A Static Extension of DynAMiT

Silvia Breu
FernUniversität in Hagen
silvia.breu@gmail.com

Jens Dörre
Universität Passau
doerre@neo.fmi.uni-passau.de

1 Introduction

With software systems becoming more and more complex, developers face increasing difficulties in building modular systems that cannot be tackled by "traditional" design and programming techniques. Anticipation of change can only be accomplished if the complexity of successive software releases is controlled and "code tangling" is limited. Aspect-oriented programming (AOP) [3, 4] has been developed to deal with these problems.

Recently, these ideas have also been used for reengineering. The major task here is to find and isolate crosscutting concerns, which is called aspect mining. Detected concerns can be re-implemented as aspects, which reduces complexity and improves maintainability and extensibility of software systems.

Several techniques have been proposed for aspect mining [5], including our DynAMiT approach [1, 2]. It mines aspects in program traces that are generated during program execution. These traces are investigated for recurring patterns of execution relations. We expect such recurring patterns to describe repeated functionality in the program and thus reflect potentially crosscutting concerns which can be replaced by aspects. Different constraints specify when a pattern is "recurring", such as the requirement that the relations have to exist more than once or even in different calling contexts in the program trace. The dynamic analysis approach has been chosen because it monitors actual program (i.e., run-time) behaviour instead of potential behaviour, as static program analysis approaches do.

However, DynAMiT's dynamic analysis has limitations that are partly due to dynamic binding at run-time. These can lead to the identification of aspect candidates that have already been encapsulated properly following object-oriented design principles. This paper describes a static extension of the approach that mitigates this problem.

2 Problem of Dynamically Mining Methods

Case studies conducted with DynAMiT have identified crosscutting concerns in small tools as well as in industrial-sized systems. Additionally, aspects that were added to systems using AspectJ were also recovered. However, while the approach is generally fairly precise, further analysis revealed that some false positives were systematically caused by dynamic binding.

Figure 1 illustrates that issue. `interface I` has two method declarations `a` and `c`. `class B` implements that interface, while `abstract class A` only implements method `a` of `I`. `abstract class A` is extended by two subclasses `C1` and `C2`, which both provide implementations of method `c` whose declaration is inherited from `I`.

```
interface I {                  class B implements I {
  public void a();               public void a(){}
  public void c();               public void c(){}
}                              }
abstract class A implements I {
  public void a(){}
}
class C1 extends A {           class C2 extends A {
  public void c(){}              public void c(){}
}                              }

class Runner {
  static void doSth(A a) {
    a.a();
    a.c();
  }
  static void doSth(B b) {
    b.a();
    b.c();
  }
  public static void main(String[] arguments) {
    A obj1 = new C1();
    A obj2 = new C2();
    B obj3 = new B();
    doSth(obj1);
    doSth(obj2);
    doSth(obj3);
  }
}
```

Figure 1: Example code of a software system

`class Runner` uses this hierarchy; its execution generates the trace in Figure 2(a). Here, the crosscutting algorithm incorrectly identifies the before-aspect candidates `A.a` $\rightharpoonup$ `C1.c` and `A.a` $\rightharpoonup$ `C2.c`. However, the underlying code pattern exists only once in the code, namely in `void doSth(A a)`. Hence, whenever there are abstract methods with several concrete implementations, the dynamic aspect mining algorithms will systematically produce false positives. This poses a real-world problem as dynamic binding is at the heart of object-oriented design and programming.

3 Idea: Static Extension of Traces

DynAMiT's algorithms can produce false positives due to dynamic binding at run-time because they do not work on the code but on method signatures only: Since each method call in the code can result in different implementations being executed, there are likely to be different signatures in the trace for this one call statement. This creates the spurious "different" calling contexts that may result in wrong aspect candidates.

If we now consider the static type of the reference objects in the traces, the program trace will change, as we see in Figure 2(b). There, the crosscutting algorithm will no longer detect the incorrect crosscutting concerns mentioned above. Thus, this could be the solution to the problem described in Section 2: An integration of static information into the traces would often allow to avoid that an invocation of the same functionality (i.e., occurring only

```
void Runner.doSth(A){
  void A.a(){}
  void C1.c(){}
}
void Runner.doSth(A){
  void A.a(){}
  void C2.c(){}
}
void Runner.doSth(B){
  void B.a(){}
  void B.c(){}
}
```

(a) 'Traditional' dynamic

```
void Runner.doSth(A){
  void A.a(){}
  void A.c(){}
}
void Runner.doSth(A){
  void A.a(){}
  void A.c(){}
}
void Runner.doSth(B){
  void B.a(){}
  void B.c(){}
}
```

(b) With static object info

Figure 2: Dynamic vs 'static' trace

once in the code) appears to be crosscutting in the traces.

4 Modified Tracing

The traces of a program for a given test suite are generated using AspectJ which offers two different kinds of pointcuts for methods. Execution pointcuts, which are used in the original DynAMiT version, use the dynamic information of the type *implementing* the method (i.e., after dynamic binding) at the callee's site. Call pointcuts, in contrast, use the static type information (i.e., the static type of the reference object) at the caller's site. Unfortunately, the choice of pointcuts influences the content of the traces. This is due to implementation limitations of AspectJ.

AspectJ currently uses byte-code or load-time weaving, and thus has no control over the API-(byte-)code. It is therefore not able to advise (i.e., use or instrument) pointcuts inside the API. A call to the API can be located in the client code, whereas an execution of an API method is located in the API; hence, it is not traced when using execution pointcuts. On the other hand, a call from the API to client code is inside the API, whereas its execution is in the client code. Such calls arise if the client code makes use of frameworks or callbacks; this is the case for AWT/Swing user interfaces using Listeners, for some other parts of the API, and even for a Java application's main method. These calls are not traced by method-call pointcuts (while calls *inside* them remain in the traces). Nevertheless, calls to the API are in general more common than callbacks, so it is safe to say that call pointcuts will monitor a much larger part of all method invocations than execution pointcuts.

5 Initial Evaluation

We evaluated the described static extension on the AnChoVis visualisation tool, which was also used in [1]. In comparison, we found less inside-aspect candidates, but in general substantially more outside-aspect candidates.

Most changes are due to the fact that the set of traced methods changed, as explained above. This has three different consequences: The set of aspect candidates tends to be larger, because more different method signatures are contained in the trace. On the other hand, it could also become smaller because some methods are no longer traced, for example `void anchovis.AnChoVis.main(String[])`. Finally, the aspect candidates themselves change because API method calls appear *in between* former relations. This affects in particular the firstIn-relations, because the parameters for a method (here the inner one) often have to be evaluated by API methods prior to the method call itself. Thus the `void anchovis.Logging.entering(String)` $\in_\top$ `...` candidate is replaced by `String java.lang.String.valueOf(Object)` $\in_\top$ `...` . This means that half of the logging concern is masked by an API method.

In the case study, the use of static object information proved to be beneficial. It eliminated wrong aspect candidates of the type shown in Section 3, and sometimes even detected new candidates as for example

```
String java.io.BufferedReader.readLine() ←
    BufferedReader anchovis.Data2Matrix.getReader(),
    BufferedReader anchovis.FunctionMapping.getReader()
```

It is a correct crosscutting concern because the two calls appear at different locations in the code: in the classes `Data2Matrix` and `FunctionMapping`. The original version of DynAMiT did not detect it, because there is only a single implementation of the `getReader()` functionality in the common superclass.

6 Conclusions and Future Work

Our idea to include static information in the analysis proved to be promising in this case study. In order to draw more general conclusions, however, it is necessary to conduct further case studies with large programs that have a deep inheritance hierarchy. Most differences to classic DynAMiT were due to additional API calls being traced (by byte-code instrumentation). Therefore the complete analysis became more fine-grained. This probably impairs program understanding and certainly the possibility of high-level mining for general crosscutting concerns like logging. However, it may enhance implementability of aspect candidates and thus ease refactoring.

We made two observations that may help to improve the proposed extended analysis: First, most of the "interfering" methods are in `class java.lang.String`, especially those implementing the concatenation operator. The number of masked aspect candidates is likely to decrease considerably if we exclude those methods/classes from the analysis. Second, the problems with string concatenation and parameter evaluation do appear in the linearised, compiled byte-code (and, of course, later on at run-time, and thus in every kind of trace). But they do not appear in or affect the analysis of source code. It could thus be worthwhile to consider a (static) analysis of the source code.

References

[1] S. Breu. Aspect Mining Using Event Traces. Master's thesis, U Passau, Germany, March 2004.

[2] S. Breu and J. Krinke. Aspect Mining Using Event Traces. *ASE-19*. IEEE Press, pp. 310–315, 2004.

[3] G. Kiczales et al. Aspect-Oriented Programming. *ECOOP 97*.

[4] P. Tarr et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. *ICSE-21*, pp. 107–119, 1999.

[5] A. van Deursen, M. Marin, and L. Moonen. Aspect Mining and Refactoring. *REFACE*, 2003.