AOP for Software Evolution: A Design Oriented Approach*

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

Sonia Pini, Massimo Ancona
Dept. of Informatics and Computer Science,
Università degli Studi di Genova
{pini|ancona}@disi.unige.it

ABSTRACT

In this paper, we have briefly explored the aspect-oriented approach as a tool for supporting the software evolution. The aim of this analysis is to highlight the potentiality and the limits of the aspect-oriented development for software evolution. From our analysis follows that in general (and in particular for AspectJ) the approach to join points, pointcuts and advices definition are not enough intuitive, abstract and expressive to support all the requirements for carrying out the software evolution. We have also examined how a mechanism for specifying pointcuts and advices based on design information, in particular on the use of UML diagrams, can better support the software evolution through aspect oriented programming. Our analysis and proposal are presented through an example.

Keywords

AOP, SW Evolution, Design Information, UML, Join Point Model.

1. INTRODUCTION

In [1], *software evolution* is defined as a kind of software maintenance that takes place only when the initial development was successful. The goal consists of adapting the application to the ever-changing, and often in an unexpected way, user requirements and operating environment.

Software evolution, as well as software maintenance, is characterized by its huge cost and slow speed of implementation. Often, software evolution implies a redesign of the whole system, the development of new features and their integration in the existing and/or running systems (this last step often implies a complete rebuilding of the system).

Besides, software systems are often asked for promptly evolving to face critical situations such as to repair security bugs, to avoid the failure of critical devices and to patch the logic of a critical system. It is fairly evident the necessity of improving the software adaptability and its promptness without impacting on the activity of the system itself. This statement brings forth the need for a system to manage itself to some extent, to dynamically inspect component interfaces, to augment its application-specific functionality with additional properties, and so on.

Nonstopping applications with a long life span are typical applications that have to be able to dynamically adapt themselves to sudden and unexpected changes to their environment. Therefore, the support for run-time adaptive software evolution is a key aspect of these systems. Software evolution of a generic system is usually carried out by stopping the system and manually, or with the aid of specific tools, changing the system behavior according to the required evolution. A more dynamic approach consists of encapsulating the system prone to be adapted in a monitoring system that wait for an event. When the event occurs, it plans a countermove that will imply the automatic and dynamic evolution of the monitored system. The monitoring system also takes care to grant the safety and stability of the monitored system against its evolution. Some examples of this approach are [3] and [6].

Independently of the mechanism adopted for planning the evolution, it requires a mechanism that permits of concreting the evolution on the running system. In particular this mechanism should be able of i) extruding the code interested by the evolution from the whole system code, ii) carrying out the patches required by the planned evolution on the located code. Both these steps must occur without compromising the system stability and the services availability (that is, without stopping the system providing them).

Aspect-oriented programming (AOP) [9] provides some mechanisms (*join points*, *pointcut* and *aspect weaving*) that allow of modifying the behavior and the structure of an application, also of a nonstopping application (*dynamic weaving* [14]). The AOP mechanisms better address functionality that crosscut the whole implementation of the application. Evolution is a typical functionality that crosscut the code of many objects in the system. Moreover, the AOP mechanisms seem suitable to deal with the detection of the code to evolve and with the instrumentation of such code. In our view, AOP will play the role of the low-level tool used to render effective the planned evolution.

The rest of this paper is organized as follows. Section 2 analyzes the AOP as a support to software evolution. This analysis has permitted to put in evidence benefits and drawbacks of the aspect-oriented (in particular of AspectJ) approach when applied to the evolution of a system. In section 3, we have proposed our approach to pointcut and join point definition based on the UML methodology. This proposal should overcome the drawbacks stressed in section 2. Finally in the sections 4 and 5 some related and future works are faced and some conclusions are drawn out.

2. AOP AND SOFTWARE EVOLUTION

AOP [9] is both a designing and programming technique that

*This work has been partially supported by Italian MIUR (FIRB "Web-Minds" project N. RBNE01WEJT_005).

SAC '05, March 13-17, 2005, Santa Fe, New Mexico, USA

takes another step towards increasing the kinds of design concerns that can be cleanly captured within source code. Its main goal consists of providing systematic methods for the identification, modularization, representation and composition of crosscutting concerns such as security, mobility and real-time constraints. The captured aspects (both functional and nonfunctional) are separated into well-defined modules that can be successively composed in the original or in a new application.

The basic mechanisms for separating the crosscutting concerns in aspects and for weaving them together again are: *join points*, *pointcut*, and *advice*. *Join points* represent well-defined points in the execution of a program, such as method calls, object field accesses and so on. *Pointcut* is a construct that picks out a set of join points based on given criteria, such as method names and so on. Pointcuts serve to define which advice has to be applied. An advice defines additional code to be executed at join points picked out by the pointcuts. Finally, an aspect represents a crosscutting concern and is composed of pointcuts definition and advices to be weaved at the corresponding join points. The frame that renders possible the proper execution of the assembled program is called *join point model*.

AspectJ [8, 11] has been the pioneer of the aspect-oriented languages and it is still one of the most relevant frameworks supporting the AOP methodology. In AspectJ an aspect looks like:

```
aspect <aspect name> {
    /* pointcut definitions */
  pointcut <pointcut designator> : <join points description>;
    /* advice definitions */
  <advice type> : <pointcut designator> { <advice body> }
}
```

As defined in [8], join points are basically described by composing explicit method signatures with predicates on the execution flow, i.e., the given kind of join points. An example of join point description is: `call(* *.print(..))`; it describes all the join points at the invocation of the methods named `print`, does not matter which object receives the message, its return type and how many arguments it needs. Whereas, the advice type suggests the point with respect to the join point where the advice body will be weaved, some examples are **`before()`**, **`after()`** and **`around()`** whose behavior comes after their name.

However, in this last few years many other frameworks have been developed, some of them are: AspectWerkz [18], Josh [5] and JMangler [10]. Notwithstanding that this paper takes AspectJ as referring AOP framework, many of the considerations we do can be also applied to other frameworks.

From AOP characteristics, it is fairly evident that AOP has the potential for providing the necessary tools for instrumenting the code of a nonstopping system, especially when aspects can be plugged and unplugged at run-time. Pointcuts should be used to pick out a region of the code involved by the evolution, whereas the advices should be used to define how the code — identified by the pointcut — should evolve. Weaving such an aspect on the running system should either inject new code or manipulate the existing code, allowing the system dynamic evolution.

Unfortunately, to define pointcuts that point out the code interested by the evolution is a hard task because such modifications can be scattered and spread all around the code and not confined to a well-defined area that can be taken back to a method call. Moreover the changes could entail only part of a statement, e.g., the exit condition of a loop or an expression, and not the entire statement. To highlight the entity of the problem we can consider the implementation of a simple bounded buffer with `get()` and `put()` operations with the usual semantics.

```
public class BoundedBuffer {
  private int first = 0;
  private final int MAX = 20;
  private int buffer[] = new int[MAX];

  public void put(int n) throws FullBufferException {
    if (first < MAX) buffer[first++] = n;
    else throw new FullBufferException();
  }
  public int get() throws EmptyBufferException {
    if (first > 0) return buffer[--first];
    else throw new EmptyBufferException();
  }
}
```

Listing 1: Bounded Buffer in Java

The Java code, reported in listing 1, despite of its simplicity, is enough complex to be used to explain the hardness of determining all the code involved by an attempt of evolution. At this point, we consider a change to the system requirements that forces a change to the applicability rule of the method `get()`. After that, the method `get()` can be invoked only after one or more invocations of the method `put()` and not immediately after another invocation of the method `get()`. At a first glance could seem that the changes are confined in the body of the method `get()` since it is the only method whose requirements change, of course, this impression is not true.

```
public class BoundedBuffer {
  private int first = 0;
  private boolean lastIsPut = false;
  private final int MAX = 20;
  private int buffer[] = new int[MAX];

  public void put(int n) throws FullBufferException {
    if (first < MAX) {
      buffer[first++] = n;
      lastIsPut = true;
    } else throw new FullBufferException();
  }
  public int get() throws NotEnoughPutsException {
    if (lastIsPut) {
      buffer[first++] = n;
      lastIsPut = false;
    } else throw new NotEnoughPutsException();
  }
}
```

Listing 2: Evolved Bounded Buffer

Rather, the whole class code is affected by the required evolution:

- a new boolean attribute (`lastIsPut`) has been introduced in the class, if it holds true the method `put()` is the last invoked, it will hold false otherwise;
- each time the method `put()` is invoked the new attribute `lastIsPut` must be set to **`true`**;
- the precondition to the call of the method `get()` changes to satisfy the new constraint[1], moreover, the flag `lastIsPut` must be set to **`false`**.

[1]Note that, if you can invoke `get()` only after a `put()` has occurred, it is impossible to invoke the method `get()` on an empty buffer.

The listing 2 shows the evolved bounded buffer class, to give more emphasis to the changes, they are written in gray. Therefore, it is fairly evident that our simple test has spawned several noisy and punctual changes that are difficult to deal with (both for maintenance, flexibility and clarity).
As said before, the AOP technology could be the right approach to deal with the evolution concern but scenarios similar to the one described by our example are difficult to administrate with the current aspect-oriented frameworks. The main issues that obstacle the use of the current AOP approaches are: the *granularity* of the requested manipulation and the *locality* of the code to manipulate. The requested granularity for the pointcut is too fine, traditional join point models refer to method invocation in several way whereas we want to be able to manipulate a single statements or a group of statements in somehow related. This means that we can manipulate the method execution at its beginning and at its ending but we can not alter its computational flow requiring the execution of another statement between two statements of its body.
In a limited way, we could work around the problem by extruding each group of statements interested by the evolution to the body of a method and replacing their occurrence with an invocation of such a method. Moreover we should (separately) define a specific pointcut (and related advices of course) for each join point that as to be manipulated. This solution is not always practicable because (neglecting the fact that it forces a manual refactoring of the original system and it is a little bit tricky):

- it is too fragmentary and therefore error prone when the spectrum of the evolution grows in size (how we could be sure that everything has been taken in consideration?);
- it is tailored on a specific case and does not permit to describe general pointcuts, for example, it can not be associated to a trace of the program execution;
- it is not applicable when the code interested by the evolution can not be promoted to a method, e.g., two interleaved statements or just part of a structured statement or expression;
- it strongly depends on the syntax of the program rather than on its semantics, that means that we can not use a single pointcut to describe the join points associated to two methods with the same behavior but with a different name;
- the removal of a statement is not so immediate and simple.

These problems are due to the poor expressiveness of the pointcut definition languages and of the related join point models provided by most of the actual AOP frameworks. Nowadays AOP languages provide very basic pointcut definition languages that heavily rely on the structure and syntax of the software application neglecting its semantics. The developer has to identify and to specify in the correct way the pointcut by using, what we call the *linguistic pattern matching* mechanism; it permits of locating where an advice should be applied by describing the join points as a mix of references to linguistic constructs (e.g., method calls) and of generic references to their position, e.g., before or after it occurs. Therefore, it is difficult to define generic, reusable and comprehensible pointcuts that are not tailored on a specific application. Moreover, current join point model is too abstract. Join points are associated to a method call whereas a finer model should be adopted to permit of altering each single statement.
Similar issues have been raised from several researchers that are providing their own pointcut language or join point model, some examples are [17, 7]. More or less each proposal addresses part of the problem but often the solution is not so intuitive as it is necessary to be usable.

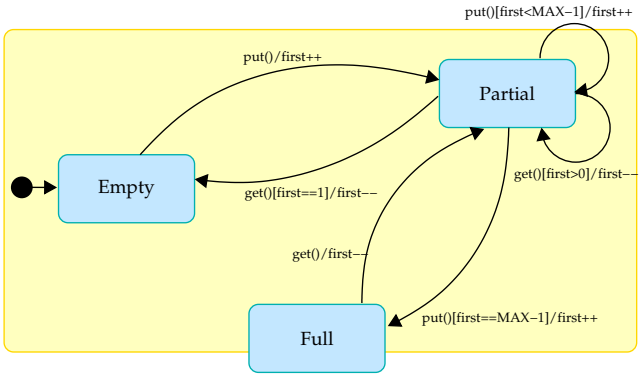

Figure 1: The Statechart of the Bounded Buffer

3. FROM UML DIAGRAMS TO ASPECTS DEFINITION

Pointcut definition and related problems have been studied by several researchers [17, 7]. In all their works they propose to use a more expressive pointcut definition language mainly based on logic deduction and pattern matching. Notwithstanding the powerfulness of their proposals, they do not provide a straightforward and easy approach to pointcut definition. Moreover the degree of abstraction and the relative granularity seems inadequate for the software evolution.
The question is: *does it exist a tool for describing the pointcuts that is independent of the program syntax, intuitive and easy to use?* In our opinion, the UML [2] methodology with its variety of diagrams fits the problem. The UML methodology forces the developer to describe each software system by using a set of diagrams. These diagrams take care of representing every aspect of the system, from its structure (e.g., class and object diagrams) up to its behavior (e.g., statecharts, sequence and activity diagrams). Moreover, these diagrams describe the behavior of the system independently of the syntax adopted in the code but rather in terms of the trace of its execution. Therefore, these diagrams should provide all the necessary data for the evolutionary purpose, they describe such data as a whole and abstracting from any syntactic description and, because of their pictorial nature, they are also characterized by an higher degree of intuitiveness.
Coming back to the *bounded buffer* example, the approach provided by AspectJ, and by other AOP frameworks, does not provides the necessary granularity and degree of abstraction to permit its evolution in a straightforward manner. In figure 1, it is shown the statechart of the bounded buffer. The statechart describes the behavior of the bounded buffer in terms of its states and of the operations that force a change of state. The transition arrows express these changes and they are labeled by a triplet $<$*event*, *condition*, *action*$>$. These triplets well identify the portion of code that provokes the change of state. The couple $<$*event*, *condition*$>$ is used to express when the corresponding action can be performed, i.e., to define which constraints limit the applicability of the action. Of course, the action represents the code whose execution effectively provokes the change of state. The *event* it is used to identify the triggering event but it could be also used (as shown in the reported diagrams) to locate the action's code by letting coincide the label with the name of the method embodying that action. Moreover, a

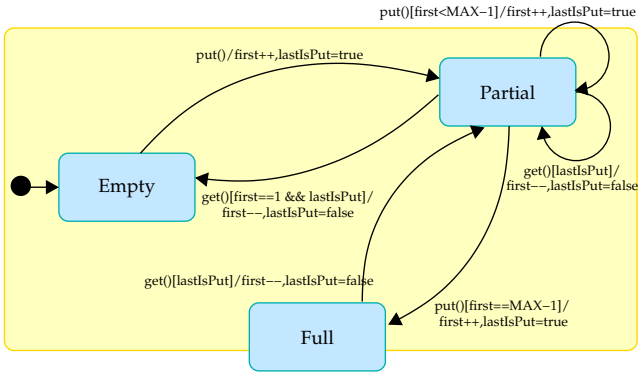

Figure 2: The Statechart of the Evolved Bounded Buffer

statechart can describe the behavior of all the instances of a class or of a specific instance as well. From these considerations, it is evident that the statechart describes the behavior of the bounded buffer in a compact and intuitive way providing also several levels of granularity (e.g., method calls are differentiated according to the state of the invoking object). Besides, the other diagrams (e.g., the sequence diagrams) permit, getting similar results, to deal with a trace of the program execution pointing out, for example, a particular sequence of method calls otherwise not manageable with the current pointcut definition language.
Hence, the UML diagrams provide a mechanism clear, intuitive and powerful for identifying portions of code associated with particular executions of the program but this is not enough to allow the evolution of a system: it is still missing a mechanism to determine how the system has to evolve, of course a mechanism that provides an high degree of intuitiveness, flexibility and granularity. In our opinion, the UML diagrams provide again the solution. The evolved system, as well as the original system, can be modeled by using the UML diagrams. The difference between such diagrams before and after the evolution represents the evolution itself. Whereas the original diagrams determine the code to be adapted, the evolved diagrams specify how the code has to be adapted, therefore the former contribute to the pointcuts/join points definition, the latter contribute to the advices definition.
Figure 2 shows the statechart of the evolved bounded buffer. By comparing the diagrams in figure 1 with the one in figure 2 it is possible to understand how the new semantics of the method `get()` affects the whole behavior of the class and which code is involved. Each transition arrow stresses the portion of code subject to the evolution in a specific state and how it evolves. Just as an example, we are going to examine how the adaptation impact on the transition arrow from the *partial* state to the *partial* state (`get()` event). In the original design the arrow was labeled with:

```
get()[first>0]/first--
```

whereas, to respect the new requirements, the label changes in:

```
get()[lastIsPut]/first--,lastIsPut=false.
```

In this case, the triggering event is still the invocation of the method `get()` but both the condition and the action change. Originally, the method `get()` could be invoked when at least a value was contained in the buffer that it is expressed by the condition `first>0`. Then this constraint has been overwhelmed by the new semantics and it has been replaced by: the method `get()` can be invoked just after the invocation of the method `put()`; condition expressed by the boolean flag `lastIsPut`. Analogously, the action varies to deal with the boolean flag `lastIsPut`, that is, it is enriched by the statement: `lastIsPut=false`.
A criticism to the use of the UML diagrams to describe join points, pointcuts and advices could be moved to the fact that the diagrams have a pictorial nature and therefore to extract information from them is a difficult job. In the RAMSES project, Cazzola et al. [3, 4] showed how to use the design information to evolve a system. They deal with the UML diagrams not in their pictorial form but encoded in the XMI [12] language. The XMI is a standard variant of the XML language designed to render easy the extraction of features from the UML diagrams. Moreover, as shown in [4], the use of XMI consents to automatically determine the extent of the required evolution by comparing the diagrams: original and modified.
In [13], Pintér et al. show how to automatically generate code from statecharts. A similar approach can be applied to recognize the code identified by the UML diagrams. In this way, it can be possible to preprocess the (byte-)code and mark by exploiting metadata code annotations (as supported by .NET or Java 1.5) the join points that could be interested by the evolution. Annotations will play the role of the hooks, in the code to be adapted, where the advices will be woven. Annotations have the benefit to be supported by standard programming environments and to be skipped during the normal execution, i.e., in this case, when no aspect is woven on that annotation; therefore they should not add extra penalties during the execution.

4. RELATED WORKS

Software evolution is a topic widely debated in the last thirty years, several proposals have been presented each with its merits and demerits. Aspect-oriented software evolution is a more recent topic but notwithstanding that several work has already been done, too much to be faced in this short overview. Hence, we will take in consideration the works that adopt the UML diagrams for driving the system evolution and the works that propose a different approach to pointcuts or join points definition.
Of course, the most related work is the RAMSES project [3] from which this work has been spawned. RAMSES is a reflective middleware that uses the design information (the UML diagrams) as meta-data automatically driving the evolution of a software system by deciding the extent of the evolution and which code is affected by such an evolution. The current work can be considered as the backbone of the RAMSES middleware.
As previously stated many researchers are providing their own extension to the current aspect-oriented mechanisms (mainly to the pointcuts definition). Tourwé et al. [17] have proposed an advanced pointcut managing environment, based on machine learning techniques. They identify three main problems for the current-day AOP languages, the first is that the pointcut language is too primitive and not expressive enough, the second is that pointcuts are very tightly coupled to an application's structure and, the last is that developers are forced to deal with pointcuts at too low level. As a solution of the above problems they propose to include the notion of *inductively generated pointcuts* in the AOP language, in this way developers can specify pointcuts by using a graphical interface and the framework will automatically generate the corresponding pointcuts. Gybels et al. [7] 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 of the naming convention, as highlighted in our work. Gybles et al. have proposed a more flexible linguistic mechanism to implement crosscutting as patterns and consequently avoiding the exposed pattern matching problem.

Sillito et al., in [15], have highlighted the importance of using *use case* diagrams in the pointcut definition, our idea is quite similar but we do not want to define a novel pointcut definition language, as AspectU, that needs a special interpreter or to be mapped on an existing AOP language, as AspectJ. Rather we would like to extend an existing pointcut language and act on the weaving mechanism to support UML-based join points. Stein et al. [16] have proposed a new design model for the development of aspect-oriented programs by using the UML. They face the problem of how to design aspect-oriented crosscutting concerns by extending the UML by exploiting its standard extension mechanisms, whereas we are exploiting the UML diagrams to automatically generate the required system evolutions.

5. FUTURE WORK AND CONCLUSIONS

In this paper, we have analyzed aspect-oriented development techniques in relation with the software evolution problem. In particular, we have focused our analysis on the AspectJ framework and its join point model, notwithstanding the work is tailored on a specific framework many considerations we have done still hold on many other aspect-oriented frameworks.

From our examination results that it is an hard job to determine the code that has to be evolved and to manage its evolution with the join point model adopted by AspectJ. The main problems are the granularity of the join point model and its dependence of the structure and syntax of the system to adapt. We have showed that the UML methodology provides a more flexible, abstract and complete model to use as a basis for a join point model more adequate for the software evolution issues.

Our analysis is the first step in the definition of a join point model based on design information and appropriate for the software evolution. In the future, we are going to develop an extension to the join point model of AspectJ based on the presented ideas. We will also define a pointcut definition mechanism and an aspect weaver for our join point model. Finally, we will integrate it in the RAMSES project.

6. REFERENCES

[1] Keith H. Bennett and Václav T. Rajlich. Software Maintenance and Evolution: a Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 75–87. ACM Press, 2000.

[2] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, third edition, February 1999.

[3] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Software Evolution through Dynamic Adaptation of Its OO Design. In H.-D. Ehrich, J.-J. Meyer, and M. D. Ryan, editors, *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software*, LNCS 2975, pages 69–84. Springer, July 2004.

[4] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. System Evolution through Design Information Evolution: a Case Study. In W. Dosch and N. Debnath, editors, *Proceedings of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE 2004)*, pages 145–150, Nice, France, on 1st-3rd of July 2004. ISCA.

[5] Shigeru Chiba and Kiyoshi Nakagawa. Josh: An Open AspectJ-like Language. In *Proceedings of the 3rd Int'l Conf. on Aspect-Oriented Software Development (AOSD'04)*, pages 102–112, Lancaster, UK, March 2004.

[6] Jim Dowling and Vinny Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Software. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of Reflection'2001*, LNCS 2192, pages 81–88, Kyoto, Japan, September 2001. Springer-Verlag.

[7] 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.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeff Palm, and Bill Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 327–353, Budapest, Hungary, June 2001. ACM Press.

[9] 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.

[10] Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler - A Powerful Back-End for Aspect-Oriented Programming. In R. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-oriented Software Development*, chapter 9. Prentice Hall, 2004.

[11] Ramnivas Laddad. *AspectJ in Action: Pratical Aspect-Oriented Programming*. Manning Pubblications Company, 2003.

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

[13] Gergely Pintér and István Majzik. Program Code Generation Based on UML Statechart Models. *Periodica Polytechnica Electrical Engineering*, 47(3-4):187–204, 2003.

[14] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic Weaving for Aspect Oriented Programming. In *Proceedings of the 1st Int'l Conf. on Aspect-Oriented Software Development (AOSD'02)*, pages 141–147, Enschede, The Netherlands, April 2002.

[15] Jonathan Sillito, Christopher Dutchyn, Andrew D. Eisenberg, and Kris De Volder. Use Case Level Pointcuts. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*, Oslo, Norway, June 2004.

[16] Dominik Stein, Stefan Hanenberg, and Rainer Unland. An UML-based Aspect-Oriented Design Notation for AspectJ. In *Proceedings of the 1st Int'l Conf. on Aspect-Oriented Software Development (AOSD'02)*, pages 106–112, Enschede, The Netherlands, April 2002.

[17] 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.

[18] Alexandre Vasseur. Dynamic AOP and Runtime Weaving for Java- How Does AspectWerkz Address It? In R. E. Filman, M. Haupt, K. Mehner, and M. Mezini, editors, *Proceedings of the 2004 Dynamic Aspect Workshop (DAW'04)*, pages 135–145, Lancaster, England, March 2004.