Aspect Mining Using Event Traces

Silvia Breu
MCT/NASA Ames, USA
silvia.breu@gmail.com

Jens Krinke
FernUniversität in Hagen, Germany
krinke@acm.org

Abstract

Aspect mining tries to identify crosscutting concerns in existing systems and thus supports the adaption to an aspect-oriented design. This paper describes the first aspect mining approach that detects crosscutting concerns in legacy systems based on dynamic analysis.

The analysis uses program traces that are generated in different program executions as underlying data pool. These traces are then investigated for recurring execution patterns based on different constraints, such as the requirement that the patterns have to exist in different calling contexts in the program trace. The implemented approach was evaluated in several case studies over systems with more than 80 kLoC. The tool was able to identify automatically both existing and seeded crosscutting concerns.

1. Introduction

The notion of *tangled code* [6] refers to code that exists several times in a software system but cannot be encapsulated by separate modules using traditional module systems because it crosscuts the whole system. This makes software more difficult to maintain, to understand, and to extend. *Aspect-Oriented Programming* [6] provides new separation mechanisms for such complex *crosscutting concerns* [10].

A major problem in re-engineering legacy code based on aspect-oriented principles is to find and to isolate these crosscutting concerns. This task is also called *aspect mining*. The detected concerns can be re-implemented as separate aspects, thereby improving maintainability and extensibility as well as reducing complexity. Aspect mining can also provide insights that enable us to classify common aspects which occur in different software systems, such as logging, timing, and communication.

Several approaches based on static program analysis techniques have been proposed for aspect mining [3, 4, 5, 7, 8, 12]. This paper describes the first dynamic program analysis approach that mines aspects based on program traces. During program execution, program traces are generated, which reflect the run-time behaviour of a software system. These traces are then investigated for recurring execution patterns. Different constraints specify when an execution pattern is "recurring". These include the requirement that the patterns have to exist in different calling contexts in the program trace. The dynamic analysis approach monitors actual (i.e., run-time) program behaviour instead of potential behaviour, as static program analysis does. As the case studies in this paper show, the technique is able to identify automatically both existing and seeded crosscutting concerns in software systems.

2. Aspect Analysis Algorithms

The basic idea behind dynamic analysis algorithms is to observe run-time behaviour of software systems and to extract information from the execution of the programs. The dynamic aspect mining approach introduced here is based on the analysis of program traces which mirror a system's behaviour in certain program runs. Within these program traces we identify recurring execution patterns which describe certain behavioural aspects of the software system. We expect that recurring execution patterns are potential crosscutting concerns which describe recurring functionality in the program and thus are possible aspects.

In order to detect these recurring patterns in the program traces, a classification of possible pattern forms is required. Therefore, we introduce so-called *execution relations*. They describe in which relation two method executions are in the program trace.

2.1. Classification of Execution Relations

The definition of execution relations in our analysis approach is based on program traces. Intuitively, a program trace is a sequence of method invocations and exits. We only consider entries into and exits from method executions because we can then easily keep track of the relative order in which method executions are started and finished. We focus on method executions because we want to analyse object-oriented systems where logically related functionality is encapsulated in methods. Formally, a *program trace* T_P of a

program P with method signatures $\mathcal{N}_P$ is defined as 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.

To make the program traces easier to read, the ent- and ext-points are represented by { and } respectively, and the redundant $name$-information is discarded from the ext-points as the trace structure implies to which $name$ the ext belongs. Figure 1 shows an example trace.

```
 1 B() {          17   J() {}       33   }
 2   C() {        18 }               34 }
 3     G() {}     19 F() {           35 D() {
 4     H() {}     20   K() {}       36   C() {}
 5   }            21   I() {}       37   A() {}
 6 }              22 }               38   B() {
 7 A() {}         23 J() {}          39     C() {}
 8 B() {          24 G() {}          40   }
 9   C() {}       25 H() {}          41   K() {}
10 }              26 A() {}          42   I() {
11 A() {}         27 B() {           43     J() {}
12 B() {          28   C() {}       44   }
13   C() {        29   G() {}       45   G() {}
14     G() {}     30   F() {         46   E() {}
15     H() {}     31     K() {}     47 }
16   }            32     I() {}
```

Figure 1. Example trace

Crosscutting concerns are now reflected by the two different *execution relations* that can be found in program traces: A method can be executed either after the preceeding method execution is terminated (e.g., `H()` in line *4* is executed after `G()` in line *3*), or inside the execution of the preceeding method call (e.g., `C()` in line *2* is executed inside `B()` in line *1*). We distinguish between these two cases and say that there are outside- and inside-execution relations in program traces. However, this distinction alone is not yet sufficient for aspect mining. For example, the execution of `B()` in line *27* has three methods executed inside its execution, `C()`, `G()`, and `F()` in lines *28* ff., but the information which of those methods comes first is lost. We thus define formally:

$\mathtt{u} \rightharpoonup \mathtt{v}$, $\mathtt{u}, \mathtt{v} \in \mathcal{N}_P$, is called an *outside-before-execution relation* if $[(\mathtt{u}, ext), (\mathtt{v}, ent)]$ is a sublist of T_P. $S^{\rightharpoonup}(T_P)$ is the set of all outside-before-execution relations in a program trace T_P. This relation can also be reversed, i.e., $\mathtt{v} \leftharpoonup \mathtt{u}$ is an *outside-after-execution relation* if $\mathtt{u} \rightharpoonup \mathtt{v} \in S^{\rightharpoonup}(T_P)$. The set of all outside-after-execution relations in a program trace T_P is then denoted with $S^{\leftharpoonup}(T_P)$.

$\mathtt{u} \in_\top \mathtt{v}$, $\mathtt{u}, \mathtt{v} \in \mathcal{N}_P$ is called an *inside-first-execution relation* if $[(\mathtt{v}, ent), (\mathtt{u}, ent)]$ is a sublist of T_P. $\mathtt{u} \in_\bot \mathtt{v}$ is called an *inside-last-execution relation* if $[(\mathtt{u}, ext), (\mathtt{v}, ext)]$ is a sublist of T_P. $S^{\in_\top}(T_P)$ is the set of all inside-first-execution relations in a program trace T_P, $S^{\in_\bot}(T_P)$ is the set of all inside-last-execution relations. In the following, we drop T_P when it is clear from the context.

For the example trace shown in Figure 1 we thus get the following set $S^{\rightharpoonup}$ of outside-before-execution relations:

$$\begin{aligned} S^{\rightharpoonup} = \{ & \mathtt{B()} \rightharpoonup \mathtt{A()}, \mathtt{G()} \rightharpoonup \mathtt{H()}, \mathtt{A()} \rightharpoonup \mathtt{B()}, \mathtt{C()} \rightharpoonup \mathtt{J()}, \\ & \mathtt{B()} \rightharpoonup \mathtt{F()}, \mathtt{K()} \rightharpoonup \mathtt{I()}, \mathtt{F()} \rightharpoonup \mathtt{J()}, \mathtt{J()} \rightharpoonup \mathtt{G()}, \\ & \mathtt{H()} \rightharpoonup \mathtt{A()}, \mathtt{B()} \rightharpoonup \mathtt{D()}, \mathtt{C()} \rightharpoonup \mathtt{G()}, \mathtt{G()} \rightharpoonup \mathtt{F()}, \\ & \mathtt{C()} \rightharpoonup \mathtt{A()}, \mathtt{B()} \rightharpoonup \mathtt{K()}, \mathtt{I()} \rightharpoonup \mathtt{G()}, \mathtt{G()} \rightharpoonup \mathtt{E()} \} \end{aligned}$$

The set $S^{\leftharpoonup}$ of outside-after-execution relations can be found directly in the trace or simply by reversing $S^{\rightharpoonup}$. The sets $S^{\in_\top}$ of inside-first-execution relations and $S^{\in_\bot}$ of inside-last-execution relations are as follows:

$$\begin{aligned} S^{\in_\top} = \{ & \mathtt{C()} \in_\top \mathtt{B()}, \mathtt{G()} \in_\top \mathtt{C()}, \mathtt{K()} \in_\top \mathtt{F()}, \mathtt{C()} \in_\top \mathtt{D()}, \\ & \mathtt{J()} \in_\top \mathtt{I()} \} \\ S^{\in_\bot} = \{ & \mathtt{H()} \in_\bot \mathtt{C()}, \mathtt{C()} \in_\bot \mathtt{B()}, \mathtt{J()} \in_\bot \mathtt{B()}, \mathtt{I()} \in_\bot \mathtt{F()}, \\ & \mathtt{F()} \in_\bot \mathtt{B()}, \mathtt{J()} \in_\bot \mathtt{I()}, \mathtt{E()} \in_\bot \mathtt{D()} \} \end{aligned}$$

2.2. Execution Relation Constraints

Recurring execution relations in the program traces can be seen as indicators for more general execution patterns. To decide under which circumstances certain execution relations are recurring patterns in traces and thus potential crosscutting concerns in a system, constraints have to be defined. The constraints will implicitely also formalize what crosscutting means.

However, for technical reasons we have to encode that there is no further method execution between nested method executions or between method invocation and method exit. This absence of method executions is represented by the designated empty method signature ϵ. Therefore, the definition of execution relations is extended such that each sublist of a program trace T_P induces not only relations defined above but also additional relations involving ϵ. Table 1 summarises this conservative extension. It shows for each two-element sublist of the trace (on the left side) the execution relations that follow from that sublist (on the right side). The execution relations added by the introduction of ϵ are annotated with an asterisk ($*$).

The program trace remains as defined before with method signatures from $\mathcal{N}_P$ whereas the execution relations now can consist of method signatures from $\mathcal{N}_P \cup \{\epsilon\}$. Thus, the sets $S^{\rightharpoonup}$, $S^{\leftharpoonup}$, $S^{\in_\top}$, and $S^{\in_\bot}$ also include execution relations involving ϵ. Now, we can define the constraints for the dynamic analysis.

Formally, an execution relation $s = \mathtt{u} \circ \mathtt{v} \in S^\circ$, $\circ \in \{\rightharpoonup, \leftharpoonup, \in_\top, \in_\bot\}$, is called *uniform* if $\forall \mathtt{w} \circ \mathtt{v} \in S^\circ : \mathtt{u} = \mathtt{w}, \mathtt{u}, \mathtt{v}, \mathtt{w} \in \mathcal{N}_P \cup \{\epsilon\}$ holds, i.e., it exists in always the same composition. $\widehat{U}^\circ$ is the set of execution relations $s \in S^\circ$ which satisfy this requirement. This constraint is easy to explain. Consider an outside-before-execution relation

Trace-sublist $(\mathcal{N}_P)$		**Relation** s $(\mathcal{N}_P \cup \{\epsilon\})$
$(\mathtt{u}, ext)$	$(\mathtt{v}, ent)$	$\mathtt{u} \rightharpoonup \mathtt{v}$, $\mathtt{v} \leftharpoonup \mathtt{u}$
$(\mathtt{v}, ent)$	$(\mathtt{u}, ent)$	$\epsilon \rightharpoonup \mathtt{u}^*$, $\mathtt{u} \leftharpoonup \epsilon^*$, $\mathtt{u} \in_\top \mathtt{v}$
BOL	$(\mathtt{u}, ent)$	$\epsilon \rightharpoonup \mathtt{u}^*$, $\mathtt{u} \leftharpoonup \epsilon^*$, $\mathtt{u} \in_\top \epsilon^*$
$(\mathtt{u}, ext)$	$(\mathtt{v}, ext)$	$\mathtt{u} \rightharpoonup \epsilon^*$, $\epsilon \leftharpoonup \mathtt{u}^*$, $\mathtt{u} \in_\bot \mathtt{v}$
$(\mathtt{u}, ext)$	EOL	$\mathtt{u} \rightharpoonup \epsilon^*$, $\epsilon \leftharpoonup \mathtt{u}^*$, $\mathtt{u} \in_\bot \epsilon^*$
$(\mathtt{w}, ent)$	$(\mathtt{w}, ext)$	$\epsilon \in_\top \mathtt{w}^*$, $\epsilon \in_\bot \mathtt{w}^*$

BOL/EOL denote begin/end of list

Table 1. Extended execution relations

$\mathtt{u} \rightharpoonup \mathtt{v}$. This is defined as recurring pattern if each execution of v is preceded by an execution of u. The argumentation for outside-after-execution relations is analogous. The uniformity-constraint also applies to inside-execution relations. An inside-execution relation $\mathtt{u} \in_\top \mathtt{v}$ (or $\mathtt{u} \in_\bot \mathtt{v}$) can only be a recurring pattern in the given program trace if v never executes another method than u as first (or last) method inside its body.

We now drop the ϵ-relations and define a further analysis constraint: An execution relation $s = \mathtt{u} \circ \mathtt{v} \in U^\circ = \widehat{U}^\circ \backslash \{\mathtt{u} \circ \mathtt{v} \mid \mathtt{u} = \epsilon \vee \mathtt{v} = \epsilon\}$ is called *crosscutting* if $\exists s' = \mathtt{u} \circ \mathtt{w} \in U^\circ : \mathtt{w} \neq \mathtt{v}, \mathtt{u}, \mathtt{v}, \mathtt{w} \in \mathcal{N}_P$ holds, i.e., it occurs in more than a single calling context in the program trace T_P. For inside-execution relations $\mathtt{u} \in_\top \mathtt{v}$ (or $\mathtt{u} \in_\bot \mathtt{v}$) the calling context is the surrounding method execution v. For outside-execution relations $\mathtt{u} \rightharpoonup \mathtt{v}$ (or $\mathtt{u} \leftharpoonup \mathtt{v}$) the calling context is the method v invoked before (or after) which always method u is executed. R° is the set of execution relations $s \in U^\circ$ which satisfy this requirement. Execution relations $s \in R^\circ$ are also called *aspect candidates* as they represent the potential crosscutting concerns of the analysed software system.

2.3. Aspect Mining Algorithm

The constraints described above can be implemented by a relatively straightforward algorithm to actually compute the sets R° of uniform, crosscutting execution relations that represent the aspect candidates. In our running example, uniformity narrows down the potential aspect candidates to the following sets of execution relations:

$$U^{\rightharpoonup} = \{\mathtt{B()} \rightharpoonup \mathtt{D()}, \mathtt{G()} \rightharpoonup \mathtt{E()}, \mathtt{G()} \rightharpoonup \mathtt{H()}, \mathtt{K()} \rightharpoonup \mathtt{I()}\}$$
$$U^{\leftharpoonup} = \{\mathtt{B()} \leftharpoonup \mathtt{A()}, \mathtt{I()} \leftharpoonup \mathtt{K()}\}$$
$$U^{\in_\top} = \{\mathtt{C()} \in_\top \mathtt{B()}, \mathtt{C()} \in_\top \mathtt{D()}, \mathtt{K()} \in_\top \mathtt{F()}\}$$
$$U^{\in_\bot} = \{\mathtt{E()} \in_\bot \mathtt{D()}, \mathtt{I()} \in_\bot \mathtt{F()}\}$$

After we enforce the crosscutting constraint, we obtain the final sets R° of aspect candidates which comply with uniformity *and* crosscutting.

$$R^{\rightharpoonup} = \{\mathtt{G()} \rightharpoonup \mathtt{H()}, \mathtt{G()} \rightharpoonup \mathtt{E()}\},\ R^{\leftharpoonup} = \varnothing$$
$$R^{\in_\top} = \{\mathtt{C()} \in_\top \mathtt{B()}, \mathtt{C()} \in_\top \mathtt{D()}\},\ R^{\in_\bot} = \varnothing$$

The full example and details of the *Dyn*amic *A*spect *Mi*ning *T*ool (*DynAMiT*) implementing the aspect mining algorithm can be found in [1].

3. Evaluation

This section presents two case studies using *DynAMiT*. All identified aspect candidates have been checked against the source code of each analysed software system, and the sources have been checked for possible aspects. Thus, the interpretation of the results is not only based on the suggestions from the algorithms but also on a manual validation using the existing program code. However, the generated traces do not monitor calls to any standard Java API method. This can have effects on the analysis results which are ignored for now.

3.1. Case Study "Graffiti"

Graffiti [2] is an industrial-sized editor for graphs and a toolkit for implementing graph visualisation algorithms, developed using Java. It currently comprises about 450 interfaces and classes, 3.100 methods and 82.000 lines, including comments. A tracing aspect, written in AspectJ, has been woven into the existing Graffiti system and the system obtained has been executed in seven different runs. In total, the traces consist of 33706 events. The analysis revealed 40 aspect candidates from before-execution relations, 40 from after-execution relations, 33 from first-execution relations, and 25 from last-execution relations.

In particular, *DynAMiT* has detected a typical crosscutting concern in Graffiti: logging. The analysis of the program traces found several calls to a method `format(LogRecord record)` of class `SimpleFormatter` as first and/or last call inside several set- and add-methods. A code investigation revealed that all executions of those methods are logged in a log-file. For that, a logger provided by Java's class `Logger` is used. We have not traced calls to the Java API but the logger uses a formatter to transform the system's log messages. The API provides an abstract class `Formatter` which is implemented by several special formatter classes but Graffiti's developers have chosen to write their own class `SimpleFormatter` implementing only basic functionality. The analysis detects the formatting of the log-messages and therefore provides us with the information that logging exists. Thus, the crosscutting logging functionality is revealed and can be encapsulated into an aspect in a re-engineering process.

Outside-aspect candidates. Figure 2(a) and (b) shows some of the before- and after-aspect candidates, resp., that *DynAMiT* has detected in Graffiti. Together, the results indicate that whenever `isSessionListener` is called,

```
(a) void editor.MainFrame.addSessionListener(SessionListener) ⇀
       boolean plugin.tool.AbstractTool.isViewListener(),
       String plugin.gui.AbstractGraffitiContainer.getId(),
       boolean plugins.inspectors.defaults.Inspector.isSelectionListener()
(b) void editor.MainFrame.addSessionListener(SessionListener) ↼
       void editor.StatusBar.updateGraphInfo(),
       boolean plugin.tool.AbstractUndoableTool.isSessionListener(),
       boolean plugins.modes.defaults.MegaMoveTool.isSessionListener(),
       boolean plugins.inspectors.defaults.Inspector.isSessionListener()
(c) Algorithm[] plugin.GenericPluginAdapter.getAlgorithms() ⇀
       String plugins.algorithms.bfs.BFS.getName(),
       String plugins.algorithms.trivialgridrestricted.TrivialGridRestrictedAlgorithm.getName(),
       String plugins.algorithms.generators.RandomGraphGeneratorNeighborConnecting.getName(),
       String plugins.algorithms.connect.Connect.getName(),
       String plugins.algorithms.bfstopsort.BFSTopSort.getName(),
       String plugins.algorithms.numbernodes.NumberNodesAlgorithm.getName(),
       String plugins.algorithms.connectspecial.ConnectSpecial.getName(),
       String plugins.algorithms.apsp.DijkstraAlgorithm.getName(),
       String plugins.algorithms.trivialgrid.TrivialGridAlgorithm.getName(),
       String plugins.algorithms.randomizedlabeling.RandomEdgeLabelingAlgorithm.getName(),
       String plugins.algorithms.fordfulkerson.FordFulkersonAlgorithm.getName(),
       String plugins.algorithms.springembedder.KKSpringAlgorithm.getName(),
       String plugins.algorithms.springembedderrestricted.KKSpringRestrictedAlgorithm.getName()
(d) boolean editor.MainFrame.isSessionActive() ∈⊤
       boolean editor.actions.ViewNewAction.isEnabled(),
       boolean editor.actions.RunAlgorithm.isEnabled(),
       void editor.actions.EditUndoAction.update(),
       void editor.actions.EditRedoAction.update(),
       boolean editor.actions.FileCloseAction.isEnabled()
(e) boolean editor.MainFrame.isSessionActive() ∈⊥
       boolean editor.actions.ViewNewAction.isEnabled(),
       boolean editor.actions.RunAlgorithm.isEnabled(),
       boolean editor.actions.FileCloseAction.isEnabled()
```

Figure 2. Example aspect candidates in Graffiti

this listener is added as a `SessionListener` to the affiliated Graffiti `MainFrame`. The remaining parts of the found pattern just provide information about the control flow, e.g., that after adding a `SessionListener` it is checked if a tool is a `ViewListener`.

Graffiti can easily be extended with graph algorithms by writing plugins. Before a plugin can be used, it has to be registered, which requires a unique string as identifier. Thus, every plugin has to implement method `getName` from interface `Algorithm` that provides the name of the corresponding algorithm. Figure 2(c) shows how this architectural principle is reflected in aspect candidates. In all appropriate algorithm classes, `getName` is always preceded by a call to `getAlgorithms` of class `GenericPluginAdapter`. Since Graffiti contains thirteen different algorithm plugins, *DynAMiT* detects thirteen individual aspect candidates; the automatic grouping as shown reveals that they all reflect the same architecture.

Inside-aspect candidates. Figure 2(d) and (e) shows some of the inside-aspect candidates detected by *DynAMiT*. In particular, it shows that the method `isSessionActive` in `MainFrame` is called as both first and last method within the method `isEnabled` in each of the classes `FileCloseAction`, `ViewNewAction`, and `RunAlgorithm`. In the system architecture, this is reflected by the fact that these three classes all extend the abstract class `GraffitiAction`. Therefore, the question arises why this functionality has not been encapsulated into `GraffitiAction` following established object-oriented design principles. The answer to that is quite simple: There are a lot more classes extending `GraffitiAction` which do not have the same functionality, e.g., `EditUndoAction` or `ExitAction`. Hence, the detected pattern is a distinct crosscutting pattern and thus a candidate for encapsulating into an aspect.

Moreover, `isSessionActive` was also found as first-aspect candidate in the `update` method in the classes `EditRedoAction` and `EditUndoAction`. A look into the code shows that `EditRedoAction` and `EditUndoAction` both extend the abstract class `GraffitiAction`. So, the question is again why the developers did not choose a better design. However, the analysis algorithm has detected this pattern in only those two classes but not all of the classes that extend `GraffitiAction`, and a quick investigation of the source code confirms that result. This suggests that the developers have not been able to provide a different design by encapsulating this concern into the superclass without overriding methods in subclasses, which would be considered bad practice. The introduction of more inher-

itance levels would not cure the problem either—in fact, there is no better solution since Java does not support multiple inheritance.

In summary, the analysis has shown that a lot of the functionality concerning actions like opening, saving, or editing files or graphs is crosscutting Graffiti's architecture. It is worth to consider restructuring the system accordingly.

3.2. Case Study "AspectJ Example telecom"

In another case study we checked whether *DynAMiT* can also detect crosscutting concerns in Java programs which are already extended by aspects written in AspectJ. For that purpose the `telecom` example from the AspectJ distribution has been chosen. It includes a small simulation of customers making telephone calls with different connection types (local and long-distance). The simulation can be executed at three different levels: `BasicSimulation` just performs the calls with the basic functionality (e.g., call, accept, hang up). `TimingSimulation` is the extension with a timing aspect which keeps track of a connection's duration and cumulates a customer's connection durations. `BillingSimulation` is a further extension with a billing aspect that adds functionality to calculate charges for phone calls of each customer based on connection type and duration. All three simulations have been traced and the resulting program traces have been fed into *DynAMiT*.

Analysis Results for BasicSimulation. The analysis of the telecom implementation with the basic functionality provides us with crosscutting concerns as well as some insights into the usual sequence of actions in phone calls. The application of the analysis constraints tells us that the simulation visualises the steps a customer is doing, such as calling someone, answering the phone, or hanging up. Also, when someone calls another person, the addition of the call to the pipeline of the customer is done as last thing. The same applies when a called person picks up the phone—the call is added to his pipeline.

Analysis Results for TimingSimulation. The resulting sets of aspect candidates in the `TimingSimulation` are larger. Generally, they include the aspect candidates already detected in the `BasicSimulation`. The analysis also discovers functionality added by the application of the timing aspect. Figure 3(a) shows these additional aspect candidates. They are clear crosscutting concerns: Before the timer can be started, stopped, or queried, one has to get hold of the timer belonging to the correct connection. Additionally, the timer is needed after a connection is completed or dropped, and when caller and receiver of a connection are determined. The reason is quite clear, as the timer of the appropriate connection is required to calculate the connection's duration.

```
(a) Timer telecom.Timing.getTimer(Connection) ⇀
        void telecom.Timer.start(),
        void telecom.Timer.stop(),
        long telecom.Timer.getTime()
    Timer telecom.Timing.getTimer(Connection) ↼
        void telecom.Connection.complete(),
        void telecom.Connection.drop(),
        Customer telecom.Connection.getCaller(),
        Customer telecom.Connection.getReceiver()
(b) long telecom.Timer.getTime() ⇀
        Customer telecom.Connection.getReceiver(),
        long telecom.Local.callRate(),
        long telecom.LongDistance.callRate()
   Customer telecom.Billing.getPayer(Connection) ↼
        long telecom.Local.callRate(),
        long telecom.LongDistance.callRate()
```

Figure 3. Add. aspect candidates in `telecom`

Analysis Results for BillingSimulation. Finally, the third simulation, which includes the timing aspect as well as a billing aspect on top of that, has been analysed. Again, the functionality already detected in the `BasicSimulation` and the `TimingSimulation` remains, and additional patterns introduced by the billing aspect are found. Figure 3(b) shows the additionally detected aspect candidates in the `BillingSimulation`. Before the call rate (local or long distance) for a connection or the receiver of a connection is determined, the actual time of the timer is needed. Furthermore, the analysis tells us that—after the correct call rate for a connection is determined—the connection's payer has to be found out.

A comparison of the analysis results for the three simulation versions clearly shows that the presented approach identifies basic functionality and the functionality seeded by the two different AspectJ aspects. The analysis is fully automatic, complete, (i.e., detects all candidates) and produces no false positives.

4. Related Work

One of the first aspect mining approaches that was developed is implemented in a tool called Aspect Browser [3]. It identifies crosscutting concerns with textual-pattern matching (much like `grep`) and highlights them. The tool assumes that aspects have a signature which can be identified by a textual regular expression. Its success in finding aspects thus strongly depends on naming conventions followed in the analysed program code.

AMT [4] 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. While the first works very well with strict naming conventions, the latter works better for objects of different types but similarly named. AMTex [12] has been built on top of AMT and offers additional analytical functionality, e.g.,

composing mining activities, managing mining tasks, and cross-analysing mining results.

The exploration tool JQuery [5] 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. The user can navigate through the browser's tree and can extend it by making additional queries.

Loughran and Rashid [7] investigated representations of aspects found in a legacy system in order to provide tool support for aspect mining. They considered three approaches: direct aspect storage coupled with meta-data, mapping of aspect anatomy to the database model, and a hybrid approach that combines the previous two.

Ophir [8] 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.

5. Conclusions and Future Work

This paper proposed a new analysis approach to identify crosscutting concerns in existing software systems that relies on the analysis of the program's run-time behaviour rather than its static structure. This is the first dynamic analysis technique for aspect mining [11]. It investigates program traces and abstracts them into execution relations called *outside-before-*, *outside-after-*, *inside-first-*, and *inside-last-execution relations*. Based on these relations, we have defined *uniformity* and *crosscutting* constraints to formalize the meaning of crosscutting concerns. An analysis algorithm that works on this abstract representation of program execution traces has been proposed; it focuses on occurrences in different calling contexts. Unlike other approaches that mine for *specific* user-defined aspects, our approach can identify automatically both existing and seeded crosscutting concerns.

The algorithm has been implemented in the *DynAMiT* tool that has been evaluated in several case studies. In the first case study, an industrial-sized graph editor has been analysed and many aspect candidates have been found. *DynAMiT* has helped to understand crosscutting concerns in the system. The second case study investigated whether the analysis approach can also be applied to detect crosscutting concerns in programs which are already extended by aspects. Here, *DynAMiT* identified the superimposed crosscutting behaviour of all these seeded aspects, without false positives.

In future work, the presented technique can easily be extended. After crosscutting concerns have been identified, they can be refactored into aspects. To validate that the refactored version works as intended, dynamic interference detection [9] can be used. This approach distinguishes intended impact of aspect application from unwanted interference by comparison of event traces. Futhermore, since our technique observes run-time behaviour, it uses resolved dynamic binding which can lead to false positives. We plan to additionally use static program information to eliminate such false positives. Finally, we intend to use quantitative information to identify crosscutting concerns in event traces that *almost* confirm to the execution relation constraints. This information can not only help to find more crosscutting concerns but can also detect possible errors in software systems. Known techniques from data mining can be used for this purpose, e.g., association rules.

Acknowledgements. A big thank you to Bernd Fischer for his valuable feedback.

References

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

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

[3] W. Griswold, Y. Kato, and J. Yuan. Aspect Browser: Tool Support for Managing Dispersed Aspects. TR CS99-0640, UCSD, 1999.

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

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

[6] G. Kiczales et. al. Aspect-Oriented Programming. In *ECOOP*, 1997.

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

[8] D. Shepherd and L. Pollock. Ophir: A Framework for Automatic Mining and Refactoring of Aspects. TR 2004-03, U Delaware, 2003.

[9] M. Stoerzer, J. Krinke, and S. Breu. Trace Analysis for Aspect Application. In *Workshop on Analysis of Aspect-Oriented Software (AAOS)*, 2003.

[10] P. Tarr, H. Ossher, W. Harrison, and S. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE-21*, pp. 107–119, 1999.

[11] A. van Deursen, M. Marin, and L. Moonen. Aspect Mining and Refactoring. In *First Intl. Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, 2003.

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