Extending Dynamic Aspect Mining with Static Information

Silvia Breu

University of Passau*
94030 Passau, Germany
E-mail: silvia.breu@gmail.com

Abstract

Aspect mining tries to identify crosscutting concerns in legacy systems and thus supports the refactoring into an aspect-oriented design. We briefly introduce DynAMiT, a dynamic aspect mining tool that detects crosscutting concerns based on tracing method executions. While the approach is generally fairly precise, further analysis revealed that some false positives were systematically caused by dynamic binding. Furthermore, some aspect candidates were blurred or not detected due to not-sufficient tracing mechanisms of method executions when using AspectJ's execution pointcuts for the trace generation. We enhanced the mining capabilities of DynAMiT by taking additional static type information into account and generating the traces using call pointcuts instead: In an initial case study with AnChoVis, a 1300 LOC Java program, the number of mined aspect candidates increased by a factor of three, while the number of false positives remained zero.

1. Motivation

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. This notion refers to code that exists several times in the system but cannot be encapsulated by separate modules using traditional techniques [16]. The problem occurs if underlying functionality crosscuts the whole software system. Thus, tangled code makes software systems more difficult to maintain, to understand, and to extend. As the object-oriented paradigm has turned out to be insufficient in many cases, several approaches have been proposed, with *Aspect-Oriented Programming* [10] being the most prominent one. It provides better separation of complex concerns that cannot be encapsulated in traditional module systems.

Recently, attention has also been drawn to the question of how these ideas can be used for re-engineering. The major task here is to find and isolate crosscutting concerns. This is called *Aspect Mining*. Detected concerns can be re-implemented as aspects: Functionality belonging together is put in one module. This reduces complexity and improves maintainability and extensibility of software systems.

Several techniques have been proposed for aspect mining [6, 7, 8, 11, 12, 14, 15, 20], including our DynAMiT [1, 3], the first dynamic approach. It mines aspects from a program's trace of method invocations. These traces are investigated for recurring patterns of method executions. Recurring patterns describe repeated functionality in the program and thus potentially crosscutting concerns, which may be replaced by aspects.

A pattern is defined as a relation between two method invocations in the trace that respect certain temporal ordering constraints. These constraints specify when a pattern is recurring. For instance, for a pattern to be recurring it must be observed more than once in different calling contexts.

Traces of Java programs, as they are analysed by DynAMiT, reflect the dynamic (or runtime) binding of method calls: The object receiving a method call depends on the *runtime type* of the variable holding the object, not just its *static type*. This currently leads DynAMiT to identify false positives: aspect candidates that already have been encapsulated properly using object-oriented techniques. To remedy this drawback, we extended DynAMiT's original approach to take static types into account as previously proposed in [2].

To our knowledge it is the first technique that combines static and dynamic analyses and pushes towards hybrid aspect mining. The fusion of static and dynamic analysis overcomes limitations inherent in both static and dynamic approaches. Re-evaluation of original DynAMiT case studies shows that the new approach reduces the number of false positives, and detects between 167% and 200% more crosscutting patterns.

*Part of the work was done while at the FernUniversität in Hagen

The paper is organised as follows: Section 2 briefly presents the original DynAMiT and summarises the results of its evaluation. Section 3 illustrates detected problems and Section 4 describes the modifications we realised with the new DynAMiT version. We discuss the impact of these extensions on two initial case studies in Section 5, while Section 6 discusses related work. Section 7 concludes with possible directions for future work.

2. DynAMiT in a Nutshell

DynAMiT mines aspect candidates from method-execution traces and is written in Java. A trace is a long list of events, each denoting either that a method is entered or exited. As both, entering *and* leaving methods are traced, it is easy to deduce what methods a given method executed. Formally, a trace is defined as follows:

Definition 2.1 (Program trace)
Given a program P with a set of method signatures $\mathcal{N}_P$, a **program trace** *T_P of P is a list $[t_1, \ldots, t_n]$ of pairs $t_i \in (\mathcal{N}_P \times \{ent, ext\})$, where ent marks entering a method execution, and ext marks exiting a method execution.*

The nesting of method execution is more conveniently expressed as nesting parentheses, like the example in Figure 1. A method signature with a curly-braced body denotes the lifetime of a method activation. Other methods invoked by a method are placed inside the body. Figure 1 shows a trace of five methods: `A`, `B`, `C`, `D`, `E`. While `B` is active it executes `A`, which executes `D`; after `D` returns, `B` executes `C`, then `E`, which finally executes `D` twice.

```
1    A() {}
2    B() {
3      A() {
4        D() {}
5      }
6      C() {}
7      E() {
8        D() {}
9        D() {}
10     }
11   }
```

Figure 1. Representation of a program trace

Two method executions u and v in a trace T_P may be related. We define the following relations, abstracting the program trace, to identify crosscutting concerns:

- **Outside-before execution**. Method u is left immediately before v is entered, written as $u \rightharpoonup v$. This means, $[(u, ext), (v, ent)]$ is a sublist of T_P. For the abstraction of the trace in Figure 1 the following holds: `A()` $\rightharpoonup$ `B()`, `A()` $\rightharpoonup$ `C()`, `C()` $\rightharpoonup$ `E()`, and `D()` $\rightharpoonup$ `D()`.
- **Outside-after execution**. Method v is entered immediately after u is left, written as $v \leftharpoondown u$. It is the reversed outside-before execution relation. This means, $[(u, ext), (v, ent)]$ is a sublist of T_P. For the abstraction of the trace in Figure 1 the following holds: `B()` $\leftharpoondown$ `A()`, `C()` $\leftharpoondown$ `A()`, `E()` $\leftharpoondown$ `C()`, and `D()` $\leftharpoondown$ `D()`.
- **Inside-first execution**. Method u is called immediately after entering v, written as $u \in_\top v$. This means, $[(v, ent), (u, ent)]$ is a sublist of T_P. For the abstraction of the trace in Figure 1 the following holds: `A()` $\in_\top$ `B()`, `D()` $\in_\top$ `A()`, and `D()` $\in_\top$ `E()`.
- **Inside-last execution**. Method u is called immediately before v is left, written as $u \in_\bot v$. This means, $[(u, ext), (v, ext)]$ is a sublist of T_P. For the abstraction of the trace in Figure 1 the following holds: `D()` $\in_\bot$ `A()`, `D()` $\in_\bot$ `E()`, and `E()` $\in_\bot$ `B()`.

The above definitions relate method executions in a trace. The inside-first relation can be used, for example, to find invocations of a logging method that is executed at the beginning of a method. The key point is that such an execution pattern must be observed multiple times in a trace before it can be considered as a candidate for an aspect. Therefore, a definition of a *recurring* execution relation needs to be established.

Definition 2.2 (Uniformity)
Two methods `u` *and* `v`*, with* $\mathtt{u}, \mathtt{v} \in \mathcal{N}_P$*, are used uniformly in a trace T_P with respect to a relation $\circ \in \{\rightharpoonup, \leftharpoondown, \in_\top, \in_\bot\}$, when the following holds:* $\forall\ \mathtt{w} \circ \mathtt{v} : \mathtt{w} = \mathtt{u}, \mathtt{w} \in \mathcal{N}_P$.

For example, in Figure 1, `A()` and `B()` are used uniformly with respect to the outside-before relation, since immediately before every `B()`, always `A()` is executed. The same applies to the outside-before relations `A()` $\rightharpoonup$ `C()`, `C()` $\rightharpoonup$ `E()`, to the outside-after relation `E()` $\leftharpoondown$ `C`, to the inside-first relations(`A()` $\in_\top$ `B()` and `D()` $\in_\top$ `E()`), and to the inside-last relations `D()` $\in_\bot$ `E` and `E()` $\in_\bot$ `B()`.

Because uniformity considers an entire trace, two uniformly used methods indicate that some functionality is used consistently in the program. If additionally it is repeatedly used it might be better implemented as an aspect. Hence, two methods are defined to be crosscutting with respect to a relation when they are used uniformly and occur in more than one calling context in a trace:

Definition 2.3 (Crosscutting)
Two methods `u` *and* `v`*, with* $\mathtt{u}, \mathtt{v} \in \mathcal{N}_P$*, are crosscutting with respect to a relation $\circ \in \{\rightharpoonup, \leftharpoondown, \in_\top, \in_\bot\}$ induced by a trace T_P, when* `u` *and* `v` *are used uniformly and the following holds:* $\exists\ \mathtt{u} \circ \mathtt{w} \wedge \mathtt{w} \neq \mathtt{v}, \mathtt{w} \in \mathcal{N}_P$.

For inside-execution relations $u \in_{\top} v$ (or $u \in_{\bot} v$) the calling context is the surrounding method execution v. For outside-execution relations $u \rightharpoonup v$ (or $u \leftharpoonup v$) the calling context is the method v invoked before (or after) which always method u is executed. In our example, `A()` is a crosscutting method execution, since `A()` is not only always executed before `B()` but also always before `C()`. It is the only crosscutting method execution that can be identified in the small trace example of Figure 1.

2.1. DynAMiT Evaluation

An evaluation of DynAMiT using Java programs demonstrated the precision and applicability of dynamically mining aspects from call traces [1]: DynAMiT found seeded and existing crosscutting concerns in AnChoVis ($\approx$ 1,300 LOC), Graffiti ($\approx$ 82,000 LOC, [5]), and rediscovered aspects that were added previously using AOP techniques. In one case an analysed program has been refactored based on the dynamic aspect mining results. Additionally, it was verified that the re-factoring did not change the software system's behaviour.

The current implementation of DynAMiT is tightly coupled with the AspectJ compiler [19], which is used to instrument Java programs for tracing. AspectJ provides two pointcuts suitable for tracing method invocations. However, it cannot be used for tracing more fine-grained control flow. For instance, AspectJ cannot be used to instrument every basic block, which could be implemented by instrumenting bytecode more directly. As a consequence, the aspects mined by DynAMiT are relatively coarse-grained, although this is not inherent to the approach of dynamic mining.

3. Problems

While the approach is generally fairly precise, further analysis revealed that some false positives were systematically caused by dynamic binding, as well as by not-sufficient tracing mechanisms of method executions when using AspectJ's execution pointcuts. This section illustrates, considering small examples, how dynamic binding can produce false positives in the resulting aspect candidates, as well as how the joice of pointcuts influences the trace generation.

3.1. Dynamic Binding of Methods

Figure 2 illustrates the problem caused by dynamic binding in the DynAMiT approach. The `interface I` has two method declarations `a()` and `c()`. The `class B` implements that interface, while the `abstract class A` only implements method `a()` of `I`. The `abstract class A` is extended by two subclasses `C1` and `C2`, which both provide

```
interface I {
  public void a();
  public void c();
}

abstract class A implements I {
  public void a(){}
}

class C1 extends A {
  public void c(){}
}

class C2 extends A {
  public void c(){}
}

class B implements I {
  public void a(){}
  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[] args) {
    A obj1 = new C1();
    A obj2 = new C2();
    B obj3 = new B();
    doSth(obj1);
    doSth(obj2);
    doSth(obj3);
  }
}
```

Figure 2. Example code of a software system

implementations of method `c()` whose declaration is inherited from `I` (via `A`).

`class Runner` uses this hierarchy; its execution generates the trace in Figure 3(a). Here, the crosscutting algorithm 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)` and thus does not exist crosscutting in the code. Hence, whenever we have abstract methods with several concrete implementations, the dynamic aspect mining algorithm will systematically produce false positives.

This poses a problem as dynamic binding is at the heart of object-oriented design and programming.

```
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 3. Dynamic vs 'static' trace

3.2. Execution vs. Call Pointcuts

DynAMiT itself uses aspects to trace method executions of a program. The AspectJ compiler used for weaving trace aspects offers two choices for a trace implemenation: execution pointcuts that are associated with the execution of a method, and call pointcuts that are associated with the call of a method (at the call site). Both gather different informations, leading to different traces and thus different abilities to mine aspects.

An execution pointcut uses the type of the actual method invoked, that is, after dynamic binding. Thus, using an execution pointcut it is thus possible to log the exact type of the invoked method. However, the static type of the method, as it is present in the source code at the call site, is lost. Since AspectJ implements weaving by bytecode instrumentation, Java API code cannot be instrumented and traced with execution pointcuts.

Conversely, a call pointcut cannot access the exact signature of a method being invoked because it is executed before dynamic binding is resolved. However, it does recognise the static signature of the called method, as it is known at the call site. Since call pointcuts are associated with call sites of methods, client calls into the Java API can be instrumented and traced.

To understand the different pointcuts with respect to API calls, consider the code fragment in Figure 4 which calls the Java API method for string concatenation. The use of method execution pointcuts produces a trace that shows the execution of method `bar()` as first inside method `foo(int, int)`. However, this is not true. It is obvious that this trace is just an approximation of what really happens: first inside the execution of `foo` the API method `concat(String)` of class `String` is executed, and the execution of `bar` occurs subsequently. The use of call pointcuts instead would lead to a correct generation of this part in the trace.

```
public void bar() {
  ...
}

public void foo(int i, int j) {
  result = result.concat("foo");
  this.bar();
  ...
}
```

Figure 4. Code snippet with API method

It may be beneficial that API methods are not traced as with each analysis Java API classes would also be analysed. Nonetheless, it would be preferable if we could control whether or not, and maybe how much of the API classes and methods should be used for mining.

With the original DynAMiT we decided to focus on method executions, and thus used execution pointcuts. They permit an exact log of the executed method to be recorded, unlike the coarser call pointcuts. However, it was subsequently realised that the additional detail provided by the dynamic types logged with execution pointcuts can make it harder to mine aspect candidates.

4. Extensions to DynAMiT

DynAMiT can produce false positives due to dynamic binding of method calls: a single call site may actually reach several distinct methods. Each such method creates a distinct entry in the trace, leading to spurious "different" calling contexts that may result in wrong aspect candidates. We describe now, how a different type of pointcut together with additional static object information helps to improve the original DynAMiT.

4.1. Static Object Type

Now, DynAMiT has to identify different methods that are invoked from the same call site using static type information: Thus, instead of logging the signature of the method actually invoked, the known static signature at the call site is logged.

A trace that uses static type information differs from a trace of the same program using dynamic (or actual) type informations, as we see in Figure 3(b). There, the crosscutting algorithm would not detect the incorrect crosscutting concerns illustrated in 3.1 (which may be part of a real

crosscutting concern, but are not on their own). Thus, this solves the problem described in Section 3.1. An integration of static information into the traces would frequently avoid the difficulty that an invocation of the same functionality (i.e., one occurring only once in the code) appearing to be crosscutting in the traces.

4.2. Method Call Pointcut

To use static type information, DynAMiT implements method tracing using call pointcuts. Since AspectJ instruments the corresponding bytecode of call sites, only calls outside of Java system classes may be traced. This has consequences for callbacks invoked from Java system classes, which no longer can be traced. Callbacks invoked from system classes arise especially in GUI-driven program that use the AWT/Swing class hierarchy.

Callbacks were not a problem for execution pointcuts, which monitor the execution of a method, not its invocation. While the call may be initiated by a Java system class, the executed class is typically part of the client code. As such, it can be traded using an execution pointcut. When moving from execution pointcuts to call pointcuts, system-related callbacks are no longer traced. Nevertheless, in the systems mined so far, calls to the API were in general more common than callbacks; thus, it is safe to say that call pointcuts will monitor a much larger part of all method invocations than previously execution pointcuts.

Moving from execution pointcuts to call pointcuts for tracing alters the trace of a program beyond the types recorded: there is a difference when a method is called, and when it is executed. If, for example, a method `x` takes arguments that have first to be evaluated using another method `y` before `x` can be executed, than the call to `x` obviously occurs before the call to `y`, but the order of execution is reversed: `y` is executed before `x`.

As we now no longer work on execution relations but on call relations, we have to think whether we still can use the notion of relative order of method executions. Because the basic idea of mining for recurring patterns in a program run remains consistent, we consider the call relations equivalent to the execution of methods, and handle them as before, referring to them still as execution relations to prevent confusion in terminology. Providing that we do not mix calls and executions of methods for monitoring program behaviour in our traces, this is safe to do.

5. Impact of Static Types and Call Pointcuts

The static extension and changes to DynAMiT have been evaluated using two programs that previously were analysed with the traditional DynAMiT [1]. First, on the AnChoVis visualisation tool. In comparison to the original DynAMiT, we found slightly less inside-aspect candidates, but in general substantially more outside-aspect candidates (see Figure 5). The traces themselves became much larger as now more method calls are monitored. In AnChoVis, the traces grew by the factor of 5.5.

Most of the differences are due to the fact that the set of traced methods changed, as explained in Section 4.2. This has three consequences: The set of aspect candidates increases considerably, some aspect candidates may have been lost, and the constellation within some aspect candidates may change.

More Aspect Candidates. The set of aspect candidates tends to be larger, because a larger number of different method signatures are contained in the trace. Consulting Figure 5 and comparing the results with those in [1] shows that with the original DynAMiT nine outside-before relations formed one aspect candidate, and ten outside-after relations formed a second candidate, whereas now the analysis detects four aspect candidates that are built from two (twice), three, or eight before relations resp., and two candidates that are built from two and nine after-relations. This corresponds to an increase of identified crosscutting patterns by 200%. For example,

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

has now been detected, which 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 was unable to detect the crosscut as there is only a single implementation of the `getReader()` functionality in the common superclass. Of course, `readLine()` is also an API method, which provides an additional reason why the original DynAMiT was not able to detect the crosscut.

A newly discovered inside candidate is `String java.lang.String.valueOf(Object)` which is found as the first call inside method `void anchovis.Logging.entering(String)`, which is again due to the fact that call pointcuts also monitor API methods like `valueOf(Object)`.

Missing Former Aspect Candidates. The hybrid approach described could also result in the set of aspect candidates becoming smaller as some methods are no longer traced when using call pointcuts, e.g., `void anchovis.AnChoVis.main(String[])` or `void anchovis.AnChoVis.mouseClicked(MouseEvent)`. Thus, the inside aspect candidates are formed from slightly less relations: 30

```
String java.lang.String.valueOf(Object) ∈⊤
  void anchovis.Logging.entering(String),
  void anchovis.AnChoVis.start(String[]),
  void anchovis.AnChoVis.init(String[]),
  boolean anchovis.AnChoVis.checkArgs(String[]),
  void anchovis.AnChoVis.parseArgs(String[]),
  int anchovis.AnChoVis.getIntValue(String),
  boolean anchovis.AnChoVis.checkMatrixProps(),
  int anchovis.AnChoVis.calcEnumSteps(),
  void anchovis.AnChoVis.setGraphicBounds(),
  void anchovis.AnChoVis.setFrameHeaders(),
  void anchovis.AnChoVis.setFrameSizes(),
  void anchovis.AnChoVis.setBackgroundColor(Color),
  void anchovis.AnChoVis.addPaintingPanels(),
  void anchovis.AnChoVis.createMainApplicationFrame(),
  void anchovis.AnChoVis.setDummyLabel(),
  void anchovis.AnChoVis.createColorMatrix(),
  void anchovis.AnChoVis
   .createMatrixEnum(boolean, int),
  void anchovis.AnChoVis.createColorScaleFrame(),
  void anchovis.AnChoVis
   .setColorPanels(ColorConstants),
  String[] anchovis.AnChoVis.calculateScaleValues(),
  void anchovis.AnChoVis.setScaleLabels(String[]),
  void anchovis.AnChoVis.createValueNamesFrame(),
  int anchovis.AnChoVis
   .setIdentifierLabels(int, int, int, int),
  void anchovis.AnChoVis
   .setValueNamesLabels(int, int, int, int, int),
  void anchovis.AnChoVis
   .setFramePositions(int, int, int),
  void anchovis.AnChoVis.setTerminationOperation(int),
  void anchovis.AnChoVis.setFramesVisible(),
  boolean anchovis.AnChoVis.insideMatrix(int, int),
  void anchovis.AnChoVis.printHelp(),
  void anchovis.AnChoVis.printVersion()

String java.io.BufferedReader.readLine() ⇁
  String[] java.lang.String.split(String),
  boolean anchovis.AnChoVis.checkMatrixProps()

void anchovis.Logging.entering(String) ⇁
  void anchovis.AnChoVis.start(String[]),
  boolean anchovis.AnChoVis.checkArgs(String[]),
  GraphicsEnvironment java.awt.GraphicsEnvironment
   .getLocalGraphicsEnvironment(),
  void anchovis.AnChoVis.setDummyLabel(),
  void anchovis.AnChoVis
   .setColorPanels(ColorConstants),
  int anchovis.AnChoVis
   .setIdentifierLabels(int, int, int, int),
  int java.awt.event.MouseEvent.getX(),
  void anchovis.AnChoVis.printVersion()

int java.lang.String.compareTo(String) ⇁
  void anchovis.AnChoVis.init(String[]),
  void anchovis.AnChoVis.printHelp()

String java.lang.StringBuffer.toString() ⇁
  void anchovis.Logging.entering(String),
  void java.io.BufferedWriter.write(String),
  void anchovis.Logging.exiting(String)

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

String java.lang.String.valueOf(Object) ↼
  void anchovis.AnChoVis.init(String[]),
  String java.lang.String.substring(int),
  double java.lang.Math.ceil(double),
  Rectangle java.awt.GraphicsConfiguration.getBounds(),
  void javax.swing.JPanel.setBorder(Border),
  void anchovis.AnChoVis.setScaleLabels(String[]),
  void anchovis.AnChoVis.setFramesVisible(),
  void anchovis.AnChoVis.printHelp(),
  void anchovis.AnChoVis.printVersion()

void anchovis.Logging.exiting(String) ∈⊥
  void anchovis.AnChoVis.start(String[]),
  void anchovis.AnChoVis.init(String[]),
  boolean anchovis.AnChoVis.checkArgs(String[]),
  void anchovis.AnChoVis.parseArgs(String[]),
  boolean anchovis.AnChoVis.checkMatrixProps(),
  int anchovis.AnChoVis.calcEnumSteps(),
  void anchovis.AnChoVis.setGraphicBounds(),
  void anchovis.AnChoVis.setFrameHeaders(),
  void anchovis.AnChoVis.setFrameSizes(),
  void anchovis.AnChoVis.setBackgroundColor(Color),
  void anchovis.AnChoVis.addPaintingPanels(),
  void anchovis.AnChoVis.createMainApplicationFrame(),
  void anchovis.AnChoVis.setDummyLabel(),
  void anchovis.AnChoVis.createColorMatrix(),
  void anchovis.AnChoVis.createMatrixEnum(boolean, int),
  void anchovis.AnChoVis.createColorScaleFrame(),
  void anchovis.AnChoVis.setColorPanels(ColorConstants),
  String[] anchovis.AnChoVis.calculateScaleValues(),
  void anchovis.AnChoVis.setScaleLabels(String[]),
  void anchovis.AnChoVis.createValueNamesFrame(),
  void anchovis.AnChoVis
   .setValueNamesLabels(int, int, int, int, int),
  void anchovis.AnChoVis
   .setFramePositions(int, int, int),
  void anchovis.AnChoVis.setTerminationOperation(int),
  void anchovis.AnChoVis.setFramesVisible(),
  boolean anchovis.AnChoVis.insideMatrix(int, int),
  void anchovis.AnChoVis.printHelp(),
  void anchovis.AnChoVis.printVersion()
```

Figure 5. Extended DynAMiT : crosscutting analysis results for AnChoVis

inside-first and 27 inside-last relations, instead of 31 each. This corresponds to gradually incomplete candidates; approximately 3% and 13% resp. of an inside aspect candidate could not be detected by the analysis.

Changing Former Aspect Candidates. Finally, the set of aspect candidates changes because API method calls appear in the program traces *in between* some non-API method calls that built the execution relations in the original DynAMiT. In particular, this affects the first-inside-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 from [1] 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 summary the switch from execution to call traces has two major consequences: Our call relations use static type information (preventing incorrect results due to dynamic binding), but the subset of methods traced also changes and increases (due to implementation issues).

In the case study, the use of static object information proved beneficial. It partly eliminated wrong aspect candidates of the structure described in Section 4.1, and sometimes even detected new candidates.

A second case study with `telecom`, the aspect-oriented phone company simulation included in the distribution of AspectJ, that was also already used in [1], reenforces the results from the AnChoVis case study. Overall, the number of aspect candidates found by the extended DynAMiT increased by a factor of 2.67. Both, the number of inside- and outside-aspect candidates increased; additional new candidates mostly include calls to methods from classes `PrintStream`, `String`, and `StringBuffer`. This is a consequence of API methods now also being traced.

6. Related Work

Several techniques have been proposed for aspect mining. Unlike the technique described here, they are based on static program analysis and often require user interaction. Also unlike the approach here, these tools don't uses type information to increase the precision of mining—with the exception of the Aspect Mining Tool [7].

- The Aspect Browser [6] identifies crosscutting concerns with textual-pattern matching (much like "grep") and highlights them. Its success in finding aspects thus strongly depends on naming conventions followed in the program code to be analysed.
- The Aspect Mining Tool (AMT) [7] is based on a multi-modal analysis for an advanced separation of concerns in legacy code. It combines text- and type-based analysis to reduce false positives.
- The exploration tool JQuery [8] offers a generic browser that allows the definition of logic queries in a specific query language. The navigation/analysis of the source code can be based on different structural relationships, regular expression matches, and complex searches for structural patterns.
- FEAT [14], the Feature Exploration and Analysis Tool is implemented as a plugin for the Eclipse Platform. FEAT visualises concerns in a system using so-called "concern graphs". A concern graph abstracts the implementation details of a concern by storing the structure implementing that concern. This way, it documents explicitly the relationships between the different elements of a concern (classes, fields, methods).
- Ophir [15], another framework for automatic aspect mining, identifies initial re-factoring candidates using a control-based comparison. The initial identification phase builds upon code clone detection using program dependence graphs. The next step filters undesirable re-factoring candidates. It looks for similar data dependencies in subgraphs representing code clones. The last phase identifies similar candidates and coalesces them into sets of similar candidates, which are the re-factoring candidate classes.
- Krinke and Breu [11] propose an automatic static aspect mining based on control-flow. The control-flow graph of a program is mined for recurring execution patterns based on constraints as introduced in [1], such as the requirement that the patterns have to exist in different calling contexts.
- Ceccato et al. [4] have conducted a comparison of three aspect mining techniques that they proposed earlier. They advocate the combination of different approaches to overcome the weaknesses of a single technique. The fan-in analysis determines methods that are called from many different places to identify aspect candidates in Java software systems [13]. This technique seems to complement the results from the second technique which is based on concept-analysis of execution traces [17]. Just as all dynamic analysis approaches, its main limitation is incompleteness. The third technique, the identifier analysis [18], produces many details but often too many false positives. Its results largely overlap with the previous two approaches. That indicates that a combination of these three techniques could make a powerful analysis by refining the united results from the fan-in and the dynamic analysis with the more detailed results from the identifier analysis.

7. Conclusions and Future Work

DynAMiT is so far the only aspect mining approach that combines dynamic analysis with static types. It identifies crosscutting aspects in a Java program by mining its method-call trace. Method calls in Java are dynamically bound, such that the same method call in Java source code may lead to different calls at run time. As a consequence, the original implementation of DynAMiT failed to recognise potential aspect candidates. Our extension of DynAMiT takes the static types of calls into account to identify dynamically different calls of the same static type. This leads to better mining results: evaluated with the same program as the original DynAMiT, the extended DynAMiT finds considerably more potential aspects and fewer false positives. With AnChoVis, the number of potential outside-aspect candidates has tripled, and for `telecom` we get a factor of 2.67.

In future work, we propose to investigate whether filtering out spurious methods can further improve the extended DynAMiT. These interferences can extensively be found in `class java.lang.String` and `class java.lang.StringBuffer`, especially in those methods that implement the concatenation operator `+`. The number of masked aspect candidates is likely to decrease considerably if we exclude those methods or classes from the analysis. Furthermore, the problems with string concatenation and parameter evaluation do appear in the linearised, compiled bytecode (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. Thus, it could be worthwhile considering a (static) analysis of the source code.

The idea to include static information in the analysis has proved promising in this case study. However, in order to draw more general conclusions, it will be necessary to conduct further case studies with large programs that have a deep inheritance hierarchy. We also plan to use JHotDraw version 5.4b1 [9], a framework for 2D graphics with $\approx$ 40 kLOC which is a benchmark for aspect mining techniques, in order to compare the new version of DynAMiT with the control-flow based approach [11].

Most of the differences between classic DynAMiT and the extended version were due to additional API calls being traced (by bytecode instrumentation). Those resulted in the complete analysis becoming 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.

Acknowledgements. Thanks to Jens Dörre who conducted parts of the case studies, and to Christian Lindig for his valuable feedback on earlier versions of this paper.

References

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

[2] S. Breu. Towards Hybrid Aspect Mining: Static Extensions to Dynamic Aspect Mining. In *1. Workshop on Aspect Reverse Engineering (WARE) at Working Conference on Reverse Engineering (WCRE)*, November 2004.

[3] S. Breu and J. Krinke. Aspect Mining Using Event Traces. In *Proceedings 19th International Conference on Automated Software Engineering (ASE)*, pages 310–315. IEEE Press, September 2004.

[4] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. A qualitative comparison of three aspect mining techniques. In *International Workshop on Program Comprehension (IWPC)*, 2005.

[5] Gravisto homepage. http://www.gravisto.org/.

[6] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspect Browser: Tool Support for Managing Dispersed Aspects. Technical Report CS99-0640, Department of Computer Science and Engineering, UC, San Diego, 1999.

[7] J. Hannemann and G. Kiczales. Overcoming the Prevalent Decomposition of Legacy Code. In *Workshop on Advanced Separation of Concerns*, 2001.

[8] D. Janzen and K. D. Volder. Navigating and Querying Code Without Getting Lost. In *2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 178–187, 2003.

[9] JHotDraw open-source project homepage. http://www.jhotdraw.org/.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[11] J. Krinke and S. Breu. Aspect Mining Based on Control-Flow. *Softwaretechnik-Trends*, 2005.

[12] N. Loughran and A. Rashid. Mining Aspects. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD Sat. Workshop)*, 2002.

[13] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *11th Working Conference on Reverse Engineering (WCRE)*, pages 132–141. IEEE Computer Society, November 2004.

[14] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *24th International Conference on Software Engineering (ICSE)*, pages 406–416, 2002.

[15] D. Shepherd and L. Pollock. Ophir: A Framework for Automatic Mining and Refactoring of Aspects. Technical Report 2004-03, University of Delaware, 2003.

[16] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *21st International Conference on Software Engineering (ICSE)*, pages 107–119, 1999.

[17] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *11th Working Conference on Reverse Engineering (WCRE)*, pages 112–121. IEEE Computer Society, November 2004.

[18] T. Tourwé and K. Mens. Mining aspectual views using formal concept analysis. In *Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 97–106. IEEE Computer Society, 2004.

[19] Xerox PARC. *Aspect-Oriented Programming with AspectJ (Tutorial)*, 1998.

[20] C. Zhang and H.-A. Jacobsen. Quantifying Aspects in Middleware Platforms. In *2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 130–139, 2003.