Semantic Join Point Models: Motivations, Notions and Requirements

Walter Cazzola, Jean-Marc Jézéquel, Awais Rashid

Semantic Join Point Models: Motivations, Notions and Requirements

Walter Cazzola
Dept. of Informatics and Communication,
Università di Milano
cazzola@dico.unimi.it

Jean-Marc Jézéquel
IRISA,
Université de Rennes 1
jezequel@irisa.fr

Awais Rashid
Computing Department,
Lancaster University
marash@comp.lancs.ac.uk

ABSTRACT

Aspect-oriented programming (AOP) has been designed to provide a better separation of concerns at development level by modularizing concerns that would otherwise be tangled and scattered across other concerns. Current mainstream AOP techniques separate crosscutting concerns on a syntactic basis whereas a concern is more a semantic matter. Therefore, a different, more semantic-oriented, approach to AOP is needed. In this position paper, we investigate the limitations of mainstream AOP techniques, mainly AspectJ, in this regard and highlight the issues that need to be addressed to design semantic-based join point models.

Keywords

AOP, Join Point Model, Pointcut Definition Language

1. INTRODUCTION

Nowadays, everywhen we ask for the better way to build a reusable, extensible and open software system we are addressed to the achievement of the *separation of concerns* property just as well as in the 70ies we were addressed to the modularity property.

Kiczales et al. [14] have introduced *aspect-oriented programming* (AOP) with the intent of capturing and untangling the *crosscutting concerns*, i.e., concerns whose implementation straddles data and behavior belonging to different objects. The main goal of AOP consists of improving the separation of concerns both in design and implementation. The programmer can implement the crosscutting concern as a separate modularity unit, called *aspect*, rather than to tangle its code with the code that implements other concerns.

The *join point model* (JPM) adopted by the programming language dictates how the crosscutting modularization takes place. Following the definition of Kiczalez at al. in [14, 17], a JPM is mainly composed of three elements:

- a set of points, called *join points*, in the computational flow of the program that can be used to compose the separated concerns with the rest of the system;
- the *pointcut definition language*, that gives means of identifying the join points; and
- a means of specifying semantics at join points (e.g., to execute code before, after or around a join point).

The join point represents the key concept in aspect orientation. Specifying a set of join points is a major task for aspect-oriented designers, and providing a suitable representation for join points is a primary task for an aspect-oriented language.

Notwithstanding the (important) role played by the pointcut definition languages, they are not always adequate to the situation (it is particularly true for the code-oriented pointcut languages). As we will see in the following, there are situations in which the desired join points cannot be accurately described in the pointcut language (e.g., because the matching requirements are too abstract) or simply their description needs information not available at weaving time.

2. LIMITS OF THE ASPECTJ-LIKE JPM

The join point model and in particular its mechanism to identify the join points (the pointcut definition language) has a critical role in the applicability of the aspect-oriented methodology. As stated by Kiczales in his keynote at AOSD 2003 [13], the pointcuts definition language has the most relevant role in the success of the aspect-oriented technology but most of them (either static or dynamic) rely on a mechanism too tailored on the syntax of the program to manipulate. Similar approaches suffer from several flaws but mainly they hinder the join points matching in a more semantic way. Let us see the problem in details[1].

Dependency on the Syntax. The AspectJ pointcut language offers a set of *primitive pointcut designators*, such as **`call`**, **`get`** and **`set`** specifying a method call and the access to an attribute. These primitive pointcut designators can be combined using logical operations (`||`, `&&`, `!`) forming more complex pointcuts. All the pointcut designators expect, as an argument, a string specifying a pattern for matching method or field signature.

Therefore, we have just two ways of describing the join points: i) by listing or ii) by using a combination of wildcards and naming conventions. For example, to identify the methods in a specific package that change, to some extent, the state of the system we have to use the following pointcut (or a very similar one):

```
call(public * com.xerox.printers..*.set*(..))
```

The problem comes when someone adds new methods to the package and does not respect the naming conventions used by the point-

[1]Note that, in the rest of the paper we will present examples written in AspectJ but similar consideration and examples can also be done adopting a different AOP language/framework that uses a code-oriented join point selection mechanism.

cut. We then have to update the pointcut definition: this is an unfortunate coupling. This problem also affects the classic object-oriented development. Something as simple as changing the naming convention or changing the package name means that aspects using such a pointcut will not be applied any more. Since the pointcut is tightly coupled to the syntax of the base code, a refactoring of the base code also implies a refactoring of the aspects applied on it. This issue is known as the *problem of the fragile pointcuts* [16].

Unrecognized Join Points. Neglecting the pointcuts' fragility aspect, they are still not usable in all possible contexts. There are some computational patterns (also very simple) that cannot be captured by a code-oriented pointcut. The sequence of two or more calls is the most simple and evident example of this problem.

```
class B extends A {
   void d() {
     a();
     b();
     c();
   }
 }
```

Listing 1: match sequence

```
class B extends A {
   void d() {
     while( aCond ) {
        b(); a();
     }
     c();
   }
}
```

Listing 2: match in loop

Considering listings 1 and 2, for example. We cannot express a pointcut that matches the join point which precedes the call of `a()` when it is followed by a call to `b()`. To some extent, we can bypass this problem by extruding a new dummy method from the sequence and expressing the pointcut in term of this dummy method. This would match the base code in listing 1 but nothing can be done, except manually unrolling the loop, to match the join point in listing 2. Moreover, this kind of refactoring cannot always be applied or gives the expected results.

In general, it is difficult to use complex patterns to recognize the join points but the difficulties increase when the pattern we are looking for is based on the properties of the computational flow and does not rely on the syntax of the base code itself. Something like capturing all the join points that occur within the dynamic extent of another join point and the dynamic extent should match the execution branches that verify a specific property, such as atomicity, or is expressed abstracting from the specific code (by using, for example, the natural language to express them). Decoupling the pointcuts from the base code representation is essential to modularize a general and then reusable concern and to free the aspect modularization from the syntactic limits of the programming languages.

Dynamic Join Points Selection or Restriction. Dynamic join points identification is neglected by most of the available join point models. Sometimes we have the necessity of describing a set of join points in terms of other join points, e.g., all the join points that can be reached from a given join point in two hops or the join point preceded by a given join point and followed by another. To determine this kind of join points the framework needs a deeper knowledge of the program's dynamic execution. The **`cflow`**[2] pointcut declarator, that allows the selection of all the reachable join points from the currently matched join point, is one of the few exceptions. Notwithstanding that **`cflow`** is not flexible enough to select a specific execution trace but just a set of join points depending on a common "ancestor", if necessary a skimming can be yielded by introspecting through reflection. A similar selection mechanism would be the basic tool to select the join points on a more semantic basis looking after the properties more than after the program syntax.

In spite of the potential of the aspect-oriented methodology, its applicability is limited by the current state-of-the-art in join point models and pointcut expression mechanisms. As stated by Gregor Kiczales [13] a lot of work needs to be done to render the mechanism more expressive and therefore usable in all contexts. In his talk, Kiczales suggested to define a more precise pointcut declarator, named **`pcflow`**, based on the prediction of the computational flow but we think that to widen the mechanism applicability we need an approach not based on heuristics but on checked and evaluated data.

3. SEMANTIC JOIN POINT MODELS

From the considerations reported in section 2 and from the issues raised from the experience of other researchers (see, for example, [8, 21]), it is fairly evident that most of the problems related to the main-stream join point models are due to the pointcut expression languages that only allow syntactic specification of join points to be matched. This is reminiscent of the traversal-based and query-based access in object databases. In case of the former, similar to current pointcut expression mechanisms, one has to specify how to traverse to the data of interest. In case of queries, on the other hand, one merely needs to specify the goals of the query and the system extracts the relevant data matching those goals. Queries on program semantics can, therefore, serve as a good means to write more intentional, semantic pointcut expressions.

Hence we perfectly agree with Kiczales [13] that next enhancement for the AOP methodology consists of extending the pointcut definition language to support join points selection on the basis of a semantic query, i.e., pointcuts that do not only take into consideration the structure of the program but also its semantics.

The Pragmatism of Semantics. Depending on the nature of the query used to identify the join points, we can give two possible interpretations of what can be a more semantic-based pointcut definition language. If the query is something like "*select the join points where the next statement writes on disk*" we are describing, independently of the syntax, a trace of the computational flow, i.e., a specific instance of the behavior of the program. In this case the word semantic must be understood as *behavior*.

On the other hand, if the query is something like "*the joint points preceding statements whose execution does not reciprocally interfere*" we are describing a property of the join points that does not directly regard the computational flow of the program, that is, a property that can or cannot be valid independently of the program behavior. In this case the word semantic must be understood as *property-oriented*.

Selecting join points on a property basis can help in separating concerns, such as concurrency, atomicity, and so on, that are formally well-described but not easy to identify in the code. Of course this kind of selection, to be practically usable should also be supported on the weaving side by providing, for example, a mechanism for extruding the identified code.

On the other hand, selecting join points on a behavioral basis does not simply abstract from the syntax improving the reusability, but also permits of capturing join points that otherwise would not be captured. Contrary to the property based selecting mechanism, a behavioral selecting mechanism would be more easily embedded in the current join point models because it is basically the same mechanism that works on a different level of knowledge.

[2]Of course, the same consideration also applies to the **`cflowbelow`** pointcut declarator.

Limiting the Expressiveness. Naturally, independence of the syntax and a more abstract selecting mechanism do not necessarily mean that we have to use natural language to select the join points.

Natural language has the benefit that it could be used to practically describe every kind of join point, but it also has the defect of being intractable from the computer. Natural language is intrinsically ambiguous and does not have a clear syntax to be parsed and evaluated.

Moreover, natural language, as well as many other high-level definition languages, permit of querying on a noncomputable knowledge base. Similar issues can be raised when the used language enables the selection from an inductively generated knowledge base by using negations and quantification (this is the case, for example, of [21] and [20]). Therefore, depending on the model of knowledge we adopt, the evaluation of a selection, for example, something like "*all the join points that do not respect a specific property/behavior*", risks to generate an infinite loop.

Therefore, providing a more expressive and semantic-oriented selection mechanism means to use a query language based on a well-defined program representation that captures its properties/behavior abstracting from the syntactic (and, why not, from the programming language) details. Moreover, the admitted queries have to be anchored on the program representation to avoid problems with quantification and negations.

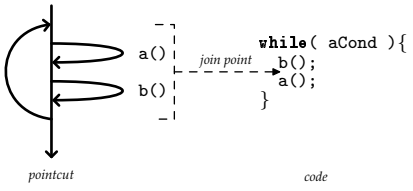
Model Driven JPMs. After these considerations, the question is: does a similar program representation exist? Or have we to model an appropriate representation?

In our opinion, *design information*, not necessarily based on UML, offers the most suitable program representation. The design information is the result of the designing phase and it describes the behavior and the interrelations inside of the code in an intuitive, often graphical, way. It can be complete and independent of the implementation details. Moreover, it focuses on describing the behavior and the features of the program rather than their realization.

The UML and Model-Driven Architecture (MDA) are the most used technologies to represent the design information of a program and the most promising to be used to build a semantic selecting mechanism. The following considerations apply to the UML but they can be easily extended to any other modeling language.

As noted by Jacobson [12], some UML diagrams, such as the use cases, the sequence and activity diagrams, interestingly slice the program behavior into separate views. These views can represent bricks used to build programs from design information as done by Jacobson. They can also, by using an aspect-oriented parlance, represent some patterns of the computational flow to select specific program behaviors, i.e., a sort of pointcuts, that can be used to disentangle the code representing a crosscutting concern.

Moreover, the UML diagrams can describe sequences of the computational flow that otherwise could not be matched through static analysis. For example, the following sequence diagram can be used as a pattern to match a sequence of calls `"a(); b();"` also in the sample of code in listing 2 and, correctly, consider the point before the call to `a()` as a join point.



The UML, by definition, abstracts from the implementation details. This means that the design information describing a program is independent of its implementation, both of the programming language characteristics and of the names used at code level. Therefore a pointcut expressed through a UML diagram is not bound to the name of the methods but can rely on only a trace of the behavior of the method. In this way, the selecting mechanism is independent of naming conventions or similar expedients.

The interrelations among the program components are immediately available and identifiable. Class and object diagrams describe clientship and inheritance among the program elements. Whereas, use cases, statecharts, activity and sequence diagrams describe how and when the clientship takes place. Therefore, through a static analysis of the design information we can get a direct outline of the system execution (only limited by decidability issues). In this, way we can statically compute almost every selectable set of join points and use them to narrow or broaden the selection in conjunction with another pointcut.

The UML, beyond its graphical representation, is defined through a meta-model that makes it easy to be analyzed and modified by a program. Based on this meta-model, it also provides a textual representation of a system based on XML (XMI [19]) that can be seen as a knowledge base for that system. Both can be used to express the whole join point model (as described, the selecting mechanism could be based on looking for a specific pattern in the knowledge base and similar diagrams can be used to describe the advice to be woven as well).

Nowadays, design-oriented JPMs are being widely pursued, e.g., state oriented [18], based on the UML [5], and model-driven [3,15].

From High- to Low-Level. Abstracting from the implementation details, as shown, is the strength of using the UML methodology as a join point model, but in it also dwells a key challenge.

Abstracting from the code is perfect to work at a higher-level of awareness but becomes a problem when you have to update the code because the level of detail of the two representations is very different and the link between them is often missing. The problem is amplified from the fact that there does not exist a unique way to code a UML specification into a program. Each programmer uses and adapts to his/her experience the specification he/she receives. From the design model, it is seldom possible to automatically deduce 100% of the code corresponding to the code to instrument, as well as inferring the association between the diagrams and the code. To some extent the discrepancy between the two representations can be contained when the UML specification is obtained from the code by reverse engineering or the code is automatically generated from the UML specification but these approaches are not always feasible especially because we are working, and we want to keep working at two different levels of detail. Another feasible approach could be entrusting this mapping to some annotation facility that can vary its level of detail ranging from the very abstract to very detailed but this operation could be difficult to automate and should be tailored on the problem (i.e., well detailed pointcuts need a very rich mapping while pointcuts loosely coupled to the code need quite a coarse mapping).

Neglecting how the mapping between the two representations is obtained and kept consistent, the big problem is how to anchor the join points to the code thanks to the flow described by the pointcuts (i.e., by a UML diagram). To have a connection between the code and its specification is fundamental for weaving the advice with respect to the identified join points. A possibility should consist of creating and then following the mapping on-the-fly. This will help with dynamic binding but it will be extremely inefficient. The most grounded mechanism would consist of embedding the mapping directly into the code (or into the bytecode) and retrieving

it at weaving time through reflection. Given a mapping between the code and its specification it could be automatically embedded by decorating/annotating the code (or better the bytecode) with the mapping. Then a pointcut will be evaluated on the decorations and not directly on the code.

Notwithstanding that, to decorate the code (bytecode) is a feasible and quite easy operation, due to the nature of the decoration it is also quite delicate an operation. In particular, several different diagrams can refer to the same portion of code and each single diagram describes several lines of code not necessarily contiguous. Therefore, the decoration mechanism has to support a scope different from a single line. Note that both Java and .NET support code annotations but they only permit annotating element declarations and not code blocks. Some extensions to their annotation mechanism are under development [4].

Thence, to adopt a model driven join point model we have to come to terms with the higher-level expressiveness of the models and the concreteness of the weaving operation that directly involves the code.

4. CONCLUSIONS AND FUTURE WORK

In this paper, we have investigated the limitations of current AOP techniques, mainly AspectJ and the other code-oriented join point models and tried to raise some issues that should lead AOP towards a more semantic separation of concerns mechanism. The need to look beyond the current join point models is now widely recognized [13], and many attempts have been made to overcome it.

Some of these attempts work on automatically selecting the join points from a high-level (often logic) description. Tourwé et al. [21] have proposed an advanced pointcut managing environment, based on machine learning techniques. To deal with pointcut expressivity problem they propose to inductively generate the pointcuts from a graphical specification. Gybels et al. [11] have dealt with the so called *arranged pattern* problem. Crosscutting languages use pattern matching to capture join points, this is a good technique to describe the intended semantics of a crosscut but it is still dependent on the naming conventions.

Different approaches are based on the observation of the application behavior. Allan et al. [1] propose to extend AspectJ with a join point selecting mechanism based on trace matching. Douence et al. [7], on the other hand, tackle the problem in terms of execution traces and aspect composition. Both of the approaches can deal with some of the listed problems but they do that at run-time whereas a model-driven approach could tackle them at compile-time improving the efficiency and the safety of the approach.

In our opinion, working at the model level offers promising possibilities to start abstracting away from the concrete syntax and to move the weaving of some aspects from the execution to the compilation. Initial investigations into such high-level models have been carried out by [2, 6, 9, 10].

We have proposed to go beyond this point by addressing semantic issues, that must be understood as either behavior based or property based. On the other hand, working at model level introduces problems of its own, such as synchronization between the model and the code. In this paper we have restricted ourselves to underlining the problems underpinning the notion of semantic joint points and just giving hints to possible research directions to solve them. Obviously future work will consist in actually taking these research roads.

Acknowledgments

The authors wish to thank Wolfgang De Meuter, María Agustina Cibrán, Jean Bézivin, Ruzanna Chitchyan and all the other ECOOP 2005 delegates that attended the brainstorming about high-level pointcut languages at the *models and aspects* ECOOP workshop. Many of the presented ideas have been refined thanks to their comments during such a brainstorming.

5. REFERENCES

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching to AspectJ. abc Technical Report abc-2005-1, The abc Group, March 2005.

[2] João Araújo, Jon Whittle, and Dae-Kyoo Kim. Modeling and Composing Scenario-Based Requirements with Aspects. In *Proceedings of the 12th IEEE International Conference on Requirements Engineering (RE'04)*, pages 58–67, Kyoto, Japan, September 2004. IEEE Computer Society.

[3] Walter Cazzola, Antonio Cicchetti, and Alfonso Pierantonio. Towards a Model-Driven Join Point Model. In *Proceedings of the 11th Annual ACM Symposium on Applied Computing (SAC'06)*, Dijon, France, on 23rd-27th of April 2006. ACM Press.

[4] Walter Cazzola, Antonio Cisternino, and Diego Colombo. Freely Annotating C#. *Journal of Object Technology*, 4(10):31–48, December 2005.

[5] Walter Cazzola, Sonia Pini, and Massimo Ancona. AOP for Software Evolution: A Design Oriented Approach. In *Proceedings of the 10th Annual ACM Symposium on Applied Computing (SAC'05)*, pages 1356–1360, Santa Fe, New Mexico, USA, on 13th-17th of March 2005. ACM Press.

[6] Siobhán Clarke and Robert J. Walker. Generic Aspect-Oriented Design with Theme/UML. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, chapter 19, pages 425–458. Addison-Wesley, 2005.

[7] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In Gail Murphy and Karl Lieberherr, editors, *Proceedings of the 3rd Int'l Conf. on Aspect-Oriented Software Development (AOSD'04)*, pages 141–150, Lancaster, UK, March 2004. ACM Press.

[8] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as Functional Queries. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, LNCS, Taipei, Taiwan, November 2004. Springer.

[9] Geri Georg, Robert B. France, and Indrakshi Ray. Composing aspect models. In Omar Aldawud, Mohamed Kandé, Grady Booch, Bill Harrison, Dominik Stein, Jeff Gray, Siobhán Clarke, Aida Zakaria Santeon, Peri Tarr, and Faisal Akkawi, editors, *Proceedings of the 4th AOSD Modeling with UML Workshop*, San Francisco, CA, October 2003.

[10] Jeff Gray, Janos Sztipanovits, Douglas C. Schmidt, Ted Bapty, Sandeep Neema, and Aniruddha Gokhale. Two-Level Aspect Weaving to Support Evolution in Model-Driven Synthesis. In Robert E. Filman, Tzilla Elrad, Siobhàn Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, chapter 30, pages 681–709. Addison-Wesley, 2005.

[11] Kris Gybels and Johan Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *Proceedings of the 2nd Int'l Conf. on Aspect-Oriented*

Software Development (AOSD'03), pages 60–69, Boston, Massachusetts, April 2003.

[12] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, Reading, Ma, USA, December 2004.

[13] Gregor Kiczales. The Fun Has Just Begun. Keynote AOSD 2003, Boston, March 2003.

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *11th European Conference on Object Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science 1241, pages 220–242, Helsinki, Finland, June 1997. Springer-Verlag.

[15] Jacques Klein, Jean-Marc Jézéquel, and Noël Plouzeau. Weaving Behavioural Models. In *Proceedings of the 1st ECOOP Workshop on Models and Aspects (MA'05)*, in 19th European Conference on Object-Oriented Programming (ECOOP'05), Glasgow, Scotland, July 2005.

[16] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the Fragile Pointcut Problem. In *Proceedings of the European Interactive Workshop on Aspects in Software (EIWAS'04)*, Berlin, Germany, September 2004.

[17] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation Semantics of Aspect-Oriented Programs. In Gary T. Leavens and Ron Cytron, editors, *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL'02)*, pages 17–25. Iowa State University, April 2002.

[18] Noorazean Mohd Ali and Awais Rashid. A State-based Join Point Model for AOP. In *Proceedings of the 1st ECOOP Workshop on Views, Aspects and Role (VAR'05)*, in 19th European Conference on Object-Oriented Programming (ECOOP'05), Glasgow, Scotland, July 2005.

[19] OMG. OMG-XML Metadata Interchange (XMI) Specification, v1.2. OMG Modeling and Metadata Specifications available at `http://www.omg.org`, January 2002.

[20] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive Pointcuts for Increased Modularity. In Andrew P. Black, editor, *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, LNCS 3586, pages 214–240, Glasgow, Scotland, July 2005. Springer.

[21] Tom Tourwé, Andy Kellens, Wim Vanderperren, and Frederik Vannieuwenhuyse. Inductively Generated Pointcuts to Support Refactoring to Aspects. In *Proceedings of Software engineering Properties of Languages for Aspect Technologies (SPLAT'04)*, Lancaster, UK, March 2004.