Language Support for Evolvable Software: An Initial Assessment of Aspect-Oriented Programming

Gianpaolo Cugola, Carlo Ghezzi and Mattia Monga

Politecnico di Milano - Dipartimento di Elettronica e Informazione
Piazza Leonardo Da Vinci, 32
20133 Milano – Italy

{cugola, ghezzi, monga}@elet.polimi.it

1. Introduction and motivations

Separation of concerns [6] is key to manage the complexity of understanding and evolving software. Software engineers have learned how to decompose a complex system into simpler sub-systems, with the goal of making the complexity of sub-problems tractable. Sub-problems are addressed relatively independently and the complete solution is built by gluing together the sub-solutions. The parts that compose the whole system are often modular units of functionality and this partitioning is well supported by existing programming languages, thanks to object orientation, functional decomposition, etc. Indeed, object-oriented programming languages, and the adoption of suitable programming techniques, are important steps in the direction of making software evolution easier. There are two major object-oriented concepts that affect evolution: encapsulation and inheritance. Encapsulation limits the effects of change to localized portion of code. Inheritance allows one to incrementally evolve a component by adding new features or redefining existing features. Both encapsulation and inheritance are instances of the more general concept of separation of concerns.

However, sometimes a concern is not easily encapsulated in a functional unit, because it *crosscuts* the entire system, or parts of it. Synchronization, memory management, network distribution, load balancing, error checking, profiling, security are all *aspects* of computer problems that are unlikely to be separated in functional units.

As an example, suppose that a Java class is used to describe the pure functionality of certain objects. Additional separate aspects may include:

- the definition of constraints on sequences of applicable operations (e.g., to get information from an object one must first apply a setup operations, and then one of a set of assignment operations);
- the definition of synchronization operations to constrain concurrent access to the object (e.g., a consumer trying to read a datum from a queue must be suspended if the queue is empty);
- the definition of how objects are distributed on the nodes of a network, either statically or through dynamic migration.

Each aspect should be clearly identifiable; it should be self-contained and easily changeable. Moreover, the various aspects should not interfere with one another. They should not interfere with the features used to define and evolve functionality, such as inheritance. That is, a composition algebra should be defined for the different linguistic features.

This position statement is organized as follows: Section 2 introduces the current stage of Aspect-Oriented Programming. Section 3 discusses the problems and pitfalls of current Aspect-Oriented languages. Section 4 provides some conclusions and motivates future briefly sets a research agenda.

2. A Brief Introduction to Aspect-Oriented Programming

The central idea of Aspect Oriented Programming (AOP) [7] is to separate the code that expresses an aspect (that is a property of the system not cleanly separable in a functional unit) from the code that expresses functional units. A *weaver* braids (not necessarily at compile-time) aspects with functional units to obtain the final system. Aspects are expressed by the means of an Aspect Oriented Language (AOL), whilst functional units are defined with a Component Language (CL). There can be a different AOL for each kind of aspect someone wants to cope with. In AspectJ 0.1 [8], an environment for aspect programming developed at Xerox PARC, the CL is Java and there is an AOL for synchronization (COOL) and an AOL for expressing remote invocation (RIDL). In the new version (0.3) of AspectJ [9], there is a unique general-purpose AOL that captures the crosscutting nature of aspects, independent of what those aspects are.

3. Problems and Pitfalls

In this section, we give a few examples taken from AspectJ and we discuss a number of drawbacks and pitfalls. More generally, our remarks enlighten the weaknesses of the current state of the art in AOLs and indicate directionsof future investigation. In our analysis of current AOLs, we found three main drawbacks:

- Possible clashes between functional code (expressed using a CL) and other aspects (expressed using one or more AOLs). Usually such clashes result from the need of breaking encapsulation of functional units to implement a different aspect. As an example, in AspectJ, aspect code may access the private attributes of a class. This can be useful in some situations but results in a potentially dangerous breaking of class encapsulation. Imagine a situation in which a class `Foo` has a private variable `i` that needs to be accessed by aspect `Bar`. Imagine also that subsequently class `Foo` is changed by changing type of variable `i` from `int` to `float`. This results in breaking the aspect code. In general, it is not possible to change the internals of a functional unit without changing other aspects.

- Possible clashes between different aspects. Suppose (see Figure 1) that a class `Point` exists with two variables `x` and `y` and two methods, `setX` and `setY`. Suppose we have developed an aspect `TraceBefore` to trace the start of execution of methods of class `Point` and an aspect `TraceAfter` to trace the end of execution of the same methods. The two aspects work perfectly when applied individually (for example, to trace the start of execution or to trace the end of it). Unfortunately, since they introduce the same method (i.e., method `print`) with different definitions, they fail when applied together.

```
class Example1 {
  public static void main(String args[]) {
    Point p=new Point();
    p.setX(1);
    p.setY(1);
  }
}

class Point {
  int x,y;
  public Point(){
    x=y=0;
  }
  public void setX(int x) {
    this.x=x;
  }
  public void setY(int y) {
    this.y=y;
  }
}
```

```
aspect TraceBefore {
  introduce private void
    Point.print(String methodName) {
    System.out.println("Tracing method " +
                       MethodName+" before");
    System.out.println("x="+x+"  y="+y);
  }
  advise void Point.setX(int i),
         void Point.setY(int i) {
    static before  {
      print(thisJoinPoint.methodName);
    }
  }
}
aspect TraceAfter {
  introduce private void
    Point.print(String methodName) {
    System.out.println("Tracing method " +
                       methodName+" after");
     System.out.println("x="+x+"  y="+y);
  }
  advise void Point.setX(int i),
         void Point.setY(int i) {
    static after  {
      print(thisJoinPoint.methodName);
    }
  }
}
```

Figure 1: An example of clash between two aspects

- Possible clashes between aspect code and specific language mechanisms. One of the best known examples of problems that falls into this category is *inheritance anomaly* [4]. This term was first used in the area of concurrent object-oriented languages [1,2,3] to indicate the difficulty of inheriting the code used to implement the synchronization constraints of an application written using one of such languages. In the area of AOP languages, the term can be used to indicate the difficulty of inheriting the aspect code in the presence of inheritance. As an example, consider class `Window` in Figure 2. Methods `show` and `paint` cannot be called before method `init` is called. This behavior is controlled by the aspect `WindowSync`. Now consider class `SpecialWindow` in Figure 3. It redefines method `show` in such a way that it does not require a previous invocation of method `init`. (Note that this way of subclassing `Window` is consistent with the OO type theory, which requires subclasses not to strengthen the precondition for redefined methods.) In principle, it should be possible to "inherit" the WindowSync aspect just modifying the code associated to method show (e.g., replacing it with the empty sequence). Unfortunately, this is not possible and it is necessary to rewrite entirely the aspect code (see aspect `SpecialWindowSync` in Figure 3).

All these problems show that AOP is still in its infancy. The experience gained in the area of concurrent object-oriented-languages [4] suggests that these problems might result more from thelinguistic choices made in developing AOLs, rather than from intrinsic limitations of the approach. The problem of finding adequate linguistic features which do not suffer from inheritance anomaly is thus an open research topic.

```
class Example2 {
  public static void main(String args[]) {
    Window w=new Window();
    w.init();
    w.show();
  }
}

class Window {
  public void init() {
    // ...
  }
  // Requires initialization
  public void show() {
    // ...
  }
  // Requires initialization
  public void paint() {
    // ...
  }
}
```

```
aspect WindowSync {
  introduce boolean Window.initDone=false;
  advise void Window.init() {
    static after {
      initDone=true;
    }
  }
  advise void Window.show(),
         void Window.paint() {
    static before {
      if(!initDone)
        System.out.println("Error: init never
called");
    }
  }
}
```

Figure 2: An aspect to control the sequence of invocation of different methods

```
class SpecialWindow extends Window {
  // This version of show does not
  // need any initialization
  public void show() {
    // ...
  }
}
```

```
aspect SpecialWindowSync {
  introduce boolean Window.initDone=false;
  advise void Window.init() {
    static after {
      initDone=true;
    }
  }
  advise void Window.paint() {
    static before {
      if(!initDone)
        System.out.println("Error: init never
called");
    }
  }
}
```

Figure 3: An example of inheritance anomaly

4. Conclusions and Open Issues

The goal of developing evolvable software should permeate all phases of software production: from requirements to specification to design and implementation. In this paper, we deliberately concentrated on the programming phase. We introduced AOP and showed that it is based on a conceptually appealing idea. AOP tries to provide linguistic mechanisms to factor out different aspects of a program, which can be defined, understood, and evolved separately. It pushes the idea of separation of concerns one step forward with respect to existing programming language constructs, which simply provide ways to encapsulate single functionality in a unit. Aspects in an AOP resemble ViewPoints in design and specification, as advocated by [10].

AOP, however, is still in its infancy. It is more an open research area than an existing technology that one can use. The problems and pitfalls we outlined in the previous section indicate that it is still unclear which constructs an AOL should provide and how they should interact with the functional language and the mechanisms provided to support functional evolution. As we observed, a fully general-purpose AOL, like AspectJ, with full visibility of the internal details of its associated

functional module, violates the principles of protection and encapsulation. On the other end, one might predefine a set of possible aspects an AOL should deal with, and then provide ad-hoc AOLs with constructs supporting limited visibility of certain features of the functional module to which the different aspects apply. The tradeoff is between flexibility and power, on one side, and understandability and ease of change on the other. (For a preliminary discussion of these points, see [12]).

In addition, we feel that aspects should be definable in a formal way. The formal definition will allow the AOL to define an algebra of aspect composition, clearly specifying when certain combinations of aspects are applicable (and what the effect is) or, conversely, when their combination is not possible or not defined, because it generates inconsistencies. Again, the problems arising here are strictly related to the ones being investigated in the case of viewpoints and viewpoint composition.

Research work at the programming language level should go hand-in-hand with experimental work, which should try to assess the usefulness and usability of the language. This is especially important since our claim is that AOP can be a vehicle to support evolvability, and this eventually will require some sort of experimental validation. [11] did an interesting initial experiment using AspectJ version 0.1. Experiments of similar kind will be needed, as further progress will be made in AOP technology.

References

1. Object-Oriented Concurrent Programming. A. Yonezawa and M. Tokoro editors. MIT Press, 1987.
2. G. Agha, Concurrent Object-Oriented Programming. Communications of the ACM, Vol. 33, No. 9, September 1990.
3. Research Directions in Concurrent Object-Oriented Programming. G. Agha, P. Wegner, and A. Yonezawa editors. MIT Press, 1993.
4. S. Matsuoka and A. Yonezawa, "Analysis of inheritance anomaly in object-oriented concurrent programming languages". *Research directions in Concurrent Object-Oriented Programming*, G. Agha, A. Yonezawa, and P. Wegner editors, the MIT Press, 1993.
5. C.Videira Lopes and G. Kiczales, "Recent Developments in AspectJ™". In Proceedings of the Aspect-Oriented Programming Workshop at ECOOP 98, 1998.
6. E.W. Dijkstra. "A Discipline of Programming". Prentice Hall 1976
7. Kiczales, Lamping, Mendhekar, Maeda, Lopes, Loingtier, Irwin, "Aspect Oriented Programming". Proceedings ECOOP97. Spinger Verlag 1997
8. AspectJ: Users Guide and Primer. XEROX Palo Alto Research Center.1998
9. AspectJ: Users Guide and Primer. XEROX Palo Alto Research Center.1999
10. A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, M. Goedicke, "ViewPoints: A Framework for Integrating Multiple Perspectives in System Development", *International Journal of Software Engineering and Knowledge Engineering*, March 1992.
11. R. J. Walker, E. L. A. Baniassad, and G. C. Murphy, "An Initial Assesment of Aspect-Oriented Programming". In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles (CA), May 1999.
12. G. Kickzales, J. Lamping, C. Videira Lopes, C. Maeda, A. Mendhekar, and G. Murphy, "Open Implementation Design Guidelines". In *Proceedings of the 19th International Conference on Software Engineering*, Boston (MA), May 1997.