 13th SEKE Conference*, pages 236–243. Knowledge Systems Institute, 2001.

[24] Naftaly Minsky. Law-governed regularities in object systems. *Theory and Practice of Object Systems (TOPAS) (John Wiley)*, 2(4), 1996.

[25] Doug Orleans and Karl Lieberherr. Dj: Dynamic adaptive programming in java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* , Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[26] Harold Ossher and Peri L. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of ICSE 2000*, pages 734–737, 2000.

[27] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. Jac: A flexible and efficient solution for aspect-oriented programming in java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* , Kyoto, Japan, September 2001. Springer Verlag.

[28] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for Aspect-Oriented Programming. In Gregor Kiczales, editor, *1st international conference on Aspect-Oriented Software Development*, april 2002.

[29] Andreas Speck, Elke Pulvermüller, and Mira Mezini. Reusability of concerns. In Tarr et al. [32].

[30] Mario Südholt and Pascal Fradet. Aop: towards a generic framework using program transformation and analysis. In Lopes et al. [18].

[31] Peri Tarr, Lodewijk Bergmans, Martin Griss, and Harold Ossher, editors. *Proceedings of the Workshop on Advanced Separation of Concerns at OOPSLA 2000*, 2000.

[32] Peri Tarr, Maja D'Hondt, Christina Lopes, and Lodewijk Bergmans, editors. *International Workshop on Aspects and Dimensional Computing at ECOOP*, 2000.

[33] Kris De Volder. Aspect-Oriented Logic Meta Programming. In *Proceedings of the Second International Conference on Metalevel Architectures and Reflection*, volume 1616 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[34] Detlef Vollmann. Visibility of join-points in AOP and implementation languages. In Pascal Constanza, Gunther Kniesel, Katharina Mehner, Elke Pulvermuller, and Andreas Speck, editors, *Second Workshop on Aspect-Oriented Software Development*, 2002.

[35] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA '98*, 1998.