Using a logic language to express cross-cutting through dynamic joinpoints.

Kris Gybels*
Programming Technology Lab - Vrije Universiteit Brussel
kris.gybels@vub.ac.be

Abstract

Aspect languages are generally comprised of a cross-cut language and an action language, with the cross-cut language describing joinpoints in a cross-cut program. It is desirable for the cross-cut language to be powerful enough so that cross-cuts and actions can be clearly separated and so that the cross-cut language is extendible with new types of joinpoints. The latter is especially needed for languages such as Smalltalk where there are few programming constructs and programming conventions are common practice. Therefore we present a cross-cut language which is based on a logic meta language and which uses a dynamic joinpoint model.

1. Introduction

An aspect language is a programming language that provides constructs for modularizing concerns that cross-cut a system's modularity in a principled way [8]. The idea is to allow for the separate implementation of cross-cutting concerns, which are also known as aspects, and the main modules of a system. This improves the overall comprehensibility and evolvability of a system's implementation. Combining aspects and modules is done by an aspect weaver.

Aspect languages have evolved from being concern-specific to concern-generic [7]. Concern-generic aspect languages have the benefit that a single aspect language can be used to capture many different concerns. With concern-specific languages a new aspect language and weaver need to be developed for every new type of concern. Concern-specific languages have the benefit however of providing constructs which capture a concern more clearly than is possible with a concern-generic language.

Many concern-generic aspect languages are based on a low-level approach where an aspect language consists of two separate languages: a *cross-cut language* and an *action language*. The cross-cut language is used to describe points in modules that are cross-cut by an aspect. These points are also refered to as joinpoints, because they are the points at which the weaver will join the aspects and modules. The action language then describes the aspect's influence on the modules at those joinpoints.

In order for a low-level aspect language to be truly generic, both the cross-cut language and action language it consists of should be generic. Making both of the languages generic and not just one of either improves the aspect programmer's ability to clearly separate cross-cuts and actions. We will not go into this any further as the benefit of such separation was previously pointed out by Douence et al. [9].

In this paper we concentrate on the genericity of the cross-cut language, as action languages are already often based on an existing generic programming language. We advocate the use of a cross-cut language based on logic programming to achieve genericity. Logic programming languages have two properties that make it interesting for expressing cross-cutting. The first is that logic languages have a declarative or descriptive nature: cross-cutting is preferably described rather than computed, which makes it easier to read an aspect. Descriptive languages have been used in a number of low-level aspect languages. In most cases however, genericity of the language was limited by the fact that it was not Turing Complete. This is the second interesting property of logic languages in that they combine declarativeness with Turing Completeness. This allows to perform arbitrary computation - or in logic terms *reasoning* - in order to determine points of cross-cutting.

Genericity in the cross-cut language also depends on the joinpoints the aspect weaver provides. Diverse types of joinpoints have so far been used, but in general a distinction can be made between static and dynamic joinpoints. Static joinpoints are defined in terms of points in a module's source code, while dynamic joinpoints are defined in terms of points in a module's execution. An example illustrating the difference is the use of method *definitions* versus method *executions* as joinpoints. The use of dynamic joinpoints allows for more powerful cross-cut expressions as data that is only available at runtime, such as arguments passed to a method, can be used in the description of cross-cutting.

We constructed an aspect language for Smalltalk to explore the ideas described above. The language was named Andrew [3]. Andrew's joinpoint model and core cross-cut language are based on the ones found in As-

*Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)

pectJ[1] [5].

The particular logic language we used in our approach is a logic *meta* language. The meta aspect of the language refers to its capability for reasoning about programs. We show how this property further enhances the genericity of the cross-cut language by showing how it can be used to extend the cross-cut language with new types of joinpoints without having to change the weaver.

In the remainder of this paper we will first explain the cross-cut language. Then some experiments with the language are shown. In the final sections we discuss work that is related to ours, future work that we need to do and our conclusions.

2. Expressing cross-cutting in Andrew

In this section we introduce our cross-cut language which is embedded in a logic language. First, the joinpoint model of the language is briefly explained. Then the basic primitives of the language and how they are used to write cross-cut expressions are described.

2.1. The joinpoints

We have used a dynamic joinpoint model where joinpoints are key events in the execution of a program written in an OO language. We specifically used Smalltalk as the OO language. There are a few different types of joinpoints: the sending of a message, the reception of a message by an object, the accessing and updating of the state of an object and the execution of Smalltalk blocks.

Each joinpoint has some relevant, possibly run-time, data associated with it. A message reception joinpoint for example is associated with the name of the received message and the arguments that are passed with the message.

2.2. The language

2.2.1 The logic language

The particular logic language we used is QSOUL [11,1], a simple variant of PROLOG. There are a few syntactical differences between the two. Variable names in QSOUL start with a question mark rather than a capital letter. Lists are written using pointy brackets rather than square brackets. Also the syntax for rules is a bit different as can be noticed in the examples later on.

The QSOUL language was specifically designed for doing meta programming about Smalltalk programs. To this end QSOUL uses a language symbiosis between itself and the Smalltalk system in which it is implemented. Basically this allows Smalltalk expressions to be embedded in logic expressions, which is done by putting the expression between square brackets. Smalltalk objects can also be bound to logic variables. The mechanism is further explained in [11]. The example uses of this mechanism in this paper are simple enough to be understood right away.

There is a rich set of standard predicates for QSOUL which are designed to reason about Smalltalk programs. We will introduce a few of these predicates which will be used in the examples later on.

- classNamed(?class, ?name)
  This predicate associates a class object with it's name.

- methodStatement(?method, ?statement)
  This predicate is used to describe that ?statement must be a statement from a method. We do not go into the details of how Smalltalk methods and statements are represented in QSOUL, it suffices to know they can be manipulated using some of the predicates presented here and in the next section.

- messageSendStatement(?statement, ?receiver, ?message, ?arguments)
  This predicate can be used to extract the receiver expression, the message and the expressions for the arguments from a message send statement. The message is simply a Smalltalk symbol representing a selector, the arguments is a list of Smalltalk expressions.

- classMethodInProtocol(?class, ?selector, ?protocol)
  Describes that a class method with the selector ?selector from a class is associated with a protocol (also known as method category) in that class.

- instanceMethodInProtocol(?class, ?selector, ?protocol)
  Similar to the predicate above, but for instance methods rather than class methods.

2.2.2 Expressing cross-cutting

The core of the cross-cut language consists of a few basic logic predicates. These are used to write cross-cut expressions, which are simply logic queries about joinpoints. This is explained in this section in more detail.

For each type of joinpoint there is a primitive predicate. All of these predicates take at least one argument: the joinpoint. Additional arguments are used for the data that is associated with joinpoints. A summary of the predicates is given below:

- reception(?jp, ?selector, ?arguments)
  Used to express that ?jp is a message reception joinpoint, where the message with selector ?selector is received with the arguments in the list ?arguments.

- send(?jp, ?selector, ?arguments)
  The joinpoint ?jp is a message send joinpoint where the message with selector ?selector is sent and passed the arguments in the list ?arguments.

[1] AspectJ version 0.8

- reference(?jp, ?varName, ?value)
  The joinpoint ?jp is a reference joinpoint 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 joinpoints, 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 joinpoint ?jp is a Smalltalk block execution joinpoint, where ?args is a list of arguments that were passed to the block.

There are also a few predicates which relate joinpoints to their execution context or their corresponding location in the source code:

- associatedJoinpoint(?jp, ?statement)
  This predicate provides for a link between QSOUL's capabilities for reasoning about source code and the dynamic joinpoint model used by our cross-cut language. Basically it associates a statement with the execution points of that statement.

- within(?jp, ?class, ?selector)
  As all executable code in Smalltalk is written in methods, all joinpoints occur in the context of a method. This predicate associates joinpoints with the class and the selector name of the methods in which they occur.

- inObject(?jp, ?object)
  All code is also executed in the context of some object. This predicate associates joinpoints with the object in which they occur.

As is always the case in logic programming, the predicates can be used in multiple ways. By specifying the values for the data arguments we can pose conditions on joinpoints. By using unbound variables as arguments to the predicates we can extract the data values associated with a joinpoint.

We stress that in contrast with the predicates in the previous section which deal with a program in terms of its source code, the predicates on joinpoints deal with a program in terms of its execution. For example with the messageSendStatement predicate ?arguments is a list of Smalltalk *expressions*, while in the reception predicate ?arguments is a list of actual *objects* passed around at runtime.

Cross-cut expressions are written as logic queries about joinpoints. These logic queries thus need to make use of the primitive predicates on joinpoints. Other logic predicates can ofcourse also be used. When resolved, the query binds a specified variable to joinpoints which are then said to match the query. An example cross-cut expression is:

```
?jp matching
  reception(?jp, ?msg, ?args),
  member(?msg, <[#test:], [#bla:]>)
```

The cross-cut expression above can be read in natural language as: "all reception joinpoints where the received message is one of `test:` or `bla:`".

3. Experiments performed

In this section we discuss some experiments performed with the cross-cut language presented in the previous section. The first two experiments show that our approach allows to easily extend the cross-cut language with new types of joinpoints whose definition depends on the ability to recognize the use of Smalltalk programming conventions. A third experiment shows the use of dynamic information in cross-cut expressions.

3.1. New types of joinpoints

A general problem in writing an aspect weaver for the Smalltalk programming language lies in the fact that the language has few different primitive constructs. Rather, Smalltalk offers the programmer a few but very flexible constructs from which more specific constructs are built simply by use of programming conventions. An example is the lack of the constructor construct which is known by Java and C++ programmers. Smalltalk programmers rather use the convention of using a combination of a class method and instance method to implement the same object creation and initialization concept. The two methods are typically tagged by putting them in the method categories 'instance creation' and 'initialization' respectively.

Extending the weaver so that it can provide more types of joinpoints may require the weaver to become aware about Smalltalk programming conventions. Suppose we wish to add a new primitive type of joinpoint to the cross-cut language presented earlier: variable initialization joinpoints. These are similar to variable assignment joinpoints, the difference being that they occur in the context of an initialization method execution. In order for our weaver to recognize this situation, it needs to be able to recognize the use of the constructor programming convention.

In principle, the recognition of programming conventions can be built into the weaver. The problem however is that programming conventions are open to change or personal preference, so the recognition of these programming conventions must be open to change as well.

We will now demonstrate how our cross-cut language can be used to extend the cross-cut language with two new types of joinpoints from within the language, so that no hard-coded extension to the weaver is needed.

3.1.1 Variable initialization joinpoints

The first new type of joinpoints we will be adding are the variable initialization joinpoints. Adding a new type of joinpoint requires defining a new logic predicate. A definition in natural language was given above, which can be easily translated to a logic rule:

```
Rule initialization(?jp, ?class, ?varName, ?initVal) if
  assignment(?jp, ?varName, ?preInitVal, ?initVal),
  within(?jp, ?class, ?selector),
  instanceInitializationMessage(?class, ?selector)
```

The first condition in the rule serves to ensure that ?jp is indeed an assignment joinpoint, while the second and third rule deal with recognizing the constructor programming convention. The within predicate is used to get at the method in whose context the assignment occurs. The instanceInitializationMessage predicate is then used to check whether this method is an initialization method. We are left to define this last predicate:

```
Rule instanceInitializationMessage(?class, ?selector) if
  classMethodInProtocol(?class, ?selector,
                        ['instance creation']),
  instanceMethodInProtocol(?class, ?selector,
                        ['initialization'])
```

The above should be fairly straightforward to understand.

Note that it is easy to allow for different variable initialization conventions in the same application. Suppose we are reusing some code written by someone who uses a slightly different convention, namely to name the instance creation protocol 'creation'. We can handle this situation by simply defining a second rule for the instanceInitializationMessage predicate in *addition* to the first:

```
Rule instanceInitializationMessage(?class, ?selector) if
  classMethodInProtocol(?class, ?selector,
                        ['creation']),
  instanceMethodInProtocol(?class, ?selector,
                        ['initialization'])
```

Ofcourse if the variation on the convention is more involved than a simple renaming of protocols we would have to define a second rule for the initialization predicate to express the other convention.

3.1.2 Exception handling joinpoints

The second new type of joinpoints are the exception handling joinpoints. In Smalltalk, exception handling is done through the use of the on:do: message. The message can be sent to a Smalltalk block which is then executed. If an exception is thrown while the block is executing and the class of this exception is the one specified as the argument to the on: part of the on:do: message, then the block given as argument to the do: part will be executed. Note though that this convention is the one specified by the ANSI, there are other conventions for exception handling in Smalltalk as well[2].

```
Rule exceptionHandlerExecution(?jp, ?exClass, ?exObject) if
  methodStatement(?method, ?statement),
  messageSendStatement(?statement, ?receiver, [#on:do:],
                <?exClassName, ?blockStatement>),
  associatedJoinpoint(?jp, ?blockStatement),
  blockExecution(?jp, <?exObject>),
  classNamed(?exClass, ?exClassName)
```

The first condition in the above rule simply states that we want the variable ?statement to be bound to a statement in some method. The second condition then states that the statement must be a message send statement in which the message on:do: is sent, the expressions for the arguments are also bound to some variables. The third condition transforms from a statement to a joinpoint using the associatedJoinpoint predicate, in the case of a block creation expression the associated joinpoint is the point in the program's execution where that block is actually executed. Finally the fourth and the last conditions simply get some extra information about the exception being handled: the class of the exception and the actual exception object that was thrown.

3.2. Dynamic joinpoints

In the following rather simple example we show the use of dynamic information in the expression of cross-cutting. The example encapsulates a constraint checking concern in a banking application. A typical constraint on a bank account is that one cannot withdraw more money than is actually on the account. In implementation terms this means that we want to generate an error at those points in the program's execution where the withdraw: message is sent to an instance of the Account class where the amount argument of the message is greater than the account's balance. These points can be easily captured in our cross-cut language:

```
?jp matching
  reception(?jp, [#withdraw:], <?amount>),
  inObject(?jp, ?account),
  greaterThan(?amount, [ ?account balance ])
```

The first condition of the cross-cut captures the joinpoints where a withdraw: message is sent to some object, it also extracts the single argument that is sent from the joinpoint. The second condition gets the object in whose context the reception joinpoint occurs, which would be the account object receiving the message. The final condition simply states that the withdrawn amount should be greater than the account's balance. Notice the use of a Smalltalk expression to get the account object's balance in the final condition.

[2]Before exception handling was standardized, different Smalltalk vendors provided different exception handling mechanisms. Some still support their own mechanism in addition to the standardized one.

While we restrict our attention in this paper to the cross-cut language, we will show here how the above cross-cut expression is used in combination with an action in our aspect language. Actions consist of Smalltalk code that is to be executed before or after joinpoint specified by a cross-cut expression.To generate an error at the joinpoints described by the cross-cut expression we would write:

```
before ?jp matching
  reception(?jp, [#withdraw:], <?amount>),
  inObject(?jp, ?account),
  greaterThan(?amount, [ ?account balance ])
do
  Smalltalk error: 'withdrawing more than is allowed'
```

The above says that before[3] the joinpoints that match the cross-cut expression are actually executed, an error is generated.

4. Related work

This section reports on related work. We first discuss the effect of the use of dynamic information in cross-cut expressions on the ability to clearly separate cross-cuts and actions. Next an overview of the use of logic meta programming in aspect-oriented programming is given.

4.1. Separation of cross-cuts and actions

As stated in the introduction the core of our cross-cut language and the joinpoint model are based on the ones found in AspectJ [5]. Until recently AspectJ did not allow the use of dynamic information on joinpoints in the cross-cut expressions, but it could be used in the action language. We note that the newest version of AspectJ[4] has been extended to allow for the use of runtime information in the description of cross-cutting as well.

When one tries to encapsulate the withdrawal constraint in the banking application using the older AspectJ, part of the cross-cut will have to be be implemented in the action language. This is because the comparing of the account's balance versus the amount being withdrawn involves values that are only available at runtime and also because the cross-cut language has no means of comparing values.

The phenomenon of having cross-cut expressions being expressed in the action language due to the limitation of a cross-cut language was previously pointed out by Douence et al. [9] in the context of an e-commerce application. They also argued in favor of more clearly separating cross-cuts and actions by making the cross-cut language more sophisticated. We have shown another example of how an existing cross-cut language can be made more expressive.

[3] A replace semantics would be better here, but is not currently available in our system.

[4] Since AspectJ version 1.0alpha1, which introduced the if pointcut.

4.2. Aspect-Oriented Logic Meta Programming

A few different researchers have been working on the use of logic meta programming in the field of aspect-oriented programming (AOLMP). The focus of each of these works is however different. This difference is related to the different usages of logic meta programming itself, we give a brief overview of AOLMP to give the reader some insight:

Kris De Volder constructed the TyRuBa logic meta programming language with the goal of using it to generate (Java) programs. His application of this system to AOP [10] focused on using it as a general framework for implementing aspect weavers as these are often implemented as source-to-source transformers. He also showed that LMP is a good formalism for expressing aspects.

The QSOUL/AOP effort is another application of LMP to AOP researched by Johan Brichau. His work extends that of De Volder by exploring further the use of LMP for the construction of domain-specific aspect languages and the problem of interaction between different aspects. In his QSOUL/AOP system aspect languages can be rapidly constructed by implementing a weaver which reduces expressions in the new language to aspects in an existing language. This results in the construction of a tree of aspect languages. Besides this vertical combination the system also allows for using logic rules to express the horizontal combination of aspects to solve the aspect interaction problem.

The author's work so far has been concentrated on the construction of the one specific aspect language presented in this paper. This work has also focused more on the use of LMP to reason about and extract information or patterns from the program that is cross-cutted to be used in the expression of cross-cutting than the other two researchers have. Though this research has been done independently from QSOUL/AOP it should be possible to combine the two by implementing Andrew as one of the languages in the QSOUL/AOP language tree. Plans to perform this experiment are underway.

5. Future work and points of discussion

In this section we report on possible areas of future work for our research. These are topics that can serve as points of discussion on the workshop.

The most important area of research we currently envision is how our aspect language can be used to further decouple aspects from the programs they cross-cut. Kersten and Murphy previously stated that decoupling an aspect from the components it cross-cuts makes the aspect reusable accross applications [4]. In studying AOP we have unfortunately come accross a problem we have dubbed the "enumeration problem". Points of cross-cutting do not just occur at random but are generally related by some pattern. However when a cross-cut

language lacks the mechanisms needed to express this pattern one has to resort to simple enumeration of occurences of that pattern in a cross-cut component. This obviously couples the aspect to that specific component. An example of this problem can be found in a publication by Lippert and Lopes [6]. In this paper we presented a few examples of the use of LMP to express patterns in code. The strength of LMP in this area has been extensively demonstrated before [11]. However this needs to be researched further as the link to AOP presents some unresolved questions: such as how to effectively make the aspects reusable or how to handle patterns from slipping through the detection rule. This topic is discussed a bit more extensively in chapter 7 of the author's licentiate's dissertation [3].

The use of dynamic joinpoints has so far not been fully explored. While demonstrated in this paper as allowing a clearer separation of actions and cross-cuts, the use of dynamic joinpoints also applies to the decoupling point of the previous paragraph: dynamic joinpoints present more information from components to aspects than static joinpoints do so they make it easier to express a pattern. However we currently seek more examples to demonstrate this.

A final point to investigate is the use of our approach in the context of another cross-cut language than AspectJ's. We think that any cross-cut language can potentially benefit from more powerful pattern matching constructs.

6. The full system

We note again that we have constructed a complete aspect language for Smalltalk but have concentrated in this paper on its cross-cut language. The action language of the aspect language is simply Smalltalk itself. The full aspect system, including a simple GUI environment, is available on the web [2].

7. Conclusion

In this paper we presented a cross-cut language based on a logic meta language and on a dynamic joinpoint model, where runtime information about joinpoints can be used in the description of cross-cutting. The use of a meta language allows one to easily extend the joinpoint model with new types of joinpoints whose definition is open to change and without the need to adapt the aspect weaver. We showed some examples of adding new joinpoints whose definition involved recognizing the use of programming conventions. Because of the use of a declarative language the definition of the new joinpoints *describes* the programming convention rather than *how* it is recognized. Finally, we showed how dynamic joinpoints aid in clearly separating cross-cut expressions from actions.

References

[1] J. Brichau and W. D. Meuter. Qsoul manual (draft). http://prog.vub.ac.be/poolresearch/qsoul/QSOULManual.pdf.

[2] K. Gybels. Andrew aop website. http://prog.vub.ac.be/~kgybels/andrew/.

[3] K. 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.

[4] M. Kersten and G. 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.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Grisworld. Getting started with AspectJ.

[6] M. Lippert and C. V. 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.

[7] W. D. Muynck. Aspect-Oriented Programming: a survey of current research. State-of-the-art article submitted for the course Capita Selecta from Computer Science in One Year Master of Science in Computer Science EMOOSE 1999-2000, 2000.

[8] X. PARC. Aspect-oriented programming faq. http://www.lifl.fr/~renaux/recherche/aop/faq.html.

[9] M. S. Rémi Douence, Olivier Motelet. Sophisticated crosscuts for e-commerce. In *Proceedings of the workshop on Advanced Separation of Concerns*, 2001.

[10] K. D. 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.

[11] R. Wuyts. *A Logic Meta Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.