Join Point Patterns: a High-Level Join Point Selection Mechanism

Walter Cazzola
Department of Informatics and Communication
Università degli Studi di Milano, Italy
cazzola@dico.unimi.it

Sonia Pini
Department of Informatics and Computer Science
Università degli Studi di Genova, Italy
pini@disi.unige.it

ABSTRACT

Aspect-Oriented Programming (AOP) is a powerful technique to better modularize object-oriented programs by introducing crosscutting concerns in a safe and noninvasive way. Unfortunately, most of the current join point models are too coupled with the application code. This fact hinders the concerns separability and consequently their reusability since each aspect is strictly tailored on the base application.
This work proposes a possible solution to this problem based on modeling the join points selection mechanism at a higher level of abstraction. In our view, the aspect designer does not need to know the inner details of the application such as a specific implementation or the used name conventions rather he exclusively needs to know the application behavior to apply his/her aspects.
In the paper, we present a novel join point model with a join point selection mechanism based on a high-level program representation. This high-level view of the application decouples the aspects definition from the base program structure and syntax. The separation between aspects and base program will render the aspects more reusable and independent of the manipulated application.

1. INTRODUCTION

Aspect-oriented programming (AOP) is a powerful technique to better modularize object-oriented programs by introducing crosscutting concerns in a safe and noninvasive way. Each AOP approach is characterized by a *join point model* (JPM) consisting of the *join point*s, a mean of identifying the join points (*pointcut*s) and a mean of raising effects at the join points (*advice*) [19]. Crosscutting concerns might be badly modularized as aspects without an appropriate join point definition that covers all the interested elements, and a *pointcut definition language* that allows the programmer of selecting those join points.
In most of the AOP approaches, the pointcut definition language allows the programmer of selecting the join points on the basis of the program lexical structure, such as explicit program elements names. The dependency on the program syntax renders the pointcuts definition fragile [14] and strictly couples an aspect to a specific program, hindering its reusability and evolvability [6,10]. The required enhancement should consist of developing a pointcut definition language that supports join points selection on a more semantic way [5]. To provide a more expressive and semantic-oriented selection mechanism means to use a language that captures the base-level program behavior and properties abstracting from the syntactic details. Several attempts in this direction have been done, some examples are [1, 7], but none of these completely solve the problem. They focus on specific behavioral aspects such as execution trace and dataflow neglecting some others. Moreover, they still rely on name conventions and on the knowledge of the implementation code. We think that the problem could be faced and solved by selecting the join points on an abstract representation of the program, such as its design information.
In this paper, we propose a join point model with a pointcut definition language that allows the selection of the join points abstracting from implementation details, name conventions and any other source code dependency. In particular, in our proposal, the aspect programmer can select the interested join points by describing their supposed location in the application through UML-like descriptions (basically, activity and sequence diagrams) representing computational patterns on the application behavior; these descriptions are called *join point patterns* (JPPs). The join point patterns are just patterns on the application behavior, i.e. they are not derived from the system design information but express properties on them. In other words, we adopt a sort of enriched UML diagrams (or portion of them) to describe the application control flows or computational properties and to locate the join points inside these contexts. Pointcuts consist of logic composition of join point patterns. In this way, pointcuts are not tailored on the program syntax and structure but only on the program behavior.
The rest of the paper is organized as follows: in section 2 we investigate the limitations of some of the other join point models, in section 3 we introduce our join point model and in particular the concept of *join point pattern*, in section 4 we outline the weaving process based on the join point patterns; finally, in section 5 and 6 we face some related works and draw out our conclusions.

2. LIMITS OF THE JOIN POINT MODELS

The join point model has a critical role in the applicability of the aspect-oriented methodology. As stated by Kiczales in his keynote at AOSD 2003 [11], in particular, the pointcut definition language has the most relevant role in the success of the aspect-oriented technology since it allows to determine where a concern crosscuts the code. Since the beginning, the pointcut definition languages are evolved to improve their expressivity, their independence of the base code and the general flexibility of the approach. The first generation of pointcut definition languages (e.g., AspectJ pre v1 [12]) were strictly coupled to the application source code because they allow of selecting the join points on the signature of the

program elements. To reduce the coupling problem, the next generation of pointcut definition languages introduced wildcards (e.g., AspectJ v1.3, HyperJ [16]), this technique reduces the coupling but introduces the necessity of naming conventions; a new problem raises since the naming conventions are not checkable by the compilers and cannot be guaranteed that the program respects these conventions. Recently, some aspect-oriented languages adopted *meta-data* or code instrumentation (e.g., trace-based aspect [7], AspectWerkz [20]) to locate the join points. This approach decouples the aspects from the base program syntax and structure. The meta-data are used as a placeholder of sorting that can be triggered to get a customizable behavior. Anyway this technique does not resolve the problem as well, it just shifts the coupling from the program syntax to the meta-data syntax. Moreover, this approach breaks in an explicit way the *obliviousness* [8] property. To get the obliviousness the aspect programmer should be unaware of the base program structure and syntax to apply his/her aspects, as well as, the base programmer must be unaware of the additional aspects.
In this situation, the aspect programmer must have a *global knowledge* of the base program to be sure that his/her pointcuts work as expected. Moreover, the JPMs based on these kinds of pointcut definition languages are suitable to select join points that are at the object interface level whereas badly fit the need of capturing join points expressed by computational patterns, such as inside loops, before/after a given sequence of statements, or reachable from a given join point after the execution of a given number of statements. Pointcuts definition heavily relies on how the software is structured at a given moment in time. In fact, the aspect developers subsume the structure of the base program when they define the pointcuts; the name conventions are an example of this subsumption. They implicitly impose some *design rules* that the base program developers have to respect when evolve their programs to be compliant with the existing aspects and to avoid of selecting more or less join points than expected.
In general, the previously described join point models are sufficient for most cases but there are situations where a more fine-grained, flexible and semantic join point model is required — more on the join point models limitations can be read in [5, 10, 14]. Therefore, notwithstanding its potentialities, aspect-oriented methodology is limited from the poorness of the join point selection mechanisms. In the rest of the paper we are going to propose a different approach to the join points selection that should relax these limitations.

3. JPP SPECIFICATION LANGUAGE

Design information (UML diagrams, formal techniques and so on) abstracts from the implementation details providing a global, static and general view of the system in terms of its behavior and should permit to locate and select the join points thanks to their properties and to the context instead of name conventions and syntactic details [5]. In this respect, we propose to overcome the limitations of the pointcut definition languages by describing the join points position (i.e., by defining the pointcuts) as a *pattern* on the *expected* application design information rather than on the code.
The *join point pattern*s are the basic elements of our pointcut definition mechanism (called *join point pattern specification language*). They describe where the local or region join points could be located in the application behavior abstracting from the implementation details; when a join point is located in the application high-level representation it will be automatically mapped on the application code. The interested behavior or computational flow can be well described by using design techniques, such as UML diagrams, that provide an abstraction over the application structure. Thanks to this abstraction, the join point patterns can describe the join point positions in terms of the application behavior rather than its code. In other words, we achieve a low coupling of the pointcut definitions with the source code since the join point pattern is defined in terms of design model rather than directly referring to the implementation structure of the base program itself.
The join point patterns are graphically specified through a UML-like description. A visual approach is more clear and intuitive and makes more evident the independence from the program source code. Finally, this approach is not limited to a specific programming language but can be used in combination with many. At the moment, we are using the `Poseidon4UML`[1] program for depicting the join point patterns but we are developing an ad hoc interface for that, and the Java as programming language.
In the application, there is a clear separation between the application structure (e.g., class declarations) and its behavior (e.g., methods execution) and the aspects can affect both the structure and the behavior. In this paper, we only focus on the behavioral join point pattern definition; since affecting the application structure simply consists on introducing and removing elements and can be faced as explained in [3].

3.1 JPP Terminology and Description

In this paper, we borrowed the terms *join point*, *pointcut* and *advice* from the AspectJ terminology but we use them with a slightly different meaning. The *join point*s are *hooks where new behaviors may be added* rather than *well defined points in the execution of a program*. In particular we consider two different kinds of join points, *local join point*s that represent points in the application behavior where to insert the advice code, and *region join point*s that represent portions of the application behavior that must be replaced by the advice code. The *pointcut*s are logical compositions of join points selected by the join point patterns rather than logical composition of queries on the application code. Whereas the term *advice* conserves the same meaning as in AspectJ. To complete the picture of the situation, we have introduced some new concepts: *join point pattern* and *join point pattern space*. The *join point pattern* is *a template on the application behavior identifying the join points in their context*. In practice the join point patterns are UML diagrams of sorting, with a name, describing where the local and region join points can be located in the application behavior. Whereas, the *join point pattern space* is the set of all join point patterns defined on the same application. A join point pattern is a sample of the computational flow described by using a behavioral/execution flow template. The sample does not completely define the computational flow but only the portions relevant for the selection of the join points, in other words the join point patterns provide an incomplete and parametric representation of the application behavior. Each join point pattern can describe and capture many join points; these join points are captured together but separately advised.

Now, we will give a glance at the join point pattern definition language "syntax" by an example. Let us consider the implementation of the *observer pattern* [9] as an aspect to observe the state of a *buffer*. The `Buffer` instances support only two kinds of operations: elements insertion (*put action*) and recovery (*get action*). The observer will monitor the work of these two families of methods, independently from their names and signatures.
The join point pattern depicted in Fig. 1 captures *all the method executions that change the state of the instances of the `Buffer` class*. The activity diagram describes the context where the join point should be found; more application details are used to describe the context and more the join point pattern is coupled to the applica-

[1] `http://www.gentleware.com`

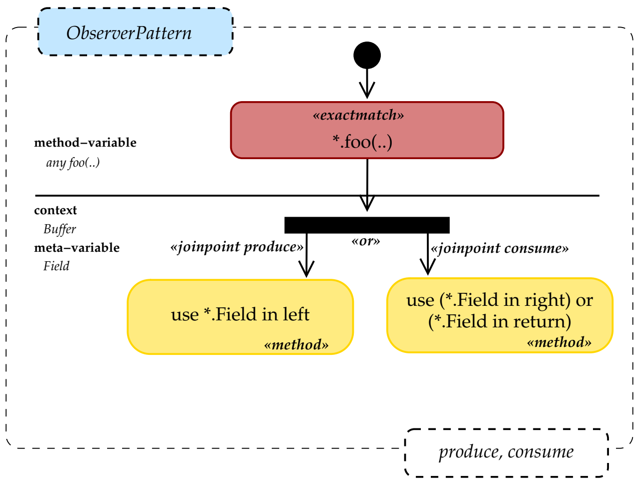

Figure 1: The Observer JPP

tion code. The exact point matched by the ≪`joinpoint`≫ depends on which element follows the stereotype: if the ≪`joinpoint`≫ is located on a flow line between two swimlanes it matches a call of a method with the behavior expressed in the other swimlane; if the ≪`joinpoint`≫ is located on a flow line inside a swimlane it matches the execution of the next instruction, if the element has a ≪`exactmatch`≫ stereotype, or the execution of the next block if the element has a ≪`block`≫ stereotype, or the execution of the next method if the element has a ≪`method`≫ stereotype. The use of *meta variables* grants the join point pattern independence from a specific case, and they are useful to denote that two elements have to refer to the same variable of the same method. Meta variables permit to access variables, methods, fields and so on used into the implementation code without knowing their exact names, but exclusively knowing their role. At the contrary, if the aspect programmer want to couple the join point pattern to the application code, they can use directly constant variable and method names (without declaring them as meta-variables).
In the example, `foo` and `Field` are meta-variables, respectively a *method meta-variable*, i.e., a variable representing a method name and a *variable meta-variable*, i.e., a variable representing a variable name. In the method meta-variables definition, the whole method signature is specified; if necessary, type meta-variables, i.e., a variable whose values range on types, can be used. Meta-variables got a value during the pointcut evaluation and their values can also be used by the advice. This permits to decouple the join point pattern from the application code.
The behavior we are looking for is characterized by: i) the call to a method with any signature, ii) whose body either writes a field of the target object (i.e., a method belonging to the `put` family with any name and signature) or reads a field of the target object (i.e., a method belonging to the `get` family with any name and signature). This join point pattern explicitly refers to the concept of a method that changes the `Buffer` state rather than trying to capture that concept by relying on implicit rules such as naming conventions about the program implementation structure. In particular, in the caller swimlane[2], we look for the invocation of a generic method named `foo(..)`[3] whereas in the callee swimlane we look at the method body for the assignment to a generic field of the class (i.e., the behavior of a method of the `put` family) or, at either the use of generic class field into the right part of an assignment or the use of the field in a return statement (i.e., the behavior of a method of the `get` family). The former should be an exact statement match, — i.e., we are looking for exactly that call — whereas in the latter we are looking for a specific use of a field in the whole method body. This difference is expressed by using the following JPP syntax:

– a yellow rounded rectangle, called *template action*, indicates that we are looking for a meta-variable into a specific kind of statements in the searching scope, indicated by a stereotype; the ≪`method`≫ stereotype limits the search to the method body whereas the ≪`block`≫ stereotype to the current block;
– we can look for the use of a meta-variable in a left (`left`) or right (`right`) part of an assignment, in a boolean expression (`booleanCondition`), and in a generic statement (`statement`) or in their logic combination;
– a red rounded rectangle identified by the ≪`exactmatch`≫ stereotype, called (according to UML) *action*, indicates one or more statements, which must exactly match a sequence of code; the names used inside these blocks can be meta-variables, constant variable names (i.e., variable names used into the code) or if not useful to the pattern definition indicated as `(i)` with $i \in \mathbb{N}$.

The join point possible location is indicated by the ≪`joinpoint`≫ stereotype attached to an arrow in the case of local join points and by the ≪`startjoinpoint`≫ and ≪`endjoinpoint`≫ stereotypes attached to the arrows to denote the borders of a region join points. All the searched join points name are listed in the window in the low-right corner of the join point pattern specification.
The join point pattern in Fig. 1 explicitly refers to the elements insertion and recovery behavior rather than trying to capture those behaviors by relying on naming conventions about the program implementation, such as `put*` and `get*`. Consequently, the pointcuts defined by using this pattern do not change when the base program evolves as long as the join points it has to capture still conserve the same properties; so if a new method inserting two elements in the buffer is added to the `Buffer` class it is captured by our join point pattern as well independently of its name.
We have adopted a loose approach to the description of the computational flow. In the join point patterns, based on activity diagrams, the lines with a solid arrowhead connecting two elements express that the first immediately follows the second, and the lines with a stick arrowhead (see Fig. 1) express that the first follows the second before or later, i.e., zero or more *not relevant* actions[4] could happen before the second action, the number of actions that could happen is limited by the scope.
In most of the existing approaches to AOP, the pointcuts refer to properties of individual join points in isolation without reference to contextual information. Our join point pattern definition language can express temporal relations between events and actions inside the join point pattern definitions. In particular, a join point can be selected only if a specific action is already occurred or will occur in the next future. The future prediction is feasible because the design description depicts all the behavioral information and therefore evaluated at compile time, if it does not involve dynamically computed values.
Our join point model is strictly based on the computational flow, so we do not need to differentiate between **`before`** and **`after`** advice but we can simply attach the ≪`joinpoint`≫ stereotype in the right position, i.e., before or after the point we want to advice. A

[2]A swimlane is a way to group activities performed by the same actor/object.
[3]Note that `foo(..)` is a method meta-variable and its signature is not specified.
[4]These actions do not participate in the description of the join point position, so they are considered not relevant.

special case is represented by the *region join point*s which match portions of the computational flow instead of a single points; the whole matched portion represents the join point and will be substituted by the advice code.

3.2 Aspects that Use Join Point Patterns

A join point pattern simply describes where the join points can be found, to complete the process we must declare an aspect where the join point patterns are in association with advice code to weave at the interested join points.

The aspect definition, like in most AOP languages, includes pointcut and advice definitions and their relations. Moreover, it declares all the used join point patterns and which join points it imports from them. Both pointcuts and advices will use these information in their definition.

The following `Observer` aspect imports the `produce` and the `consume` join points from the `ObserverPattern` join point pattern (see Fig. 1). The join point patterns can define many join points but only a subset of these join points could be imported by the aspects.

```
public aspect Observer {
  void notify() { ... }
  public joinpointpattern ObserverPattern(produce,consume);
  public pointcut p() : produce();
  public pointcut c() : consume();
  advice() : p() && c() {notify();}
}
```

The pointcuts could also be defined as a logical combination of the imported join point definitions.

4. WEAVING IN JPP

One central component in AOP is the weaver. Given a set of target programs and a set of aspects, the weaver introduces the code of the advices at the captured join points in the target programs during the weaving process. Our approach does not differ in that and the weaving process must be realized.

Notwithstanding the join point patterns are language independent, the weaving process strictly depends on the program it has to modify. At the moment, we have chosen the Java 5 programming language because its meta-data facility, by providing a standard way to attach additional data to program elements, has the potential to simplify the implementation of the weaving process.

The weaving process in JPP consists of the following phases:

- *pre-weaving phase*: the abstraction level of the join point patterns and of the Java bytecode is equalized;
- *morphing/matching phase*: the matching is performed by traversing the model/graph of the pattern and the model/graph of the program in parallel;
- *join points marking phase*: when the pattern and the program models match, each captured join point is annotated at the corresponding code location;
- *advice weaving phase*: the annotated bytecode is instrumented to add the advice code at the captured join points.

4.1 Pre-Weaving Phase

The target program and the join point patterns are at a different level of abstraction. To fill this gap and allowing the weaving, it is necessary to build a common representation for the target program and the join point patterns (the *pre-weaving phase*). Structured graphs [2] perfectly fit the problem; both program computational flow (through its control flow graph) and join point patterns can be represented by graphs and the structured graphs provide a graph representation and manipulation mechanisms at variable level of details. Relaxing the quantity of details used in the control flow graph it is possible to fill the gap with the join point patterns.

A structured control flow graph is generated from the control flow graphs of each method by using BCEL [5] on the application bytecode and imposing a structure on that. Each instruction is a node of the structured flow graph and each method call is a macro-node, i.e., a node that can be expanded to the called method control flow graph. The structured graphs are stored in a special structure that separates the content from the layout saving the space and improving the efficiency of the navigation and of the layout reorganization (particularly useful on already partially woven programs). To simplify the matching of some join point patterns, an index has been built on the graph to provide access points that differ from the `main()` method.

Analogously, each join point pattern is stored into a structured graph. Since, the join point patterns already have a graph structure the conversion is less problematic. In this case the macro-structure provide a mechanism to navigate between different swimlanes and to skip some context details, e.g., in the case of a template actions.

4.2 Morphing/Matching Phase

The morphing/matching phase consists of looking for (matching) the join point patterns in the application control flow graph. Since, the basic elements of our pointcut definition language are expressed in a UML-like form, and the UML is a diagrammatic language, it is reasonable and promising to apply techniques developed in the graph grammar and graph transformation field to get our goal.

In particular, we have to solve a *graph matching problem* or better a model-based recognition problem, where the model is represented as a (structured) graph (the *model graph*, G_M), and another (structured) graph (the *data graph*, G_D) represents the program control flow graph where to recognize the model. In model-based pattern recognition problems, given G_M and G_D, to compare them implies to look for their similarities.

Graph and sub-graph isomorphism are concepts that have been intensively used in various applications. The representational power of graphs and the need to compare them has pushed numerous researchers to study the problem of computing graph and sub-graph isomorphisms. What we need is a inexact graph matching [18], or better a *sub-graph inexact matching* since one graph (the join point pattern representation) is smaller than the other (the program control flow graph representation).

Ours are attributed graphs, i.e., their vertices and edges contain some extra information, such as instructions (both for application and join point patterns actions), template actions and *loose connection* (stick arrows) for the join point patterns. Therefore, the type of our matching algorithm cannot be exact because the matching between corresponding parts of two graphs is possible even if the two parts are not identical, but only similar according to a set of rules involving predefined transformations. The inexactness implies that the join point pattern graph is considered matchable with the application one if a set of syntactic and semantic transformations can be found, such that the transformed join point pattern graph is isomorphic to a portion of the application graph.

During the morphing/matching phase, all the join point pattern graphs must be compared against the application control flow graph, the number of necessary matching can be reduced by using the context information stored for every pattern. For each node of both graphs:

- if the current pattern element is a real Java instruction (i.e. it

[5]`http://jakarta.apache.org/bcel/`

is from an *action* element), the algorithm tries to unify[6] it with the current application node;
– if the current pattern element is a *template action*, the algorithm matches it with the current application node when there is a semantic transformation that transforms the first into the second node; if the match fails, the algorithm iterates on the next application node inside the scope defined by the scope stereotype.

In both cases, the unification process can fail or success with a set of variable bindings, known as a *unifier*. Found a match, the outgoing edge of the current pattern element gives to the algorithm information about how to continue the match: according to the kind of edge, the next pattern element should match exactly the next application node (solid arrowhead) or should match one of the next nodes not necessary the first (stick arrowhead).
At the end of this process, the algorithm returns a set (that could be empty) of code locations for the captured join points.

4.3 Join Points Marking Phase

Each captured join point is marked directly in the bytecode by annotating its location. As already stated, Java 5 incorporates the concept of custom annotation that we could exploit in this phase. Unfortunately, Java annotation model permits of annotating only the element declarations, whereas we need to annotate the method body since the join points may be at every statement.
To overcome this problem, we have extended Java, codename @Java, to arbitrary annotate program elements such as blocks, single statements, and expressions. This work is partially based on the experience we have done in extending the annotation model of C# [4] that suffers of the same limitations. @Java minimally extends Java. The only real difference from the standard mechanism is related to the possibility of annotating a code block or an expression. In these cases, the annotation must precede the first statement of the block or the beginning of the expression to annotate and the whole block or expression must be grouped by braces to denote the scope of the annotation.
Every join point annotation, contains the join point pattern name, the join point name, the join point parameters, and when necessary, the run-time residual. Current implementation provides a compiler based implementation that does almost of the weaving work at compile time. This solves and then captures most of the join points at compile time and avoids unnecessary run-time evaluations and overheads. Some pointcuts, that needs run-time information to be evaluated, still cannot be completely evaluated at compile-time; in these cases the pointcut is reduced and a small residual for the not evaluated part is annotated at the potential join points waiting for the dynamic evaluation. Its evaluation will determine if the join point really has to be advised or not. However, the corresponding overhead is contained and the necessary residuals with our approach are less then in AspectJ. At the end of this phase the application is ready to be advised and to speed up the last phase an index on the captured join points is built.

4.4 Advice Weaving Phase

In this phase the bytecode application will be really modified. For every advice associated to a pointcut declaration we generate the bytecode of the advice by using the BCEL and in particular the `InstructionLists` structure. We retrieve the annotations associated to that pointcut to locate where the advice code must be inserted, then we use the `InstructionList.append()`, the `InstructionList.insert()` or the `InstructionList.delete()` methods to insert the advice `InstructionList` into the application `InstructionList`. Finally, when the instruction list is ready to be dumped to pure bytecode, all symbolic references must be mapped to real bytecode offsets. This is done by a call to the `getMethod()` method.

[6]Unification is the operation of merging information-bearing structures, without loss information if the *unificands* are consistent (monotonicity).

5. RELATED WORKS

This paper propose to decouple base programs and aspects by approaching the join points selection on a less syntactical and structural basis. To get a semantic join point model to avoid the fragile pointcut problem is a quite hot topic and several approaches are currently under investigations.
In [15], Noguera et al. present a mechanism to express type-safe source code templates in pure Java that improves the expressiveness of the pointcut definition languages. To have a more semantic pointcut language, they propose to match, not only on the signature, but also on the structure of the method. The basic idea is similar to ours, i.e., to identify join points not only on the base of method signature but also on method behavior, but their approach is less decoupled from the base program syntax and structure because it is applied at the code level.
In [10], Kellens et al. propose a novel technique of model-based pointcuts, which shifts the fragile pointcut problem to a more conceptual level where it is easier to solve. This is done by decoupling the pointcut definitions from the actual structure of the base program, and defining them in terms of a conceptual model of the software instead.
In [13] Klein et al. address aspect weaving at model level, including with behavioral modeling languages such as scenario languages. With behavioral models, aspect weaving can be performed either at the abstract syntax level, or at the semantic level. The two approaches have contact points, since UML sequence diagram and message sequence chart are very similar. But our work is more general because it considers also a different grain of details thanks to the use of the activity diagrams.
In [1], Allan et al. present a new history based language feature called *trace matches* where the programmer can trigger the execution of extra code by specifying a regular pattern of events in a computational trace. This is an interesting feature, but since it is based on AspectJ join point model a program's trace is modeled as a sequence of going in and out from AspectJ join points.
In [7], Douence et al. present a model which takes into account the history of the program execution to decide which aspect behavior to invoke. Trace-based aspects are defined on traces of events occurring during program execution, this kind of aspects are more expressive than those based on atomic points because they can express relations between execution events but exactly as in [1] the events are limited to the method calls and returns.
In [17], Ostermann et al. define a pointcut language (`ALPHA`) that allows to specify pointcuts at a high-level of abstraction by providing different models of the program semantics and abstraction mechanisms analogous to functional abstraction. This approach is an important step towards the semantic selection of the join points, but it is still related to the program implementation. In fact, the library of pointcuts used to query the program execution models, refers to the operations chosen to implement the business logic, and not to the semantic of the operation itself.

6. CONCLUSIONS

Current AOP approaches suffer from well known problems that rely on the syntactic coupling established between the application and

the aspects. A common attempt to give a solution consists of freeing the pointcut definition language from these limitations by describing the join points in a more semantic way.
This paper has proposed a novel approach to join points identification and to decouple aspects definition and base-code syntax and structure. Pointcuts are specified by using join point patterns expressed on the application design information. More precisely, a join point pattern is a template on the application behavior identifying the join points in their context. In particular join points are captured when the pattern matches portion of the application behavior.
Compared with current approaches, we can observe some advantages; first of all, we have a pointcuts definition more behavioral. In the join point pattern definition we identify the context of the computational flow we want to match, and the precise point we want to capture. Notwithstanding that, we can still select the join points by using syntactic and structural specification, it is only necessary a more detailed join point pattern. The graphical definition of join point patterns is more intuitive and comprehensible for programmers. Moreover, a visual view of the context where locating the join points would be preferred since it better demonstrates where and how an aspect can affect a program. Finally, since the design model provides an abstract and intuitive representation of the program behavior and component interrelations, we have at compile-time a detailed view of the static structure and/or dynamic behavior of the whole program. Thanks to this characteristic, we can manage the weaving phase nearly completely at compile-time. Last but not least, our approach is quite general, it can be applied to every programming language (at the cost of adapting the weaving algorithm to the characteristics of the new language) and used to mimic all the other approaches to AOP.
There is also a drawback, the matching phase is very complex, and it demands time and space. Fortunately, thanks to the abovementioned characteristics most of the weaving phase is done once during the compilation and does not affect the performance of the running program.

7. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching to AspectJ. TR abc-2005-1, The abc Group, Mar. 2005.

[2] M. Ancona, L. De Floriani, and J. S. Deogun. Path Problems in Structured Graphs. *The Computer Journal*, 29(6):553–563, June 1986.

[3] W. Cazzola, A. Cicchetti, and A. Pierantonio. Towards a Model-Driven Join Point Model. In *Proceedings of ACM SAC'06*, pages 1306–1307, Dijon, France, Apr. 2006. ACM Press.

[4] W. Cazzola, A. Cisternino, and D. Colombo. Freely Annotating C#. *Journal of Object Technology*, 4(10):31–48, Dec. 2005.

[5] W. Cazzola, J.-M. Jézéquel, and A. Rashid. Semantic Join Point Models: Motivations, Notions and Requirements. In *Proceedings of SPLAT'06 Workshop*, Bonn, Germany, Mar. 2006.

[6] W. Cazzola, S. Pini, and M. Ancona. Design-Based Pointcuts Robustness Against Software Evolution. In *Proceedings of RAM-SE'06 Workshop*, Nantes, France, July 2005.

[7] R. Douence, P. Fradet, and M. Südholt. Trace-Based AOP. In *Aspect Oriented Software Development*, chapter 9, pages 141–150. Addison-Wesley, Oct. 2004.

[8] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, USA, Oct. 2000.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Ma, USA, 1995.

[10] A. Kellens, K. Gybels, J. Brichau, and K. Mens. A Model-driven Pointcut Language for More Robust Pointcuts. In *Proceedings of SPLAT'06 Workshop*, Bonn, Germany, Mar. 2006.

[11] G. Kiczales. The Fun Has Just Begun. Keynote AOSD 2003, Boston, Mar. 2003.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and B. Griswold. An Overview of AspectJ. In *Proceedings of ECOOP'01*, pages 327–353, Budapest, Hungary, June 2001. ACM Press.

[13] J. Klein, L. Hélouët, and J.-M. Jézéquel. Semantic-based Weaving of Scenarios. In *Proceedings of AOSD'06*, pages 27–38, Bonn, Germany, Mar. 2006. ACM Press.

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

[15] C. Noguera and R. Pauwlak. Open Static Pointcuts Through Source Code Templates. In *Proceedings of Open and Dynamic Aspect Languages Workshop (ODAL'06)*, Bonn, Germany, Mar. 2006.

[16] H. Ossher and P. Tarr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proceedings of ICSE'01*, pages 729–730, Toronto, Ontario, Canada, 2001. IEEE Computer Society.

[17] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In A. P. Black, editor, *Proceedings of ECOOP'05*, LNCS 3586, pages 214–240, Glasgow, Scotland, July 2005. Springer.

[18] L. G. Shapiro and R. M. Haralick. Structural Descriptions and Inexact Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(5):504–519, 1981.

[19] N. Ubayashi, G. Moriyama, H. Masuhara, and T. Tamai. A Parameterized Interpreter for Modeling Different AOP Mechanisms. In *Proceedings of ASE'05*, pages 194–203, Long Beach, CA, USA, 2005. ACM Press.

[20] A. Vasseur. Dynamic AOP and Runtime Weaving for Java-How Does AspectWerkz Address It? In *Proceedings of the 2004 Dynamic Aspect Workshop (DAW'04)*, pages 135–145, Lancaster, England, Mar. 2004.