Trace-based Debugging for Advanced-Dispatching Programming Languages

Christoph Bockisch¹  Marnix van ’t Riet²  Haihan Yin²  Mehmet Aksit²  Ziyi Lin³  Yuting Chen³  Jianjun Zhao³

¹Open University, the Netherlands, ²University of Twente, the Netherlands, ³Shanghai Jiao Tong University, China
¹bockisch@acm.org, ²{m.vanriet,h.yin,aksit}@cs.utwente.nl, ³{linziyi,chenyt,zhao-ji}@cs.sjtu.edu.cn

Abstract
Advanced-dispatching programming languages allow to implicitly alter the behaviour of a program, depending on runtime program context. While this improves modularity, it also impedes comprehensibility. The use of advanced-dispatching programming languages can give rise to complex debugging scenarios, which cannot efficiently be resolved with traditional debugging approaches such as breakpoint-based debugging. Therefore, tool support for analysing the full history of runtime behaviour of advanced-dispatching programs is of importance for efficient debugging.

In this paper, we characterise debugging scenarios for which existing work does not apply well, because they require access to the program execution history for efficient resolution. We present our design and implementation of a trace-based debugger for advanced-dispatching that supports such debugging scenarios efficiently. Our approach is the first one based on an XML-representation of the execution trace, giving rise to using very powerful standard tools such as the XQuery language for searching and navigating the trace, or scalable visualisations such as tree-maps.

Categories and Subject Descriptors D.2.5 [Software Engineering]: Testing and Debugging—Debugging aids

Keywords Advanced-dispatching, aspect-orientation, debugging, trace-based debugging, omniscient debugging, XML, XQuery, tree-maps

1. Introduction
Several programming languages are proposed to improve program modularity by altering runtime behaviour implicitly, i.e. other modules do not have to call this functionality explicitly. Examples are AspectJ [16] and JPred [18]. Modules defined with such languages define functionality together with a specification of when this functionality is applicable and possibly when it should be invoked implicitly. Advanced-Dispatching (AD) programming languages realise this by manipulating the dispatching of, for example, method calls [2]. In this way, the behaviour of a module can be influenced and altered without manipulating the source code of that module. The down-side of this additional flexibility is the reduced comprehensibility—and thus debuggability—of programs written in such advanced-dispatching languages. Debugging support for this new generation of programming languages is challenged by their typical implementation strategy which is cross-compilation to a conventional language. As a consequence several source-level abstractions are lost in the compiled code and debugging defects they contain becomes difficult [32]. Thus dedicated debugging techniques and technologies are required.

The debugging process consists of three main steps: localising, understanding and fixing a program failure. Finding the root cause for a program failure appears to be the most time-consuming step [27]. Breakpoint-based debugging approaches for advanced-dispatching languages do exist [7, 8, 31]. However, not all kinds of defects can efficiently be localised in such an approach. Marc Eisenstadt [9] found that debugging becomes difficult when dealing with a significant cause-effect chasm, i.e. when a significant amount of computation happens between the execution of the root cause and the observed symptom of a program failure. In such scenarios, programmers cannot make accurate guesses about parts in the source code that are responsible for the program failure, because the relation between the root cause and symptom is less direct. In a previous study [26], we have shown that are also several cause-effect-chasm debugging scenarios specific to AD programming language.

To resolve cause-effect chasm debugging scenarios, the entire runtime information of the cause-effect chasm has to be collected, such that the programmer can navigate from the symptom up to the root cause. In this paper, we present the design and implementation of such a trace-based debugger which collects all relevant execution events to support rich operations to query and navigate the execution of advanced-dispatching programs, respecting the source-level abstractions. For the trace representation, we employ XML. Throughout this paper we argue that the structure of an XML document matches very well with the structure of a program call tree giving rise to expressing debugging tasks very naturally with XML-based technology such as XQuery.

The primary contributions of our work are:

• a trace-model for AD programming languages based on a distinction between AD and non-AD activities,
• an implementation of an execution environment for collecting traces,
• an intuitive visualisation layout of an AD program trace based on tree-maps, and
an identification of the potential of a query-based debugging approach to locate AD specific information in a trace.

2. Approach overview

The approach consists of three main components. Firstly, the target program, i.e. an AD program that is subjected to the trace-based debugging approach. Furthermore, a back-end and front-end, which are decoupled by a representation of an execution trace of the target program.

![Image](image.png)

Figure 1: Relational overview of the approach

The back-end observes the program’s execution, captures runtime information, and produces a trace of the target’s program execution. The trace-recorder instruments the AD program. These instrumentations allow the trace-recorder to capture relevant runtime information according to a trace-model. A trace-model defines a selection of program statement types, e.g. an assignment to a local variable, for which the runtime behaviour needs to be recorded. From the captured runtime information, the trace-recorder produces an execution trace conforming to the structure of the storage-model.

The front-end provides navigational aids to track down failures in the execution trace. The visualisation employs a tree-map design to support sufficient scalability. This visualisation allows to navigate through the execution trace by following a top-down strategy, i.e. navigating from high-level information towards low-level detail in the call hierarchy of the trace. A query engine provides a generic query language that permits programmers to quickly locate specific runtime values. The query engine works hand-in-hand with the trace visualisation. Query results are reflected on the visualisation. Consequently, a query-based debugging approach is combined with the trace visualisation to improve navigability.

3. The Back-end

Our back-end is implemented as an extension of ALIA4J [2], an architecture for implementing advanced-dispatching languages. A key concept of this architecture is to maintain a first-class model of declarations that influence dispatch, such as the pointcut-advice in AOP or predicate methods. These models influence the execution of statements for which ALIA4J supports dispatching (e.g., method calls, field accesses, or object creation). The dispatched statements as well as events during the evaluation of dispatch models constitute the execution elements in our trace, and we commonly refer to them as joinpoint. Since our approach is heavily influenced by ALIA4J’s metamodel (Figure 2) of dispatch declarations, we give a brief overview below.

![Image](image.png)

Figure 2: ALIA4J meta-entities

An Attachment is the root entity. It functions as a container of all entities for a dispatch description. A Specialization consists of entities that dictate which joinpoints must be intercepted. The Pattern entity allows to select statements by lexical conditions, e.g. a method name. The Predicate entity is for expressing runtime conditions for the dispatch function. The AtomicPredicate represents a condition with no deeper propositional structure. This allows the Predicate entity to represent a composition of atomic conditions. The Action entity represents the functionality which is invoked whenever its correlating specialization is satisfied. The Context entity represents runtime values during the execution of dispatch. The ScheduleInfo and PrecedenceRule together specify the ordering of Actions. A CompositionRule declares constraints on Actions being executed together.

There are several execution environments implemented following the ALIA4J architecture. For the work presented in this paper, we have extended the interpretation-based execution environment, because this is most flexible and simple. The implementation strategy of this interpreter is to intercept the execution of all relevant dispatch sites, identify the effective dispatch declarations and interpret them.

3.1 General requirements

Trace-capturing must include sufficient information to support the following features offline: object state reconstruction for a given moment in time, and causality-link tracing. The latter means to trace an observed value back to its (last) assignment to a variable. Joinpoints such as the evaluation of an if-condition or reading a local-variable do not have to be recorded because they induce no state change to the program. Hence, the runtime information of those joinpoints can be acquired by a computed data reconstruction.

Because an AD program extends a traditional base-program, the aforementioned defined requirements apply in this context as well. Moreover, debugging the implicit behaviour of AD source abstractions raises additional questions. These questions are related to (1) the bidirectional relation...
3.2 Storage-model

This section describes a storage-model that produces a call tree representation of the program execution. Each joinpoint type that is associated with the call to a subroutine is rendered as an inner node. Recursive calls, for example, are naturally supported by this representation as nested sub-trees with root-nodes referring to their parental call joinpoint. The scope of a call joinpoint lasts until the execution of its corresponding exit joinpoint, i.e. joinpoint types that can be associated with the exit to a subroutine. The execution order of the joinpoint nodes can be reconstructed by performing a pre-order traversal on the call tree.

Figure 3 shows a simplified call tree representation of the execution of the program in Listing 1. The pre-order traversal of the call tree is labeled by the numbers assigned to each node. The nodes 1, 2, 3, 7 and 9 are inner nodes, because they represent the function calls of the program's execution. The term calling depth is used to refer to the depth, of a specific node, in the call tree. For example, node 5 was executed at a calling depth of three. We use the term execution path to refer to a walk from the root to a specific node in the tree, effectively representing the call stack of that node. For example, the execution path "main(args) / init() / n() / o()" led to the execution of the joinpoint node 10.

Listing 1: An example Java program.

```java
public class Main {
    private static int f;
    public Main() { m(); n(); }
    public int m() { int a = 1; int b = 2; for(int i=0;i<150;i++) { doSomething(); } return a + b; }
    public void n() { int d = 4; o(); }
    public void o() { int e = 5; f = 6; }
    public static void main(String args[]) {
        new Main();
    }
}
```

Figure 3: Call tree representation of the execution of the program in listing 1.

3.3 Implementation and Performance

As mentioned in the beginning of this section, our implementation is based on an interpreter of the ALIA4J architecture. This intercepts the execution of all relevant statements, such as method invocations or field accesses, and interprets all applicable Attachments. Our prototype of a trace-based debugger hooks into the interpretation of relevant joinpoints to gather the trace information. In addition, we instrument the bytecode of the executed application to insert tracing for local variable accesses. This is necessary because these statements are not considered relevant for dispatching and are thus not intercepted by the ALIA4J interpreter. But local variable accesses are relevant for recovering program states from traces.

While we did not target runtime efficiency yet in our prototype, we nevertheless assessed its performance. For this purpose, we applied micro-measurements [12] to measure the overhead of taking the trace information for specific kinds of statements. In particular, we measured the execution time of local variable accesses (local), field accesses (field), function calls (function) and action calls (action) in the four settings: original executing on a standard JVM without the ALIA4J interpreter or bytecode instrumentation, interpreter with the standard version of the interpreter, tracing with additional instrumentation and gathering of trace information, and storing with additional storing of the trace. The results presented in Figure 4 are presented as overhead, i.e., as multiples of the execution in the original setting. The interpreter setting already shows an enormous overhead in the thousands. However, we would like to point out that we measured only single operations that are interpreted here, while in an actual program a mix of operations is executed, containing many that are not interpreted and executed at full speed.1

As the overhead of the local variable write in the tracing category shows, the overhead can be as low as 242 times, if the trace is not gathered through the interpreter but through bytecode instrumentation. This overhead is similar to that of other approaches discussed in the related work section, which range between 176 and 252.

The storing data category shows that storing the trace as an XML document is very slow, which is not unexpected due to the verbose nature of XML. Again, we want to point out that we have measured worse-case scenarios here and that performance can be increased significantly, e.g., through

---

1 Besides, the performance overhead is not inherent to the ALIA4J approach but due to the interpretation-based implementation.
caching. Early experiments also show that choosing for more compact tag and attribute names in the XML representation significantly improves the performance.

Due to the verbosity of our storage format and the naming scheme we apply, traces get relatively big. However, their size is still comparable to the size of trace files created by contemporary trace-based debugging approaches. Pothier et al. [22] present a trace size of 3.6GB after running a program that executed tens of millions of joinpoints. We approximate that our approach would currently create a trace file of size 7.6GB for the same run. In the future, we would like to apply pruning techniques to reduce the trace size.

A full account on the implementation details of our prototype, as well as a more detailed evaluation is presented in the Master Thesis of the second author [25].

4. Front-end
The front-end of our approach is implemented by the prototype debugger tool called ALIA-TBD (“ALIA trace-based debugger”), which is a standalone Java tool. ALIA-TBD uses the core modules of BaseX\(^2\) to parse the XML trace file, visualise it, and process XQueries.

4.1 Trace visualisation
A typical program can produce tens of thousands joinpoints in a couple of seconds. Programmers may get overwhelmed and even lose track because of the huge amount of information that is given to them. A visualisation can assist in locating parts of interest in large information spaces. An important challenge that needs to be addressed, when designing such a visualisation, is navigability. Navigability refers to the ability of allowing users to locate parts of interest in the trace in an efficient and convenient way.

Since the back-end of our work produces a call-tree representation of the trace, we further refine the navigability fundamental to this representation. Foltz [11] identified three main requirements to create an effective, and navigable visualisation of a large information space. Each requirement can be mapped to the context of a program trace visualisation, which is described next.

\(\text{R1 }\) “Information in the space should be placed according to an organising principle, and this principle should be communicated explicitly to users.” For examples, join points should be presented in chronologic order.

\(\text{R2 }\) “Users should be able to make a correct navigation decision to take the right next step, even if the eventual destination is imprecisely known.”

\(\text{R3 }\) “Orientation should be recoverable at every point in the space. Users should never feel lost.”

Another important design fundamental is related to the AD concept. The main aspect that distinguishes AD programs from traditional object-oriented programs is the implicit behaviour that can be expressed. We therefore add as requirement:

\(\text{R4 }\) A notation of AD program behaviour must make implicit behaviour obvious.

In the following we will discuss how our approach achieves these requirements.

4.1.1 \(\text{R1 Tree-map layout as organising principle}\)
Tree-maps are two-dimensional, space-filling visualisations of hierarchical data sets [23]. Each node in the data set is represented by a rectangle. A tree-map visualises the composition of these rectangles, and the parent-child relations that the dataset defines. The latter is achieved by letting rectangles surround multiple smaller rectangles.

A tree-map can visualise the trace by providing a rectangular representation of each joinpoint. Rectangles of high-level calls surround those of low-level detail.

An example of the organising principle is shown in Figure 5, it represents the trace of executing the base program in Listing 1 together with the aspect 2. The programmer can conveniently eyeball the high-level calls of the example program, i.e. \(\text{Main.init()}, \text{m()}\) and \(\text{n()}\), because of their rectangular size. The details of joinpoints deeper down the call hierarchy are not shown, because there is not enough space to show all detail. But since the top-level calls are the most important ones for comprehending the program behaviour, hiding the details of low-level calls in this view is the right thing to do. In the next sections, we will discuss how to obtain information about the lower-level joinpoints and how to focus the view on them.

The execution order of the trace is preserved by using the \(\text{Split}\) algorithm [1] to construct the tree-map layout. The order is preserved in a two-dimensional fashion. The top-left of the visualisation is considered as the origin. The following order-preserving condition holds: if joinpoint \(v\) preceded the execution of joinpoint \(w\) then \((v.xPos < w.xPos)\) or \((v.yPos < w.yPos)\). A diagonal arrow is added to Figure 5 which depicts the order.

Note that the tree-map uses a colouring scheme based on calling depth. Joinpoints that were executed deep down the call-tree are represented by a relatively darker colour. In Figure 5, the rectangles that are surrounded by the function call to \(o()\) have a darker grey colour than those surrounded by the function call to \(n()\). This further improves the comprehension of the visualisation because it helps users to distinguish low-level information from high-level information.

4.1.2 \(\text{R2 Efficient navigational decisions based on exposed joinpoint details}\)
Showing plain rectangles does not help the programmer to localise joinpoints of interest. The tree-map is therefore

\(\text{R2 E}\)

\(\text{R2}\) by exposing details.
extended with context-awareness, i.e. it “knows” what kind of joinpoint it is drawing. This context-awareness consists of short textual messages, shown as tooltip, which summarise the activity of a joinpoint. By consulting the textual messages, users can make informed decisions about the relevance of joinpoints for a debugging task, thus without having to explicitly ask for details. These messages permit users to make efficient navigation steps in a multi-staged strategy.

Table 1 presents the format of the contextual message for each joinpoint type. An acronym is defined for each joinpoint type. Examples of contextual messages are also included in the table. Important to note is that primitive types are displayed by value and object types by a unique object identifier (OID).

To examine all the details of a joinpoint, an additional view, called the joinpoint-info view, is positioned next to the tree-map visualisation. An example of this view is shown in Figure 6. This view is triggered when the user selects a joinpoint in the tree-map visualisation. Once triggered, it extracts all detail of the selected joinpoint and outlines the information in a tree structure. This structure reflects the XML trace representation. Attributes and leaf element nodes are shown by value.

4.1.3 R3 Location orientation and trace navigation through interactive multi-staging

The tree-map visualisation employs an interactive strategy to provide multi-staging. Initially, an overview of the trace is visualised, where only the execution information about high-level calls can be observed because they are represented by rectangles that are large enough to include a context-aware message. The user can choose any displayed rectangle in this overview and request to zoom in to it. The tree-map then uses the entire display space to visualise the chosen rectangle, thus enlarging it. In effect, the rectangles, that are surrounded by the chosen rectangle, are also enlarged. The same steps can be applied at the currently displayed stage, thus navigating deeper down the call-tree representation of the trace. While using multi-staging, the programmer can deduce the location in the trace by consulting the contextual message of the rectangle that was zoomed into, because this rectangle is always displayed in the consecutive stage.

An example of the multi-staging in action is shown in Figure 7. This figure presents the trace visualisation when the user has double-clicked the selected joinpoint rectangle of Figure 5. The selected joinpoint rectangle represents an ActionCallJoinPoint to the around-advice of Listing 2. This can be concluded from the contextual message in Figure 5, marked by a 1, and helps the programmer to deduce the location in the trace. Unrelated joinpoint rectangles are removed from the visualisation such that more detail about the around-advice can be displayed. AD activity is coloured by a separate scheme, as shown by the legend on the right-hand side of the figure. This colouring scheme is described next.

Listing 2: An example AspectJ aspect which is applied to the base program of listing 1.

```java
1 public aspect Azpect {
 2  pointcut pc() : set(static int Main.f);
 3  int around() : pc() {
 4    System.out.println("around"); return proceed();
 5 }
 6  before(): pc() {
 7    System.out.println("before");
 8  }
 9 }
```

Figure 7: Detailed stage. The selected joinpoint rectangle of Figure 5 is now used as the root of the visualisation 2.

4.1.4 R4 AD awareness

The tree-map design adds several techniques to become aware of AD activity and gain insight about the relation between AD and base-program. The first technique is a colouring scheme that highlights the AD activity of the trace. Colouring can help to spot AD activity quickly, because the programmer does not have to inspect the context of a joinpoint individually to become aware of AD activity. By default, all joinpoints that can be associated with AD activity are given a purple colour, whereas base-program joinpoints are shown in gray. An example of this colouring scheme is shown in Figure 7.
Though this indicates nothing more than the existence of a Tanter’s vision on execution levels in AOP. This can help to trace the activity of a specific attachment, vice call raises the level of execution whereas the message of an advised joinpoint is prefixed with a '#' symbol. Furthermore, the contextual stage of the visualisation, a tooltip above its related rectangle is drawn. This tooltip helps users to locate the advised stage of the visualisation, a tooltip above its related rectangle. The contextual message of an AD joinpoint always includes a different colour to each attachment that can be found in the trace. Programmers can select attachments in the legend, triggering the tree-map visualisation to be redrawn according to the legend selections. This can help to trace the activity of a specific attachment, as shown in Figure 8. The activity of the around and before advice is coloured red and yellow respectively.

A single-colouring scheme for AD activity, however, becomes less useful when trying to localise the activity of specific AD entities. The tree-map visualisation therefore includes a legend. This legend assigns a different colour to each attachment, i.e. an advice call raises the level of execution whereas proceed() calls lower the level of execution. Joinpoints that are executed during a call to proceed, which lowered the execution level down to zero, are considered as base-program activity. The advised joinpoint in Figure 8 and Figure 7 is therefore coloured gray, because the execution level of zero (base) is reached because the execution level of zero (base) is reached because the function call to proceed() by the around-advice of Listing 2.

The second technique is related to the bidirectional relation between AD and base-program. As shown by table 1, the contextual message of an AD joinpoint always includes a representation of the advised joinpoint ID. This helps programmers to locate the advised joinpoint of AD activity in the trace. If the advised joinpoint is displayed in the current stage of the visualisation, a tooltip above its related rectangle is drawn. This tooltip helps users to locate the advised joinpoint rectangle conveniently. Furthermore, the contextual message of an advised joinpoint is prefixed with a '#' symbol. Though this indicates nothing more than the existence of a relation to AD activity, the awareness of its presence is raised. More information about the related AD activity can then be acquired by, for example, consulting the joinpoint-info view. Examples of the contextual message of AD joinpoints, the tooltip of an advised joinpoint rectangle, and the contextual message prefix are shown in Figure 8, see (3) and (4). The third technique is related to the ability of "around" actions to skip the execution of the function call or field access in the source code. The trace-model keeps track of skipped joinpoints as attributes of the ActionExitJoinPoint and the trace visualisation represents them by a black tree-map rectangle, which is a colour that is not used by other colouring-schemes. This can be helpful when the programmer is exploring the trace and looking for a particular joinpoint that was unexpectedly skipped.

### Table 1: Format of contextual message per joinpoint type.

<table>
<thead>
<tr>
<th>Joinpoint type</th>
<th>Acronym</th>
<th>Contextual message format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocalVariableWriteJoinPoint</td>
<td>LVWP</td>
<td>&lt;varName&gt;=&lt;writeValue&gt; (varType)</td>
<td>var=10 (int)</td>
</tr>
<tr>
<td>FieldWriteJoinPoint</td>
<td>FWJP</td>
<td>&lt;fieldName&gt;.&lt;varName&gt;=&lt;writeValue&gt; (&lt;varType&gt;)</td>
<td>Obj:1.aField=10 (int)</td>
</tr>
<tr>
<td>FieldReadJoinPoint</td>
<td>FRJP</td>
<td>&amp;&lt;fieldName&gt;.&lt;varName&gt;=&lt;readValue&gt; (&lt;varType&gt;)</td>
<td>&amp;Class1.aStaticField=10 (int)</td>
</tr>
<tr>
<td>FunctionCallJoinPoint</td>
<td>FCJP</td>
<td>&lt;context&gt;.&lt;functionName&gt;()</td>
<td>Class1.aStaticFunction()</td>
</tr>
<tr>
<td>FunctionExitJoinPoint</td>
<td>FEJP</td>
<td>&lt;context&gt;.&lt;functionName&gt;()</td>
<td></td>
</tr>
<tr>
<td>ThrowJoinPoint</td>
<td>TJJP</td>
<td>Throw Obj:1()</td>
<td></td>
</tr>
<tr>
<td>ArrayWriteJoinPoint</td>
<td>AWJP</td>
<td>&lt;array&gt;[&lt;index&gt;]=&lt;writeValue&gt;</td>
<td></td>
</tr>
<tr>
<td>ActionCallJoinPoint</td>
<td>ACJP</td>
<td>Action:&lt;advisedJoinPointId&gt; (&lt;scheduleInfo&gt;)</td>
<td></td>
</tr>
<tr>
<td>ActionExitJoinPoint</td>
<td>AEJP</td>
<td>Action:&lt;advisedJoinPointId&gt; (&lt;scheduleInfo&gt;) / actionResult (returnType)</td>
<td>Action:1(Before-1234)</td>
</tr>
<tr>
<td>PredicateEvaluationCallJoinPoint</td>
<td>PCJP</td>
<td>Pred:&lt;predictedOID&gt; (&lt;interceptedJoinPointId&gt;) / evalResult</td>
<td>Pred:1(1234) false</td>
</tr>
<tr>
<td>PredicateEvaluationExitJoinPoint</td>
<td>PEJP</td>
<td>Pred:&lt;predictedOID&gt; (&lt;interceptedJoinPointId&gt;) / evalResult</td>
<td>Pred:1(1234) true</td>
</tr>
<tr>
<td>PrecedenceRuleEvaluationJoinPoint</td>
<td>PRJP</td>
<td>Pred:precedenceRuleOID= (&lt;advisedJoinPointId&gt;)</td>
<td>Pred:1(1234)</td>
</tr>
<tr>
<td>CompositionRuleEvaluationJoinPoint</td>
<td>CRJP</td>
<td>Comp:compositionRuleOID= (&lt;advisedJoinPointId&gt;)</td>
<td>Comp:1(1234)</td>
</tr>
<tr>
<td>AttachmentDeploymentJoinPoint</td>
<td>AUJP</td>
<td>AttDep: &lt;attachmentOID&gt;</td>
<td>AttDep:1</td>
</tr>
<tr>
<td>AttachmentUndeploymentJoinPoint</td>
<td>AUJP</td>
<td>AttUndep: &lt;attachmentOID&gt;</td>
<td>AttUndep:1</td>
</tr>
</tbody>
</table>

Figure 8: Same stage as in Figure 7. The attachment legend is used to colour the activity of the two advices separately (3).

5. **Query-based debugging for AD programs**

The trace visualisation, of Section 4.1, provides the means to localise and inspect the joinpoints in the trace. However, these localisation techniques do not work well in three situations:

- Firstly, the joinpoints of interest can be scattered throughout the trace, whereas the trace visualisation displays one region at a time. Consequently, one must reiterate the navigational steps of the trace visualisation to localise each joinpoint.
- Secondly, joinpoints can be located at a low-level in the call tree. Since the trace visualisation follows a top-down navigation approach, programmers may have to follow a relatively long path, which may require many interactive steps with multi-staging to arrive at the level where the joinpoints of interest are located.
- Thirdly, the program behaviour that must be validated may be related to specific runtime conditions of the program, e.g. an if condition, or complex inter-object relationships. With the top-down navigation approach of the trace visualisation, programmers cannot instantly see where these runtime conditions were met. Thus, programmers have to navigate to all parts in the trace that are candidate to these conditions, and then manually check if they are satisfied.

To conclude, using the trace visualisation, when dealing with one or more of these three situations, results in a tedious
debugging process, especially if the programmer desires to compare joinpoints of interest with one another.

5.1 Visualising query results

Querying can aid localisation by letting programmers express complex questions about the trace in a dedicated language. The result of such a query is an ordered collection of joinpoints which represents a sub-tree of the trace. The query result can be presented instantly to programmers, even if the matching joinpoints are scattered throughout the trace, or located deep-down the call tree. Moreover, previously generated query results can be used as a search domain on which future querying is processed. This provides the localisation of specific program behaviour by executing a sequence of queries, where each query further refines the search domain of the next.

The results of a query, executed from the Query-editor view, are projected on the trace visualisation. This highlights, with a colouring scheme, the tree-map rectangles that correspond to the joinpoints of the query result.

Query results may be located too deep in the call tree compared to the level of detail that is currently displayed by the trace visualisation. To ensure that programmers can locate such query results, the trace visualisation highlights the last call joinpoint, in their corresponding execution path, that is viewable in the current stage of the trace visualisation. This strategy enables programmers to eyeball query results, no matter how deep they are located down in the call tree. The navigation techniques, which are provided by the trace visualisation, can then be used to walk the execution path that led to a query result.

For illustration, consider the following example. During debugging of the AspectJ program from Section 4.1.3 we are using this query, selecting all join points at which an advice is implicitly invoked:

```
1:  tr//JoinPoint[Isadvised="true"]
```

Figure 9 shows the visualisation of the results of this query, highlighted in green (see 1 and 2). If more than one join point matches a query, we would see multiple rectangles highlighted in green. In the initial visualisation (left-hand side of the figure) the ActionCallJoinPoint shows the same treemap as Figure 5 with the only difference that the action at 1 is now highlighted in green because it represents a query result. More precisely, the highlighted rectangle represents the deepest visible call in the execution path in whose control flow a join point is located that is selected by the active query. By using multi-staging, one can locate the exact query result, as shown in the right-hand side of Figure 9.

Once a query has been executed, the programmer can choose to remove all joinpoints, that did not match, from the trace visualisation. This feature is called query-results-only, and can be useful when query results differ in calling-depth, or if the programmer does not bother about their location in the trace.

5.2 Common debugging tasks as queries

An important note is that the queries, which are presented throughout this section, are templates, and thus do not exhaust the full extent of our query-based approach. Nevertheless, the purpose of this section is to give an impression of the kind of information that queries can target, and demonstrate how well they fit in the context of debugging AD programs. Arbitrary query constraints can be added, which can refer to specific program context, thereby enabling programmers to express more powerful queries.

Several query languages, which are specifically designed for XML (e.g. XQL[5], XQuery), can process the XML trace of our back-end. For our approach, we have chosen the XQuery standard. The trace structure, which is a call tree, conforms to the design philosophy of XQuery, i.e. traversing a tree structure. Therefore, the powerful expressiveness of XQuery can be exploited to efficiently, and conveniently, traverse the trace. This allows to translate, among others, common debugging facilities into compact and readable queries. Furthermore, XQuery’s axes reflect meaningful directions in which a trace can be explored. For example, the following and preceding axes relate to the execution order of the trace. These axes can thus be used to traverse the trace in a forward or backward-in-time direction.

An illustrative example of the adoption of XQuery to our trace is to locate causality links. Given a FieldReadJoinPoint called \( tr \) \( jp \), the previous assignments to the field can be retrieved by the XQuery template expression shown in this listing:

```
1:  [frjp/preceding::JoinPoint[Type="FWJP" and FieldId=frjp/FieldId] ]
```

By using the preceding axis, the query locates FieldWriteJoinPoints, that preceded the execution of \( tr \) \( jp \), and refers to the same field as \( tr \) \( jp \). The result of this query is an ordered set of joinpoints. Such a set can be explored, in a backward-in-time fashion, up to first assignment to the field.

Inspecting object state can be realised by performing the above query for each field of an object. The query finds the last assignment to the field before the currently selected joinpoint and, thus, the field’s current value. Program states can be reconstructed in a similar way: by querying the last assignment to the LocalVariableWriteJoinPoints that are applicable in a given scope.

Stepping is a well-known debugging feature that lets programmers examine the execution of a program with several metaphors, e.g. step-next, step-over, step-into. This allows programmers to step through the execution of the program from the moment it was suspended.

As mentioned, the axes of XQuery correspond to a meaningful way in which the trace is navigated. Several of these axes can be used to translate the stepping metaphors to compact and readable queries. Moreover, since our querying approach targets an exhaustive trace, the conventional stepping metaphors of breakpoint-based debuggers can be augmented with metaphors to navigate back-in-time, e.g. step-back.

To explain the stepping queries we introduce two terms: initial-joinpoint refers to the joinpoint before a stepping query is performed; step-joinpoint refers to the joinpoint that is reached after a stepping query has been performed. Each stepping query determines, based on its semantics, by which axis the call tree must be traversed, in order to reach the step-joinpoint from the initial-joinpoint. Figure 10 depicts the effect of each stepping query in the example call tree, which was introduced in Section 3.2.

These stepping metaphors accept any initial-joinpoint and step to the next, or previous, joinpoint in time. The corresponding queries therefore realise a single step of a (reversed) pre-order traversal of the call tree, starting from the initial-joinpoint. The listing below shows the XQuery expressions of step-next (line 2) and step-back (line 4). These queries traverse to the first joinpoint in the axis that corresponds to the order in which the call tree needs to be traversed, i.e.
Figure 9: Left: initial stage of trace visualisation highlighting a query result (1). Right: detailed stage highlighting same result (2).

Figure 10: Depiction of the stepping metaphors.

The functional composition abilities of the XQuery language give rise to two novel features in trace-based debugging. Firstly, the stepping commands can be composed with each other, and secondly they can be combined with XQuery predicates. In effect, more powerful stepping queries can be formulated, which can be helpful to skip unrelated joinpoints while stepping through a trace.

For example, suppose the programmer is debugging the trace of the AspectJ program, which was introduced in listings 1 and 2. During stepping, the programmer encounters the loop of line 6 of the base-program. If the programmer wishes to inspect the $n$-th iteration, she must manually execute `step.next` a total number of $n$ times. This can become time-consuming. By adding an XQuery predicate to the stepping query, the programmer can instantly step to the joinpoint that represents the $n$-th iteration in the loop. An example of a conditional stepping query, where the debugger steps to the 76th iteration of the loop, is shown in this listing:

```
| step.next[LocalVariableId/VariableName="i" and NewValue/Int > 75] |
```

5.3 Querying AD-specific information

Given an AD joinpoint type, the relation to the joinpoint in the trace that was intercepted, can be located with the query in Listing 3, on line 1. This query performs a join on the `joinPointId` attribute of every joinpoint in the trace. This query can be further specialised based on the properties, or type, of the AD joinpoint. For example, when the AD joinpoint is related to the call to a before-action, the intercepted joinpoint was executed afterwards. Thus, the search domain of the query can be limited by traversing the trace only in a forward direction, regarding execution order. This can be realised by adding an additional predicate to the XQuery expression. Line 3 in Listing 3 adds an XQuery predicate, such that the query limits the search domain by removing the joinpoints that preceded the execution of the before-advice.

```
| tr//JoinPoint[@joinPointId=$adJp/InterceptedJoinPointId/text ()] |
```

Listing 3: XQuery template expression to locate the intercepted joinpoint.
**Predicates** reflect the dynamic conditions that need to be satisfied before an intercepted joinpoint is advised. This can make it difficult for programmers to predict which joinpoints will be advised at runtime. The following query templates can locate predicate evaluations with specific properties in the trace. These templates can therefore assist in locating the runtime behaviour of suspicious predicates, such that the programmer can verify if they behaved correctly.

A typical example in AOP is the *unmatched joinpoint* bug pattern. This bug pattern arises when programmers expect certain joinpoints to be advised during execution, but they never were. The query template of the listing below (line 1) locates each predicate evaluation that dissatisfied a given joinpoint. The starting point of the query is the ID to the unmatched joinpoint. The corresponding *PredicateEvaluationExitJoinPoints*, which evaluated to *false*, are then located in the trace by reusing the query template of Listing 3. The result of the query template is therefore a set of *PredicateEvaluationExitJoinPoints*. From each such joinpoint in the result set the programmer can explore the runtime behaviour of the predicate evaluation to find out why it resulted to *false*. The query template can be further refined if the programmer suspects that a specific predicate is responsible for the unmatched joinpoint. This can be realised by adding an extra XQuery predicate to the original query, as shown at line 3 in the following listing:

```xquery
1. tr//JoinPoint[Type="PEJP" and InterceptedJoinPointId=$sum]/@joinPointId
2. for ($shared := tr//JoinPoint[Isadvised="true" and EvaluationResult="false" and
3. PredicateId=$shared]/@joinPointId)

Since AD allows to advise a *joinpoint with multiple attachments*, the execution sequence of these attachments may affect the outcome of the program. The missing of an explicit precedence declaration, among such attachments, can cause the program to behave unexpectedly, as demonstrated by Zhang and Zhao [33].

The listing below presents a query template to locate shared joinpoints, for which no precedence declaration evaluated the applicable sequence of attachments. First, the shared joinpoints (i.e., a joinpoint at which more than one AD attachment is applicable) of the trace are retrieved (line 1). Then, the query iterates over each shared joinpoint (line 2) and locates the corresponding *PrecedenceRuleEvaluationJoinPoint* (line 3). Finally, if no corresponding evaluation of a precedence rule was found (line 4), the shared joinpoint is added to the result set of the query (line 5).

```xquery
1. let $shared_jps := tr//JoinPoint[Isadvised="true" and count(
2. ActionIdsPlanned//ActionId) > 1]
3. for ($shared_jp in $shared_jps)
4. let $precedence := tr//JoinPoint[Type="PRJP" and
5. advisedJoinPointId=$shared_jp/@joinPointId]
6. where empty($precedence)
7. return $shared_jp
```

6. **Related Work**

Several debugging approaches for advanced-dispatching programming languages have been proposed in the recent years. Most of this work has focused on online, i.e., breakpoint-based debugging. A comprehensive account on these works can for example be found in our previous publication [30].

In this section, we focus on related trace-based debugging approaches.

**Trace-based debugging approaches with AD support** To our best knowledge, there exists only one trace-based debugger that provides special support for AD. This work by Pothier et al. [22] extends their earlier trace-based debugger for Java [20] with support for the AspectJ language. Their approach works on the woven Java bytecode, wherein a reference to several source abstractions is lost. Therefore, their approach can be recognised more AD-specific elements in the debugging process than the work by Pothier et al.

**Non-AD-specific exhaustive trace approaches** The work of Bohnet et al. [3] and De Pauw et al. [19] presents trace visualisation strategies that are based on call trees, similar to our approach. Both works use a graph-like visualisation layout. Tree-maps as used in our approach are known to scale better than graph representations.

There are a number of approaches that support debugging based on querying execution traces, such as JHyde [13], the JIVE system [6] or WhyLine [17]. These approaches have in common that they employ a proprietary query language with limited expressiveness.

**Replay-based approaches** Replay-based debuggers such as Xu et al. [28], Bdb [4], Igor [10], STIQ [21], EXPOSITOR [15], and ReTrace [29] produce a trace by replaying the execution of the program between well-defined checkpoints. In these approaches, trace collection has less impact on the program execution than an exhaustive trace as in our case, but replays are not generally possible.

7. **Conclusions and future work**

This paper has shown our design and implementation of a trace-based debugging approach that copes with the special properties of advanced dispatching. This approach makes it possible to efficiently locate the root cause in such debugging scenarios, and improve the overall comprehensibility of the developed system.

The implementation of the back-end employs a call-tree representation to structure the trace, which groups runtime information according to its respective function call in the trace. This call-tree structure and its representation as an XML document naturally matches the functional decomposition of programs. We have thus shown that debugging using XML-based queries on the program trace are an instrument that naturally matches the intuition of programmers. We believe that our approach of representing the trace as XML document lays the foundation of very promising future work in the general field of post mortem program analysis such as test coverage analysis or fault prediction [14].

Future improvements of our prototype include better support for multi-threaded programs and the provision of an extensive library of re-usable XQueries for re-occurring debugging tasks. A more distant future work is to develop support for online XML-based debugging. In this project we will upgrade our prototype to produce a steady XML stream and use corresponding querying tools such as stream XQuery.

**References**


