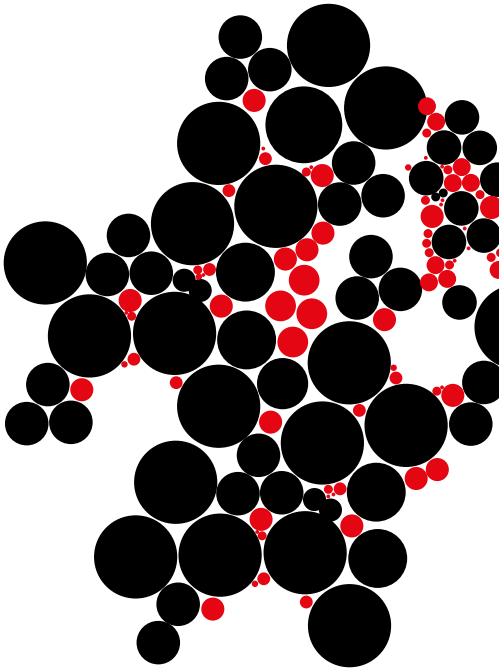


# Trace-based debugging for Advanced-Dispatching Programming Languages

Marnix van ´t Riet

August, 2013

A thesis submitted to the University of Twente
for the degree of Master of Science

Department of Computer Science
Software Engineering

**Examination committee**
Dr. Ing. Christoph Bockisch
Haihan Yin, MSc
Dr. Andreas Wombacher

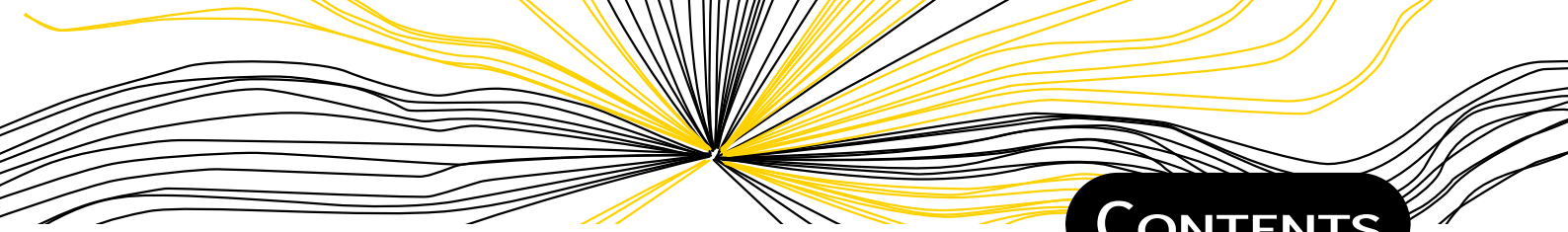
UNIVERSITY OF TWENTE.

# Abstract

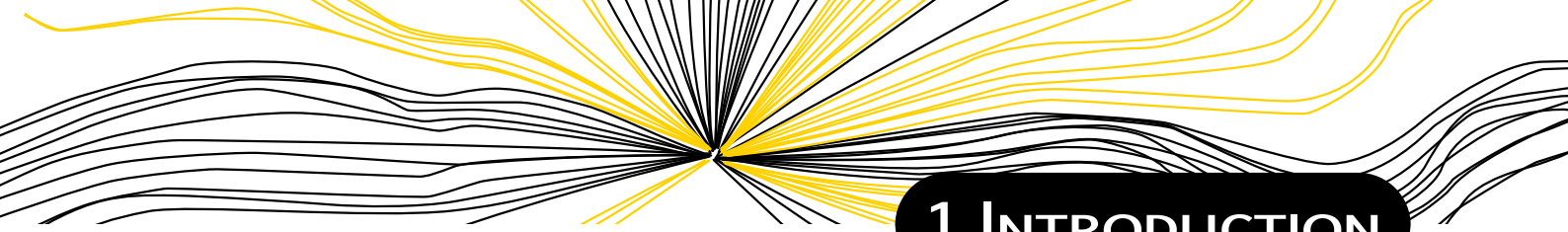
Recently, several programming languages have been proposed to improve modularity by using advanced-dispatching mechanisms. These mechanisms allow programmers to express implicit program behaviour: an advanced-dispatching module can alter the runtime behaviour of program modules, which contain no explicit reference to the advanced-dispatching module. The language elements, that enables programmers to express implicit program behaviour, are compiled away and transformed such that their semantics are expressed in terms of a well-known existing programming language. This transformation creates complex synthetic structures of the language elements, and can even result into the loss of several source abstractions. Consequently, existing debugger tools that target these transformations cannot offer tool support in terms of the original source abstractions, and can become unaware of the presence of source abstractions that were compiled away. This limits existing debuggers because they cannot offer proper support regarding said language elements. This thesis proposes the design and implementation of a trace-based debugging approach, which can capture the activity of language elements of the advanced-dispatching concept, and offers several features to debug such elements accordingly. Because of the trace-based fashion of our approach, it becomes particularly useful in situations where the symptom of a bug is located relatively far away from its actual cause (which can be related to an advanced-dispatching language element), because the program trace that lead to the bug is captured and can be explored. The novelty about this approach is that it offers multiple navigational techniques to locate the activity of advanced-dispatching language elements, and help programmers to comprehend their interplay with the execution of the program.

# Acknowledgements

Almost three years spent at the University of Twente is a long time, and I hope to thank everyone who helped, encouraged and inspired me over these years. First of all, I would like to express my gratitude to my supervisors for all the time they invested in this master project, and assisted me continuously. My sincere thanks to Dr. Ing. Christop Bockisch, who gave me the opportunity to work on an interesting research field, along with offering all the time and freedom that I needed to work out ideas of my own. His knowledge and expertise about the subject matter always lead to fruitful discussions, which were invaluable for the quality of my work and writing. Furthermore, I would like to thank Haihan Yin, MSc. His office was always open for me, and our meetings ever inspired new ideas and suggestions that improved the quality of my work. I would also like to thank Dr. Andreas Wombacher for giving valuable remarks and critique about my writing. My deepest gratitude extends to my family and friends who always supported my efforts and helped me to occasionally relieve my mind from this master project.



# Contents



# 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 [27] and JPred [40]. Modules defined with these languages consist of functionality and furthermore define at what moment this functionality has to be performed during program execution. Advanced-Dispatching (AD) programming languages realize this by manipulating the dispatching of, for example, method calls [5]. The execution of a dispatch can be altered without modifying the source abstraction, thus improving modularity.

This thesis is concerned with the debugging of AD programs. Our preliminary study showed that the runtime behaviour of AD can lead to debugging scenarios that require trace-based debugging technologies to be resolved efficiently [54]. This thesis reveals additional debugging scenarios based on experimentation. Furthermore, the results of a systematic design and implementation for a trace-based debugging approach are presented. This approach provides several techniques to explore AD program traces. These techniques assist in resolving the revealed debugging scenarios, including the ones that were identified by our preliminary study.

## 1.1 Motivation

Yin et al. [59] identified the problems of a discrepancy between the emergent of new programming paradigms and their supportive debugger tools. The debugging support of new programming languages often targets a transformation of the source code. Debuggers therefore present information about the transformed code, whereas programmers ought to be provided with information they are most familiar with. Thus, the new generation programming languages, although promising, are hampered by the lack of proper tool support. This phenomenon is labelled as the next generation of the *debugging scandal*, a term first introduced by Lieberman [34].

The debugging process consists of three main steps: localizing, understanding and fixing a program failure, i.e. any incorrect behaviour with respect to the specification of a software system. Finding the root cause for a program failure appears to be

the most time-consuming step [56]. Marc Eisenstadt found that debugging becomes difficult when dealing with a significant *cause-effect chasm*, i.e. the executed program behaviour between the root cause and the observed symptom of a program failure [12]. 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. This can make conventional debugging techniques inapplicable because they expect the programmer to have suspicious pieces of source code in mind. For example, breakpoint-based debugging requires to specify source-code locations where a program must be suspended during execution. 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. Collecting this runtime information may result in a tedious and time-consuming debugging process when appropriate debugging techniques are inapplicable for the scenario at hand. Research in software debugging proposes omniscient debugging, which is a popular type of a trace-based debugging approach, to improve the localization of the root cause in cause-effect chasm scenarios [32]. The AD concept extends the traditional Object-oriented concept and thus inherits similar debugging scenarios.

Commonly, a well-accepted programming language provides the building blocks of an AD programming language implementation. The well-accepted language is extended with additional elements to express AD behaviour. This might raise the impression that omniscient debuggers that target the extended language are applicable. In theory this is correct. But since these debugger tools are not aware of AD, both a conceptual and structural mismatch arise, which can lead to confusing and insufficient information being displayed by the debugger [61]. For example, they do not allow to examine the runtime context on which a dispatch relied, while this information is essential for understanding the runtime behaviour of AD source abstractions. A structural mismatch arises because of the inability of the extended language to express the AD source abstractions. The source abstractions are therefore transformed to base-language constructs during compilation. The debugger thus targets transformations of the source abstractions. This particular structural mismatch is an example of the next generation debugging scandal.

Another result of this transformation is the possible loss of AD source abstraction locations. For example, the semantics of precedence rules, which define a partial order among AD modules, are often woven into the transformed code. The transformed

code therefore loses presence awareness of these source abstractions. The work of Pothier and Tanter [46] presents an omniscient debugger called TOD which includes support for the AD programming language called AspectJ [27]. Java [2] is used as the extended language of AspectJ. A strategy is provided to present information closer to the source code. Nevertheless, their work targets the transformed code and thus loses the ability of being used for debugging the mentioned source abstractions.

To conclude the motivation for this work, we firstly claim that any element of an AD programming language can potentially be the root cause of a program failure. Any programming language element generates certain runtime behaviour which may conflict with the expected outcome. Thus, tracing the activity of every language element appropriately is crucial to guarantee localization of the root cause. Secondly, we advocate the development of generic tool support for AD programming languages. Several research works propose debugging strategies which target a specific flavour of AD, or even a specific programming language. These strategies are therefore designed to cope with specific peculiarities. This makes these strategies less flexible to be adopted to other programming languages. Designing and implementing debugger tools for the AD concept in a generic way can raise the maturity of existing languages and the new to come.

## 1.2 Research questions

A separate study has been carried out to define the focus and research domain for this work. This study formulated the objectives for the master thesis as research questions. A summarized overview of these research questions is presented here. This gives an impression what has inspired the work that is described in this thesis. More detail about the research questions can be found in [53].

**RQ-1**: *In what ways can a trace-based debugging approach, that is aware of AD, give insight into the runtime behaviour of AD programs?*

**RQ-1.1**: *For which debugging scenarios is a trace-based debugger for AD programming languages useful?*

**RQ-1.2**: *In what ways can a trace-based debugger assist in localizing the program failures of an AD program?*

**RQ-2**: *How can a trace-based debugging approach, that is aware of AD, be efficiently implemented?*

**RQ-1.1**: *What is a suitable recording method to capture the runtime information of an AD program?*

**RQ-1.2**: *What is a generic architecture for the trace-based debugging approach such that several AD programming languages can be supported?*

Our preliminary study proposes parts of the design for a trace-based debugger that is aware of the AD concept, and furthermore proposes the design of a visualization to locate program failures in an AD program trace [54]. This study is therefore focussed on RQ-1. The work of this thesis builds upon this study and implements the fundamental properties of its design proposals, thereby mainly focussing on RQ-2.

## 1.3 Approach overview

This thesis proposes to adopt the trace-based debugging approach to AD in order to improve the debugging process of AD programs. The components and interrelationships of our approach are depicted by Figure 1.1.

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.

The back-end observes the program's execution, captures runtime information, and produces a trace of the target's program execution. The trace-recorder ② instruments the AD program ①. These instrumentations allow the trace-recorder to capture relevant runtime information according to a trace-model ③. 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 ④ produces an execution trace ⑥ that conforms to a storage-model ⑤. A storage-model defines the structure of the execution trace. Commonly, a storage-

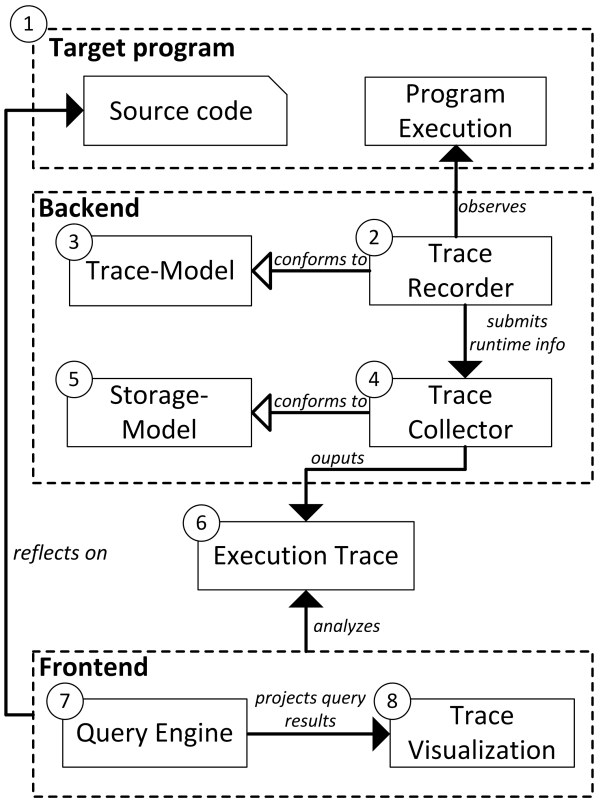



**Figure 1.1:** Relational overview of the approach

model provides a file-based structure that is well suited to be stored on a hard disk, and processable by the front-end.

The front-end provides navigational aids to track down failures in the execution trace. The visualization employs a tree-map design to support sufficient scalability ⑦ [50]. This visualization 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. An off-line indexing strategy is applied to the execution trace. This imposes less runtime-overhead and allows efficient query processing. A query engine ⑦ provides a generic query language that permits programmers to quickly locate specific runtime information in the trace. Several query templates are provided in this thesis. These templates represent common debugging metaphors to release programmers from the burden of mastering a new query language, and indicate the abilities of our query-based debugging approach. The query engine works hand-in-hand with the trace visualization. Query results are reflected on the visualization. Consequently, a query-based debugging approach is combined with the trace visualization to improve navigability.

The techniques that are employed by this work are closely related to those of omniscient debugging. Our approach however cannot be categorized as such, because programmers can express arbitrary queries to explore the trace. Whereas an omniscient debugger supports a limited set of queries to provide basic navigation, such as forward and backward stepping [45]. These basic navigational means can be supported through the query-based debugging approach, thus making omniscient debugging a feature that is also offered by our work.

The back-end of our approach is implemented as an extension to an existing AD framework proposed by Bockisch et al. [6]. The front-end of our approach is implemented by the prototype debugger tool called ALIA-TBD, which is a standalone Java tool. This thesis shows the applicability of ALIA-TBD to several debugging scenarios for which conventional debugging techniques do not work well. In addition, specific and more complex debugging scenarios are presented to expose and evaluate the limits of the approach.

## 1.4 Contributions

At the beginning of this research project, we found that there were opportunities to contribute to the discipline of debugging AD programs, and we believe that this thesis contains such contribution. This work presents the next step in solving the daunting problem of debugging AD programs. This is accomplished by studying the properties of the AD concept, and showing how a trace-based approach can be used to resolve debugging scenarios that can arise from these properties, and which could not be resolved conveniently by existing debugging solutions. The primary contributions of our work are as follows:

- A **trace-model for AD programming languages** based on a distinction between AD and non-AD activities. An AD activity is any program behaviour that supports to realize the AD concept at runtime [60]. The trace-model identifies the required entities of an AD program for which the runtime behaviour needs to be captured, such that the trace is provided with sufficient information to resolve the debugging scenarios of our preliminary study. This identification is mainly based on object state reconstruction and tracing AD activity.

- An **intuitive visualization layout of an AD program trace**. A tree-map based approach is proposed to visualize an AD program trace. The visualization is equipped with several strategies that focus on scalability, and which allow programmers to conveniently locate and explore AD activities in the trace.
- An **identification of the potential of a query-based debugging approach to locate AD specific information in a trace**. This contribution consists of studying and implementing the adoption of a query-based debugging approach to the AD concept. This approach allows programmers to express queries, in a dedicated query language, to locate complex relations within the trace, e.g. tracing the activity of specific AD entities.
- A **storage-model that constructs a tree structure** of the trace. We designed our storage-model in a way that existing query languages, which are designed to traverse tree structures efficiently, can be reused. In addition, the storage-model structures the recorded runtime information such that the navigational philosophies of these query languages correspond to a meaningful way in which program traces often are traversed in software debugging.
- An **implementation of the trace-based debugging approach**. The implemented ALIA-TBD debugger tool realizes the aforementioned contributions. This tool therefore allows to assess our approach on practical debugging scenarios in AD programs. Consequently, this also helps to identify remaining limitations.

## 1.5 Outline

The remainder of this thesis is organized as follows:

Chapter 2 presents the required technical background information. It covers the two most relevant concepts for this thesis, i.e. trace-based debugging and AD. The required terminology is described and illustrative examples are presented.

Chapter 3 presents the back-end of our approach. At first, the most important requirements for the trace-model are identified based on an analysis of the language elements in the AD domain. A diagram of the trace-model, that conforms to these requirements, is portrayed. Furthermore, this chapter presents a storage model that can be processed by the front-end of our approach. The characteristics and

design decisions of the storage-model are hereby discussed. Important implementation details of both the trace and storage model are presented throughout the chapter.

**Chapter 4** presents the front-end of our approach. A visualization layout is constructed based on studying several existing works on trace analysis. This study encompasses an identification of the important requirements to visualize the large information space of a program trace. The techniques that are employed, in order to satisfy these requirements, are elaborately described. Furthermore, this chapter proposes the adoption of a query-based debugging approach to the AD domain. This includes a study of common debugging scenarios, which can arise from the AD concept, and it is shown how these scenarios can be resolved when executing queries over the trace. The implementation details of both the trace visualization and query engine, by ALIA-TBD, are presented throughout the chapter. Most importantly, this chapter describes how ALIA-TBD unites the query-based debugging approach together with the trace visualization.

**Chapter 5** evaluates the proposed approach according to an evaluation plan that was drafted in advance of this work by the author and supervisors of this thesis [53]. The validity of each scientific contribution will be shown by using benchmarking approaches or by demonstrating convincing examples.

**Chapter 6** discusses related work.

**Chapter 7** summarizes and concludes this thesis and presents the remaining questions that can be suggested as future research. Moreover, this chapter relates future work to known limitations of the current implementation of our approach.

# 2 Background

This chapter covers the required background information of this thesis. Firstly, Section 2.1 is describes the debugging process. Relevant terms are introduced and the motivation behind trace-based debugging is explained. Then, Section 2.2 describes the AD concept. We introduce the relevant terminology of AD, since no standardized terminology exists in this field. Furthermore, the complications in debugging AD programs are presented, along with several illustrative examples. The AD system which was extended during this work is presented in Section 2.2.4. The relevant subsystems and strategies are explained in detail. Finally, Section 2.3 presents the background information of the programming language *XQuery*[1]. This section starts with a short introduction about the structure of XML. Furthermore, it describes the parts of the XQuery syntax, and the query mechanisms are explained.

## 2.1 Debugging

Debugging is a task, may it be a subtask, of Software Engineering due to the imperfection of our nature. We make mistakes because of the number of interrelationships and amount of data that is given to us [31]. This phenomenon makes it almost impossible to create a moderately complex program flawlessly, i.e. having no faults. Therefore, debugging techniques are necessary that help programmers fixing the faults [21].

### 2.1.1 Basic terminology

In computer science, the term *"debugging"* is used to refer to the process of diagnosing the precise nature of a known fault and then correcting it [21]. The program that is submitted to the debugging process is called the *target program*. The process of debugging is explained in more detail in the following paragraph. A relational overview of the terms of the debugging process is presented in figure 2.1.

A failure is any incorrect behaviour with respect to the specification of a software system. The actual programming error that causes the failure is called the fault. A

[1] XQuery: An XML query language. See http://www.w3.org/TR/xquery/

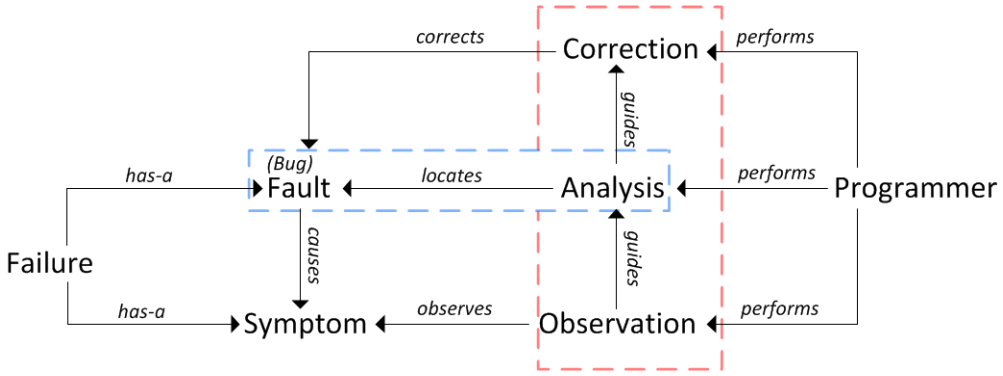



**Figure 2.1:** Relational overview of the debugging process

similar term that refers to the cause of a failure is *"bug"*. The term bug was introduced by Admiral Grace Hopper when a moth got stuck in the Mark 2 Computer and caused a failure. Today, bug is a commonly used term in computer science. The symptom of a failure is noticed by the programmer during observation of the output of the target program, e.g. an unhandled exception, and indicates the presence of a failure. The term *"symptom location"* is used to refer to the source-code location that can be associated with the symptom. The programmer performs an analysis on the symptom and the system behaviour to find the source-code location of the fault. This step is called fault-localization (enclosed by a blue rectangle in Figure 2.1). Finally, the programmer performs an edit at the location of the fault in order to correct the failure. The term "debugging scenario" is used to refer to a particular instance of the relational overview of Figure 2.1. Programmers thus perform three tasks during this process: Observation, analysis and correction of a failure (enclosed by a red rectangle in Figure 2.1). Debuggers and other analytical program tools assist programmers with performing these tasks more efficiently and accurately.

### 2.1.2 Trace-based debugging

#### Motivation

Conventional debugging techniques have several drawbacks. Log-based debugging has three limitations: Firstly, *print* statements are required to be manually inserted into the source code of the target program. This results in source-code pollution. Secondly, It becomes difficult to acquire and locate the parts in the output that are relevant for the observed failure. In practice, programmers often insert either *too* many

or *too* few log statements. Thirdly, the output of log-based debugging does not scale up. It becomes difficult to navigate to parts of interest in a large output, because the inserted print statements often do not create a structured log. This results in a tedious and time-consuming fault-localization process. Breakpoint-based debugging requires programmers to place breakpoints at source-code locations in the target program. In effect, this debugging technique relies on the programmer to have suspicious source-code locations in mind, which is not the case in cause-effect chasm debugging scenarios. A breakpoint-based debugger presents the information that is on the call stack during suspension. This information is often not enough for cause-effect chasm debugging scenarios, because the root cause may be located in a different program state that was entered far before the program suspended [25]. Acquiring missing information, e.g. old object states, requires several program executions and multiple breakpoint locations, which can make the debugging process inefficient.

### Trace-based approach

To overcome the drawbacks of the conventional debugging techniques, software debugging research proposes trace-based debugging approaches. These approaches aim at giving insight into the runtime behaviour of a program by either recording or replaying the program's execution. The main idea of this concept is that: The bug is recorded, thus the recorded trace contains it, and a trace-based debugger can help you navigate to it's cause. Debuggers that use recording create an exhaustive trace of the program execution. This trace contains the runtime information for a specific debugging purpose, e.g. reconstructing any program state of the program's execution. It is important to point out that such an approach does not record every program statement that is executed. Rather, it selects the program statements or language elements that are relevant for the debugging purpose. Replay-based debuggers save the required information of several points in the program's execution. The debugger can re-execute the program from point to point, thereby reconstructing portions of the trace.

A popular flavour of trace-based debugging is an omniscient debugger, also known as a back-in-time debugger, which creates an exhaustive trace. It defines each state change of a program to be part of the trace-model. By doing so, the omniscient debugger can reconstruct any state of the program's execution. After a recording, an omniscient debugger provides several user-friendly interfaces which allows program-

mers to navigate through the program execution history. For example, by means of stepping forward and backward.

### Challenges of trace-based debugging

The programmer tries to find answers to several questions during fault-localization, e.g. *"Where was this variable set?"*, *"Where was that variable assigned a nil value?"*. A trace-based debugger can answer these questions by letting the programmer explore the execution history. Trace-based debuggers that create an exhaustive trace have to deal with massive portions of information, because a program that executes on a common PC can process tens of thousands of instructions in a matter of seconds. This gives rise to two challenges [45]:

- Performance. This challenge is twofold. The first part is inherent to recording a program's execution. Capturing the relevant runtime information, and structurally storing it comes with a performance penalty. A trace-based debugger that creates an exhaustive trace imposes most runtime overhead regarding part one, because a lot of information needs to be captured. A replay-based approach on the other hand imposes most runtime overhead on part two, because the debugger has to re-execute parts of the program. The second part of this challenge is related to the performance of the navigation interface. During navigation, the programmer executes queries on the trace that need to be processed within a reasonable time.
- Storage. A program may execute a massive amount of instructions at runtime. The recording technique of a trace-based debugger has to save the relevant information of each executed instruction. This challenge mostly applies to approaches that create an exhaustive trace, because a lot of runtime information is collected and needs to be saved. A replay-based approach only needs to store information about certain points in the program's execution.

## 2.2 Advanced-Dispatching

### 2.2.1 Basic terminology

The term *dispatching* refers to the mechanism that binds concrete functionality when resolving abstractions at runtime. The classification to "advanced" is made when a programming language allows to express complex conditions and relations for choosing among dispatching alternatives. A classic example of dispatching is *receiver-type polymorphism*, also known as dynamic-binding. It is generally supported by Object-Oriented programming languages. There may be different implementations of a method in the type hierarchy. Receiver-type polymorphism dispatches based on the dynamic type of the receiver object.

The term *dispatch site* refers to the location of the usage site of an abstraction, e.g. a method call or field access. The execution of a dispatch site is referred to as a *joinpoint*, a term generally used by Aspect-Oriented Programming (AOP) [28]. In addition, we borrow the term *adviced joinpoint* to refer to joinpoints for which one or more dispatch function alternatives are available. The term *intercepted joinpoint* is used for joinpoints that are submitted for evaluation of available dispatch functions, hence it becomes an adviced joinpoint if the evaluation is satisfied. A joinpoint is considered a *shared joinpoint* if two or more AD function alternatives are applicable.

The term *base language* is used to refer to the programming language that is extended by an AD programming language. The parts of a program that are expressed in the base-language is referred to as the *base program*.

Several AD programming languages, e.g. AspectJ, compile the AD source abstractions to structures in terms of the base-language. The result of the transformation is commonly called the *intermediate representation* (IR). The IR can be executed directly, or is transformed further into machine-executable form.

### 2.2.2 Characteristics

Raising the level of abstraction helps programmers to implement a system closer to its design. AD programming languages do this by permitting programmers to express functionality alternatives at a dispatch of a program. This is realized by applying

*late-binding*, meaning that the actual functionality that will be executed upon a call depends on the runtime state.

Two popular types of AD are: Predicate dispatching [13] and Pointcut-Advice dispatching [38]. Predicate dispatching allows programmers to attach arbitrary predicates to control the function alternatives of a dispatch. Pointcut-Advice dispatching deals with selecting joinpoints and allowing to modify their behaviour, or attach additional program behaviour that must be executed nearby a joinpoint. A pointcut is a program construct that picks out a set of joinpoints and collects relevant context. Advice defines the program behaviour which executes whenever a pointcut satisfies. The AOP paradigm utilizes this type of dispatching, which is particularly helpful to modularize concerns for which the implementation would be crosscutting when using an Object-Oriented approach. AOP introduces a new module called the *aspect*, which is a container for Pointcut-Advice pairs.

### Illustrative example

The example program in listing 2.1 is written in AspectJ, today's most popular AOP language [27]. AspectJ extends Java with several language constructs to create aspects, e.g. pointcut expressions.

Suppose the concern of event-logging is required to be implemented to log the activity of a hypothetical `EventHandler` class. Without AOP, the implementation of event-logging requires to add additional lines of code to each event-handler body within the class. This would tangle the concern of both event-handling and event-logging in class. Listing 2.1 presents the `EventLogger` aspect. Therein, a pointcut is defined (line 2) which advices the function calls to the `EventHandler`. This example shows, in a simplified manner, how the concern of event-logging is encapsulated and relieved from being managed by modules unrelated to this purpose.

**Listing 2.1:** AspectJ Example of an EventLogger

```
1 public aspect EventLogger {
2   before(call(EventHandler.handle(Event) && args(e))):
3     this.logger.log("EventHandler handling event " + e.toString());
4 }
```

### 2.2.3 Complications in debugging

When using AD properly, the base-program is unaware of the existence of AD source abstractions. This creates an implicit connection at runtime, i.e. runtime modifications are distributed at places in the base-program which contain no explicit reference to these modifications at their respective source-code locations.

runtime modifications are distributed at places without having explicit reference. Programmers comprehend poorly with this implicit connection, as shown by a recent study [15] where 42 out of 104 reported AOP-related faults were due to the lack of awareness of the implicit connection between aspects and other program modules. In effect, a mental reconstruction of the runtime behaviour of a piece of code in the base-program can conflict with the outcome of the actual execution. This decreases the comprehension of AD programs and thus complicates debugging them.

Other complications arise when using a debugger which targets the base-language, instead of the AD programming language in which the source code was expressed. This was already discussed in Chapter 1 shortly, but the cause will now be explained in more detail. Figure 2.2 shows the compilation and execution (right-hand side) combined with the debugging process (left-hand side) when using a traditional Java debugger for AspectJ programs.

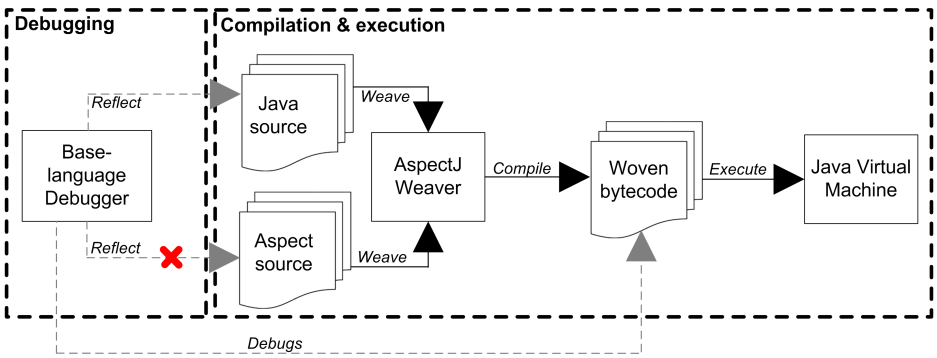



**Figure 2.2:** Structural mismatch after the bytecode instrumentations that are deployed by the AspectJ weaver

The compilation process starts by collecting the source code of the base-program and the aspects. The weaver of AspectJ applies *bytecode weaving*, i.e. it transforms the bytecode of the Java base program in order to enable aspect functionality. The woven bytecode is then submitted to the Java Virtual Machine for execution. A base-language debugger *debugs* the bytecode of the program. During debugging it

presents the runtime information of a corresponding location in the source code. However the references to the aspect source were lost during weaving, because they were compiled and woven into the bytecode of the base-program. Therefore, a base-language debugger cannot refer to the aspect source, but displays information which does not correspond to the source code. An example of this effect is shown by Listings 2.2 and 2.3.

**Listing 2.2:** Example of a base program

```
1 public class Main {
2   public static void main(String[] args){
3     EventHandler eventHandler =
          new EventHandler();
4
5     ResizeEvent resizeEvent =
          new ResizeEvent(20,10);
6
7     eventHandler.handleResizeEvent(resizeEvent);
8   }
9 }
```

**Listing 2.3:** Bytecode of line 7 in listing 2.2 after AspectJ bytecode weaving with aspect in listing 2.1

```
1 invokestatic #42 = Method
      EventVerifier.aspectOf(()LEventVerifier;)
      aload_3
2
3 invokevirtual #46 = Method
      EventVerifier.ajc$before$EventVerifier
      148a5a855((LEvent;)V)
      aload_3
4
5 invokevirtual #25 = Method
      EventHandler.handleResizeEvent
      ((LResizeEvent;)V)
      return
```

Listing 2.2 presents a class that creates an instance of an `EventHandler` (line 3). Furthermore, an instance of `ResizeEvent` is created (line 5) and passed to the `handleResizeEvent` method of the `EventHandler` (line 7). Listing 2.3 shows the bytecode of line 7 after performing bytecode weaving together with the `EventVerifier` aspect in Listing 2.1. Lines 3 and 4 are instructions of method invocation that are inserted by the bytecode weaver. These instructions are not written in the source code. The programmer will have to mentally map the presented information by the debugger to the corresponding AspectJ source abstraction.

### 2.2.4 ALIA

Bockisch et al. [7] revealed that AD is the foundation of many new programming paradigms, e.g. predicate dispatching, pointcut-advice dispatching, domain-specific language design. They propose the *Advanced-dispatching Language-Implementation*

*Architecture* (ALIA) for implementing languages with AD concepts. This architecture offers a meta-model called LIAM which can act as an intermediate representation (IR) for AD source abstractions. The design of LIAM supports several AD paradigms, thus improving the potential re-use of different language implementations. ALIA4J [6] implements the approach for the Java programming language.

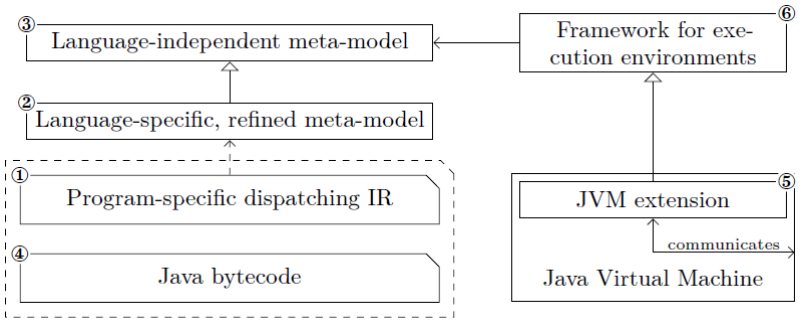



**Figure 2.3:** Overview of compilation and execution flow of ALIA4J

Figure 2.3 presents the flow of compilation and execution of an AD program that is based on ALIA4J. The AD source abstractions are compiled to the IR ① based on a refined Java implementation ② of LIAM ③. The base-program is compiled to the IR of the base-language, in this case Java bytecode ④. After compilation, both the LIAM instances and the base-program are submitted to a JVM including a FIAL-based extension ⑥. This extension can be implemented as a Java agent for example.

The two most important components of ALIA are: The Language-Independent Advanced-dispatching Meta-model (LIAM) and Framework for Implementing Advanced-dispatching Languages (FIAL). Both are described in sections 2.2.4 and 2.2.4 respectively. Part of this thesis is carried out an on execution environment for ALIA, called the Non-Optimizing Interpreter-based Reference Implementation (NOIRIn). Therefore, section 2.2.4 is dedicated to describe its relevant details.

### LIAM

LIAM acts as the IR of an AD programming language that is implemented with ALIA4J. The meta-model defines the entities of the AD concept. The AD source abstractions can be translated to a language specific extension of LIAM, e.g. a Java implementation where each LIAM entity is represented by a Java class. The AD language constructs are then represented by instances of LIAM entities, e.g. an *AtomicPredicate* Java class. Figure 2.4 shows the entities of LIAM and their respective relations.

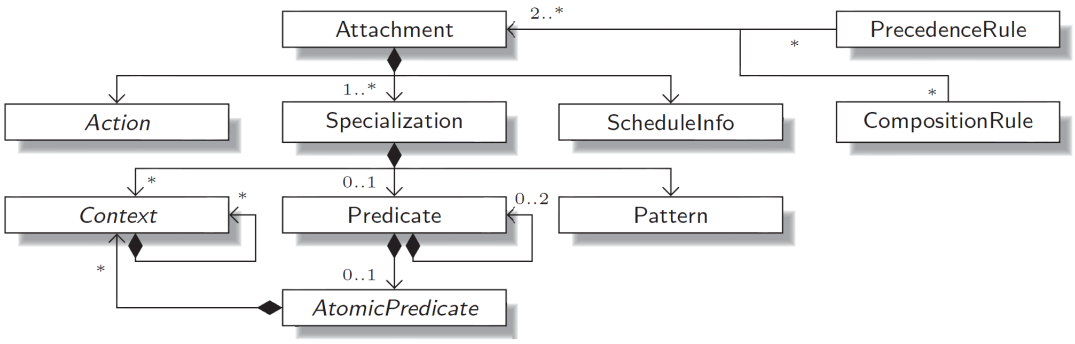



**Figure 2.4:** LIAM entities

An *Attachment* is the root entity. It functions as a container of all entities for a dispatch description. In AOP, an attachment refers to a pointcut-advice pair. A *Specialization* consists of entities that dictate which joinpoints must be intercepted. A pointcut expression in AOP is an example of a specialization. The *Pattern* entity allows to define lexical conditions, e.g. a method name, to target a set of joinpoints statically. 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 *Context* entity can be used to make runtime context of the base-program available to a *Predicate*. The *Action* entity represents the functionality which is invoked whenever it's correlating specialization is satisfied. This entity is similar to an advice in terms of AOP. The *ScheduleInfo* specifies when the *Action* should be executed relative to the adviced joinpoint. ALIA supports three types of *ScheduleInfo*: before, after and around. The latter can be used to manipulate both the control and data-flow of the joinpoint.

LIAM provides additional entities to control the dispatching behaviour at shared joinpoints. These entities define relational rules among attachments. A *PrecedenceRule* declares a partial order of *Action* execution. A *CompositionRule* declares constraints on *Actions* being executed together.

### FIAL

The *Framework for Implementing Advanced-dispatching Languages* (FIAL) provides several common components and workflows required for developing execution environments based on LIAM. It supports code generation and an interpreter-based approach. Most importantly, FIAL offers a generic service for deriving an execution

model based on LIAM entities per dispatch site. The execution model acts as the meta-object protocol of AD.

At runtime, FIAL derives a dispatch function for each intercepted joinpoint of the program. The dispatch function is a Boolean function which evaluates the specializations of all deployed attachments on the intercepted joinpoint. As such, the dispatch function identifies which *Actions* must be executed, depending on the evaluation outcome of their correlated Specializations. This evaluation is performed efficiently by constructing an Ordered Binary-Decision Diagram (OBDD) representation of the Boolean function. Each inner node is an *AtomicPredicate* and the leaf nodes represent ordered sets of actions. The leaf node that is reached after traversing the OBDD represents the set of actions that must be executed for the intercepted joinpoint. After the traversal, the set of actions is modified according to the applicable relational rules. Finally, the set of actions, called the execution model, can be invoked by the execution environment. More details about the dispatch-function can be found in [49].

### NOIRIn

The Non-Optimizing Interpreter-based Reference Implementation (NOIRIn) is an execution environment based on the meta-object protocol of FIAL. NOIRIn does not apply code generation but interprets the execution model produced by FIAL at runtime. It is build for ALIA4J and thus targets the Java programming language. NOIRIn is implemented as a Java agent extension to the JVM. Figure 2.5 depicts the workflow of NOIRIn.

The base-program source-code ① and a LIAM-based IR of the AD source abstractions ② are submitted for compilation. By using bytecode manipulation, every dispatch site of the base-program is transformed to a statically-bound call to a site method ③. This approach separates the dispatch logic from the base-program. At runtime, the attachments are deployed by FIAL ④. Then, NOIRIn intercepts the execution of each dispatch site thanks to the earlier inserted callback ⑤. The site method receives the actual dispatch represented by a first-class object, including relevant context values. This first-class object is called *CallContext*. Based on the *CallContext*, a dispatch function is derived. NOIRIn's interpretation requires that LIAM entities implement a "compute" method to evaluate the dispatch function. This method receives context values as arguments and returns the evaluation result as a Boolean value. The OBDD is traversed by interpreting the compute method of the AtomicPredicates that

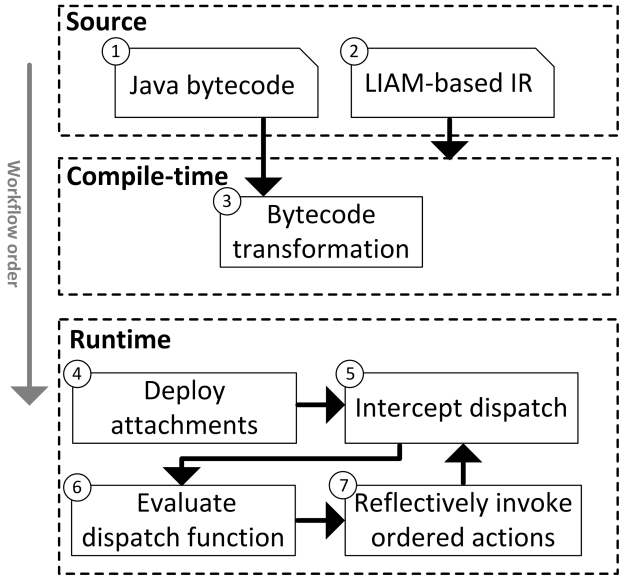



**Figure 2.5:** The workflow of NOIRIn

it encounters. This creates a set of applicable actions ⑥. Schedule information and relational rule constraints are solved against this set to determine their execution order. NOIRIn interprets the dispatching function by reflectively invoking each applicable action ⑦. Finally, NOIRIn awaits the next dispatch by going back to ⑤.

## 2.3 XQuery

XQuery is a programming language which provides a declarative syntax to traverse an XML document. We start this section with a brief description of an XML document, thereby introducing relevant terminology. Then, a subset of XQuery, called *XPath*[2] , is described. XPath provides the query expressions that are most used by our work. Finally, several other features that are offered by XQuery are briefly mentioned.

### 2.3.1 XML documents

Trees are the underlying structure of XML documents. Each node in the tree has a *kind* property, which represents the type of the node. The following nodes are relevant in our context:

[2]Xpath: An XML path language. See http://www.w3.org/TR/xpath/

- The entire XML document is referred to as the **document-node**. This node, though not visible, contains an arbitrary number of child elements, but only one root element.
- **Elements** represent the inner nodes of the tree structure. They are defined by a *start* and *end* tag. Tags are represented by angle brackets, e.g. `<element>..</element>`, and represent the scope of an element. Elements have a *name*, a unique *parent* (document-node or another element), an arbitrary number of *child* elements, and *attributes*.
- **Attributes** are part of an element, and consists of a *name* and a *value*. They are defined within the start tag of an element, e.g. `<element attribute="value"/>`
- **Texts** are leaf nodes of the tree. They consist of textual content which is enclosed by a *start* and *end* tag, e.g. `<leaf>content</leaf>`.

An example document is shown in Listing 2.4. The element *a* is the root element of the document, containing three child elements: *b*, *e* and *h*. Each of these elements further has two child elements, which contain textual content. A pictorial representation of its corresponding tree structure is shown in Figure 2.6.

**Listing 2.4:** Example of an XML document.

```
1  <a>
2    <b>
3      <c>C</c>
4      <d>D</d>
5    </b>
6    <e>
7      <f>F</f>
8      <g>G</g>
9    </e>
10   <h>
11     <i>I</i>
12     <j>J</j>
13   </h>
14 </a>
```

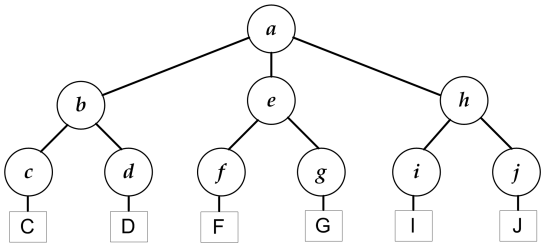



**Figure 2.6:** The tree structure of the XML document of Listing 2.4. Elements nodes are represented by a circle; textual content of an element is represented by a rectangle.

### 2.3.2 XPath

XPath is W3C standardized query language to traverse XML documents. The most relevant part of XPath is related to the compact, and yet powerful way, in which nodes can be selected from the tree structure. This is accomplished by offering *path expressions*, which define a logical traversal through the tree structure. A path expression is composed of *steps*, which are separated by forward slashes. Each step is evaluated according to a node set. The result of this evaluation consists of a refined node set, which is then passed to the following step in the path expression. If a node set consists of multiple items, the step is evaluated once for each node, and the results are concatenated. The result of the path expression is the node set which is produced by it's last step. A step consists of three elements, each element is more elaborately described by the rest of this section.

1. An *axis* specifies a relationship, regarding the tree structure, between the input and output node set. For example, the relationship between a parent and child element.
2. A *node test* specifies the type or name of a node.
3. A composition of *predicates*, which are Boolean expressions, to filter the generated node set before it is passed to the following step in the path expression.

XPath defines thirteen axis. These axis refer to the hierarchical relationships between nodes in a tree structure, and define in which way a step needs to traverse the tree. Figure 2.7 shows a pictorial representation of the most relevant axis, in our context, applied to the example XML document of Listing 2.4, where *e* (lines 6 - 9) is the initial node. The axis can be further categorized into *forward* (e.g. following) and *backward* (e.g. preceding), depending on the direction in which the tree is traversed. The steps of path expressions can be separated by a single forward slash, or a double forward slash. The first refers to the *child* axis, the latter refers to the *descendant* axis, which selects all nodes that are encompassed by the initial node (e.g. children, grandchildren, etc).

Node tests are expressed by adding two colons after the axis specification. While the axis is traversed, the node tests filter the result set based on the node type, e.g. *text* nodes, or the node name. The latter can be used to filter element or attribute names, where attribute names must be prefixed with a **@**.

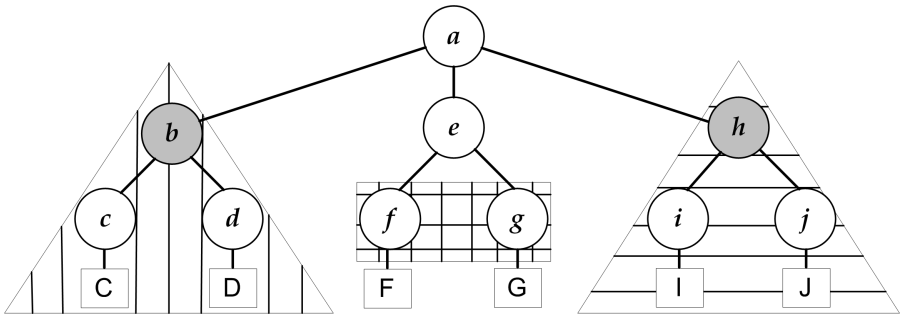



**Figure 2.7:** Several XPath axes projected on the tree structure of the XML document of Listing 2.4, where element *e* is the initial node. Nodes that are surrounded by the raster pattern refer to the **child** axis, the vertical striped pattern refers to the **preceding** axis, and the horizontal striped pattern refers to the **following** axis. The gray coloured nodes are the siblings of *e*.

Predicates can be added to a step with brackets [ . . . ]. XPath offers comparison operators, e.g. <, >, =, !=, that can be used in a predicate expression. Predicates can be composed by the Boolean operators *and* and *or*. Nodes for which the predicate evaluation yields *true* are added to the result set of the step. If the predicate refers to an integer value *n*, then only the node with index *n*, of the result set, is passed to the next step.

An example of an XPath path expression is shown in Listing 2.5. In this example, the document-node is referred to as *doc*. From the document-node, a path expression, which follows the child-axis, traverses the document till node *b*. Then, a predicate evaluates the textual content of the child element *c* against the value "C". The result of this predicate yields true because *b* has a child element called *c* that contains the textual content "C". Hence, the result of the path expression is node *b*.

**Listing 2.5:** Example of an XPath path expression, which applies on the XML document of Listing 2.4.

```
1 doc/a/b/[c="C"]
```

### 2.3.3 FLOWR expressions

XQuery extends XPath by offering additional features, the most important being FLOWR expressions. FLOWR is the equivalent of SQL's SELECT, and is named after its five clauses: *for*, *let*, *order by*,*where* and *return*. Because every XPath expression is a valid XQuery, we can use path expressions and bind their result to a variable,

which is defined by *let*. An example of an XQuery using the FLOWR expression is shown in Listing 2.6. First, a path expression uses a node test to select the child nodes of the root node (i.e. *a*) in the XML document of Listing 2.4 (line 1). The result of this path expression is bound to a variable (line 1). A sequence of nodes is created by the *for* clause on line 2, whereby each iteration refers to a node in the sequence, which is bound to the variable `child`. The *where* clause, on line 3, filters the sequence by a Boolean expression, in this case it uses the same predicate expression as the XPath example of Listing 2.5. Thus, the node *b* is the only node that passes the *where* clause. Finally, the *return* clause defines what should be added to the result set for a matching node. In this example, a path expression that leads to the textual content of the element *c* is returned.

**Listing 2.6:** Example of an XQuery FLOWR expression, which applies on the XML document of Listing 2.4.

```
1 let $children_of_a := doc/a/child::node()
2 for $child in $children_of_a
3 where $child[c="C"]
4 return $child/c/text()
```

# 3 Recording program executions

This chapter discusses the design and implementation for recording the trace of an AD program execution. Section 3.1 identifies the general requirements for trace capturing of AD programs. Based on this identification, a fine-grained trace model is proposed. This granularity makes the trace-model independent of the implementation of various storage-models. Section 3.2 proposes a suitable storage-model to support the navigation towards low-level details in a step-wise strategy. Several advantages of this storage-model are discussed in this section, and which become evident in Chapter 4. Finally, Section 3.3 discusses the implementation of the fine-grained trace-model. The implementation is realized as an extension to the NOIRIn execution environment of ALIA4J.

## 3.1 Trace-model

### 3.1.1 Custom terminology

Several trace-based debugging approaches, e.g. TOD [48] and ODB [32] use the term *event* to refer to the execution of a program instruction that is relevant for trace-capturing. Unlike these approaches, we use the term *joinpoint*, since it can be used to refer to *events*, and it is a well-known term in the research community of AD and AOP.

Programming languages often provide several types of method calls. Examples are: *virtual*, *special*, *static*. The type depends on the resolution strategy in which the call is performed. We use the term *function* for unification. Thus, this term refers to the call to a subroutine in any resolution strategy.

### 3.1.2 General requirements

Common questions, which programmers have during debugging, refer to the past while inspecting the current state of a program [32]. Examples are: *What was the last assignment to variable x?*, *What was the state of object y before being passed to method z?*. Trace-capturing must therefore include sufficient information to support the

following features offline: object state reconstruction for a given moment in time, and causality-link tracing [45]. 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 either be acquired by a computed data reconstruction, or by a mental observation of the programmer.

Because an AD program extends a traditional base-program, the aforementioned defined requirements apply in this context as well. Moreover, to trace the implicit behaviour of AD source abstractions raises additional questions. These questions are related to: the bidirectional relation between AD entities and base-program, the runtime behaviour of AD entities, and in what ways AD entities can modify the runtime behaviour of the base-program. Examples are: *Which joinpoints were adviced during program execution?*, *Was joinpoint* `y` *adviced? If so, by whom?*, *What was the evaluation result of predicate* `x`*?*, *Was the value of variable* `y` *modified by an AD entity?*. To answer these questions, the trace must include information that is specific to AD. Hence, we argue that the set of requirements for trace-based debugging is augmented because of the AD concept. An AD program trace must therefore include sufficient information to:

1. Reconstruct the bidirectional relation between adviced joinpoints and AD activity. This relation can help, for example, to raise the awareness of implicit invocations.
2. Trace the activity of AD entities according to their type. Each AD entity has characteristics that must be reflected in the information of the trace. For example, when tracing a *Predicate* it is important to include the evaluation result. Furthermore, the trace must provide information which reflects the defined interrelationships by *Attachments*, e.g. *Predicates* that are part of the same *Specialization*.
3. Track the control and data-flow modifications to the base-program, as a result of AD activity.

Trace-based debugging approaches may target specific runtime behaviour of a program, e.g. its control-flow. The trace-model of various trace-based debuggers can therefore differ. An example of such variation is an omniscient debugger that supports scoped tracing, i.e. the trace-model can be customized to a subset. Programmers

have to specify, before runtime, what can be omitted from the trace-model. A typical example usage is when programmers want to analyse only the functional flow of a program execution. The activity of variables can then be omitted from the trace-model. Note that scoped tracing can also be applied in a lexical way, i.e. by letting programmers specify, before runtime, a lexical environment which encapsulates only the program modules that are of interest for the debugging task at hand. This kind of scoped tracing can be particularly helpful when one "trusts" the runtime behaviour of, for example, a third party library and wishes to omit its runtime activity from the trace. This kind of scoped tracing is not supported by the current implementation of our trace-model, and scheduled for future work, see Section 7.2.3.

We cannot accurately predict to what ends a trace-based debugging approach for AD may be used in the future. Supporting scoped tracing is therefore important. This requires a flexible design of the trace-model, i.e. any subset of entities in the trace-model can be selected without violating existing constraints in the design. This flexibility can be met by constructing the trace-model out of modular entities, i.e. each entity encapsulates the relevant information on its own. In effect, scoped tracing can be supported without the lack of relevant information that would otherwise be located in missing entities.

Though we cannot guarantee that no information is missing, we define the information that is required to satisfy the aforementioned requirements as an *exhaustive trace*. The design of the trace-model is based on capturing an exhaustive trace. Optionally, a subset of the trace-model can be specified to capture a scoped trace.

### 3.1.3 Fine-grained trace-model for AD

Figure 3.1 presents a simplified version of the trace-model for AD programming languages as an UML class diagram. The design philosophy of the trace-model is based on a distinction between the activity of the base-program and AD activity. This enables the trace-recorder to separate joinpoints, that can be associated with the base-program, from AD activity. The AD related joinpoint types are named according to their respective LIAM entities, e.g. `ActionCallJoinPoint`. What makes the trace-model fine-grained is that it defines a joinpoint type for each element of an AD programming language. *ActionCallJoinPoints* and *PredicateCallJoinPoints* act no differently than a *FunctionCallJoinPoint*. However, by unifying them one would lose the ability to distinguish between these joinpoint types. In effect, a trace-based

debugging approach, which aims at identifying the activity of specific AD language elements, would refrain from adopting such a trace-model.

The required flexibility is supported by defining a modular entity for each joinpoint type, thus there exist no dependencies between the entities in the model. As a consequence, a joinpoint type entity can contain attributes for runtime information that would otherwise be located in a different joinpoint type entity. For example, the *LocalVariableWriteJoinPoint* entity, which corresponds to the assignment of a local variable in the program, contains runtime information about the function it belongs to. This information is also located in the corresponding *FunctionCallJoinPoint* entity, however this entity may not be captured by a scoped trace-model. In such a case, the runtime information of the function to which a *LocalVariableWriteJoinPoint* belongs would not be part of the trace that is eventually created, thus loosing relevant information of a *LocalVariableWriteJoinPoint*.

We assume that most components of the trace-model make sense to the reader. In fact, the trace-model parts for capturing the base-program joinpoints closely resembles that of existing omniscient debuggers, e.g. TOD [48] and Unstuck [24]. The rest of this section describes the structure of the base-program, and AD, related parts of the trace-model. Lastly, we explain how the trace-model is refined at a lower-level to improve its modularity.

#### General entities

Every joinpoint type inherits from the abstract class *JoinPoint* (top left entity in Figure 3.1). This class contains the general fields which represent the general information of a joinpoint, e.g. the source-code location of the dispatch site. Each subclass further refines the `JoinPoint` class by declaring fields that represent the information specific to its type.

The *JoinPoint* class further contains several attributes that can optionally be set by the AD system that implements the trace-model. These attributes are added to conform to the requirements of the trace-model. For example, if the joinpoint was executed during an invocation of an *Action*, the *executingActionId* can be set to refer to the *Action*. This attribute enables a trace-based debugger to identify which joinpoints, regardless of the type, were part of an *Action* execution.

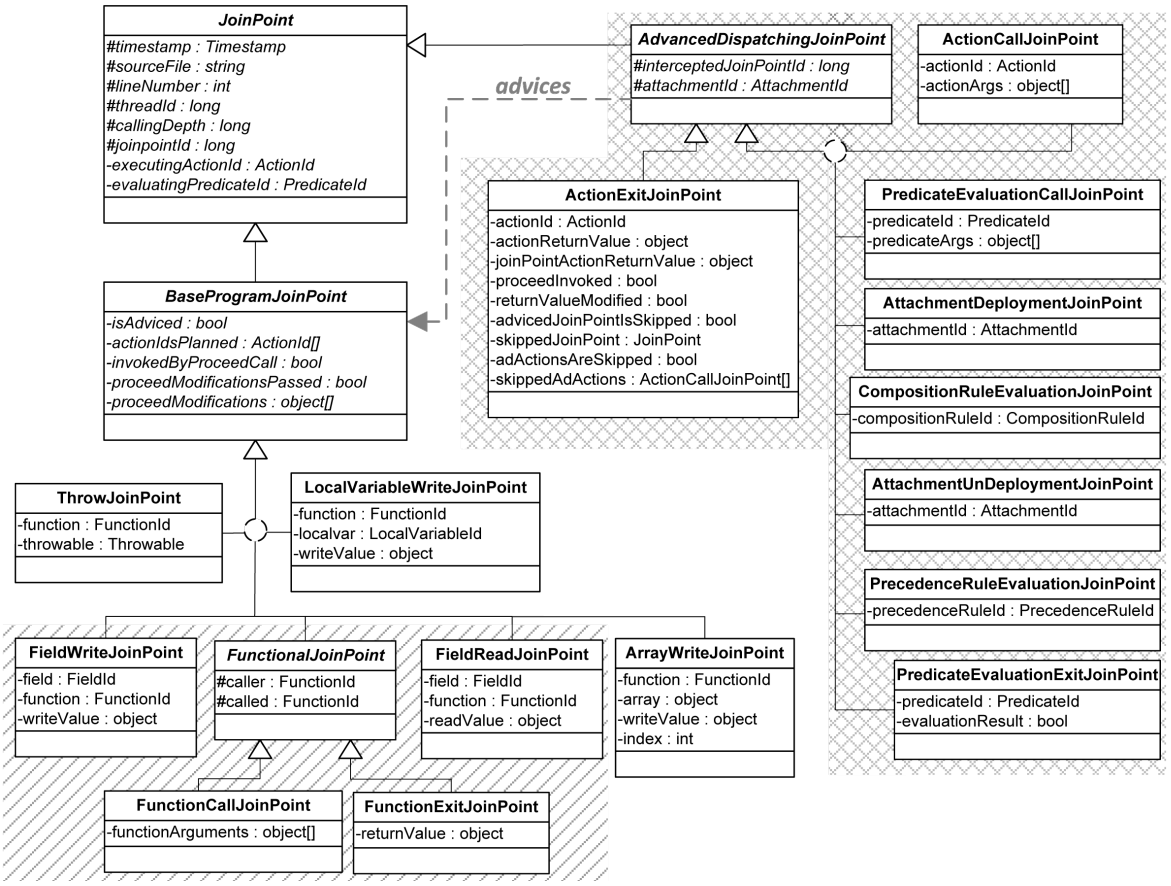



**Figure 3.1:** A simplified UML class diagram of the trace-model for AD programming languages. The joinpoint-model of NOIRIn is enclosed by a striped pattern. The AD-related joinpoints are enclosed by a X-filled pattern.

An example of an entity, that is related to base-program entity, is the *FunctionalJoinPoint*. This entity defines the general attributes for a joinpoint that changes the control-flow. It consists of a representation for the caller and the called (also known as callee). A subclass of this abstract entity can add additional attributes that are relevant for the call or exit of a function. For example, the *FunctionCallJoinPoint* contains the arguments that were passed.

The reader might already have noticed that the inclusion of the *FieldReadJoinPoint* is superfluous, because this joinpoint does not induce a state-change. The reason for its inclusion is inherent to the implementation phase of this trace-model, which is discussed in Section 3.3.1. The AD system, that is extended by this implementation, includes *FieldReadJoinPoint* to its joinpoint model. This allows to attach AD behaviour when a field is read. Though the AD behaviour is recorded in such case, the joinpoint that was adviced would not be part of the trace. Consequently, the bidirectional

relation between the intercepted joinpoint and the corresponding AD-activity cannot be reconstructed. We therefore argue that the trace-model must be a superset of the target AD system's joinpoint model.

### AD entities

Joinpoint types that refer to AD activity inherit from the abstract class *AdvancedDispatchingJoinPoint*. This class contains several general fields for recording the activity of AD source abstractions. For example, a reference to the joinpoint that it intercepted. Note that the *Attachment(Un)DeploymentJoinPoint* does not intercept a *BaseProgramJoinPoint*, this reference therefore remains empty. The `Action` and `Predicate` entity may enclose a subroutine, e.g. an advice body in AspectJ. Separate joinpoint types for their respective call and exit are therefore defined.

The trace-model defines joinpoint types for every AD language element that performs dynamic behaviour. This explains the absence for the LIAM entities: `ScheduleInfo` and `Pattern`. The behaviour of these entities does not change at runtime, hence they do not induce a state change of the program. Besides, several existing debuggers can show how these entities will manifest at runtime by performing a static analysis of the program, e.g. the work of Pfeiffer and Gurd [43].

Note that *PrecedenceRules* do not perform dynamic behaviour, and their activity during program execution could be reconstructed offline by performing a static analysis on the source code. This static analysis consist of, for example, comparing the attachment composition at shared joinpoints to existing precedence rules in the source code, e.g. looking for applicable *declare precedence* statements in AspectJ. However, such a static analysis would need to be implemented for a specific AD programming language syntax, thereby loosing the generic properties of the trace-model. We therefore trace the activity of precedence rules during program execution, such that the trace-model is independent of the syntax in which they are expressed in the source code.

An example of an AD joinpoint is the *ActionExitJoinPoint*. This joinpoint represents the exit of an *Action* invocation to an AD source abstraction. It contains attributes for each modification that an *Action* can impose to the adviced joinpoint. For example, an *Action* can perform a data-flow modification by returning a value other than the adviced joinpoint would have returned. This information is reflected by the

*ActionExitJoinPoint* by providing the return value of both the adviced joinpoint and the *Action* itself. These attributes enable a trace-based debugger to notify the programmer about modifications to the execution of the base-program, as a result of AD activity.

### Low-level modularization

The trace-model makes use of an *identifier-model*, which provides classes to refer to the language elements of an AD program. This helps the trace-model to become more fine-grained and improves the potential for reuse. Figure 3.2 presents a simplified version of the identifier-model as a UML class diagram. Important to note is that the LIAM IR already structures the AD language elements, thus allowing easy lookup and reference. This makes it unnecessary to include AD language elements in the identifier-model.

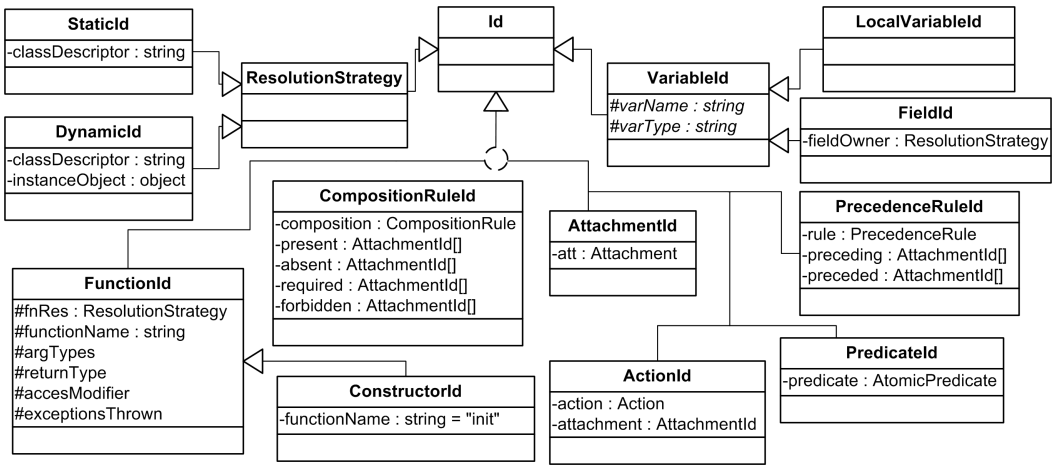



**Figure 3.2:** A simplified UML class diagram of the identifier-model.

An example of an entity, which corresponds to a base-language element, is the *FunctionId*. This entity can uniquely refer to a function in a certain resolution strategy, i.e. static or dynamic. For example, if object `x` and object `y` are both instances of the same class, then `x.m()` and `y.m()` are represented by two different *FunctionIds*. This entity furthermore contains the information about the function signature.

One might argue that the information of the identifier-model could well be integrated with the trace-model. This, however, would destroy the potential for reference. This can be clarified with an example. Suppose a trace-recorder keeps the currently calling function of a thread in memory until it exits. The identifier-model allows to construct a *FunctionId* for the calling function. This object can be kept in memory.

Once the function exits, the trace-recorder can easily create a *FunctionExitJoinPoint* by referring to the previously created *FunctionId*.

## 3.2 Storage-model

This section describes a storage-model that produces a *call tree* representation of a program execution. A call tree is a tree graph which is rooted, ordered and directed. Each joinpoint type that is associated with the call to a subroutine is rendered as an inner node. Other joinpoint types are rendered as the leaf nodes, which are connected 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 3.3 shows a simplified call tree representation of the execution of the program in Listing 3.1. 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. For example, the execution path: `"main(args)/init()/n()/o()"` led to the execution of the joinpoint node ⑩.

Figure 3.4 depicts the storage-model as a simplified UML class diagram. The entities, which represent the call to a subroutine, inherit from the abstract entity *CallJoinPoint*[1]. This entity composes the joinpoints that are part of its execution. Respectively, the entities that inherit from the abstract class *ExitJoinPoint* represent the exit of a subroutine. Thus, the execution trace is represented as a call tree, where *CallJoinPoints* are the inner nodes.

The advantage of a call tree representation of a trace is the abstraction from low-level detail that it permits, because it groups the runtime information that belongs to a particular call. This can avoid the situation where programmers get overwhelmed with a lot of information during debugging. Well-known navigation metaphors such as *step-over* and *step-return* are supported by this storage-model, because it preserves the

[1] We use *CallJoinPoint* and *call-joinpoint* interchangeably.

**Listing 3.1:** An example Java program.

```
1  package baseprogram;
2  public class Main{
3    private static int f;
4    public Main(){
5      m();
6      n();
7    }
8
9    public int m(){
10     int a = 1;
11     int b = 2;
12     return a + b;
13   }
14
15   public void n(){
16     int d = 4;
17     o();
18   }
19
20   public void o(){
21     int e = 5;
22     f = 6;
23   }
24
25   public static void main(String args[]){
26     new Main();
27   }
28 }
```

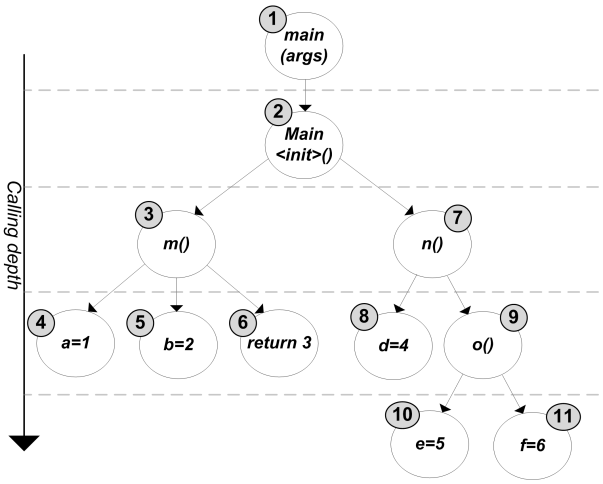



**Figure 3.3:** Call tree representation of the execution of the program in listing 3.1.

relation between inner and child nodes. Furthermore, a call tree nicely structures the trace to answer debugging questions that refer to the functional decomposition of the system that is being debugged, e.g. *"what happened during this function call?"*. A more detailed explanation of the application of a call tree representation, in trace-based debugging, is given in the next chapter.

## 3.3 Extending NOIRIn to support trace-capturing

The implementation of the trace and storage model was carried out as an extension to the ALIA4J framework. The design philosophy of this framework is to provide a generic framework for AD languages. This philosophy fits to that of the trace and storage model that were described in the previous sections.

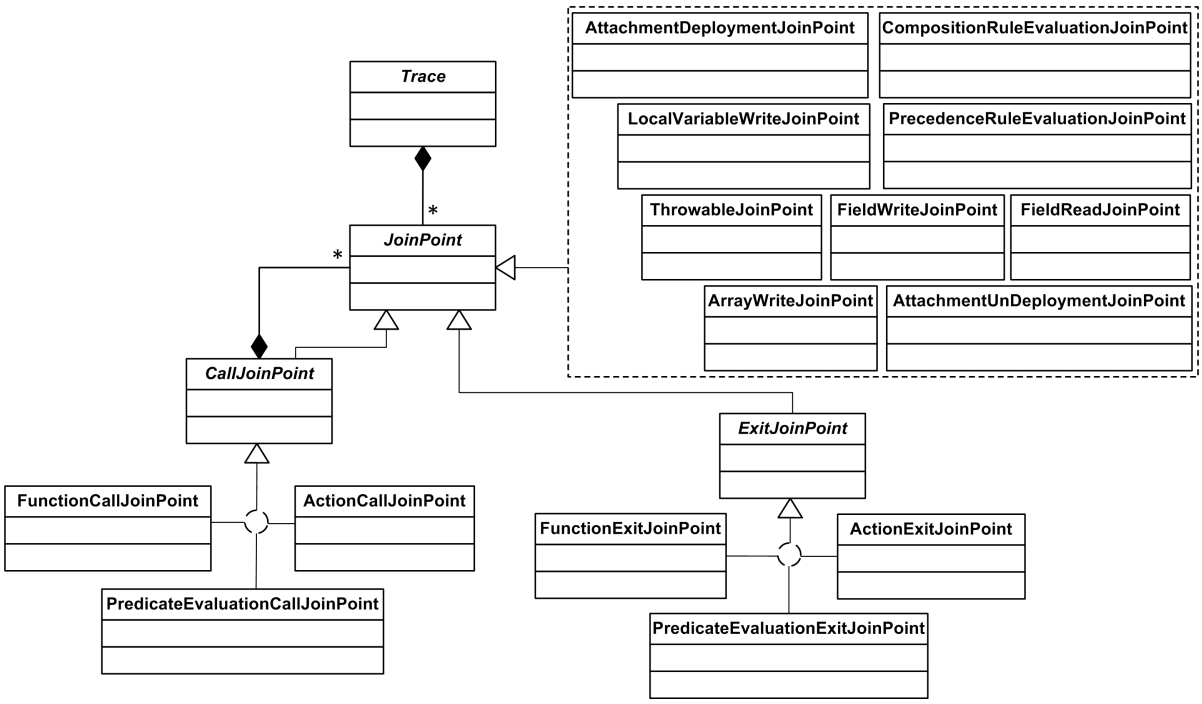



**Figure 3.4:** A simplified UML class diagram of the storage-model.

The trace and storage model are implemented as extensions to the NOIRin execution environment that is offered by ALIA4J. NOIRIn handles all dispatch logic by interpretation. It therefore provides all desired AD-related information at runtime. This makes NOIRIn a suitable execution environment for developing debugging support. Several works have shown that NOIRIn can be extended to implement debugging support that gives insight into the dispatching [4] [61]. The rest of this section describes the step-by-step implementation process that was carried out on NOIRIn.

### 3.3.1 Implementing the trace-model

A mixture of two approaches is applied to collect all the runtime information that is defined by the trace-model: bytecode instrumentation [32] and interpreter-instrumentation [33]. The first is used to capture base-program joinpoints. The latter is used to capture AD-related joinpoints.

Figure 3.1 shows the joinpoint types that are subject to NOIRIn's dispatch logic: *FunctionCallJoinPoint, FunctionExitJoinPoint, FieldReadJoinPoint and FieldWriteJoinPoint*. In NOIRIn, all AD-related information is available. This consequently allows to capture the AD joinpoint types of the trace-model. The remaining joinpoint types of

the trace-model are: *ThrowJoinPoint*, *ArrayWriteJoinPoint and LocalVariableWriteJoinPoint*. We abbreviate these three joinpoint types to *TAL* for simplicity.

### Tracing TAL by using bytecode instrumentation

NOIRIn applies bytecode transformation to intercept joinpoints. TAL is captured by extending this bytecode transformation strategy. To give an impression of this extension, we now explain the bytecode instrumentations that are necessary to capture *LocalVariableWriteJoinPoints*. NOIRIn uses the ASM[2] bytecode manipulation library, hence the same library is used for this implementation. Several bytecode opcodes indicate a local variable write, e.g. `ISTORE`. With ASM, one can traverse through a class file with the provided ASM nodes, e.g. `MethodInsnNode`. While traversing, additional bytecode instructions can be inserted before, for example, a local variable write instruction. This allows to capture both information about the local variable, the method in which the local variable is declared, and the value that is newly assigned.

A simplified example of the bytecode instrumentation to capture two *LocalVariableWriteJoinPoints* is shown in Listing 3.2. The associated Java source code of Listing 3.2, without bytecode instrumentations, is shown in Listing 3.3. A value, which is on top of the stack, is duplicated before it is stored into a local variable. This duplication is then boxed to unify local variable writes among primitive and object types. By using the registry index of an local variable write instruction, additional information about the variable can be obtained by consulting ASM. This information can pushed onto the stack, for simplification this is shown in dots in Listing 3.2. Finally, a statically-bound call is inserted which refers to a function that further performs the trace-capturing. This call receives the required runtime information as function arguments, and constructs a *LocalVariableWriteJoinPoint* of the trace-model accordingly.

**Listing 3.2:** Simplified example of capturing the local variable writes of listing 3.3 by using bytecode instrumentation. Lines 2-5 and 8-11 represent the bytecode instructions that are inserted to enable trace capturing.

```
1 iconst_2 //Load an integer constant onto the stack
2 DUP  //Duplicate the integer constant that is about to be written to a variable
3 invokestatic #5; //Box the integer value by invoking: java/lang/Integer.valueOf(I)Ljava/lang/Integer;
4 ...  //Push variable information and function information onto the stack
5 invokestatic #6 //Invoke: traceCapturing.localVarWrite(Ljava/lang/Object;....)V
```

[2]Homepage of the ASM bytecode manipulation library. See http://www.asm.ow2.org/

```
istore_1 //Store the integer value into local variable with register index 1
lconst_2 //Load a long constant onto the stack
DUP2  //Long values occupy two words on the stack, duplicate both
invokestatic #7; //Box the long value by invoking: java/lang/Long.valueOf(J)Ljava/lang/Long;
...  //Push variable information and function information onto the stack
invokestatic #6 //Invoke: traceCapturing.localVarWrite(Ljava/lang/Object;....)V
lstore_2 //Store the long value into local variable with register index 2
```

**Listing 3.3:** The source code of two local variable writes.

```
int i = 2;
long l = 2;
```

A separate setting can be passed to NOIRIn to enable trace-capturing. Only if this setting is passed will the extension to the original bytecode transformation of NOIRIn be applied. Thus, if the trace-capturing extension is not used, no runtime overhead is imposed on NOIRIn.

### Overview of the high-level extensions to NOIRIn

Figure 3.5 depicts the extended workflow of NOIRIn when trace-capturing is enabled. Please refer to Section 2.2.4 for a recap of steps one through seven. At compile-time, the bytecode transformation of NOIRIn is extended to capture TAL. Statically-bound calls are inserted before the execution of TAL, which collects the relevant runtime information (8). These calls refer to a function in NOIRIn different than the one for interpretation (9). This separates the tracing of TAL from NOIRIn's interpretation process. Once TAL executes, the trace-capturing module is called to create the corresponding joinpoint type entity, and in turn submit this entity to the coupled storage model (10). Note that the trace-capturing module generates and manages several attributes of the trace-model without bytecode instrumentation. For example, the trace-capturing module manages a sequential counter, which is updated when a joinpoint is encountered, to assign unique values to the *joinPointId* attribute.

NOIRIn's interpretation process is instrumented to trace the joinpoint types other than TAL. When NOIRIn intercepts a dispatch (5), an additional callback is performed. The trace-capturing module hereby receives the context of the joinpoint (11) in a first-class representation provided by NOIRIn. From this representation, the trace-capturing module can create the corresponding joinpoint type entity. At this point, the trace-capturing module reserves a unique identifier (ID) for the intercepted joinpoint.

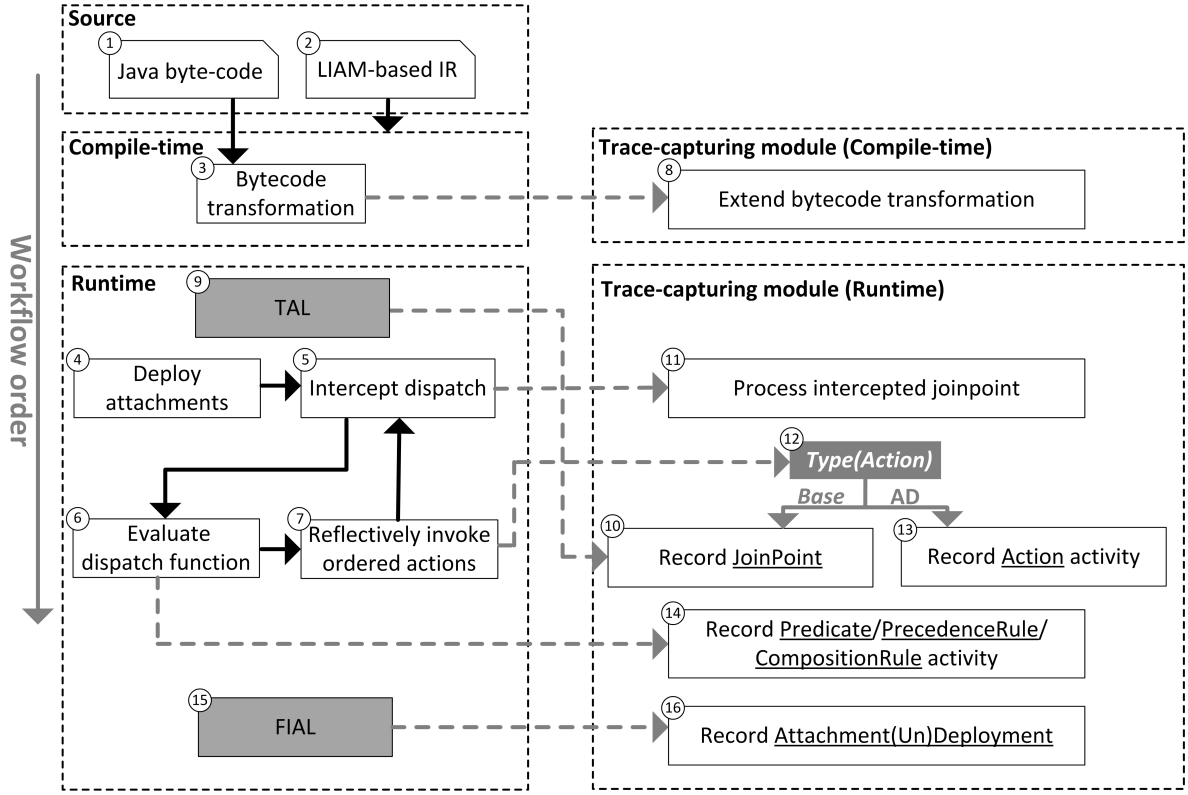



**Figure 3.5:** The extended workflow of NOIRIn to support trace-capturing according to the trace-model of Figure 3.1.

This ID is available during the interpretation process of the joinpoint, and allows to relate possible AD activity with the joinpoint that is adviced. This realizes half of the bidirectional relation between adviced joinpoints and their corresponding AD activity. The other half is realized by maintaining, per base-program joinpoint, a set of *ActionIds* that adviced the base-program joinpoint.

The BDD traversal of the dispatch function (6) is instrumented with additional callbacks. This enables to capture: *PredicateEvaluationCallJoinPoints*, *PredicateEvaluationExitJoinPoints*, and the program behaviour that was executed during its evaluation (14). The evaluation of relational rules is performed after the BDD traversal. This evaluation is also instrumented by an extra callback to capture its activity (14).

An inserted callback is performed right before NOIRIn reflectively invokes an action of the derived execution model (7). During this callback, the trace-capturing module checks if the type of the invoked action corresponds to activity of the base-program or an AD Action (12). If the invocation refers to base-program activity, the reserved ID for the adviced joinpoint is consumed and its trace-model representa-

tion, which was constructed at (11), is recorded by the storage-model. Otherwise an *ActionCallJoinPoint* or *ActionExitJoinPoint* is registered (13).

The implementation of NOIRIn leaves the task of controlling the (un)deployment of attachments at runtime upto FIAL. For this reason, FIAL is instrumented with additional callbacks to capture *Attachment(Un)DeploymentJoinPoints*.

### 3.3.2 Implementing the storage-model

Trace-analysis and query-based debugging are examples of well studied trace-based debugging approaches. Their use and potential for the AD domain is, to our knowledge, not yet studied exhaustively. One of the primary aims of our implementation is interoperability, i.e. allowing 3rd party tools to process the trace representation. Taking interoperability into account allows future research, that also target a trace-based debugging approach for AD programs, to reuse this implementation as a back-end.

This section presents one way to implement the storage-model. The strategy employs the eXtensible Markup Language (XML) [1], because it can nicely reflect the nested structure of a call tree, that was proposed in Section 3.2. XML is also becoming a more and more commonly used standard for storing large data instances [20], aims at a widespread deployment, which makes interoperability an important design factor, and it is becoming a well-accepted format to exchange data between multiple parties [1]. XML tends to describe information in a verbose way, which has its impact on the scalability of this implementation. More detail about this impact is shown Chapter 5. When aiming for a practical solution, a custom database technique suits better, as shown by Pothier [44]. The downside of such a solution is the poor interoperability, because 3rd party tools may have to implement custom drivers to cope with the provided trace database.

For simplicity, the implementation is referred to as the *ADT2XML* (AD trace to XML writer). ADT2XML performs the following set of steps. Before the target application starts, a new XML document file is created on the HDD. At runtime, every joinpoint, that is part of the trace-model, is captured and serialized to an XML element. This serialization creates an XML element, named `JoinPoint`, which contains child elements for every field of the joinpoint's type. An extra child element is added which reflects the type of the joinpoint. The fields of the *JoinPoint* entity of the trace-model are serialized as XML attributes. These fields can be considered as the metadata of

each concrete joinpoint type, because they represent information that a programmer, who reads the XML data, is not interested in at first glance [22]. For example, the field *JoinPoint.joinPointId* is most likely not of interest to the programmer when a particular joinpoint is inspected. Only in special debugging circumstances will the programmer need to know the value of *JoinPoint.joinPointId*. The XML structure of each joinpoint serialization hereby follows the design principle of Harold: *"data goes in elements, metadata in attributes"* [22]. Consequently, the joinpoint element headers throughout the trace have a consistent structure. Every joinpoint element is, after its creation, appended to the XML document. Lastly, when the target application shuts down, the XML document file is closed.

A simplified DTD of the output that ADT2XML generates is shown in listing 3.4. The aim of this simplified version of the DTD is to give the reader an impression of the trace document structure. Dots are therefore positioned at places to omit unnecessary detail. The trace contains a root element called `Trace` (line 1), which consists of arbitrary number of `JoinPoint` elements. A `JoinPoint` element (line 3) contains a child element that defines its `Type`, and contains XML elements to reflect the fields of the corresponding entity in the trace-model, e.g. a *FieldWriteJoinPoint* contains a `FieldId` element. Attributes are defined for each field of the abstract *JoinPoint* entity of the trace-model (lines 4-9), and serve as the metadata of a `JoinPoint` element. A *call-joinpoint*, e.g. *FunctionCallJoinPoint*, contains an extra element that is not defined in the trace-model. This element is called *SubJoinPoints* (line 11), and serves as the parent for the joinpoint elements that are executed during the *call-joinpoint*. This subsequently creates the call tree representation in XML.

**Listing 3.4:** Simplified DTD which depicts the output XML structure of ADT2XML.

```
1  <!ELEMENT Trace ( JoinPoint* ) >
2
3  <!ELEMENT JoinPoint ( Type & ( CallerId | CalledId | LocalVariableId | FieldId . . . . | SubJoinPoints
       )* ) >
4  <!ATTLIST JoinPoint timeStamp CDATA #REQUIRED >
5  <!ATTLIST JoinPoint sourceFile CDATA #REQUIRED >
6  <!ATTLIST JoinPoint lineNumber CDATA #REQUIRED >
7  <!ATTLIST JoinPoint threadId CDATA #REQUIRED >
8  <!ATTLIST JoinPoint callingDepth CDATA #REQUIRED >
9  <!ATTLIST JoinPoint joinPointId CDATA #REQUIRED >
10
11 <!ELEMENT SubJoinPoints ( JoinPoint* ) >
12 ....
```

13 ..

ADT2XML serializes primitive data by value. Objects are serialized by an unique identifier, called *OID*. For example, the *FieldWriteJoinPoint.writeValue* attribute is serialized as an OID. The OID allows to trace the activity of specific objects throughout the trace. For example, the state of an object o can be reconstructed by querying for *FieldWriteJoinPoints* that match the constraint: *field.owner == OID(o)*. ADT2XML implements the creation of OIDs by utilizing a *WeakIdentityHashMap* that consists of `<Object, OID>` pairs. Every object that ADT2XML detects is looked up in the hashmap to obtain the OID. If the object does not exist within the hashmap, a new OID is generated and stored in the hashmap.

Implementing the call tree representation of the storage-model requires to write out joinpoint XML serializations without a closing tag. For example, the serialization of a *FunctionCallJoinPoint* ends with an open XML element, i.e. its *SubJoinPoints* element. This open element acts as the parent of all recorded joinpoints during the function's execution. The open element is closed when its corresponding *FunctionExitJoinPoint* executes. This strategy realizes the required nested structure of the trace.

Unfortunately, existing XML generation libraries for Java often do not provide this kind of flexibility. Their implementation does not allow to add incomplete XML elements to an existing document, because they aim at preserving well-formedness. Thus, using such a library requires to maintain a *FunctionCallJoinPoint* XML element in memory until the *FunctionExitJoinPoint* executes. This may consume a significant amount of memory and performance when the execution of a function produces a lot of joinpoints that need to be recorded.

ADT2XML therefore generates XML elements by using a custom and primitive implementation for XML generation. Unlike existing XML generation library, ADT2XML refrains from a complex object model that enforces the constraints of the XML structure, and uses only primitive Java language elements to create XML serializations, e.g. String concatenations. It supports a limited set of XML construction features. No checks on the document structure are carried out for performance reasons. This implementation provides the required flexibility and allows to append joinpoint XML serializations to the trace file at the moment they execute. Thus, nothing has to be maintained in memory.

Existing Java XML libraries often construct complex inter-object relations to maintain the integrity of an XML document. Furthermore, several checks are performed before new elements are added to an existing document. These checks ensure proper XML integrity after the adjustments. ADT2XML omits such checks because the structure of each element in the XML document is known in advance. Thus, integrity of the trace XML document is provided by a valid implementation of these structures. Because ADT2XML refrains from integrity checks, its XML serialization outperforms that of existing XML Java libraries. This was verified by implementing the storage-model with *dom4j*[3], and measuring the execution time needed to record a certain number of joinpoints in a controlled environment (The same environment was used to benchmark the back-end, see Section 5.1 for more detail).

The execution time to record 100K *LocalVariableWriteJoinPoints* with the dom4j storage-model was roughly 37 times larger than measuring the same number of *LocalVariableWriteJoinPoints* with ADT2XML. Furthermore, dom4j restricts to write out XML elements without a closing tag. In effect, the entire XML element of inner nodes in the call tree, e.g. a *FunctionCallJoinPoint*, has to be preserved and updated in-memory until the execution of such an inner node is completed, e.g. the corresponding *FunctionExitJoinPoint*. Therefore, the memory consumption and performance of the dom4j storage-model becomes relative to the calling depth of the program. On the other hand, ADT2XML allows to write out XML elements with an open tag, thus it can write out an inner node, e.g. *FunctionCallJoinPoint*, directly and no in-memory structure has to be preserved.

An important implementation detail of ADT2XML is the proper dealing with an unexpected termination of the target program. ADT2XML's flexible XML generation causes the XML document to be in a continuous ill-formed state at runtime, because it contains an open element without a closing tag during the execution of a function. An unexpected termination of the program can therefore result into an ill-formed XML file. To deal with this scenario properly, ADT2XML keeps track of the XML elements that are currently opened, and checks if they are all closed when the program terminates. If not, a corresponding closing tag is added for each opened XML element, such that the integrity of the trace XML document is preserved.

A limitation of the current implementation of ADT2XML is that multi-threaded programs are unsupported. This due to the strategy in which the call tree representa-

[3]dom4j an open source library for working with XML, see http://dom4j.sourceforge.net/.

tion is created: a new joinpoint serialization is always appended to the most recently opened *SubJoinPoint* element in the XML file. Thus, new joinpoint serializations cannot be added to the *SubJoinPoint* element of another call-joinpoint in the XML file, while this is required in a multi-threaded situation, because per thread a different call-joinpoint is under execution at a particular moment in time. This limitation is further discussed in Chapter 7, wherein an extension to ADT2XML is proposed to support multi-threading.

# 4 Visualizing and exploring traces

Having a program trace, that conforms to the trace model of the previous chapter, gives the opportunity to experiment with the adoption of existing trace-based debugging techniques for the AD domain. This chapter aims to adopt two trace-based debugging approaches and shows their potential when combined. Firstly, Section 4.1 presents a trace-analysis strategy to construct an interactive visualization of an AD program trace. We start with summarizing several important fundamentals for a visualization of large information spaces. These fundamentals are then mapped to the context of an AD program trace. Based on these fundamentals a layout, called tree-map, is used to visualize an AD program trace. Secondly, Section 4.2 adopts a query-based debugging approach to the AD domain. Several queries that are specific to the behaviour of AD are identified. The queries that are introduced by this section are XQuery expressions. Section 4.3 presents a standalone debugging tool called ALIA-TBD. This tool implements the aforementioned debugging strategies and shows how they can cooperate to improve the navigability in large program traces. Implementation details and design choices are also discussed. The debugging strategies that are proposed in this chapter are designed for XML traces that can be produced by the back-end of our work, which was described in the previous chapter.

## 4.1 Trace visualization

The design principles of the visualization, which are introduced in this section, aim for a standalone debugging strategy. The reason for this is that we do not want to force programmers to (possibly) learn a new query language before they can make use of our debugging tools. The visualization therefore implements features to support a convenient and efficient way to explore the trace. The primary focus of the visualization lies on improving the comprehension of an AD program, and localizing parts of interest in the trace.

### 4.1.1 Fundamental properties

#### Providing navigability

A program that executes on a common PC can produce tens of thousands joinpoints in a couple of seconds. Programmers may get overwhelmed and even loose track because of the huge amount of information that is given to them. A visualization can assist in locating parts of interest in large information spaces. An important challenge that needs to be addressed, when designing such a visualization, is navigability. Navigability refers to the ability of allowing users to locate parts of interest in the trace in an efficient and convenient way. An additional design fundamental that is inherent to navigability is scalability, i.e. the ability to create an instructive representation of the trace that can be comprehended by programmers.

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 [16] identified that multi-staging is an important property that must be supported by a visualization which targets a large information space. Multi-staging states that a limited part of the information space is visible to users at once. Each stage can therefore be considered as a subset of the information space, whose size is limited by the available display space and the ability of users to grasp a certain amount of information. To support multi-staging for a trace visualization, we can exploit the hierarchical structure of the call-tree representation by following a top-down comprehension strategy. The first stage presents a summarized overview of the program trace, where the execution detail of high-level calls is initially pruned. Programmers can select a call and request further detail if desired. The visualization then removes unrelated information such that the entire display space can be occupied with information about the call of interest. The user is then presented with information one level deeper down the call hierarchy. From there, the same strategy can be applied to provide the user with an iterative way of navigating towards low-level detail in the call hierarchy. Foltz further identified three main requirements to create an effective, and navigable visualization of a large information space. Each requirement can be mapped to the context of a program trace visualization, which is described next. Note that the underlined number of each requirement is used for convenient reference in the rest of this chapter.

**"1. Information in the space should be placed according to an organizing principle, and this principle should be communicated explicitly to users."**
Two key aspects are important for the visualization to accustom an organizing principle. Firstly, the joinpoints in the trace are ordered by time of execution, this ordering must be reflected by the layout of the visualization. If not, the visualization makes it unclear when joinpoints were executed compared to one another, while this is useful information in trace navigation. Secondly, the layout should reflect the functional decomposition of the system implementation in an intuitive way. Every call in the call tree encompasses a certain portion of runtime behaviour, i.e. the number of joinpoints that were part of its execution. Based on this portion, one can identify the importance of the call for trace navigation, as shown by Bohnet et al. [8]. The visualization can use this property as a key aspect to further organize the layout.

**"2. Users should be able to make a correct navigation decision to take the right next step, even if the eventual destination is imprecisely known."**
In the aforementioned strategy to provide multi-staging, every stage presents a certain amount of nodes, of the call tree, in the available display space. These nodes represent multiple joinpoints of the trace. Based on the joinpoint type, a short textual message can summarize the activity of the joinpoint. This summary should reflect information that makes sense to users, and provide sufficient information to let users decide if the joinpoint is relevant for the debugging task. For example, when the visualization encounters a *FieldWriteJoinPoint*, the name of the field can be considered an important attribute, because this is defined by the developer in the source code and therefore instantly gives an impression to which field the joinpoint refers.

**"3. Orientation should be recoverable at every point in the space. Users should never feel lost."**
System features are often implemented by a hierarchical functional decomposition. This decomposition helps programmers to implement high level system behaviour by reusing lower level system behaviour, thereby enabling the reuse of system functionality. This hierarchical way of implementing system behaviour can be reflected by a multi-stage trace visualization to meet the requirement. Each stage must present sufficient information that corresponds to the functional decomposition, and can serve as an indicator of the location in the trace. This information should make sense to programmers assuming that she is familiar with the functional decomposition of the system implementation.

#### Raising AD awareness

Other important design fundamentals are 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. A graphical notation of AD program behaviour, trace-based or not, must reflect this implicitness. Additionally, implicit runtime behaviour must not be exposed at an *on-demand* basis, because programmers are often unaware of its presence, as shown by a study of Ferrari [15]. Rather, pro-active notifications must be used such that programmers recognize certain runtime behaviour as a result of AD.

Two fundamentals must be included in the trace visualization to deal with the implicitness of AD. Firstly, AD activity needs to be visualized in a way such that programmers can instantly become aware of its presence. Thus, without having to manually look or ask if the currently displayed information belongs to AD activity. This can be accomplished by, for example, labeling joinpoints that are related to AD activity. Secondly, the visualization must reconstruct the bidirectional relation between AD and the base-program. When the user inspects AD activity, the adviced joinpoint must somehow be highlighted, and vice versa.

### 4.1.2 Navigable and AD aware visualization design

Tree-maps are 2-d space-filling visualizations of hierarchical data sets [50]. Each node in the data set is represented by a rectangle. A tree-map visualizes 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. What makes tree-maps more intuitive than graph visualizations is their ability to include an additional, often contextual, attribute in its drawn process. The value of this attribute is reflected by assigning a size, which is proportional to the attribute value, to the rectangular representation of a node. This property of a tree-map visualization makes it easy for users to eyeball relevant nodes. The rest of this section explains how a tree-map layout can be adopted for trace visualization, and implement the fundamental properties of the previous section. We use the previously assigned numbers of each fundamental property to clarify which feature implements it.

Several examples of the trace visualization are presented throughout this section. The trace information, which is visualized by these examples, is based on the execution

of an AspectJ program. The base-program is shown in Listing 4.1, which extends the example program that was introduced in the previous chapter (Section 3.1). The extensions are explicitly marked in the listing, and represent isolated loops, i.e. the control and data-flow of the original program are not altered. The loops produce joinpoints during execution, e.g. a local variable write to `i, j or k` with every iteration, thereby increasing the size of the trace. This is necessary because the navigational aids, which are provided by the trace visualization, can best be demonstrated with a relatively large program trace.

The AspectJ source code is shown in Listing 4.2, wherein an *aspect* is defined. This aspect contains two advices: an around advice (line 7) and a before advice (line 12). Both advices use the same pointcut definition (line 5), the adviced joinpoints of this AspectJ program are therefore shared. The pointcut definition picks out the field write to `f` (line 24) in the base-program of Listing 4.1.

Though this work is based on ALIA, we have chosen for AspectJ in this particular example for convenience reasons. AspectJ is a well-accepted programming language and allows to express AD behaviour in a clear syntax. Besides, AspectJ adopts pointcut-advice dispatching, which can be implemented by ALIA, and is therefore also supported by our work.

#### 1. Tree-map layout as organizing principle

A tree-map can visualize the trace by providing a rectangular representation of each joinpoint. The size of this rectangle is based on the amount of runtime behaviour that the joinpoint encompasses. This information can be extracted from the call-tree representation of the trace. Joinpoint types, that do not define functional behaviour, e.g. *FieldWriteJoinPoint*, are therefore represented by rectangles with a static size. The rectangles of call-joinpoints on the other hand have a flexible size, which depends on the number of joinpoints that they encompassed during execution. This way of organizing the layout reflects the functional decomposition of the system, because the rectangles of high-level calls surround those of low-level detail.

An example of the organizing principle is shown in Figure 4.1. The functional decomposition of the example program is well reflected by the trace visualization. The programmer can conveniently eyeball the high-level calls of the example program, i.e. `Main.init()`, `m()` and `n()`, because of their rectangular size. Information about

**Listing 4.1:** An extended version of the Java program of Listing 3.1. The extensions are marked by a gray color.

```
1  package baseprogram;
2  public class Main{
3    private static int f;
4    public Main(){
5      m();
6      n();
7    }
8
9    public int m(){
10     int a = 1;
11     int b = 2;
12     for(int i=0;i<150;i++){}
13     return a + b;
14   }
15
16   public void n(){
17     int d = 4;
18     for(int j=0;j<200;j++){}
19     o();
20   }
21
22   public void o(){
23     int e = 5;
24     f = 6;
25     for(int k=0;k<50;k++){}
26   }
27
28   public static void main(String args[]){
29     new Main();
30   }
31 }
```

**Listing 4.2:** An example AspectJ aspect which is applied to the base program of listing 4.1.

```
1  package aspects;
2  import baseprogram.Main;
3  public aspect Azpect {
4
5      pointcut pc() : set(static int Main.f);
6
7      around() : pc(){
8          System.out.println("around");
9          proceed();
10     }
11
12     before(): pc() {
13         System.out.println("before");
14     }
15 }
```

the textual messages that are printed inside joinpoint each tree-map rectangle can be found in the next section.

The execution order of the trace is preserved by using the *Split* algorithm [3] to construct the tree-map layout. The order is preserved in a two-dimensional fashion. The top-left of the visualization 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 4.1 which depicts the order.

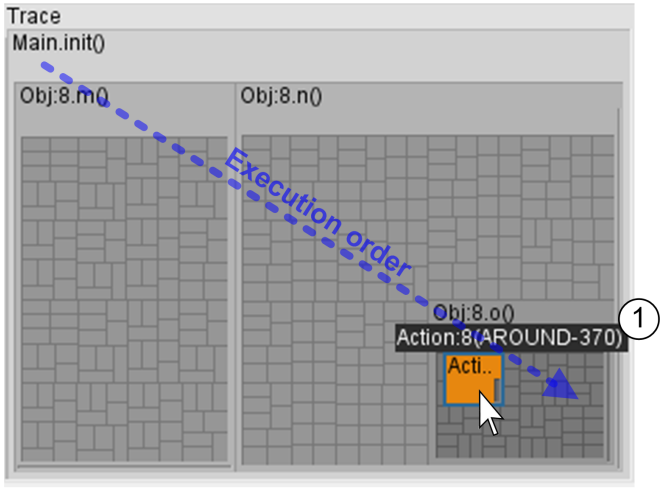



**Figure 4.1:** Initial stage, presenting a condensed overview of the trace. A tooltip of a contextual message is shown at (1). A selected joinpoint rectangle is marked with an orange color. The diagonal arrow depicts the execution order of the trace.

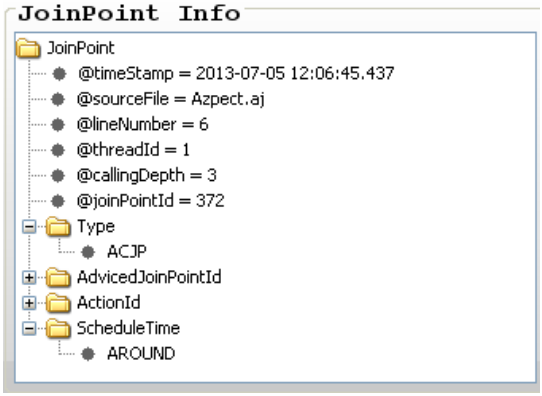



**Figure 4.2:** Joinpoint-info view. Presents the detailed information, in a tree structure, of the selected joinpoint in 4.1.

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 color. In Figure 4.1, the rectangles that are surrounded by the function call to `o()` have a darker gray color than those surrounded by the function call to `n()`. This further improves the comprehension of the visualization because it helps users, just by looking, to distinguish low-level information from high-level information.

### 2. Efficient navigational decisions based on exposed joinpoint details

Showing plain rectangles does not help the programmer to localize joinpoints of interest. The tree-map is therefore extended with context-awareness, i.e. it "knows" what kind of joinpoint it is drawing. This context-awareness consists of short textual messages which summarizes 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.

Because the size of every joinpoint rectangle is limited, certain abbreviations are used to save display space. The textual messages use the following abbreviations to refer to common program operations:

- '&' refers to reading the value of a variable, e.g. *FieldReadJoinPoint*.
- '=' refers to assigning a value to a variable, e.g. *FieldWriteJoinPoint*.

Table 4.1 presents the format of the contextual message for each joinpoint type. An acronym is defined for each joinpoint type. Since these acronyms are introduced by this table, the remainder of this thesis occasionally uses them for convenience. Examples of contextual messages are also included in the table. Important to note is that primitive types are displayed by value, whereas objects by a textual representation of their respective OID.

These contextual messages, shortened as they are, may still exceed the available display space of a joinpoint rectangle. A tooltip of the contextual message is therefore constructed when hovering over a joinpoint rectangle with the mouse. The tooltip is displayed above the joinpoint rectangle. This is also shown in Figure 4.1, marked by a ①.

To examine all the details of a joinpoint, an additional view, called the *joinpoint-info view*, is positioned next to the tree-map visualization. An example of this view is shown in Figure 4.2. This view is triggered when the user selects a joinpoint in the tree-map visualization. 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.

### 3. Location orientation and trace navigation through interactive multi-staging

The tree-map visualization employs an interactive strategy to provide multi-staging. Initially, an overview of the trace is visualized, 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 visualize 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

**Table 4.1:** Format of contextual message per joinpoint type.

| Joinpoint type | Acronym | Contextual message format | Example |
|---|---|---|---|
| *LocalVariableWriteJoinPoint* | LVWJP | <varName>=<writeValue> (varType) | var=10 (int) |
| *FieldWriteJoinPoint* | FWJP | <fieldOwner>.<varName>=<writeValue> (<varType>) | Obj:1.aField=10 (int) |
| *FieldReadJoinPoint* | FRJP | &<fieldOwner.<varName>/<readValue> (<varType>) | &Class1.aStaticField/10 (int) |
| *FunctionCallJoinPoint* | FCJP | <context>.<functionName>() | Class1.aStaticFunction() |
| *FunctionExitJoinPoint* | FEJP | <context>.<functionName>()/<returnValue> (<returnType>) | Obj:1.aFunction()/10 (int) |
| *ThrowJoinPoint* | THJP | Throw <throwableOID>() | Throw Obj:1 |
| *ArrayWriteJoinPoint* | AWJP | <array>[<index>]=<writeValue> | Obj:1[0]=10 |
| *ActionCallJoinPoint* | ACJP | Action:<actionOID>(<scheduleInfo>-<advicedJoinPointId>) | Action:1(BEFORE-1234) |
| *ActionExitJoinPoint* | AEJP | Action:<actionOID>(<scheduleInfo>- <advicedJoinPointId >) / <actionReturnValue>(<returnType>) | Action:1(AROUND-1234)/10 (int) |
| *PredicateEvaluationCallJoinPoint* | PCJP | Pred:<predicateOID>(<interceptedJoinPointId>) | Pred:1(1234) |
| *PredicateEvaluationExitJoinPoint* | PEJP | Pred:<predicateOID>(<interceptedJoinPointId>)/<evalResult> | Pred:1(1234)/true |
| *PrecedenceRuleEvaluationJoinPoint* | PRJP | Prec:<precedenceRuleOID>(<advicedJoinPointId>) | Prec:1(1234) |
| *CompositionRuleEvaluationJoinPoint* | CRJP | Comp:<compositionRuleOID>(<advicedJoinPointId>) | Comp:1(1234) |
| *AttachmentDeploymentJoinPoint* | ADJP | AttDepl:<attachmentOID> | AttDepl:1 |
| *AttachmentUndeploymentJoinPoint* | AUJP | AttUnDepl:<attachmentOID> | AttUnDepl:1 |

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 4.3. This figure presents the trace visualization when the user has double-clicked the selected joinpoint rectangle of Figure 4.1. The selected joinpoint rectangle represents an *ActionCallJoinPoint* to the around-advice of Listing 4.2. This can be concluded from the contextual message in Figure 4.1, marked by a ①, and helps the programmer to deduce the location in the trace. Unrelated joinpoint rectangles are removed from the visualization 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 in the next section.

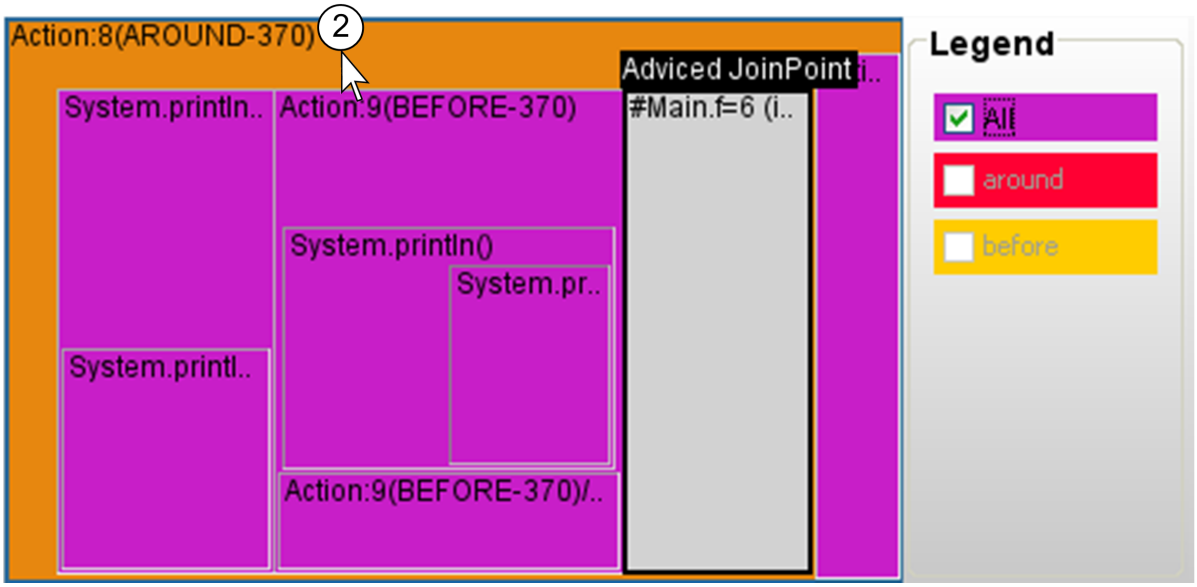



**Figure 4.3:** Detailed stage. The selected joinpoint rectangle of Figure 4.1 is now used as the root of the visualization ②.

### 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 color, whereas base-program joinpoints are shown in gray. An example of this colouring scheme is shown in Figure 4.3.

A single-colouring scheme for AD activity, however, becomes less useful when trying to localize the activity of specific AD entities. The tree-map visualization therefore includes a legend. This legend assigns a different color to each LIAM *attachment* that can be found the trace. Programmers can select attachments in the legend, triggering the tree-map visualization to be redrawn according to the legend selections. This can help to trace the activity of a specific attachment. Figure 4.4 shows the usage of the attachment for the example program. The activity of the around and before advice is coloured red and yellow respectively.

We have chosen to provide *colour-attachment* pairs in the legend, because the *attachment* entity represents the main unit of dispatch in ALIA. Defining a separate colour for each AD entity may cause problems for programmers to grasp the visualization, because of the possible large number of different colours. Besides, the activity of specific AD entities can be efficiently localized with other approaches, e.g. querying. This approach is discussed later in Section 4.2.

The colouring scheme of the legend is partially based on Tanter's vision on execution levels in AOP [52], 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 adviced joinpoint in Figure 4.4 and Figure 4.3 is coloured gray, because the execution level of zero (base) is reached because of the the call to `proceed()` by the around-advice of Listing 4.2.

The second technique is related to the bidirectional relation between AD and base-program. As shown by table 4.1, the contextual message of an AD joinpoint always includes a representation of the adviced joinpoint ID. This helps programmers to locate the adviced joinpoint of AD activity in the trace. If the adviced joinpoint is displayed in the current stage of the visualization, a tooltip above its related rectangle is drawn. This tooltip helps users to locate the adviced joinpoint rectangle conveniently. Furthermore, the contextual message of an adviced 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

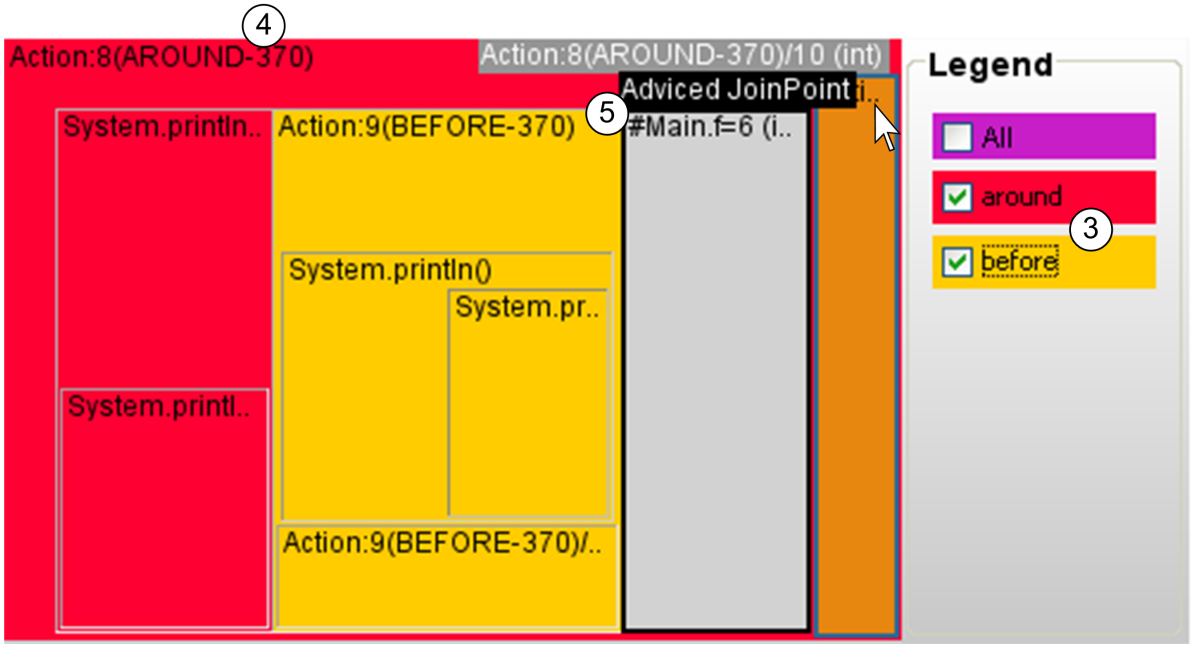



**Figure 4.4:** Same stage as in Figure 4.3. The attachment legend is used to colour the activity of the two advices separately ③. The contextual message of AD joinpoints contain a reference to the adviced joinpoint ID ④. The selected joinpoint by the mouse represents AD activity, its corresponding adviced joinpoint is therefore marked ⑤ by a tooltip. The '#' symbol at ⑤ indicates that the joinpoint is adviced.

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 adviced joinpoint rectangle, and the contextual message prefix are shown in Figure 4.4, see ⑤ and ④.

The third technique is related to skipped joinpoints. An around-action has the special usage of optionally preempting the original computation of the intercepted joinpoint, thus skipping its execution. Since the trace-model keeps track of skipped joinpoints as attributes of the *ActionExitJoinPoint*, the trace visualization represents them by a black tree-map rectangle, which is a colour that is not used by other colouring-schemes. As such, programmers can become aware of skipped program behaviour as a result of AD activity. This can be helpful when the programmer is exploring the trace and looking for a particular joinpoint that was unexpectedly skipped. Section 5.2 demonstrates a practical debugging scenario where this technique proves to be valuable.

## 4.2 Query-based debugging for AD programs

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

- Firstly, the joinpoints of interest can be scattered throughout the trace, whereas the trace visualization displays one region at a time. Consequently, one must reiterate the navigational steps of the trace visualization to localize each joinpoint.
- Secondly, joinpoints can be located at a low-level in the call tree. Since the trace visualization 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 visualization, 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 satisfied.

To conclude, using the trace visualization, 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.

Querying can aid localization 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 subset 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 localization of specific program behaviour by executing a sequence of queries, where each query further refines the search domain of the next.

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. Chapter 5 demonstrates the usage of several templates in concrete debugging sessions.

The rest of this section exhibits the adoption of a query language to the context of AD program traces. We start by explaining the conformance of the XQuery programming language to the storage-model of our approach. Then, Section 4.2.2 presents several templates for high-level queries, i.e. queries that are considered useful in debugging and independent of any specific program context. Finally, Section 4.2.3 focusses on identifying queries that target AD behaviour.

### 4.2.1 Adopting a query language to AD program traces

Several query languages, which are specifically designed for XML (e.g. XQL[10], XQuery), can process the XML trace of our back-end. For our approach, we have chosen XQuery as its query language, because it has been standardized. This instantly adopts a well-established query language to our work. Moreover, 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 axis reflect meaningful directions in which a trace can be explored. For example, the *following* and *preceding* axis relate to the execution order of the trace. These axis 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 $frjp$, the previous assignments to the field can be retrieved by the XQuery template expression shown in Listing 4.3. By using the *preceding* axis, the query locates *FieldWriteJoinPoints*, that preceded the execution of $frjp$, and refers to the same field as $frjp$. 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.

**Listing 4.3:** XQuery template expression to trace the causality links of a *FieldReadJoinPoint* called $frjp$.

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

### 4.2.2 High-level queries

The debugging requirements, which were used as the basis for the trace-model design in Section 3.1.2, can be translated to queries. The following sections show the translations as XQuery template expressions, which are applicable for traces that the back-end of our approach produces. Furthermore, this section demonstrates that the query-based approach can support several debugging facilities that are commonly offered by existing debuggers, e.g. stepping through the execution of a program.

#### Reconstructing object and program states

Object state reconstruction can be realized by performing the query of Listing 4.3 for each field of an object. Program states can be reconstructed in a similar way: by querying the last assignment to the *LocalVariableWriteJoinPoints* that are applicable in a given scope.

#### Reconstructing the control flow

Control flow reconstruction consists of locating the joinpoints that executed in the top-level control-flow of a given *call-joinpoint*. Given a *call-joinpoint*, called $cjp$, the control flow can be reconstructed by the query of Listing 4.4. This query first navigates to the *SubJoinPoints* element, and then uses the child-axis to traverse over the joinpoints that are childs of $cjp$ in the call-tree.

**Listing 4.4:** XQuery template expression to reconstruct the control flow of a *call-joinpoint* called $cjp$.

```
1 cjp/SubJoinPoints/child::JoinPoint
```

#### Limiting the search domain

Normally, a query is evaluated upon the entire execution trace. However, XQuery predicates can be used to specify the range of a query. Such predicates consists of

defining the boundaries of the desired range. This range is called the search domain, and represents a subset of the trace. Given a trace *tr*, Listing 4.5 shows how such a search domain can be defined by expressing XQuery predicates on the *joinPointId* attribute. Any other available attributes or elements of a joinpoint can be used to specify a search domain, e.g. *timeStamp*.

**Listing 4.5:** XQuery template expression to specify a search domain for a trace called *tr*. The *minRange* and *maxRange* represent arbitrary *joinPoindId* values that reflect the boundaries of the range.

```
1 tr//JoinPoint[@joinPointId>minRange and @joinPointId>maxRange]
```

A typical example usage of expressing a search domain, is to take the lexical scopes of the program into account. If a search domain is not expressed, a query can target joinpoints of different lexical scopes. This can lead to a confusing result set of a query, especially when querying for joinpoints that are related to local variables, because they have to be distinguished by their respective name and lexical scope.

This problem can best be illustrated by example. Consider the example program of Listing 4.6; the corresponding call tree is depicted by Figure 4.5. The trace contains assignments to the variable $i$ in two different methods, thus two different scopes (lines 8-10 and 14-16 in the source code).

Suppose the programmer is inspecting joinpoint (10) in the call tree, which corresponds to line 16 in the source code, and wishes to locate the previous assignments to the variable $i$. The query of Listing 4.7 accomplishes this. The result of this query contains five joinpoints, which refer to the assignments of values one through five to $i$. These joinpoints are encompassed by the rectangles in Figure 4.5. However, this result set may lead to confusion for the programmer, because it contains joinpoints that are not related to the runtime behaviour that is currently inspected.

**Listing 4.7:** XQuery expression to locate the assignments to variable $i$ from a joinpoint called *jp*, which is (10) in Figure 4.5.

```
1 jp/preceding::JoinPoint[Type="LVWJP" and LocalVariableId/VariableName="i"]
```

What the programmer was expecting to find, by using the query of Listing 4.7, are the joinpoints that assigned a value to $i$ and executed in the same scope that is currently inspected. These joinpoints are encompassed by the rectangle with diagonal lines in Figure 4.5. However, the search domain of the query contains all joinpoints

**Listing 4.6:** An example Java program.

```
1  public class Main{
2    public Main(){
3      m();
4      n();
5    }
6
7    public int m(){
8      int i = 1;
9      i = 2;
10     i = 3;
11   }
12
13   public void n(){
14     int i = 4;
15     i = 5;
16     i = 6;
17   }
18
19   public static void main(String args[]){
20     new Main();
21   }
22 }
```

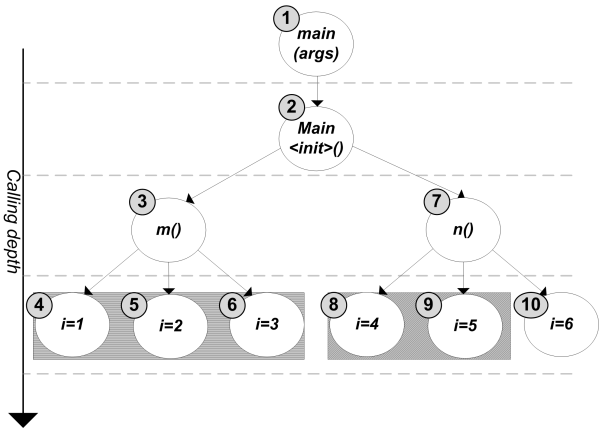



**Figure 4.5:** Call tree representation of the execution of the program of Listing 4.6. The joinpoints that satisfy the query of Listing 4.7 are encapsulated by striped rectangles.

that preceded (10), including the assignments to variable $i$ that executed in an entirely different scope and runtime state.

Situations, that are similar to the one that was just described, can lead to confusing query results. To resolve this problem, the programmer needs to able to limit the search domain of a query according to the scope that is currently examined. The lexical scopes of the program are reflected by the call tree representation. For example, the search domain, that corresponds to the expected scope in the scenario that was depicted before, can be constructed by traversing the preceding-siblings of joinpoint (10) in the call tree. This can be seen in Figure 4.5, where joinpoints (8) and (9) are siblings of joinpoint (10). Listing 4.8 uses the *preceding-sibling* axis of XQuery to define the search domain for the query (line 1). Then, a predicate is expressed over the search domain to find the local variable assignments to $i$. The result set of this query consists of joinpoint (8) and (9), which is what the programmer initially expected.

**Listing 4.8:** XQuery expression to locate the assignments to variable $i$ in the same scope as a joinpoint called $jp$, which is joinpoint (10) in Figure 4.5.

```
1 let $search_domain := jp/preceding-sibling::JoinPoint
```

```
2 return $search_domain[Type="LVWJP" and LocalVariableId/VariableName="i"]
```

### Stepping through the program execution

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*. Breakpoint-based debuggers typically implement these metaphors, and allow them to be used when the target program is suspended. 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 set of conventional stepping metaphors of breakpoint-based debuggers can be augmented with metaphors to navigate back-in-time, e.g. *step-back*.

The stepping queries are classified into two categories:

- **Forward-steps** relate to queries that step forward in time, e.g. step-next.
- **Backward-steps** relate to queries that step backward 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. The rest of this section explains several forward and backward-step queries. Since the required effect of several forward and backward queries is similar, despite of their traversal direction, we explain them in pairs. Figure 4.6 depicts the effect of each stepping query in the example call tree, which was introduced in Section 3.2.

### Step-next & Step-back

These stepping metaphors accept any initial-joinpoint and step to the next, or previous, joinpoint in time. The corresponding queries therefore realize a single step of a (reversed) pre-order traversal of the call tree, starting from the initial-joinpoint. Listing

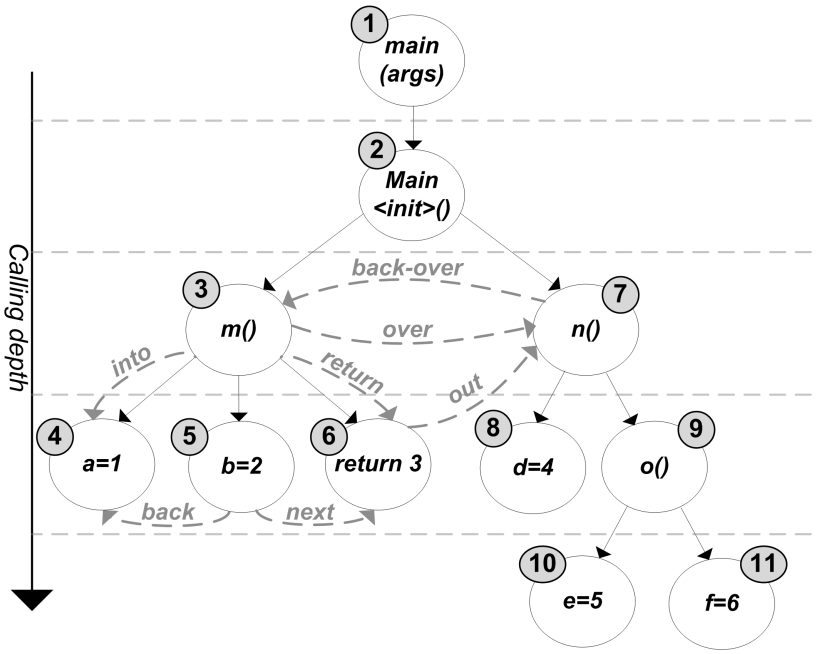



**Figure 4.6:** Visual depiction of the stepping metaphors. The stepping metaphors step-next and step-back are initiated from node ⑤, step-out from node ⑥, step-back-over from node ⑦, and step-into, step-return, step-over are initiated from node ③.

4.9 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. *following*-axis for a forward traversal, and *preceding*-axis for a backwards traversal. The effect of the queries is shown in Figure 4.6, where the initial-joinpoint is node ⑤, and the step-joinpoint is either node ④ or ⑥.

**Listing 4.9:** XQuery expressions for step-next and step-back, where the initial-joinpoint is called *jp*.

```
1 (: STEP–NEXT :)
2 jp/following::JoinPoint[1]
3 (: STEP–BACK :).
4 jp/preceding::JoinPoint[1]
```

### Step-over & Step-back-over

These stepping metaphors can only be applied when the initial-joinpoint is a call-joinpoint, and steps over all joinpoints that were executed during the call, either forwards (step-over) or backwards (step-back-over) in time. The corresponding query

therefore always traverses the call tree on the X-axis of its two-dimensional structure, hence there is no change between the initial-node and step-node regarding calling depth. Listing 4.10 shows the XQuery expressions of step-over (line 2) and step-back-over (line 4). These queries traverse to the first sibling, in a forwards or backwards direction, from the initial-joinpoint in the call-tree. The *following-sibling* and *preceding-sibling* can be used to realize the effect of traversing forward or backward respectively. The effect of the queries is shown in Figure 4.6. The initial-joinpoint for step-over is node ③, thereby ending at node ⑦, the step-back-over works the other way around.

**Listing 4.10:** XQuery expressions for step-over and step-back-over, where the initial-joinpoint is called *jp*.

```
1 (: STEP–OVER :)
2 jp/following–sibling::JoinPoint[1]
3 (: STEP–BACK–OVER :).
4 jp/preceding–sibling::JoinPoint[1]
```

### Step-into, Step-return & Step-out

These stepping metaphors are categorized as forward-steps. Step-into and step-return can only be applied when the initial-joinpoint is a call-joinpoint. Step-into traverses to the first joinpoint that was executed during the call-joinpoint, step-return traverses to the last joinpoint, which refers to the return statement of the function, and step-out traverses to the next joinpoint with the same calling depth as the call-joinpoint during which the initial-joinpoint executed. Step-return has a different meaning in our approach than in other popular debuggers, e.g. the Eclipse Java debugger. The reason why it is supported differently is inherent to locate the root cause when navigating a trace. Whenever the programmer encounters a call-joinpoint in the trace, she may verify that the call behaved correctly by examining its return value, e.g. checking if the returned value was not *null*. With the step-return of our approach, one can instantly locate the exit-joinpoint of the function call and inspect the return value. If the return value conflicts with the programmer's expectation, a causality link query, or backward-steps, can be used to track down the cause for the incorrect return value.

Listing 4.11 shows the XQuery expressions of step-into (line 2), step-return (line 4) and step-out (line 6). The queries for step-into and step-return first access the *SubJoinPoints* element, which gives access to the child joinpoints of the call. Then, step-into indexes the sequence of child joinpoints at one, thus selecting the first joinpoint

that executed during the call. Step-return uses the function *last()*, which returns the size of a sequence, to index the sequence of child joinpoints at its last element, hence the exit-joinpoint of the call. The query for step-out first traverses to the the call-joinpoint during which the initial-joinpoint executed, i.e. it's parent joinpoint in the call-tree. From there, the following joinpoint with the same calling depth in the call-tree is accessed by performing a step-over, i.e. the *following-sibling* axis, as shown in Listing 4.10.

**Listing 4.11:** XQuery expressions for step-into, step-return and step-out, where the initial-joinpoint is called *jp*.

```
1 (: STEP–INTO :)
2 jp/SubJoinPoints/child::JoinPoint[1]
3 (: STEP–RETURN :).
4 jp/SubJoinPoints/child::JoinPoint[last()]
5 (: STEP–OUT :)
6 jp/parent::JoinPoint/following–sibling::JoinPoint[1]
```

### 4.2.3 AD-specific queries

Queries can be formulated based on the AD specific information that is included in the trace-model. Such queries can be useful to determine if AD entities behaved as expected at runtime. In the following, we present several query templates that target specific AD activity. These templates define variables which must be substituted with runtime information of a particular trace, such that the template can be used for a debugging session of a program. Some of the templates were motivated by the study of Zhang and Zhao [62] on bug patterns in AOP . We start with a generic query that reconstructs the relation between AD activity and the base-program.

#### Querying the relation between AD and base-program

Given an AD joinpoint type, the relation to the joinpoint in the trace that was intercepted, can be located with the query in Listing 4.12, on line 1. This query performs a join on the *joinPointId* attribute of every joinpoint in the trace. This query can be further specialized 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 realized by adding an additional predicate to the XQuery expression. Line 3 in Listing 4.12 adds an XQuery predicate, such that the query limits the search domain by removing the joinpoints that preceded the execution of the before-advice.

**Listing 4.12:** XQuery template expression to locate the intercepted joinpoint, for a given AD joinpoint called *adJp*, in a trace called *tr*.

```
1 tr//JoinPoint[@joinPointId=$adJp/InterceptedJoinPointId/text()]
2 (: adJp == before–action :)
3 tr//JoinPoint[@joinPointId=$adJp/InterceptedJoinPointId/text() and
    @joinPointId>$adJp/@joinPointId]
```

### Querying the behaviour of predicates

Predicates reflect the dynamic conditions that need to be satisfied before an intercepted joinpoint is adviced. This can make it difficult for programmers to predict which joinpoints will be adviced at runtime. The following query templates can locate predicate evaluations with specific properties in the trace. These templates can therefore assists 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 adviced during execution, but they never were. The query template of Listing 4.13 (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 4.12. 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 realized by adding an extra XQuery predicate to the original query, as shown at line 3 in Listing 4.13.

**Listing 4.13:** XQuery template expression to locate the predicates evaluations, for an unmatched joinpoint called *umJp*, that resulted to *false*, in a trace called *tr*. This query template can optionally be refined to consider only a given predicate by inserting a *predicateId*, as shown on line 3.

```
1 tr//JoinPoint[Type="PEJP" and InterceptedJoinPointId=$umJp/@joinPointId and
      EvaluationResult="false"]
2 (: Specific predicate :)
3 tr//JoinPoint[Type="PEJP" and InterceptedJoinPointId=$umJp/@joinPointId and
      EvaluationResult="false" and PredicateId=predicateId]
```

The unmatched joinpoint bug pattern can also work the other way around, i.e. given a predicate, the programmer wants to verify if each joinpoint, that it intercepted, was evaluated correctly. This can be accomplished by locating the joinpoints that were evaluated by the given predicate. Depending on the verification strategy that the programmer wants to perform, a query can either return the set of joinpoints that matched or unmatched. Listing 4.14 presents a query template that locates all evaluations that resulted to *false*, for a given predicate.

**Listing 4.14:** XQuery template expression to locate the predicates evaluations, of a given predicate with *predicateId*, that resulted to *false*, in a trace called *tr*.

```
1 tr//JoinPoint[Type="PEJP" and EvaluationResult="false" and PredicateId=predicateId]
```

### Querying the behaviour of actions

The execution of before and after-actions can be considered isolated from the base-program, i.e. they cannot change the control or data-flow of the base-program. An around-action however, can alter the control and data-flow of the base-program, this is termed as *aspect interference* by AOP, we therefore adopt this term and use *AD interference*. As AD interference changes the program behaviour dynamically, it causes a gap between static text and dynamic execution, which can make it difficult for programmers to predict when these changes are performed. Examples of AD interference are:

- **Skipping** the execution of the intercepted joinpoint. This leads to a control or data-flow modification, depending on the type of the intercepted joinpoint.
- **Modifying the arguments** on which the intercepted joinpoint operates.

- **Modifying the return value** of the original computation of the intercepted joinpoint.

As shown in our preliminary study [54], AD interference can lead to complex debugging scenarios, which require the execution history to be resolved efficiently. Our trace-model keeps track of AD interference, and this information is registered in the attributes of the *ActionExitJoinPoint*. In the following, we present two examples of query templates that exploit these attributes to locate AD interference in the trace.

Listing 4.15 presents a query template to locate, for a given action, the intercepted joinpoints that it skipped. First, the *ActionExitJoinPoints* of the given action, and who skipped the execution of the intercepted joinpoint, are stored in a variable (line 1). Then, the XML representation of the intercepted joinpoint, which is a child element of a matching *ActionExitJoinPoint*, is returned (line 2). The programmer can examine the result of this query template to gain insight into the joinpoints that were skipped by AD interference of a particular action.

**Listing 4.15:** XQuery template expression to locate the intercepted joinpoints that were skipped by an around action, with `actionId`, in a trace called `tr`.

```
1 let $act_skipped := tr//JoinPoint[Type="AEJP" and ScheduleTime="AROUND" and
    JoinPointActionSkipped="true" and ActionId=actionId]
2 return $act_skipped/JoinPointActionSkipped
```

Listing 4.16 presents a query template to locate the intercepted joinpoints of which the data-flow was altered, regarding its return value, by a given action. We consider the return value unaltered, if either of the following conditions is met:

- The action invoked the computation of the intercepted joinpoint, without modified arguments,
- and the return value of the action is equal to the return value of the intercepted joinpoint. We consider the return values equal if an invocation to Java's `equals()` yielded *true*.

First, the *ActionExitJoinPoints* of the given action, and who returned a value other than the intercepted joinpoint, are stored in a variable (line 1). Then, the query template of Listing 4.12 is reused to return the corresponding intercepted joinpoints (line 2).

**Listing 4.16:** XQuery template expression to locate the intercepted joinpoints of which the data-flow, regarding its return value, was altered by an around-action, with *actionId*, in a trace called *tr*.

```
1 let $act_alt_dataflow := tr//JoinPoint[Type="AEJP" and ScheduleTime="AROUND" and
      ReturnValueModified="true" and ActionId=actionId]
2 return tr//JoinPoint[@joinPointId=$act_alt_dataflow/InterceptedJoinPointId/text()]
```

Furthermore, the return value of the intercepted joinpoint is registered by the trace-model, provided that it was ever computed during the execution of the around-advice. This allows programmers to compare the return value of the around-advice and the intercepted joinpoint, in order to examine the actual data-flow alteration that was imposed. Listing 4.17 presents a query template that locates the same information as the template of Listing 4.16, but returns a custom element called *DataFlowAlteration*. This element is constructed by the XQuery expression, and presents the differences in return value between the around-action and the intercepted joinpoint. An example of the output is shown in Listing 4.18, wherein the return values of the around-action and the intercepted joinpoint are highlighted.

**Listing 4.17:** XQuery template expression to create an overview of all the data-flow modifications, regarding return value, that were imposed by an around-action, *actionId*, in a trace called *tr*.

```
1  let $act_alt_dataflow := tr//JoinPoint[Type="AEJP" and
       ScheduleTime="AROUND" and
       ReturnValueModified="true" and ActionId=actionId]
2  for $alt in $act_alt_dataflow
3  return
4  <DataFlowAlteration>
5   <Action actionId="{$alt/ActionId}">
6    {($alt/AdActionReturnValue)}
7   </Action>
8   <InterceptedJoinPoint
        joinPointId="{$alt/AdvicedJoinPointId}">
9    {($alt/InterceptedJoinPointActionReturnValue)}
10  </InterceptedJoinPoint>
11 </DataFlowAlteration>
```

**Listing 4.18:** An example output of the query template of Listing 4.17.

```
1  <DataFlowAlteration>
2    <Action actionId="9">
3      <AdActionReturnValue>
4        <Int>300</Int>
5      </AdActionReturnValue>
6    </Action>
7    <InterceptedJoinPoint
         joinPointId="10">
8      <InterceptedJoinPointReturnValue>
9        <Int>250</Int>
10     </InterceptedJoinPointReturnValue>
11   </InterceptedJoinPoint>
12 </DataFlowAlteration>
```

### Querying shared joinpoints and precedence rules

Since AD allows to advice 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 [62].

Listing 4.19 presents a query template to locate shared joinpoints, for which no precedence declaration evaluated the applicable sequence of attachments. First, the shared joinpoints of the trace are retrieved (line 1), a quick recap: an intercepted joinpoint is considered shared if two or more AD function alternatives are applicable, hence the comparison at the end of line 1. Then, the query iterates over each shared joinpoint (line 2) and locates the corresponding *PrecedenceRuleEvaluationJoinPoint* (line 3), by using the strategy of Section 4.2.3. 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).

**Listing 4.19:** XQuery template expression to locate shared joinpoints for which no precedence declaration evaluates the applicable sequence of attachments, in a trace called $tr$.

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

## 4.3 Implementation

This section gives an overview of the implementation of the trace-based debugger tool called ALIA-TBD. This tool uses the Java programming language to implement the aforementioned techniques for trace navigation. It expects two sources of input: an XML trace file, and the source code of the program which produced the trace. This section starts by describing the general information about ALIA-TDB, i.e. its graphical layout, view synchronization, XML parsing, and query processing. Then, Section 4.3.2 describes the features, which ALIA-TBD provides, to combine the query-based debugging approach with the trace visualization. Finally, Section 4.3.3 introduces several features that are added to further improve trace navigation. Most importantly,

ALIA-TBD provides separate commands to support the stepping queries of Section 4.2.2. The implementation of these commands give the programmer the ability to express more powerful stepping queries.

### 4.3.1 General

#### Views

ALIA-TBD contains the following four views:

1. Source-code view. Constructs a relation between the trace and source code of the program. It can display Java and AspectJ source-code files, highlight syntax keywords, and highlight a given line-number.
2. Tree-map visualization. Implements the visual design and interactive strategies of Section 4.1.
3. Query editor. A text editor where XQueries can be expressed and executed. Syntax highlighting is supported, and syntax checking is performed dynamically, i.e. while the programmer expresses the query.
4. JoinPoint info view. Implements the proposed tree structure to expose the information of a single joinpoint. An example was provided by Figure 4.2.

A screenshot of ALIA-TBD is shown in Figure 4.7. This figure shows how the views are positioned. All four views, including the tree-map visualization, are fully resizable. This makes ALIA-TBD independent of the available display resolution. ALIA-TBD synchronizes views 1, 2 and 4, i.e. they react to events which are fired by the trace visualization. This synchronization ensures that the views display information about the same joinpoint(s), and programmers do not have to explicitly request to update a specific view. If the programmer selects (or double-clicks) a joinpoint in the trace visualization, the joinpoint-info view and source code view are triggered. The joinpoint-info view shows all detail of the selected joinpoint, and the source-code view highlights the dispatch site in the source code.

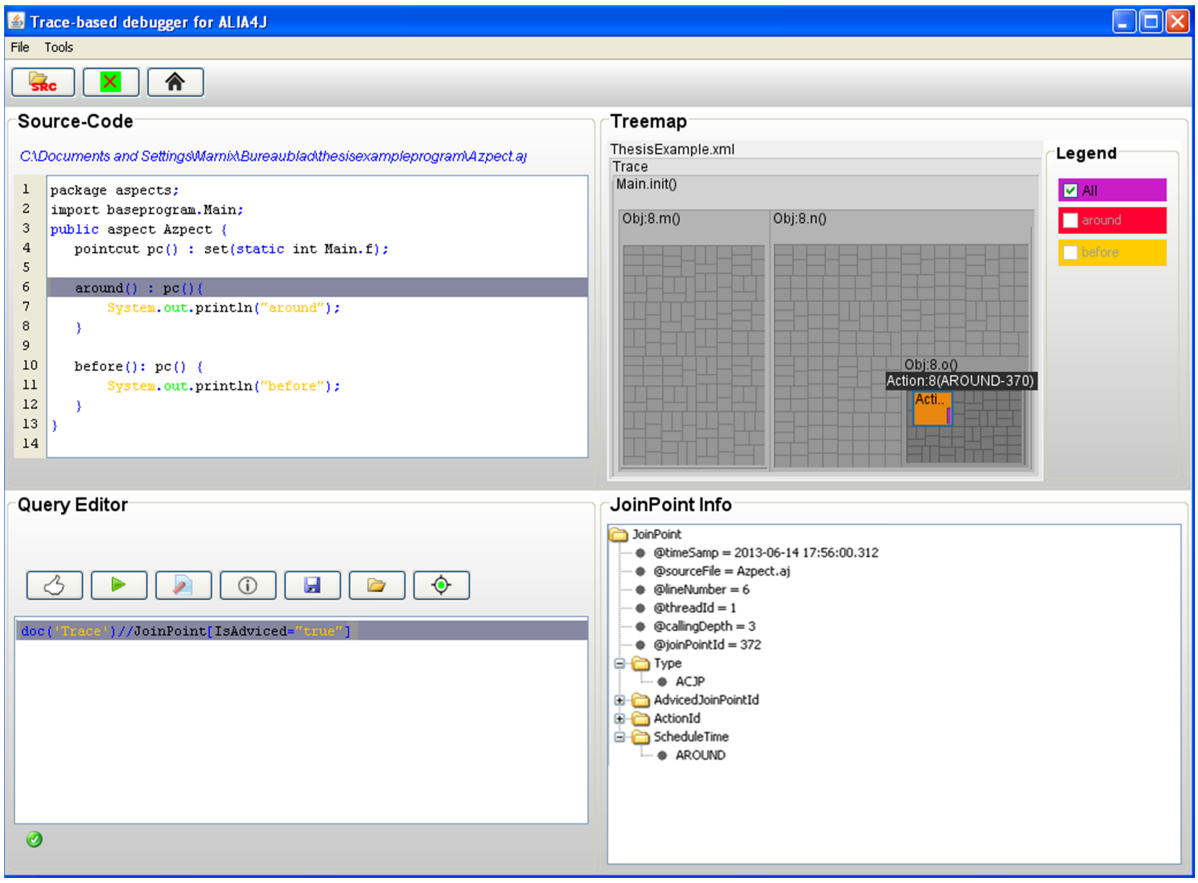



**Figure 4.7:** A screenshot of ALIA-TBD. Source-code view ①, trace visualization ②, query editor view ③, joinpoint-info view ④. Button which triggers the trace visualization to show query matches only ⑤. Button to remove query highlights from the trace visualization ⑥.

### Processing XQueries

ALIA-TBD uses the core modules of BaseX[1] to parse the XML trace file, and process XQueries. During parsing, BaseX applies an indexing scheme to the trace file, in order to efficiently process XQueries. This indexing scheme can be considered off-line regarding the back-end of our approach, and thus imposes no runtime overhead on the target program.

[1] BaseX is an open source, light-weight, high-performance and scalable XML Database engine and XPath/XQuery 3.0 Processor. See http://www.basex.org/

#### Implementation of the tree-map visualization

A tree-map implementation is provided by BaseX. This implementation is modified and extended by ALIA-TBD to fit the design of the trace visualization. The primary modification consist of removing rectangles that are unrelated to the trace, i.e. only rectangles that reflect a joinpoint element in the trace must be visualized. The extensions consist of adding the techniques to raise AD awareness, and synchronizing the views.

### 4.3.2 Combining query-based debugging with the trace visualization

The results of a query, executed from the Query-editor view, are projected on the trace visualization. This highlights, with a colouring scheme, the tree-map rectangles that correspond to the joinpoints of the query result. Three important properties are added to this colouring scheme, which assist programmers to locate the query results in the trace visualization:

- Firstly, the colouring scheme to highlight query results overrules all others, e.g. the AD activity colouring scheme. Otherwise, query results may not be visible in the trace visualization, which can lead to confusion for the programmer.
- Secondly, the colouring scheme is independent of calling depth. Query results may be located *too* deep in the call tree compared to the level of detail that is currently displayed by the trace visualization. To ensure that programmers can locate such query results, the trace visualization highlights the last call joinpoint, in their corresponding execution path, that is viewable in the current stage of the trace visualization. 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 visualization, can then be used to walk the execution path that led to a query result.
- Thirdly, the colouring scheme remains active until an explicit request is made by the programmer to remove query highlights. This ensures that, while one is navigating through the trace by multi-staging, the query results remain highlighted.

Figures 4.8, 4.9 and 4.10 show the aforementioned colouring scheme in action. The query of Listing 4.20 is executed on the trace of the AspectJ program, which was introduced in Section 4.1.2. The result of this query refers to the adviced joinpoint of the example program. In Figure 4.8 the *ActionCallJoinPoint*, during which the adviced joinpoint executed, is highlighted because its corresponding tree-map rectangle is viewable, and it represents the deepest call in the execution path that led to the query result. The tree-map rectangle is highlighted with a dark green color, because it does not represent the query result itself.

By using multi-staging, one can locate the query result, as shown in Figure 4.9. The query result is highlighted with a bright green color, because the tree-map rectangle, which represents it, is viewable in the current stage of the trace visualization. The programmer finally clicks button ⑥ in Figure 4.7 to remove the highlights of query results, as shown in Figure 4.10.

**Listing 4.20:** XQuery expression to localize adviced joinpoints in a trace called *tr*.

```
1 tr//JoinPoint[IsAdviced="true"]
```

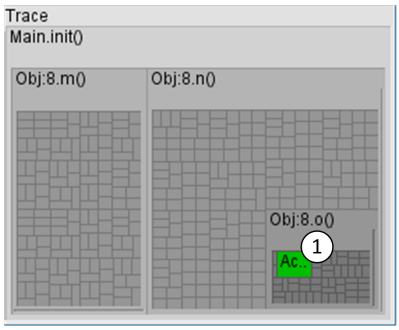



**Figure 4.8:** The initial stage of the trace visualization, which highlights the query result of Listing 4.20 ①.

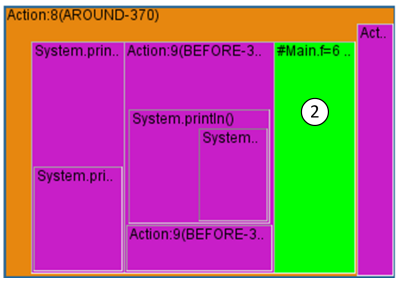



**Figure 4.9:** Detailed stage of the trace visualization, which highlights the query result of Listing 4.20 ②.

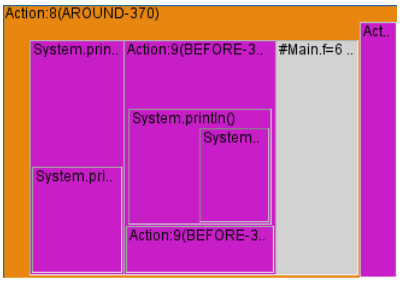



**Figure 4.10:** Detailed stage of the trace visualization, query highlights are removed.

The primary gain of projecting query results on the trace visualization, is that it gives the programmer an idea of the location of query results in the trace. This further improves program comprehension because it visualizes the differences between query results, regarding their execution path and execution time. For example, programmers can eyeball which query results were executed at an early stadium in the trace, or which execution path led to a particular query result. If query results are presented

as a sequence, programmers may find it difficult to deduce their location in the trace. This becomes even more difficult if query results need to be compared with one another on the matter of location. This would require the programmer to lookup the required information, that indicates the location of a query result in the call-tree, by using the navigational features of the trace visualization. This can become a time-consuming task especially if the query result was executed by a relatively long execution path, because the programmer will have to perform many interactive steps with multi-staging.

### 4.3.3 Utilities

During the implementation of ALIA-TBD, we added several features that can improve our approach. In this section we describe two such features. Firstly, a strategy that limits the search domain to an acquired query result. And secondly, a novel way of letting programmers express more powerful stepping queries to navigate through the trace.

#### Query-based navigation

Once a query has been executed, the programmer can choose to remove all joinpoints, that did not match, from the trace visualization. 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. Button ⑤ in Figure 4.7 enables this feature, after a query has been executed.

From this point, the programmer can further explore the query results with the trace visualization's multi-staging strategy, or by executing a new query. In case of the latter, the previously acquired query result is set as the search domain for the new query. This permits programmers to perform a step-wise querying approach, where each step further refines the matching set of joinpoints.

Examples of the previously described problem, and the effect of the utility, are shown by Figures 4.11 and 4.12. Suppose the query of Listing 4.21 is executed on the AspectJ program trace, that was introduced in Section 4.1.2. The query result matches two joinpoints: (1) the field write to $f$; (2) the local variable write, with value 153, to

$j$. The first match executed at calling depth three, and the second match executed at calling depth two.

The query matches are highlighted in the initial stage of the trace visualization, as shown in Figure 4.11. But because of their difference in calling depth, they are highlighted differently. The first match is highlighted by the strategy of Section 4.3.2, because it's corresponding tree-map rectangle is not visible in the initial stage. Thus, at this stage, the programmer can view the second match, but has to use multi-staging to reach the level of detail that makes the second match visible, and in this new stage the first match is not visualized. Thus, without *query-results-only*, the programmer cannot compare the query matches.

Figure 4.12 shows, that by using *query-results-only*, the query matches are visualized as a sequence. The programmer can then inspect, and compare their differences without having to perform navigational steps.

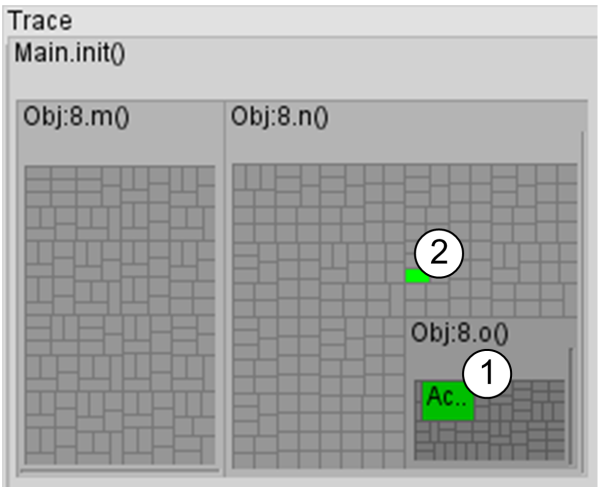



**Figure 4.11:** The initial stage of the trace visualization, which highlights the query result of Listing 4.20. The query matches are labeled with ① and ②.


**Figure 4.12:** Trace visualization when using the *query-results-only* feature after executing the query of Listing 4.21.

**Listing 4.21:** XQuery expression that matches two joinpoints in the AspectJ program, that was introduced in Section 4.1.2, which differ in calling depth.

```
1 tr//JoinPoint[(Type="FWJP" and FieldId/VariableName="f" and @callingDepth="3") or
    (Type="LVWJP" and NewValue/Int=153) and LocalVariableId/VariableName="j" and
    @callingDepth="2"]
```

### Stepper

ALIA-TBD supports the stepping queries, which were introduced in Section 4.2.2, as a set of stepping commands. The commands can be expressed by the programmer in the query editor view, and can be executed when a single joinpoint is selected in the trace visualization. The syntax of these commands are: *step.next*, *step.back*, *step.into*, *step.return*, *step.over*, *step.out*. Supporting these commands facilitates the programmer in using the stepping queries, because the underlying XQueries are more complex, and do not have to be formulated whenever stepping is desired.

The implementation of the stepping commands is realized by performing a translation step before a query is administered for execution. This translation step consists of locating the stepping commands, in the provided query, and replacing them with their corresponding XQuery expression. This implementation can be considered as an extension to the XQuery language, and allows to blend the stepping commands with the original syntax of XQuery. Two novel features thereby arise. 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 section 4.1.2. During stepping, the programmer encounters the loop of line 12 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 a XQuery predicate to the stepping query, the programmer can instantly step to the joinpoint that represents the $n$-th iteration in the loop. Listing 4.22 shows an example of a conditional stepping query, where the debugger steps to the 76th iteration of the loop.

**Listing 4.22:** Example of a stepping query composed with an XQuery predicate.

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

### Locate the activity of source-code lines

In the source-code view, the programmer can right click on a line and order ALIA-TBD to highlight the joinpoints in the trace that match the dispatch site. This utility is helpful because a bug is often associated with a region or line in the source code. For example, the stack trace of a bug contains the dispatch site at which the execution

failed. This dispatch site can be used by this utility to locate the corresponding runtime behaviour. From there, the programmer can use the navigational aids of ALIA-TBD to inspect the preceding runtime behaviour, thus enabling to backtrack from the symptom location to the root cause.

Moreover, since all joinpoints that match the dispatch site are highlighted, the programmer can locate similar joinpoints to the bug, but which passed during execution. By comparing these joinpoints, the programmer can deduce possible causes for the bug.

# 5 Evaluation

During our preliminary work [53], we drafted a detailed plan to correctly evaluate our work. This evaluation plan is modified to better suit the outcome of this work. This chapter describes the setup of this modified evaluation, how it was carried out, and what the results are. Since our approach consists of two separate parts, we structured the evaluation plan as such. Firstly, the back-end of our approach is evaluated in Section 5.1. The evaluation plan dictates to perform benchmarks that focus on performance and scalability, which are important when aiming for a practical solution. Though fulfilling such properties are beyond the scope of our work, the experiments give an indication of the state of the back-end and reveals existing bottlenecks, which may be resolved in future work. Secondly, the front-end of our approach is evaluated in Section 5.2. This evaluation is focused on effectiveness, as the evaluation plan suggests. Since no de facto benchmarks exist on this regard, we evaluate the front-end by conducting a case study, with ALIA-TBD, on an example system which contains several AD related bugs. This evaluation assesses and demonstrates the use of ALIA-TBD to debug a real program. Finally, in Section 5.2.3 we discuss the efforts that were necessary to debug this system with ALIA-TBD, and compare it to other debugging approaches.

## 5.1 Benchmarking the back-end

At first glance, it looks trivial to measure the performance of the back-end. However, Georges et al. [17] have shown that the execution time of a Java application can differ from run to run. These differences are caused by non-determinism in the experimental setup. Well-known sources of non-determinism are Just-In-Time (JIT) compilation, thread scheduling, external system events and garbage collection. Georges et al. therefore advocate to add statistical rigor to the performance evaluation of Java systems, and propose several methodologies for different kind of performance evaluations, i.e. start-up performance, steady-state performance. The latter concerns long-time running applications. We adopted the steady-state performance methodologies, because it is valuable to find out how the performance evaluation scales over relatively long execution times.

The performance evaluation of the back-end is conducted by a microbenchmarking platform. The design of this platform is shown in Figure 5.1 and consists of two main components, the *BatchExecuter* and the *OperationHarness*. The *BatchExecuter* defines the configurations to measure each operation in the microbenchmarking platform. The *OperationHarness* performs the actual measurement, and outputs a single data point once the measurement is finished. Both components are described in detail in the following two sections.

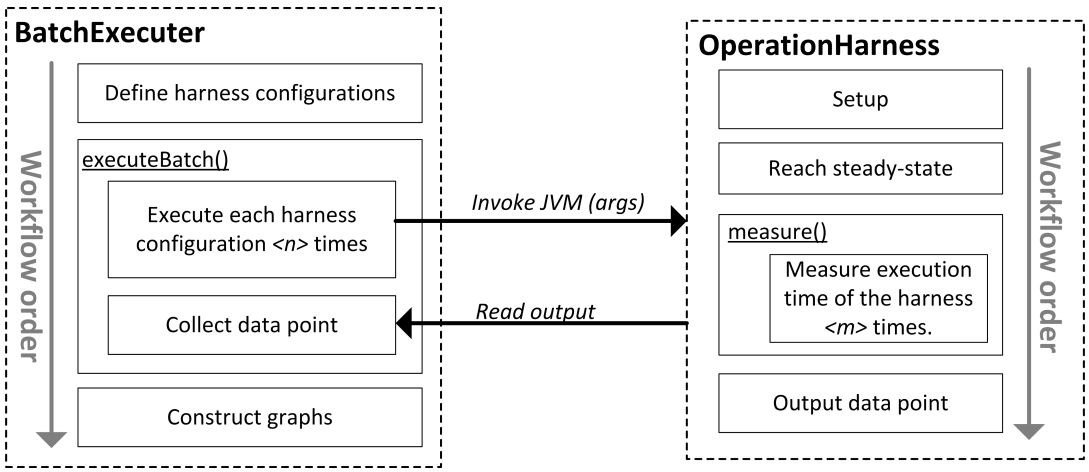



**Figure 5.1:** The components and workflow of the microbenchmarking platform.

### 5.1.1 BatchExecuter

The BatchExecuter first defines the harness configurations. Each harness configuration contains several attributes which determine how an operation is measured. The two most important attributes are:

- The *joinpoint type* that must be evaluated, e.g. *FieldWriteJoinPoint*.
- *IterationCount*, the number of joinpoints that must be performed to measure a single data point.

Separate harness settings are defined to calculate the performance overhead of the back-end, and identify the bottle-necks. Each setting disables the activity of certain sub-components of the back-end before executing a harness configuration. The following harness settings are defined:

- **Original setting**, measures the execution of an original JVM run, i.e. without any interference of our approach.

- **NOIRIn setting**, measures the execution time when only NOIRIn is activated.
- **Trace-model setting**, measures the execution time when the trace-model is activated. The implementation of the trace-model relies on NOIRIn, thus in this setting NOIRIn is also activated.
- **Storage-model setting**, measures the execution time when the trace-model and the storage-model are activated. Hence, all components of the back-end are enabled in this setting.

Each harness configuration applies an iterationcount of 100K joinpoints. This number was determined by trail and error: measuring the execution time of 100K joinpoints showed steady results in the original setting, which is the fastest setting, and 100K joinpoints can also be measured in the Storage-model setting in a feasible execution time (about 2 minutes per data point) regarding benchmarking.

The joinpoint types of the trace-model are captured by using multiple strategies, as described in Section 3.3.1. Each strategy may have a different performance impact, and therefore must be evaluated separately. The following joinpoint types encompass each strategy, and are therefore part of the microbenchmarking platform:

- *FieldWriteJoinPoints* are captured by reusing the interception of joinpoints by NOIRIn.
- *LocalVariableWriteJoinPoints* are captured by a separate extension to NOIRIn, which does not involve AD interpretation.
- *FunctionCallJoinPoints* are captured in the same way as *FieldWriteJoinPoints*, however the corresponding *FunctionExitJoinPoints* also need to be captured, which makes it slightly different.
- *ActionCallJoinPoints* are AD-related joinpoints, which are captured by instrumenting the interpretation process of NOIRIn.

For convenience, we refer to a harness configuration by using the following abbreviation style: `<HarnessSetting>-<Joinpoint type acronym>`. The acronyms for each joinpoint type were introduced in Section 4.1.2. For example, the harness configuration: *Storage-Model-FWJP*, outputs one data point which represents the execution time to perform 100K consecutive *FieldWriteJoinPoints* in the storage-model setting.

The harness configurations of the *BatchExecuter* can be determined by taking the Cartesian product of the defined joinpoint types and the harness settings. *ActionCallJoinPoints* can not be measured in the original setting, because it is related to AD activity, which a *original* java program does not support. This yields a total of 15 harness configurations. Each harness configuration is executed by *n=36* separate JVM invocations, such that the number of measurements can be considered sufficiently *large* [17]. Consequently, the distribution of these measurements is a normal distribution, thereby allowing to apply respective statistical calculations, e.g. confidence intervals.

After defining all the harness configurations, the *BatchExecuter* invokes its batch process, called *executeBatch()*. During this process, each harness configuration is serialized to a String object, which can be passed as an argument to the JVM invocation of the *OperationHarness*. The *BatchExecuter* then waits till the execution of the *OperationHarness* is finished. The output of the *OperationHarness*, which represents a data point, is read out and added to the collection of data points. Finally, after the batch processes is finished, the *BatchExecuter* presents a graph of the performance evaluation. This graph is presented and discussed later in this section.

The *BatchExecuter* is implemented as a Python script, because Python has an extensive offer of libraries for statistical functions and graph plotting, e.g. SciPy[1], Numpy[2] and Matplotlib[3].

### 5.1.2 OperationHarness

The *OperationHarness* is a Java program that receives a harness configuration, as a command-line argument, from the *BatchExecuter*. The attributes of the harness configuration are then parsed, and a setup phase is performed accordingly. The setup phase performs the initial steps that are required to execute the harness configuration, e.g. deploying an *Attachment*.

Before the actual measurement starts, the *OperationHarness* reaches steady-state, which means that the performance of the harness suffers less from variability due to JIT compilation. Steady-state is reached once the coefficient of variation, after $k$ harness measurements, falls below a threshold of 0.02 [17].

[1] Library of scientific tools. See http://www.scipy.org/
[2] Multi-dimensional arrays in Python. See http://www.numpy.org/
[3] Plotting library for Python. See http://matplotlib.org/

Once steady-state is reached, the *OperationHarness* invokes the *measure()* routine. Listing 5.1 presents a simplified version of this routine. The execution time of the harness is measured (lines 7-13) during each iteration of a while-loop (line 3). Note that the execution time of the call to `System.nanoTime()` is taken into account by this measurement (lines 11,12). The measured execution time, which represents a data point, is added to a list (`performanceDataPoints` on line 15). The while-loop keeps iterating when either: (line 2) the harness has not been executed for a minimum number of times (`BenchmarkSetup.MIN_ITERATIONS` is set to 5 by default); or (line 3) the data points of the list are not converging, i.e. dividing the maximum and minimum value, of the last `BenchmarkSetup.MIN_ITERATIONS` data points in the list, exceeds a certain threshold, which is set to 0.05. By demanding convergence, the routine can be more certain to measure a data point that does not represent an outlier, because it behaved similarly to several previous measurements. In effect, we take the last measurement of the list as the data point for a JVM invocation of the *OperationHarness* (line 20), because it is representative for the previous `BenchmarkSetup.MIN_ITERATIONS` measurements. The `BenchmarkSetup.PANIC_ITERATIONS` (line 4) is set to 100, which means that the *OperationHarness* failed if the data points in the list did not converge during any of the 100 iterations of the while-loop.

**Listing 5.1:** Simplified version of the measure() routine of the *OperationHarness*.

```
1  public void measure(List<Long> performanceDataPoints){
2    int iteration = 0;
3    while((iteration < BenchmarkSetup.MIN_ITERATIONS
4              || !converging(performanceDataPoints))
5           && iteration < BenchmarkSetup.PANIC_ITERATIONS){
6
7        long startTime = System.nanoTime();
8
9        executeHarness();
10
11       long endTime = System.nanoTime();
12       long nanoCallTime = System.nanoTime() - endTime;
13       long executionTime = endTime - startTime - nanoCallTime;
14
15       performanceDataPoints.add(executionTime);
16
17       iteration++;
18   }
19   if !(panic(performanceDataPoints))
20     output(getLastDataPoint(performanceDataPoints));
```

```
21 }
```

All harness settings that are defined by the *BatchExecuter*, except for the *original setting*, use bytecode instrumentation to capture joinpoints of the underlying Java program. Because this instrumentation can influence the execution of the *OperationHarness*, we have configured the *OperationHarness* such that only the body of the *executeHarness()* method is subjected to the bytecode instrumentation.

The body of the *executeHarness()* method consists of a for-loop that ranges from 0 till *IterationCount*, and executes a single joinpoint per iteration. The executed joinpoint is of the type that was defined by the harness configuration. For the joinpoint types: LVWJP, FWJP and FCJP; the content of the for-loop consists of a single-line. For example, to measure *FieldWriteJoinPoints* the content of the for-loop consists of the line: `field=value;`.

To measure *ActionCallJoinPoints*, the setup phase of the *OperationHarness* deploys the *Attachment* of Figure 5.2. The content of the for-loop consists of a *FieldWriteJoinPoint*. Consequently, during each iteration of the for-loop, the *FieldWriteJoinPoint* is intercepted by NOIRIn and the method *BenchmarkAction.call* is called, which is captured as an *ActionCallJoinPoint* by the trace-model. The *Attachment* uses the AROUND schedule time, and the method body of *BenchmarkAction.call* is empty, such that only a *ActionCallJoinPoint* and *ActionExitJoinPoint* are captured during each iteration of the for-loop.

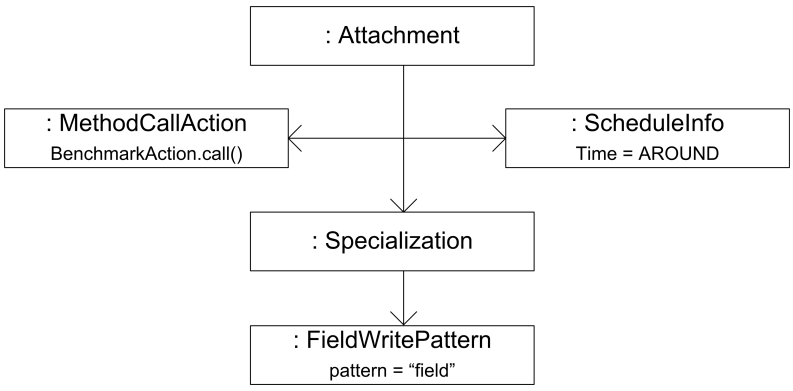



**Figure 5.2:** Object diagram of the *Attachment* used to benchmark *ActionCallJoinPoints*.

### 5.1.3 Environment

The measurements of the *BatchExecuter* were conducted on a 2GHz Processor, with 4GB of RAM running the Windows NT 5.1 kernel. The version of the Sun JVM was 1.6.0_ 24 with a heap size of 256MB, which is the default maximum heap size for this JVM when operating on a 32-bit system.

### 5.1.4 Data analysis: performance overhead

After the *BatchExecuter* is finished, 36 data point samples of the underlying population are gathered for each harness configuration. Each data point represents the execution time that was necessary to perform a particular harness configuration. The actual mean execution time of a harness configuration can be approximated by computing the confidence intervals around the mean of the data point samples. These intervals are based on a certain probability, and define a range, $c_1 - c_2$, wherein the actual mean lies. For example, a confidence interval of 95% means that there is a 95% probability that the actual mean lies within the respective range.

The unit, in which we define performance overhead, is *times the original execution*. This means that if the performance overhead of a harness configuration is for example 100, then the execution time of the harness configuration is 100 times the execution time of its counterpart in the original setting. The counterpart is determined by taking the harness configuration in the original setting with the same joinpoint type. For example, the counterpart of the harness configuration *Storage-Model-FWJP* is *Original-FWJP* in the original setting.

The performance overhead is calculated as follows: Firstly, given a harness configuration, determine its counterpart in the original setting. Then, calculate the mean of both the harness configuration $\bar{x}_1$ and its corresponding counterpart $\bar{x}_2$. The performance overhead is calculated by: $\bar{x}_1/\bar{x}_2$. Note that we calculated a 95% confidence interval of the performance overhead of each harness configuration. These intervals were so close to the mean such that any performance overhead calculations, based on the confidence intervals instead of the mean, gave the same results after rounding to a whole number. For example, the range of the confidence interval of the harness configuration *Trace-model-FWJP* spanned 0.2% over its mean value. Calculating the performance overhead by using for example the upper bound (i.e. mean + 0.1%) of

the confidence interval yields the same performance overhead, after rounding, when using the mean.

Because the *BatchExecuter* does not define a harness configuration for *Original-setting-ACJP*, the performance overhead of harness configurations that involves the *ActionCallJoinPoint*, is calculated by taking *FunctionCallJoinPoint* in the original setting as its counterpart. This is a valid counterpart regarding performance, because the *ActionCallJoinPoint* is interpreted as a function call by NOIRIn. Furthermore, no other AD activity is performed by NOIRIn before the function call is invoked, because the Attachment of this benchmark uses a *TruePredicate*, such that every candidate joinpoint is adviced without predicate evaluations.

Figure 5.3 presents a bar chart of the performance overhead of each harness configuration, grouped by harness setting. We now describe the findings of the data analysis, and refer to the bars in the chart by using the numbers in their corresponding labels.

Bars 1-4 correspond to the NOIRIn setting. LVWJP imposes no performance overhead in this setting, because this joinpoint type is not part of NOIRIn's joinpoint-model. Thus, no byte-code instrumentation is performed, letting LVWJP behave the same as in the original setting. NOIRIn already imposes a performance overhead in the order of thousands for FWJP (bar 2) and FCJP (bar 3). This is due to NOIRIn's interpretation process. For example, every field write is captured by NOIRIn, and submitted to the interpretation process, even if no attachments are deployed. ACJP (bar 4) imposes about half of the performance overhead of FCJP, because ACJPs are not submitted to NOIRIn's interpretation process.

Bars 5-8 correspond to the trace-model setting. The performance overhead of FWJP, FCJP and ACJP (bars 6-8) is a little higher than in the previous setting, because the trace-model has to hook into NOIRIn's interpretation process to collect and structure the runtime information. LVWJP (bar 5) imposes an overhead 242 in this setting, which is much lower than the performance overhead of the other joinpoint types. The reason for this is that LVWJPs are captured by an extension to NOIRIn, which is separated from the interpretation process, as described in Section 3.3.

The performance overhead of this extension is relatively close to that of existing omniscient debuggers, e.g. WhyLine [29] has a performance overhead 252, Unstuck [25] 250 and TOD [48] 176. However, the performance overhead of these omniscient debuggers was measured with their original storage-model enabled (WhyLine), an

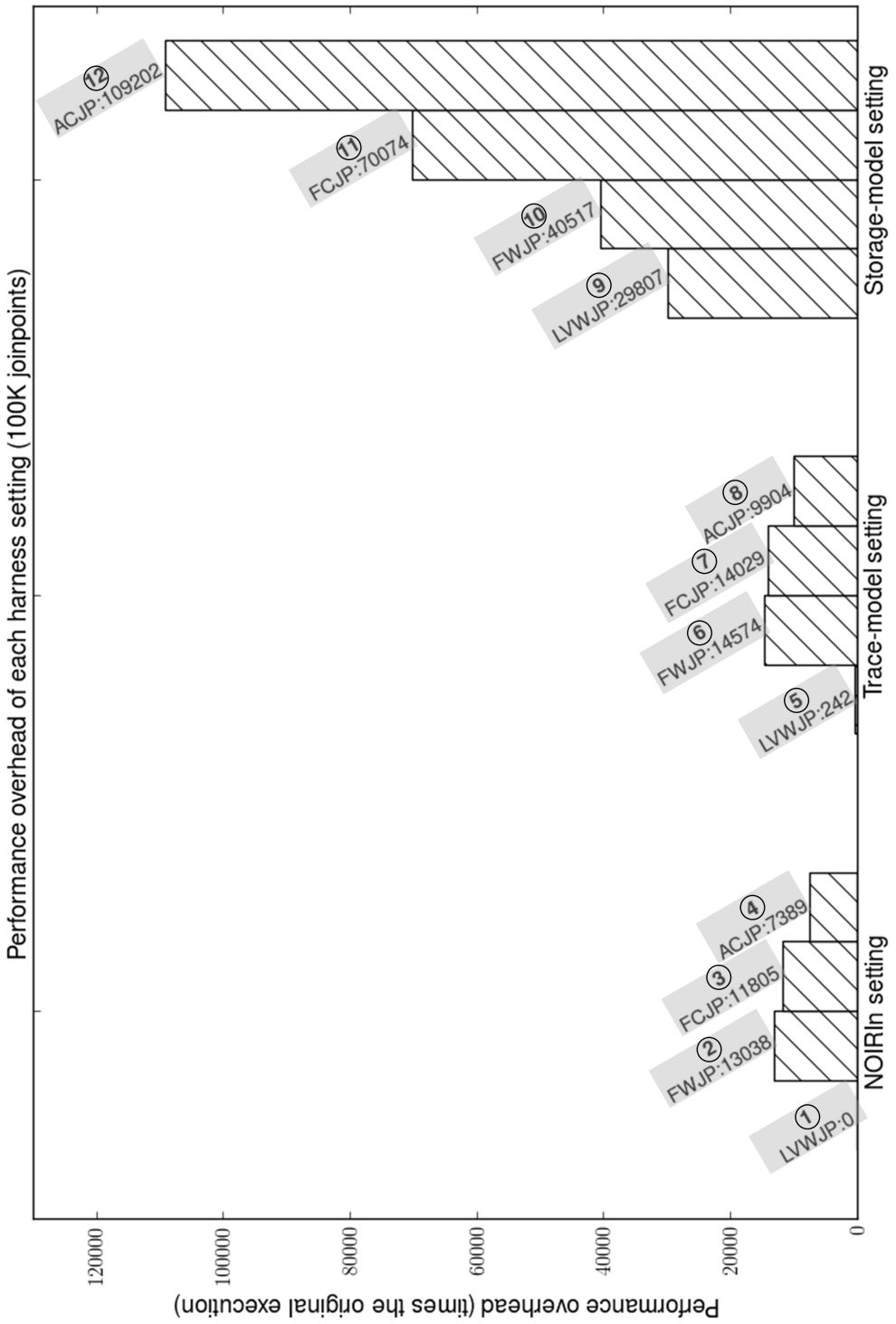



**Figure 5.3:** Bar chart constructed by the *BatchExecuter*.

in-memory storage-model (Unstuck), or a simplified implementation of their storage-model (TOD). To make a fair comparison with our performance overhead in this setting, we performed an extra experiment. In this experiment, the performance overhead was measured for LVWJP, in the same setting as bar 5, and a customized storage-model was enabled. This customized storage-model creates compact textual representations of LVWJPs, where only the runtime information is written out that would be sufficient to offer state reconstruction. The outcome of the experiment showed a performance overhead of 410. This outcome can be compared fairly to other omniscient debuggers, because the performance overhead was measured in a similar setting, i.e. where both a trace-model and a storage-model were enabled.

The reason why the performance overhead of our experiment is higher than that of existing omniscient debuggers is due to the amount of runtime information that is captured per LVWJP. Our trace-model also captures the runtime information of the function to which a LVWJP belongs. By doing so, the trace-model becomes flexible, as discussed in Section 3.1.2. Existing omniscient debuggers, in contrast, use a fixed trace-model that contains entities to capture functional behaviour of the program, therefore it is guaranteed that the runtime information of the corresponding function of a LVWJP is included in the trace, and thus this runtime information does not have to be captured per LVWJP.

Bars 9-12 correspond to the storage-model setting. The performance overhead in this setting are the highest, because all captured information by the trace-model is written out to disk, which requires a lot of I/O operations. Moreover, a lot of bytes are written out per joinpoint, because of XML's verbosity. The performance overhead of ACJP (bar 12) is the highest because it contains the most attributes, as can be seen from the trace-model in Figure 3.1.

The data analysis showed that the bottlenecks, which are mainly responsible for the performance overhead, are NOIRIn's interpretation process and the implementation of the storage-model. However, the design and implementation of NOIRIn does not target efficiency. By reusing NOIRIn for the development of our approach, we inherit its inefficiency, which is amplified by the unoptimized implementation of the storage-model. Several optimizations to reduce the performance overhead of both bottlenecks are planned as future work, and are presented in Section 7.2.3.

### 5.1.5 Storage model scalability

For each joinpoint type in the trace-model, it can be approximated by how many bytes it is represented in the XML of the storage-model, because the XML file is encoded using UTF-8, thus every character in the XML file occupies one byte. Consequently, we can estimate the XML trace size for a given set of joinpoints. The approximations are based on a static analysis of the XML representation of each joinpoint type. This analysis consists of determining the minimum amount of bytes of the XML representation of a certain joinpoint type. The minimum amount can be determined by taking the smallest possible value for the attributes of a particular joinpoint type. For example, the number of bytes for the attribute *FieldWriteJoinPoint.fieldName* is one byte at minimum, because the name of a field variable consists of at least one character. To give an impression of the outcome of this analysis: a FWJP occupies 920 bytes at minimum. Thus, a trace consisting of 100K FWJPs yields an minimum XML file size that weighs in at 92MB.

From this evaluation we can conclude that the storage-model scales linearly, because the XML representation of each joinpoint has a bounded size. Existing trace-based debuggers hardly discuss the trace size that they generate. Only Pothier et al. [48] present a trace size of 3.6GB after running a program that executed tens of millions of joinpoints. However, little detail is given about the program that generated the joinpoints, which makes it unfeasible to fairly compare their results to ours. Nevertheless, in our approach, a program that executed tens of millions of joinpoints will yield a file size larger than 3.6GB. For example, executing ten million *ThrowJoinPoints*, which can occupy the smallest number of bytes (760 bytes), will yield a file size of 7.6GB.

Pothier, one of the authors of TOD, studied the applicability of existing database management systems for a trace-based debugging approach [44]. This study showed that a program that executes just over 100K joinpoints, randomly distributed over three joinpoint types (FWJP, LVWJP, FCJP), results in a database size of 61.2MB in PostgreSQL and 67.5MB in Berekely DB. Again, no exact details are presented about the program that was used for this study, such that we cannot make a fair comparison to our storage-model. However, if we evenly distribute the three joinpoint types over 100K, we can estimate the XML file size and make a reasonable comparison to the aforementioned outcomes. The estimation is calculated by: $(33.3K * (minimumBytesXMLrepresentationofFWJP)) +$

$(33.3K * (minimumBytesXMLrepresentationofLVWJP)) +$
$(33.3K * (minimumBytesXMLrepresentationofFCJP)) = 127.2MB$. Thus, our storage-model creates a trace size which is roughly twice as large as the database sizes of PostgreSQL and Berekely DB.

## 5.2 Assessment of the front-end

The case study, to evaluate the front-end, consists of a AD program where three bugs are deliberately introduced. Section 5.2.1 describes the AD program, and presents the source-code locations of each bug, which are given a number such that we can conveniently refer to them in the rest of this section. The bugs that are introduced were inspired by our prelimanary study [54]. The goal of this evaluation is to show that these bugs can be efficiently resolved when the programmer possesses the trace-based debugging features that are offered by our approach.

Section 5.2.2 outlines how the programmer observes each bug, and shows how the features of the front-end can be used to localize the corresponding root cause efficiently and increase the program comprehension. Finally, Section 5.2.3 discusses the differences between our front-end and other existing debugging approaches, regarding the debugging features that are required to resolve each bug.

There are two important notes regarding the limitations of this evaluation. Firstly, we outline merely one way in which each bug is resolved, while they could be resolved in less or more steps, depending on the level of knowledge that the programmer has about the case study program, and which features the programmer can utilize during debugging, e.g. the programmer may not be familiar with XQuery and therefore cannot use the query-based debugging approach.

Secondly, this evaluation relies on the assumption that the programmer is familiar with the AD concept, and the debugging features that are offered by the front-end. These limitations can be overcome by conducting a user experiment, where programmers of different levels (e.g. senior developer, student) use the front-end to resolve the bugs. However, such an experiment is beyond the scope of this thesis, and is scheduled as part of future work (Section 7.2.7).

### 5.2.1 Case study program

The case study program that is used for this evaluation represents a grocery store. A simplified UML diagram of the grocery store program is shown in Figure 5.4. This diagram contains only the classes and corresponding function signatures that are relevant for this evaluation.

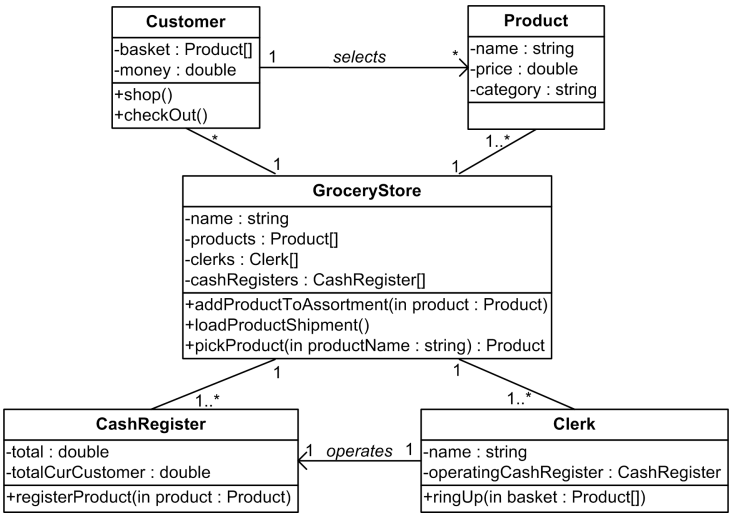



**Figure 5.4:** A simplified UML class diagram of the case study program.

The real-life scenario that this program is trying to represent is described in the next section. During this description we occasionally refer to entities of Figure 5.4.

#### Program description

The program starts with delivering a shipment to the grocery store. This shipment is loaded (*GroceryStore.loadProductShipment()*) into the assortment of the grocery store, which is represented by *products : Product[]*. A customer can visit the store and start shopping (*Customer.shop()*), whereby the customer picks out several products of the store's assortment (*GroceryStore.pickProduct(String)*). Every product that is picked out is put in a basket, represented by *Customer.basket : Product[]*. When the customer has found all the desired products, she checks out (*Customer.checkOut()*), which means that a clerk is visited that operates a cash register of the store. The clerk then rings up (*Clerk.ringUp()*) each product such that they are registered by the cash register (*CashRegister.registerProduct(Product)*). The cash register keeps track of the total price for the current customer, represented by *CashRegister.totalCurCustomer*. This price

is shown after the ring up is completed. Finally, the customer pays the clerk who subsequently inserts the money into the cash register, thereby increasing the value of *CashRegister.total*.

There are three concerns implemented by using AOP:

- Firstly, the concern of **ticket printing**, i.e. every customer is given a ticket after checking out. This ticket has a unique ID (e.g. a bar-code), contains an overview of the products that were bought, and shows the total price.
- Secondly, the concern of **discounts**, i.e. the price of several products may be temporary discounted. The grocery store system has to be aware of these discounts, and calculate the product prices accordingly.
- Thirdly, the concern of **bonus products**, i.e. products that are involved in a sales event, e.g. *two for the price of one*, can be given away for free.

The implementation of each concern is represented by an AspectJ aspect. The rest of this section describes the code of each aspect.

**Ticket printing aspect**

Listing 5.2 presents the ticket aspect, which implements the concern of ticket printing. This aspect has two pointcuts, and three advices. Each advice is given a name in Java comments for easy reference. The *productRegister* pointcut (line 6) matches whenever a product is registered. The *ringUpTransaction* (line 7) pointcut matches whenever the clerk rings up the products of a customer's basket.

The *ticketBeforeRingup* (line 10) advice is executed before the clerk rings up the products of a customer, and prints the header of the ticket. This header contains the name of the clerk, and a unique ID of the ticket that is being printed. During the ring up, the *ticketBeforeRegister* (line 16) advice is executed before a product is registered, and prints its name and price on the ticket. After the ring up, the *ticketAfterRingup* (line 22) finalizes the ticket by adding a footer.

The ticket aspect contains two fields: (1) the ticket ID, represented by an integer value, and (2) the total of the current customer, such that it can be printed on the ticket after the ring up. Thus, the ticket aspect and the cash register maintain the total price of the customer, by the fields *TicketAspect.curCustomerTotal* and *CashRegister.totalCurCustomer* respectively.

**Listing 5.2:** AspectJ aspect that implements the concern of ticket printing.

```
1  public aspect TicketAspect {
2
3    private int ticketId = 0;
4    private double curCustomerTotal = 0;
5
6    pointcut productRegister(Product product) : call(public void
         CashRegister.registerProduct(Product)) && args(product);
7    pointcut ringUpTransaction(Clerk clerk) : call(public double Clerk.ringUp(*)) && target(clerk);
8
9    //ticketBeforeRingup
10   before(Clerk clerk) : ringUpTransaction(clerk){
11     PrinterDriver.printTicketHeader(ticketId, clerk.getName());
12     curCustomerTotal = 0;
13   }
14
15   //ticketBeforeRegister
16   before(Product product) : productRegister(product){
17     PrinterDriver.printProduct(ticketId, product.getName(), product.getPrice());
18     curCustomerTotal += product.getPrice();
19   }
20
21   //ticketAfterRingup
22   after(Clerk clerk) : ringUpTransaction(clerk){
23     PrinterDriver.printTicketFooter(curCustomerTotal);
24     ticketId += 1;
25   }
26 }
```

### Discount aspect

Listing 5.3 presents the discount aspect, which implements the concern of discounts. The *DISCOUNT* (line 2) field declares the discount as a percentage that must be subtracted from the original price of a product. The *productRegister* pointcut is the same as the one of the ticket aspect (Line 6 in Listing 5.2). The around advice (line 7) checks if the price of the registered product needs to be discounted (line 8). If so, the advice checks if the price of the product was already discounted before (line 9). If not, a new value to the price field of the product is assigned (line 10). After the discounted price is assigned, a field of the Product object is set to *true* such that future purchases use the same price instead of applying discounts over and over for the same product (line 11). Finally, the call to *CashRegister.registerProduct* is performed by calling *proceed()* (line 13).

**Listing 5.3:** AspectJ aspect that implements the concern of discounts.

```
1  public aspect DiscountAspect {
2    private final static double DISCOUNT = 25.0;
3
4    pointcut productRegister(Product product) : call(public void
         CashRegister.registerProduct(Product)) && args(product);
5
6    //discountAroundRegister
7    void around(Product product) : productRegister(product){
8      if(isDiscountedProduct(product)){
9        if(!product.priceIsDiscounted()){
10         product.setPrice(calcDiscountedPrice(product.getPrice()));
11         product.setPriceIsDiscounted(true);
12       }
13     }
14     proceed(product);
15   }
16 }
```

### Bonus aspect

Listing 5.4 presents the bonus aspect, which implements the concern of bonus products. The *productRegister* pointcut is the same as the one of the ticket aspect (Line 6 in Listing 5.2). The around advice (line 5) checks if the registered product is a bonus product. If not, the original call to *CashRegister.registerProduct* is performed by calling *proceed()* (line 7). If it is a bonus product, the around advice skips the call to *CashRegister.registerProduct*, such that it is not processed by the cash register, and thus not added to the total price.

**Listing 5.4:** AspectJ aspect that implements the concern of bonus products.

```
1  public aspect BonusAspect {
2
3    pointcut productRegister(Product product) : call(public void
         CashRegister.registerProduct(Product)) && args(product);
4
5    void around(Product product) : productRegister(product){
6      if(!isBonusProduct(product)){
7        proceed(product);
8      }else{
9        //FIXTO: product.setPrice(0); proceed(product);
10     }
11   }
12 }
```

**Ordering aspect**

Each call to *CashRegister.registerProduct* is adviced by the ticket aspect, discount aspect and bonus aspect. It is therefore a shared joinpoint. Listing 5.5 presents the OrderingAspect, which defines an explicit order between the three aspects in the event of a shared joinpoint.

**Listing 5.5:** AspectJ aspect that defines an order between the bonus, ticket and discount aspect.

```
1 public aspect OrderingAspect {
2   declare precedence: BonusAspect, TicketAspect, DiscountAspect; //FIXTO: DiscountAspect,
        TicketAspect
3 }
```

### Introduced bugs

**1: Incorrect precedence declarations** When a product is registered, the ticket aspect precedes the execution of the discount aspect. Consequently, the original price of the product is printed onto the ticket, even if the product's price is discounted. The source-code location of the bug is shown on line 2 in Listing 5.5.

*Symptom:* the symptom of this bug is most likely to be observed after the ticket is printed out, because the ticket reflects an invalid price for a discounted product. Furthermore, the total price that is shown on the ticket is also invalid, because it used the original product price to calculate the total of the ring up.

**2: Cause-effect chasm bug** When the product shipment is loaded, a product is wrongly inserted into the store's assortment. A Brussels sprout is categorized as a snack, while it should have been categorized as a vegetable. The source-code location of the bug is shown on line 3 in Listing 5.6.

**Listing 5.6:** Code of the cause-effect chasm bug.

```
1 public void loadProductShipment(){
2       ...
3       addProductToAssortment(new Product("Brussels sprout", 4.50, "SNACK")); //FIXTO:
            VEGETABLE
4       ...
5 }
```

*Symptom:* the effect of this bug is most likely to be observed when the product is registered, and the ticket is printed out, because the ticket contains the price that was accounted for the Brussels sprout. Thus, creating a cause-effect chasm between the moment where the shipment is loaded, till the point when the Brussels sprout is bought by a customer.

**3: Incorrect control-flow** The grocery store system does not keep track of bonus product registrations. This is because the bonus aspect skips the call to *CashRegister.registerProduct*, see line 8 in Listing 5.4.

*Symptom:* the symptom is most likely to be observed after the ticket is printed out, because the ticket misses any information about bonus products.

### 5.2.2 Using ALIA-TBD

#### Program execution and bug symptoms

The back-end of our approach records the execution of the grocery store system. The discount aspect is configured to apply for all products that are categorized as vegetable or bread. The bonus aspect applies for the bounty product, without any additional conditions. Thus, every customer can pick up a free bounty.

During execution of the grocery system, a customer shops for seven products as shown in Listing 5.7. The cash register presents a total of $ 10,54 after the clerk performed the ring up. The corresponding ticket is shown in Listing 5.8, wherein each bug symptom is highlighted.

#### Resolving Bug 1

The symptom of each bug is located in the ticket, as described in Section 5.2.1. Based on the available data of each symptom, we can write a query that locates the joinpoint that can be associated with the bug symptom. For example, the query of Listing 5.9 locates the joinpoint that corresponds to the bug symptom of line 11 of the ticket (Listing 5.8). This query finds the function call to the ticket printer when the Brussels

**Listing 5.7:** An implementation of *Customer.shop()*.

```
1  public void shop(){
2   //Vegetable –> discount of 25%, original price =
        $4,50
3   basket[0] = pickProduct("Brussels sprout");
4   //Bonus, original price = $1,50
5   basket[1] = pickProduct("Bounty");
6   //Bread –> discount of 25%, original price = $1,95
7   basket[2] = pickProduct("Ciabatta");
8   basket[3] = pickProduct("Apple");
9   basket[4] = pickProduct("Beef steak");
10  basket[5] = pickProduct("Milk");
11  basket[6] = pickProduct("Lollipop");
12 }
```

**Listing 5.8:** Ticket that is printed out after the clerk rang up the products of Listing 5.7. The bug symptoms are highlighted with a gray color.

```
1  = AdvancedDispatching–Mart =
2
3  Date: 2013/07/05 17:16:46
4
5  Ticket ID: 0
6
7  Clerk: Jasper
8
9  ––––Products––––
10
11  Brussels sprout - $4,50  //<– No discount (Caused
        by Bug 1 and 2)
12  Ciabatta - $1,95  //<– No discount (Caused by Bug
        1)
13 Apple – $0,35
14 Beef steak – $3,95
15 Milk – $0,95
16 Lollipop – $0,45
17  ....  //<– No bounty registered (Caused by Bug 3)
18
19  TOTAL: $12,15  //<– Incorrect total (Caused by
        Bug 1,2,3)
20
21 = Thank you and goodbye =
```

sprout product, which is wrongly reflected on the ticket, is passed as the function argument.

**Listing 5.9:** XQuery expression to locate the joinpoint that can be associated with the symptom of bug 1, in the trace called $tr$.

```
1 let $symptom := $tr//JoinPoint[Type="FCJP" and
2              CalledId//FunctionName="printProduct" and
3              .//FunctionArgument[@argIndex=0]/Int="0" and
4              .//FunctionArgument[@argIndex=1]/String="Brussels sprout"]
```

Figure 5.5 presents the initial stage of the trace visualization, after this query has been executed. The programmer can use the trace visualization to navigate towards the bug symptom ①, e.g. by double-clicking on the call to *Clerk.ringUp*. Figure 5.6 shows the next stage, where the bug symptom is still highlighted as a query result

②, and the programmer uses the attachment legend to identify the activity of each pointcut-advice pair ③. When the programmer inspects the activity of the ticket aspect, the adviced joinpoint gets highlighted, as shown by ④ in Figure 5.6. This joinpoint refers to the call to *CashRegister.registerProduct(Product)* where the function argument is related to the Brussels sprout product, which is the bug symptom. From the visual representation of the aspect composition, the programmer can conclude that the product was printed onto the ticket before the discount aspect even applied, thus the advice ordering is wrong. Based on this information, the programmer can use the query template, which was introduced in Section 4.2.3, to find out if a precedence rule was applied for the adviced joinpoint. This query is shown in Listing 5.10.

**Listing 5.10:** XQuery expression to locate precedence rule evaluations that can be associated with the adviced joinpoint (*CashRegister.registerProduct(Product)*), called `registerProduct`, in a trace called `tr`.

```
1 tr//JoinPoint[Type="PRJP" and AdvicedJoinPointId=$registerProduct]
```

After executing the query, ALIA-TBD refers to the source abstraction of the precedence rule, enabling the programmer to observe that the ticket aspect precedes the discount aspect, hence the cause of the bug is located. Finally, the programmer can apply the fix to the precedence declaration, as shown in Listing 5.5.

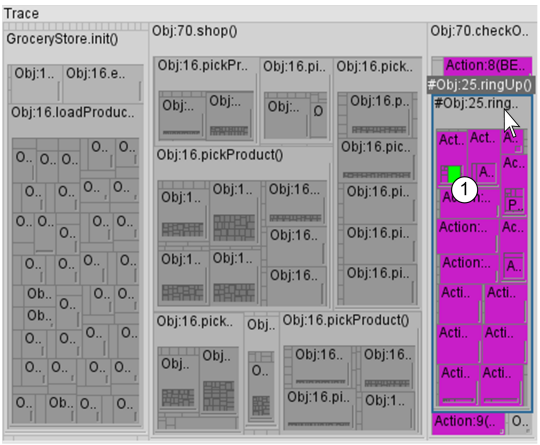



**Figure 5.5:** Initial stage, presenting a condensed overview of the grocery store trace. The query result of Listing 5.9 is highlighted, see ①.

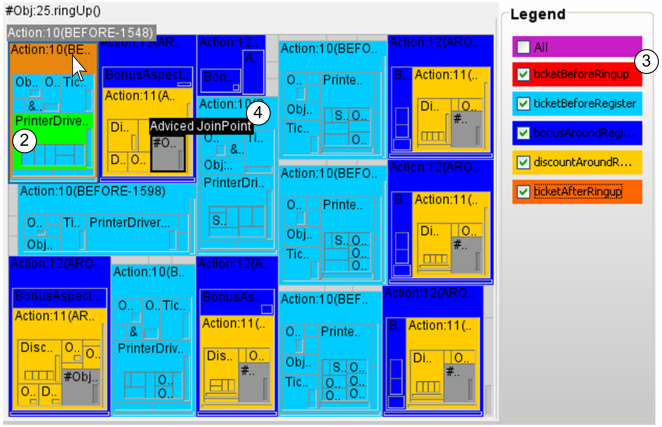



**Figure 5.6:** Presenting a more detailed view of the call to *Clerk.ringUp* . The query result of Listing 5.9 is still highlighted, see ②; the attachment legend is used for every pointcut-advice pair ③, and the adviced joinpoint is labeled ④.

After the programmer applied the fix to the precedence declaration in Listing 5.5, the aspect composition is visualized as in Figure 5.7. This visualization helps the

programmer to acknowledge that the aspect composition is now correct, because the discount aspect is executed before the ticket aspect.

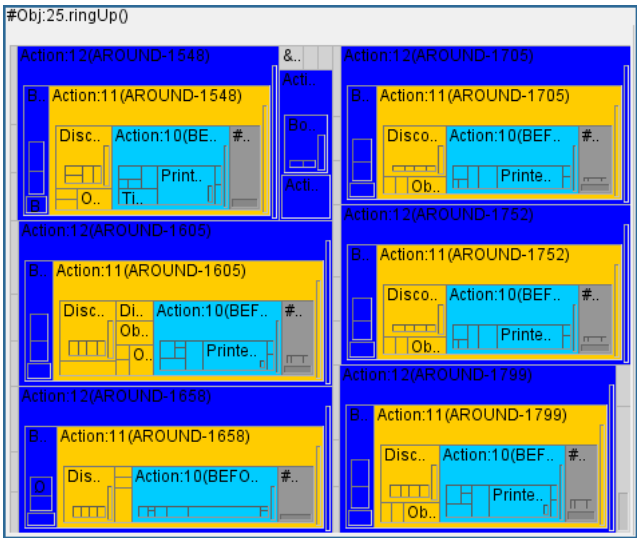



**Figure 5.7:** Trace visualization, showing the stage as in Figure 5.6, after bug 1 is fixed.

### Resolving bug 2

After bug 1 is resolved, the ticket correctly reflects the discount of the ciabatta product, because it is categorized as a bread product. However, the discounted price of the Brussels sprout product still does not appear on the ticket.

This bug has the exact same symptom as bug 1, therefore we can reuse the query of Listing 5.9 to locate the joinpoint that can be associated with the symptom. Thanks to the trace visualization, the programmer can see that the discount aspect executed before the ticket aspect, as shown in Figure 5.7. By navigating backwards, the programmer can examine how the discount aspect behaved, and inspect the call to *DiscountAspect.isDiscountedProduct(Product)*, as shown in Figure 5.8.

The return value of this call, and the OID of the Brussels sprout object, can be inspected by consulting the joinpoint-info view. The programmer notices that the return value was *false*, which gives the programmer the impression that the properties of the Brussels sprout product are wrongly set. Thus, the programmer reconstructs the state of the Brussels sprout product by utilizing the query template that was introduced in Section 4.2.2. The corresponding query is shown in Listing 5.11. This query reconstructs the bounty product object state at the moment when the bug

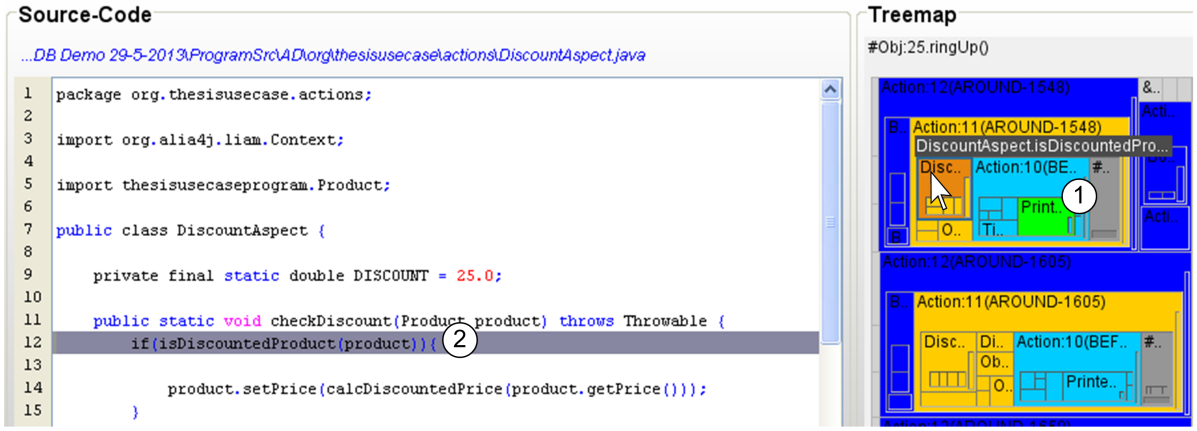



**Figure 5.8:** Trace visualization, showing the stage as in Figure 5.6. The symptom is highlighted as a query result ①; the activity of the discount aspect is selected by the mouse; The corresponding source-code is shown ②.

symptom (Listing 5.9) was executed. The result of this query refers to the three field write joinpoints that executed within the constructor of the Brussels sprout product object. The programmer notices that the category of the Brussels sprout object was set to "SNACK", which is wrong. Subsequently, the programmer can step backward to find out who called the constructor of the Product object. This brings the programmer to the source-code location of Listing 5.6, hence the cause of the bug is found and the fix can be applied.

**Listing 5.11:** XQuery expression that reconstructs the state of the bounty product OID, called *BountyOID* at the moment when the bug symptom was executed.

```
1 return $symptom/preceding::JoinPoint[Type="FWJP" and
      FieldId/FieldOwner//Object[ID=$BountyOID]
2                     and (.//VariableName="name"
3                         or .//VariableName="price"
4                         or .//VariableName="category")]
```

### Resolving bug 3

For this bug, the ticket contains no data which can be used to track the symptom of the bug, because the bonus product (Bounty) was never printed on it. Therefore, the programmer can write a query to find the function exit of *GroceryStore.pickProduct("Bounty")* in the control-flow of *Customer.shop()*, and from there navigate through the trace to find out how the product was handled when it was ringed up by the clerk. This is expressed by the query of Listing 5.12.

**Listing 5.12:** XQuery expression to locate the joinpoint that can be associated with the symptom of bug 3, in the trace called $tr$.

```
1 return $tr//JoinPoint[Type="FEJP" and CallerId/FunctionId/FunctionName="shop" and
2         CalledId/FunctionId/FunctionName="pickProduct"and
3         .//FunctionArgument[@argIndex=0]="Bounty"]
```

By inspecting the function exit joinpoint of the query result, the programmer can find out what the OID of the bounty product is, because the return value of *GroceryStore.pickProduct("Bounty")* is a Product object. Subsequently, the programmer can express a query to find the activity of the OID when the customer's basket was ringed up by the clerk. With this query, the programmer can examine how the bounty Product object was handled during the ring up, and find out why it did not appear on the ticket. The query is expressed in Listing 5.13.

**Listing 5.13:** XQuery expression to locate the activity of the bounty product OID, called *BountyOID*, in the trace called $tr$.

```
1 return $tr//JoinPoint[Type="FCJP" and CallerId/FunctionId/FunctionName="ringUp" and
2         .//Object[ID=$BountyOID]]
```

By looking at the query results, reflected in the trace visualization, the programmer can see that the call to *CashRegister.registerProduct* was skipped by the bonus aspect, because it is represented by a black tree-map rectangle, as shown in Figure 5.9. This indication may already be sufficient input for the programmer to conclude why the bounty product was never printed onto the ticket. Moreover, when further examining the trace visualization, the programmer can see that a call to an AROUND action was also skipped ①, because it is also shown by a black tree-map rectangle. The programmer can find out which action was skipped by clicking on the AROUND action's tree-map rectangle and consulting the joinpoint-info view. This AROUND action corresponds to the discount aspect, and the programmer knows that the discount aspect precedes the ticket aspect, because she corrected the precedence declaration when resolving scenario 1 (Section 5.2.2). Hence, the cause for the bug is located, and is fixed by the programmer at line 9 in Listing 5.4.

The reader may notice that this fix changes the price of a Product object to zero forever. However, our implementation resets the price of each product back to its original value when the store opens up the next day. Thus, only if the product is part of bonus on the next day will its price be set to zero again.

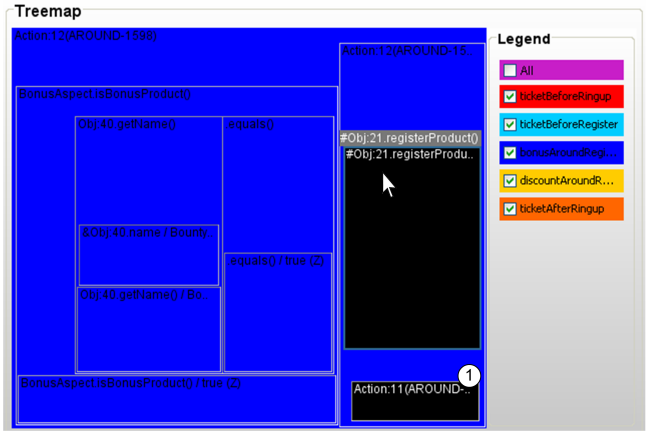



**Figure 5.9:** Trace visualization, showing the joinpoints that were skipped (in black) by the bonus aspect.

### After debugging

After all fixes of the bugs have been applied, the ticket of the same customer is printed out. This ticket is shown in Listing 5.14.

**Listing 5.14:** Ticket that is printed out, after the bug fixes, when the clerk rings up the products of Listing 5.7. The lines that are highlighted in gray indicate the bug fixes.

```
1  = AdvancedDispatching–Mart =
2
3  Date: 2013/07/05 17:16:46
4
5  Ticket ID: 0
6
7  Clerk: Jasper
8
9  ---Products---
10
11 Brussels sprout - $3,38 //<– Correctly discounted
12 Bounty - $0 //<– Bonus product registered
13 Ciabatta - $1,46 //<– Correctly discounted
14 Apple – $0,35
15 Beef steak – $3,95
16 Milk – $0,95
17 Lollipop – $0,45
18
19 TOTAL: $10,54 //<– Correct total
20
21 = Thank you and goodbye =
```

### 5.2.3 Discussion

This discussion compares our front-end to existing debugging approaches which aim for similar goals in debugging. We first identify, for each bug of this evaluation, which debugging features can help to resolve them efficiently. Then, we explain why several debugging approaches are irrelevant to this discussion. Finally, we check if relevant debugging approaches offer similar features, and if so, how these features differ from our approach.

#### Required debugging features

We briefly summarize how each bug of this evaluation was resolved, and underline the specific debugging features that were needed. Each bug of this evaluation can be classified as a cause-effect chasm bug, because the symptom is located relatively far away from the cause, especially in bug 2, where the cause was executed during shipment loading, and the symptom was executed when the ticket was printed out. Thus, exploring the execution history is an important requirement for resolving these kind of bugs.

Exploring the execution history can be done in multiple ways, however from the bugs described in this evaluation, we can conclude that querying can be very helpful to navigate towards the cause of a bug, because it allows the programmer express powerful navigation steps in a dedicated syntax. This is shown by bug 2, where the programmer can navigate from the symptom directly to the cause by expressing a query.

From bug 1, we can see that the trace visualization helped the programmer to find the cause, because it enables visual navigation through the trace and presents the composition of AD activity. Without such a visualization, the programmer may not have gotten aware of the incorrect composition of AD activity, which would have made it more time-consuming to locate the bug's cause.

AD activity can change the execution of the base-program, which can cause bugs that the programmer may not have expected at first glance. Therefore, allowing the programmer to query and explore control and data-flow changes is important, otherwise the debugger cannot inform about such changes, as was necessary in bug 3.

### Irrelevant debugging approaches

We omit three major debugging approaches from this discussion, because they apply a vastly different approach than ALIA-TBD, and the debugging goals differ. Firstly, **Program slicing** focuses on finding regions of the source-code that are related to a particular debugging task. A slicing approach therefore differs from ALIA-TBD, because we focus on letting the programmer navigate and explore the execution history to localize bugs, and increase the program comprehension. Besides, slices are often based on data-dependency graphs, while the actual cause of a bug is not always present in this graph. For example, in bug 1 the precedence declaration was the cause for the bug, which has no data-dependency to the symptom of the bug.

Secondly, static analyzer tools can perform an analysis on the source code of the AD program, and can construct a visualization which makes it apparent how AD activity may affect the base-program execution. However, such debugging approaches cannot give insight into the runtime behaviour of the AD program, while this is one of the main goals of our approach, because bugs often reveal under specific input, as shown by the bug examples of this evaluation.

Thirdly, the conventional debugging approaches, i.e. **breakpoint-based debugging** and **log-based debugging**, do not work well for cause-effect chasm bugs, as explained in Section 2.1.2. We therefore omit these approaches from the discussion.

### Comparison with relevant debugging approaches

Table 5.1 presents an overview of the debugging approaches that (can) support one or more of the debugging features, which were identified in the previous section.

Omniscient debuggers allow to navigate through the execution history with the well-known debugging metaphors, e.g. *step-forward*, *step-into*. Therefore, they do not allow the programmer to express an arbitrary query, which can be very useful as was demonstrated by bug 2. Furthermore, several omniscient debuggers, e.g. IntelliTrace[39], ODB [32], implement a similar look and feel as breakpoint-based debuggers, with the special abilities of stepping backwards. Thus, no visual representation of the trace is offered, which can be useful for navigational purposes and increasing the program comprehension, as shown by bug 1.

**Table 5.1:** Evaluation of debugging features compared to other debugging approaches.

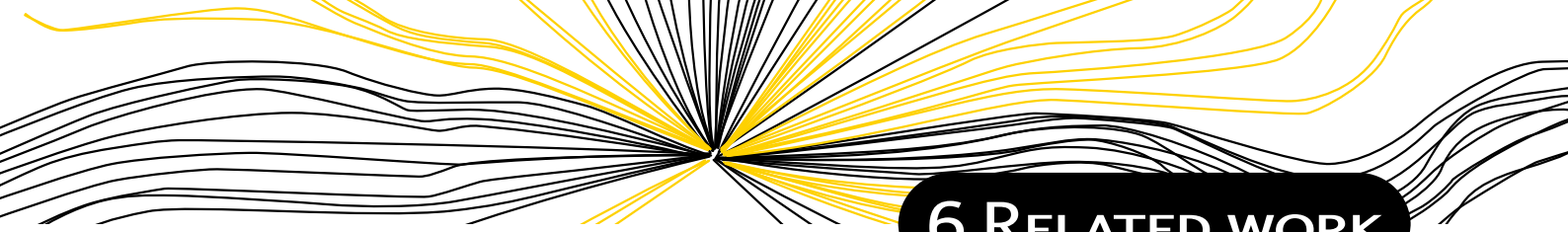
| | **Omniscient debugging** | **Query-based debugging** | **ALIA-TBD** |
|---|---|---|---|
| Execution history is accessible | ✓ | **O** | ✓ |
| Visualization of the trace, including a visual representation of the composition of AD activity | ✗ | ■ | ✓ |
| Express and execute arbitrary queries | ✗ | ✓ | ✓ |
| Explore control and data-flow changes of AD activity | ■ | ■ | ✓ |

✓- supported
✗- not supported
**O** - the debugging approach can optionally support the debugging feature.
■ - we are not aware of an existing omniscient or query-based debugging approach that supports the debugging feature.

Several existing query-based debugging approaches, e.g. JavaDD[18], record the entire execution history and allow programmers to execute queries. However, we are not aware of existing query-based debugger that contain a visual representation of the trace, wherein AD activity is explicitly marked, and which depicts the composition of AD activity, as was helpful to resolve bug 1 and 2.

To our best knowledge, we are not aware of an existing trace-based debugging approach (omniscient or query-based) that records the control and data-flow changes of AD activity. We consider our work unique on this regard, and bug 3 of this evaluation has shown that this feature can be helpful in debugging AD programs.

To summarize, ALIA-TBD provides a trace visualization, offers the same features as an omniscient debugger, and allows to express and evaluate an arbitrary query on the trace. This combination makes ALIA-TBD suitable for resolving the bugs of this evaluation.



# 6 Related work

We categorize this chapter into two groups. At first, the related work of the back-end of our approach is discussed. The approach of existing trace-based debuggers are shortly described. Their trace collection strategies and storage structures are then compared to our approach. Secondly, the related work of the front-end of our approach is discussed. Existing software trace visualizations and query-based debugging approaches are described and compared to our work. These comparisons focus on the visualization layout, and the way queries are evaluated.

## 6.1 Related work of the back-end

This section starts with an elaborate discussion about existing trace-based debuggers with special support for AD. Then, we devote two sections that discuss conventional trace-based debuggers which produce an exhaustive trace, or use an replay-based approach.

### 6.1.1 Trace-based debugging approaches with AD support

To our best knowledge, there exists only one trace-based debugger that provides special support for AD, i.e. an extension to TOD [46] in order to support AOP. For convenience we refer to this extension as TOD-AOP. The following paragraphs shortly discusses TOD, followed by an elaborate comparison between our approach and TOD-AOP.

TOD [48] is a trace-based debugger that creates an exhaustive trace, i.e. it records every state change of the program execution, and can be used for Java programs. TOD uses two key strategies to provide scalability and impose as little performance overhead as possible (176x the execution time of the original program). Firstly, the storage-model consists of a custom *event-database*, which offers a high throughput. Secondly, A custom indexing scheme is applied to the event database, such that a good response time can be provided when information is retrieved. Thirdly, TOD is designed to use a distributed database architecture, consisting of several database

nodes in a server cluster, which can further improve the performance. Such hardware demands are however not always available for programmers.

Unlike a custom database structure, our approach uses a storage-model that produces an XML file of the trace, which increases the potential of reuse. Furthermore, the indexing of a trace in TOD is performed during execution, and is thus partially responsible for the performance overhead. Our approach, in contrast, indexes the trace in an off-line fashion. This entails no performance overhead regarding indexing, but increases the loading time of the front-end when a trace is opened.

TOD-AOP is designed to support trace-based debugging for AspectJ programs. It applies a tagging-scheme on the woven bytecode of the AspectJ program in order to identify AO-specific activities, such as advice execution. Several features are offered, which allows programmers to inspect AO related runtime information in an appropriate intimacy level. The history of pointcut evaluations can be inspected, and an overview of aspect activity within the trace can be generated. We list the main differences between the extension to TOD and our approach (some of these differences were already identified by our preliminary study [54]):

- TOD-AOP works on the woven bytecode, wherein a reference to several source abstractions can be lost. For example, the execution order between multiple aspects, which can be defined by *precedence rules*, is enforced when AspectJ weaves the aspects with the base-program. Therefore, the semantics of precedence rules do not have to be considered at runtime, such that the woven bytecode does not contain a reference to their corresponding source abstractions. Thus, TOD-AOP can neither trace the activity of such source abstractions, nor can it inform the programmer about their presence. Our approach is based on ALIA4J, which can map AO concepts to LIAM instances, wherein no source abstractions are lost.
- TOD-AOP reuses the trace-model and storage-model of TOD, which do not contain AO-specific attributes. Therefore, we can deduce that no AO-specific entities are defined for these models, and that the activity of AO elements are unified to base-program elements. For example, the call to an advice is traced as a normal function call. Consequently, no AO-specific attributes exist in the trace, e.g. a reference to the intercepted joinpoint, leaving it to the programmer to reconstruct this information while debugging the activity of aspects. Our trace-model defines several entities for AD elements, which contain attributes

that reflect the relation to the corresponding joinpoint that was intercepted. The storage-model stores this information appropriately in the trace.

- TOD-AOP inherits the navigational means of TOD, which consists of the conventional stepping metaphors, and respective metaphors to navigate in a backwards direction. This limits the way in which information can be located in the trace. As shown in this thesis, our query-based debugging approach gives programmers more flexibility on this regard, and also supports the conventional stepping metaphors.
- TOD-AOP cannot refer to the individual parts of which a pointcut evaluation is composed. Pointcut evaluations can consist of a composition of dynamic tests which need to be satisfied, for a given joinpoint, before the corresponding advice is executed. TOD-AOP can show the number of tests that were satisfied during a pointcut evaluation, but cannot refer to the source abstraction that corresponds to a test. If a test unexpectedly (dis)satisfied, the programmer must manually analyse the source code to locate the corresponding source abstraction. In ALIA4J, each test is represented as an *AtomicPredicate* entity of LIAM which contains the location of the respective source abstraction. This enables a debugger to refer to the source abstractions of each individual part of the pointcut evaluation. To use this feature in our approach, the front-end must become aware of the source abstractions that can be associated with the LIAM entities of the target program, this is further discussed in Section 7.2.6 of the next chapter.

### 6.1.2 Exhaustive trace approaches

IntelliTrace [39] is a commercial trace-based debugger that is developed by Microsoft, it is available in the Visual Studio IDE, and it can be used to debug .NET programs. IntelliTrace aims to record as little as possible, in order to impose minimal runtime overhead. Therefore, by default, only the following joinpoints are recorded: Exceptions, common .NET Framework events (e.g. mouseclick event). Furthermore, the program state is recorded when a statement is executed that contains a breakpoint. Then after execution, the programmer can step forward and backward between the recorded information and examine its content, e.g. values of local variables in a recorded program state. The disadvantage of this approach is that the trace merely contains information about source-code locations for which the programmer informed the debugger to record its runtime behaviour. While, in cause-effect chasm scenarios,

the programmer often cannot accurately point to a source-code location that is related to the root cause. Thus, the trace may not contain any information about the root cause. Consequently, the programmer may have to place new breakpoints, and record a new execution of the program. Optionally, the programmer can specify for which modules the functional behaviour must be recorded. This captures similar information as the *FunctionCallJoinPoint* and *FunctionExitJoinPoint* of our approach. However, the same problem arises because programmers need to explicitly mark for which modules this recording strategy is applied, thus assuming that the programmer has an idea which module contains the source-code location of the root cause.

Chronon [51], which is another commercial trace-based debugger, fully integrates with the Eclipse IDE[1], and can be used to debug Java programs. Chronon uses three key strategies to provide scalability and a low performance overhead:

- Firstly, Chronon records only the program itself, and nothing about third-party libraries or core classes of the Java programming language.
- Secondly, Chronon uses a *prediction strategy* to reduce the performance overhead, and limit the amount of information that needs to be collected during program execution. Before runtime, a static analysis of the program is performed. This analysis creates an initial set of predictions, consisting of the execution paths which the program most likely will take. During execution, Chronon only collects the runtime behaviour that differs from the initial prediction, and possibly adjusts the prediction set to take into account the new information. If the predictions match accurately, no data is collected at all, otherwise only the information that differs from the predictions is collected.
- Thirdly, Chronon uses so-called *Flusher Threads*. The collected information from the program is submitted to an in-memory buffer. Flusher threads, which are background processes, extract chunks from the buffer, process their content, and *flush* them to a trace file. This strategy can decrease the performance overhead because the execution of the program is hardly interrupted by the processing of trace information. However, this does require extensive hardware power. Chronon recommends to use a 64-bit system, assigning a minimum of 1GB to the JVM stack, and a multi-core processor. Furthermore, it recommends to optimize the multi-core processor such that there exists a balance between the execution of the program and the process of trace generation. Such optimization consists

[1] The Eclipse project, see http://www.eclipse.org.

of assigning an equal number of cores to flusher threads, and the application program, e.g. when having a quad processor and a program that spawns two threads, then two cores should be assigned to flusher threads.

ODB [32] and Unstuck [25] are trace-based debuggers for Java and Smalltalk programs respectively. These debuggers use an in-memory storage-model, i.e. they consume the RAM space that is assigned to the debugged program. Because the amount of memory is limited, the execution trace only holds the newest joinpoints that can be stored in-memory, and discards old ones. Such approaches work well regarding performance and scalability, because in-memory can process instructions fast and only a limited amount of information is stored at any moment in time. However, if the cause-effect chasm consists of a number of joinpoints that exceeds the available memory, the root cause is not present in the execution trace, hence the programmer cannot locate it.

Liebenhard et al. [36] propose a more recent work with a different in-memory approach, and also targets the Java programming language. Their trace collection strategy works at the level of the JVM, and keeps track of object changes in the same memory space as the program. This decreases the performance overhead compared to other approaches, because the trace data is collected at a lower level. The in-memory trace is cleaned up according to the garbage collection of the JVM. Thus, the trace contains no data about objects that are discarded as a result of garbage collection. However, program failures can be caused by objects that were already garbage collected before the symptom location was executed. In such a case the root cause is not present in the trace.

ZStep95 [35] is an early back-in-time debugger for the programming language Lisp. This debugger can present a graphical representation of the currently examined statement. Moreover, the state of the graphical representation is updated when the programmer steps back or forward through the trace. ZStep95 does not provide special features, other than stepping, to explore the trace. Scalability and performance issues are not addressed.

### 6.1.3 Replay-based approaches

Replay-based debuggers produce a trace by replaying the execution of the program between well-defined points, called checkpoints. The idea is to reconstruct the pro-

gram state, called a snapshot, at a given moment time by re-executing the program between two consecutive checkpoints. The advantage of the replay-based approach over the creation of an exhaustive trace is their low performance impact, as only non-deterministic operations need to be recorded. The disadvantages of this approach are (1) a slowdown of the navigation process because the debugger has to re-execute part of the program, (2) the evaluation of queries, with a relatively large search domain, can become slow because a large part of the program needs to be re-executed to generate the query result, and (3) replay cannot be guaranteed, depending on the behaviour of the program (e.g. if the program deletes a file from the HDD, then this often cannot be reconstructed). Xu et al. [57] have addressed the performance issue and propose an strategy, which is based on a static analysis of the program, to choose checkpoints efficiently, rather than periodically.

Bdb [9] and Igor [14] are recently proposed replay-based debuggers that take periodic checkpoints of the program execution. STIQ [47] is a replay-based debugger developed by the same authors as TOD. Their approach creates a *semi-exhaustive* trace consisting of data that is required for replay, and to apply an indexing scheme that guarantees good query response times.

EXPOSITOR [26] is a recent *scriptable debugger* that uses a replay-based approach. With a scriptable features of EXPOSITOR, the programmer can write Python scripts which perform complex debugging tasks. Traces are created via a replay-based approach, and are implemented as a set of first-class objects, e.g. a *snapshot* Python object. Scripts can therefore utilize the first-class objects of which the recorded trace consists. The key idea to execute these scripts efficiently is *laziness*, meaning that trace information is retrieved on-demand, thereby minimizing the requests for re-execution. An evaluation of EXPOSITOR's efficiency is however not presented.

The idea that drives scriptable debuggers is that debugging often involves a lot of manual repetition of common operations. These operations can be expressed in a script, which is reusable and can be executed by the debugger. This philosophy closely resembles that of our approach, as we offer programmers query templates that refer to debugging tasks which often occur in the context of AD. As in a scriptable debugging approach, the query templates can be provided as XQuery functions, which can be saved, distributed, reused and extended with expressions that refer to the specific problem domain of the program. However, a scripting language generally better fits the purpose of constructing automation processes than a query language. For example, a scripting language allows to define and maintain an auxiliary data structure which is

strictly used for the purpose of monitoring and debugging, while such a data structure can be more difficult to express in a query language such as XQuery.

Another interesting replay-based debugging approach is ReTrace [58], which is based on VMWare's virtual machine technology. This approach has an extremely low runtime overhead (commonly around 5%) and scales very well. ReTrace however does not provide features such as backward stepping, querying capabilities or trace visualization techniques.

## 6.2 Related work of the front-end

To our best knowledge, there exists only one trace visualization that depicts the AD behaviour in an overview of the execution history, which is TOD-AOP (also discussed in the previous section), and there exists no query-based debugging approaches with special support for AD. We therefore start with a section that discusses about the trace visualization of TOD-AOP, followed by two sections that discuss other existing trace visualizations and query-based debugging approaches respectively.

### 6.2.1 Trace visualization with AD support

TOD-AOP use the *information mural* to present a scalable summarized overview of the entire trace. The visualization has to be invoked explicitly, and highlights the activity of AspectJ aspects, that are selected by the programmer, with a colouring scheme. The horizontal axis of their mural view represents the execution time, and can be used for navigational purposes. Consequently, the execution time (represented as a timestamp) of a joinpoint is the only visual indication of its respective location in the trace. Furthermore, their colouring scheme for highlighting aspect activity does not address the situation where an advice execution encompasses another. For example, the execution of an around-advice can encompass the execution of other advices when calling *proceed()*.

Our trace visualization uses the tree-map layout, whereby two attributes indicate the location of a joinpoint in the trace: the execution time is reflected by the ordered layout of the tree-map, and the nested structures of the visualization conform to their respective execution paths in the call tree. The execution time of a joinpoint is often

meaningless to programmers, whereas the execution path that lead to the joinpoint can be much more informative, because it represents a sequence of function calls that was coded by the programmer. Furthermore, our trace visualization is implemented as a stand-alone debugging tool, rather than a supplementary feature. We employ several techniques to assist programmers in navigating the trace visually, i.e. interactive multi-staging and embedding summarized textual messages of the runtime information that is currently examined.

The colouring scheme of our trace visualization implements a *pro-active* strategy, rather than highlighting merely the activity of selected AD entities *on-demand*. This can be more useful during debugging because programmers are often unaware of the presence of AD activity, as shown by Ferrari et al. [15]. Furthermore, our trace visualization adopts the vision of execution levels, proposed by Tanter [52], to visualize encompassing AD activity appropriately.

### 6.2.2 Conventional trace visualizations

The work of Bohnet et al. [8] and De Pauw et al. [42] present trace visualization strategies that are based on call trees, similar to our approach. The strengths of both works lies in the powerful pattern recognition techniques to make repetitions of similar behaviour explicit. This can be used to prune information of the visualization, making it scale better. Bohnet et al. go one step further and classify the function calls that are present within the trace. They identified the characteristics of function calls that are most significant during trace exploration, and expressed these characteristics in a pruning algorithm which is applied by the trace visualization. This can massively prune the visualization and yet still allows programmers to efficiently navigate towards the low-level detail that they look for. Both works use a graph-like visualization layout.

Our approach however uses the tree-map layout, which occupies the available display space more efficiently than a graph layout, such that the visualization can present more information at a time. Furthermore, the tree-map layout can take an additional attribute into account which determines the sizes of the rectangles. Combining this attribute with the classification strategy of function calls by Bohnet et al. seems a promising strategy. For example, a function call that is classified as significant for trace exploration, can be represented by a relatively large tree-map rectangle.

The JIVE system [11] combines a query-based debugging approach with visualizations that depict the runtime behaviour of Java programs. This combination is similar to our approach. JIVE records the program's execution and provides two visualizations to depict the runtime behaviour. These visualizations are triggered when the programmer is stepping through the trace, and helps programmers to comprehend the runtime behaviour of the program. Firstly, an object diagram can show the relations between objects of a given runtime state. Secondly, a UML-like sequence diagram shows the interactions, regarding method calls, between the objects at runtime. This visualization provides several interactive features allowing the programmer to conveniently locate relevant joinpoints in the trace. Querying the execution history is supported by letting programmers express queries through a *search dialog*, where the relevant runtime information is filled in. Programmers therefore do not need to learn a query language, however the provided expressiveness of the *search dialog* is rather limited.

The trace visualization of our approach has the same goal of increasing program comprehension, but shows object interactions by presenting the call tree as a tree-map. Furthermore, our visualization is implemented as a stand-alone navigation tool to explore the trace, which is not possible with the visualizations provided by JIVE, because it primary realizes trace navigation through the common debugging stepping metaphors.

### 6.2.3 Query-based debugging approaches

Our front-end is not the first implementation that combines multiple trace-based debugging approaches. A recent debugger called JHyde [23] combines an omniscient debugging approach with a declarative debugging approach. The latter approach offers debugging by generating questions while the trace is explored with the navigational features of the omniscient debugger. These questions are related to the validity of the joinpoints that are currently examined. Based on the provided answers, JHyde can narrow down the location of the bug. The strength of this localization strategy depends strongly on the way questions are generated. The generated questions must reflect the programmer's goal, because JHyde does not allow to express an arbitrary query, as supported by our approach.

Lencevicius et al. [31] proposed the static query approach, where a query has to be formulated before runtime. The semantics of the query are woven into the program,

and evaluated during execution. There are several other query-based approaches that utilize this strategy, we discuss three:

- PQL [37] is a query language to match patterns against an abstraction of the trace. It offers a domain-specific syntax. A prototype of this language, implemented in Java, uses bytecode instrumentations that represent query evaluations. Several optimization techniques are applied to insert a minimal number of bytecode instrumentations, thereby imposing less runtime overhead.
- The PQTL [19] approach defines separate tables that store a specific joinpoint type, e.g. method invocation. The syntax of the *Structured Query Language* (SQL) is adopted such that relational queries can be expressed over the tables. In effect, SQL joins can be used to express powerful queries that target specific runtime information, e.g. the runtime behaviour that belongs to a certain execution path. Our approach also supports this, because of the call tree structure that is defined in the trace file, and the XQuery language which focuses on navigating a tree structure.
- Squirrel [55] is a query-based debugger for Java. An interesting implementation detail is that it uses AspectJ pointcut expressions to collect runtime information. The Java syntax can be used to express queries, and Squirrel provides a syntactical extension which refers to a commonly used concept in query languages, i.e. *forall*, which returns the set of joinpoints, of a certain search-domain, that satisfied an arbitrary condition. However, Squirrel only traces object creations, therefore only object relationships can be queried.

Unlike static querying, our approach allows to evaluate queries without having to re-execute the program, because the entire trace is captured. This can make the debugging process more efficient when the programmer is unable to express a query, that is powerful enough to locate the root cause, after the first time the program failure was observed. In such a situation, a static-based query approach requires the programmer to re-execute the program every time a new query is expressed, which can make the debugging process time-consuming.

Lencevicius et al. [30] proposed another query-based debugging approach called the dynamic query approach. Their implementation of this approach targets Java programs, and queries can also be expressed in the Java programming language. Thus, programmers do not have to learn a new language, as may be the case when

using our work. This query-based debugging approach lets programmers evaluate queries at runtime, i.e. whenever it is suspended by a breakpoint. This resolves the main limitation of a static query approach, however the search domain of the query consists of the program state at suspension, thus old object states are lost. Besides the information about the control flow of the program is not accounted for. Hence, programmers can only query about object relationships, and the values of variables of the suspended program state, while our approach allows to evaluate queries over the entire execution history and can target both data and control flow.

Mirghasemi [41] proposed an interesting dynamic query approach called *Querypoints*. When a breakpoint is hit, queries can be expressed in a domain specific language called *traceQuery*. This language provides several functions to locate a joinpoint, termed a *Querypoint*, that preceded the breakpoint and satisfies certain runtime conditions. The program is re-executed to evaluate a query, and the execution is suspended once a match is found. From there, a new query can be executed, thus allowing programmers to navigate backwards in time, up to the root cause. The advantage of *Querypoints* is that no trace information needs to be collected in order to provide features that are similar to backward stepping. However, the number of program executions is equal to the number of queries that need to be evaluated during a navigation process. This can slowdown the debugging process. Furthermore, the debugging strategy is dependent on the initial breakpoint location that has to be chosen manually, as with other dynamic query approaches.

JavaDD [18] is a declarative debugger which extends the previous JIVE system with a more expressive query-based approach. Prolog style expressions can be executed to query the execution history. This query language provides several special keywords that are related to commonly used requests in a query-based approach. The trace-model of JavaDD is a bit more fine-grained, regarding base-program joinpoints, than that of our approach. For example, JavaDD defines a separate entity that captures the start and stop of a thread object. Our trace-model refrains from defining such joinpoints separately, but rather allows to reconstruct the corresponding runtime information by querying. For example, an XQuery function can be provided that locates the joinpoint that invoked the *start()* method of a given thread object.

WhyLine [29] is a trace-based debugger for Java programs that creates an exhaustive trace, and employs a user-friendly strategy to locate suspicious runtime behaviour in the trace. This strategy consists of generating *why did* and *why didn't* questions about the program state that is displayed. The programmer can select one of these

questions such that the WhyLine jumps to the runtime behaviour that corresponds to the answer. Programmers thus do not have to learn a new query language, nor having to express it on another way. The downside of this strategy is that the generated queries may not always reflect the information that the programmer desires to locate. Moreover, in such a situation the programmer cannot express these desires to the WhyLine. Furthermore, WhyLine supports no additional features to explore the trace, other than using stepping and jumping to query answers. An interesting note about WhyLine is that it uses the same strategy that we proposed (Section 7.2.2) to support multi-threaded programs, i.e. every execution thread has a separate file that contains it's activity.

# 7 Conclusions and Future work

This chapter starts with presenting the conclusions that we have deduced from the design and development of our work. Next, we outline several candidate topics for future work. These topics discuss the open problems of our approach, potential features to further improve our approach, and suggest new directions for future research.

## 7.1 Conclusions

This thesis has shown that debugging scenarios, that result from a significant cause-effect chasm involving AD activity, cannot be resolved efficiently by approaches that are unaware of the AD concept, because AD activity is debugged in terms of base-program constructs. In effect, the influence of the AD activity throughout the program execution, and how it affected the base-program is not explicitly presented. This insight has driven the design and implementation of a trace-based debugging approach that copes with the special properties of AD. This approach makes it possible to efficiently locate the root cause in such debugging scenarios, and improve the overall comprehension of the developed system.

The design philosophy of our approach is primarily based on tracing AD programs close to the way they are expressed in the corresponding source code. This consists of tracing the execution of the base-program, AD activity, and the interplay between the base-program and AD entities. The latter two design attributes distinguish our approach from conventional trace-based debuggers, and makes it more suitable for debugging AD programs. Consequently, our approach increases the comprehension of AD programs, and assists programmers to efficiently resolve debugging scenarios that were caused by the unexpected behaviour of AD activity.

The back-end of our approach, consisting of a trace-model and storage-model, has been developed to capture the execution of an AD program. The trace-model represents a fine-grained representation of the runtime information that is collected during program execution, whereas the storage-model defines how this information needs to be structured and serialized in order to produce a trace. Both models define

separate entities for every language element of the AD concept that performs dynamic behaviour. Hereby, traces contain AD related information, which can be used to provide specialized features that assist programmers in comprehending AD specific program behaviour.

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. Therefore, the call-tree is structured according to the functional decomposition of the program, which makes the tree intuitive for the programmer regarding navigational purposes. The back-end produces an XML tree that reflects the call-tree representation. By doing so, existing query languages for XML that are designed to traverse tree structures, such as XQuery, can be used to retrieve and locate information in the trace. Consequently, the facilities of XQuery to traverse tree structures correspond to traversing a trace in a meaningful way, e.g. common debugging metaphors, which are often the only navigational features in existing debuggers, are supported in our approach by using the axes of XQuery to define a direction in which the call-tree must be traversed. We consider our efforts on this matter as a novel way to make use of existing technologies regarding tree structures in the field of trace-based debugging. Besides, XML provides convenient interoperability with future research projects, thereby increasing the potential reusability of our work.

The front-end of our approach can process a trace that is produced by the back-end and provides two debugging strategies to explore the trace. The software trace visualization strategy provides visual means to efficiently locate runtime behaviour that is relevant for a debugging scenario. Several features are included to inform programmers about the presence of AD activity while exploring the trace. The query-based debugging approach allows programmers to express complex questions about the trace in XQuery. The XQuery expressions of common debugging metaphors are offered by special commands that can be used in the query editor of the front-end. The corresponding XQuery expressions of these commands can be extended with arbitrary XQuery predicates, which contain boolean expressions that are related to the problem domain of the program. As such, more efficient navigational steps can be expressed by programmers, without requiring expert knowledge of the XQuery language. Furthermore, based on the AD related information in the trace, queries can be formulated to track down the runtime behaviour of AD entities and their interplay with the base-program, as shown by the query templates that were presented in this thesis. This helps programmers to make powerful navigational steps in order to track

down the root cause in either of the debugging scenarios that originally motivated our work.

The performance impact of the back-end has been extensively measured by applying a statistically rigorous benchmark approach. These measurements showed that parts of the trace-model are competitive to several existing omniscient debuggers. Furthermore, it was shown that the implementation of the storage-model, which constructs an XML file of the trace in a custom way, is the main bottle-neck of our approach regarding performance. Despite the performance overhead that is imposed, the benefits of our approach in resolving hard-to-find bugs in AD programs can outweigh this inconvenience. Furthermore, a case-study has been designed to assess if our approach works well for debugging scenarios where the cause is related to AD activity. This case study consisted of a program where the runtime behaviour of different AD entities were the cause of several program failures, and the symptoms of these failures were observable much later. The case study showed that our approach provides convenient navigation towards the cause of the program failures, because no re-execution of the program was needed. Suspicious source-code locations did not have to be guessed by the programmer, and a relatively low number of navigational steps were needed because of the query-based debugging approach. Furthermore, the case study showed that our approach can improve program comprehension while one utilizes its navigational means.

We believe that trace-based debugging approaches are very promising to tackle the complex debugging scenarios that may arise from the AD concept. Our approach showed that several trace-based debugging strategies can be extended to support the AD concept. This leads to more powerful ways to localize and comprehend AD activity of the program. We hope that future trace-based debuggers will integrate similar functionalities to cope with AD programming languages, thereby enriching the offered tool support to programmers and facilitate the robust development of AD programs.

## 7.2 Future work

The implementation of the back-end, and the usability of the front-end presented in this thesis are still subject to further improvements. Note that without these improvements our work is fully operational as described in this thesis. Several future

improvements and topics for future research projects, that are based on the work presented in this thesis, are discussed in the following sections. Per improvement, we describe a suggestion or a potential solution, if available, to support it with our approach. It is important to point out that these suggestions are based on our personal knowledge of the approach and underlying technologies; a more thorough study is required to get a reliable indication about the validity, relevance and importance.

### 7.2.1 Improving the indication of the trace location in the visualization

The visualization requirement: *orientation should be recoverable at every point in the space*, is currently supported by the hierarchical way in which the data is shown, i.e. the root of the trace visualization, in a particular stage, represents the call to which the programmer explicitly navigated. However, any previous navigation steps, that were performed by the programmer, are not visualized. This can make it hard for the programmer to deduce the location in the trace when she followed a relatively long execution path in the trace.

**Suggested solution**

To deal with the aforementioned situation, the visualization should keep track of the navigational steps that the programmer performed while exploring the trace. Based on this information, the visualization can present the programmer, at any moment in time, what the execution path was that lead to the currently displayed stage.

Figure 7.1 presents a snapshot of the trace visualization where a text area is placed above it ①. This text area outlines the navigational steps that were taken until the currently displayed stage. When the programmer is inspecting low-level detail, e.g. a *getter()* method as in the figure, the text area shows the execution path that lead to the currently displayed stage. In this way the programmer does not feel lost, because there is always an indicator that presents the path that lead to the currently displayed information.

### 7.2.2 Supporting multi-threaded programs

The trace-model supports multi-threading, because it defines an attribute that corresponds to the ID of the thread that executed the joinpoint (See Figure 3.1). However,


**Figure 7.1:** Trace visualization that presents the navigational steps ①.

the storage-model does not deal with multi-threaded programs well, because it maintains a single call tree representation of the entire program, while each thread yields a separate call tree. Thus, the current implementation of the storage-model tangles the joinpoints of mutliple threads into a single XML file, which may lead to an ill-formed XML state.

**Suggested Solution**

A solution to make our work fully compatible with multi-threaded programs is to create a separate XML file for each thread of the program. By doing so, the call trees of different threads do not tangle. This strategy is also employed by the trace-based debugger called WhyLine [29].

The implementation of this solution does not encompass a lot of programming effort to the back-end, however the front-end needs several adjustments. Most importantly, the trace visualization must be able to display multiple call trees and yet be informative to the programmer. Furthermore, the query-based debugging approach must become aware of the multi-threaded situation, because the programmer may write a query that matches joinpoints in multiple threads, while she is interested in only one particular thread. Thus, it must be explicitly shown that certain query results are part of a particular thread.

### 7.2.3 Improving the performance of the back-end

The evaluation of the back-end showed that the bottlenecks, which mainly cause the performance overhead, are NOIRIn's interpretation process and the implementation of the storage-model. These bottlenecks can be resolved such that the implementation of our approach becomes applicable for relatively large applications.

**Suggested Solution**

There may exist several low-level optimizations that can be carried out on the implementation of the back-end. Apart from these low-level optimizations, we describe several suggestions to improve the performance of the back-end briefly.

- Offering an additional scoped tracing strategy. Currently, the implementation of the trace-model supports one strategy to create a scoped trace, i.e. by defining (a priori) which joinpoint types must be captured. This can be helpful when the programmer only wants to debug the control or data-flow of the program. However, there may be other situations when certain program activity can be omitted from the trace. The programmer may be certain, based on knowledge of the program, test results, or previous debugging sessions, that certain classes, or even entire packages, of the program are unrelated to the debugging task at hand. Therefore, the trace-model does not have to record the activity of these classes, while still providing a trace that contains sufficient information for the debugging task. This can significantly lower the performance overhead, because only certain parts of the program have to be captured by the trace-model.
- Bypassing the performance overhead of NOIRIn by implementing our approach in combination with a different execution environment for ALIA, e.g. Steamloom$^{ALIA}$.
- Constructing more compact representations to write out joinpoints. The implementation of the storage-model creates an XML representation of a joinpoint, which contains redundant data (e.g. opening and closing tags). Though an XML representation provides interoperability with third parties, the number of bytes that needs to be written out per joinpoint is relatively large, as was shown during the evaluation of the back-end. Thus, when aiming for a practical implementation of our approach, the storage-model can be optimized by generating compact representations of joinpoints.

### 7.2.4 Providing a query library

In order to prevent programmers from having to express common debugging operations over and over again, an XQuery library module can be constructed. This library can contain XQuery functions for the query templates that were presented in this thesis, and other functions that target different kinds of debugging operations. This relieves programmers from expressing potential complex XQuery expressions themselves, and enables programmers which are less experienced with XQuery, to facilitate the query-based debugging approach.

During our work we have started with developing an XQuery library that contains several query templates as functions. However, a thorough study has to be conducted to identify the questions that often arise during debugging. Based on these questions, a corresponding XQuery function can be created and added to the library, making it more complete.

### 7.2.5 Further improve the navigability and scalability of the trace visualization

A program trace can contain information that can be related to repetition, e.g. a *for-loop* that iterates thousands of times. Such information may occupy a significant amount of display space in the trace visualization. This can lead to navigational problems when the programmer is trying to locate the joinpoint that executed just before/after the loop entered/exited, because the display space of the trace visualization is filled with information of the loop. Bohnet et al. [8] proposed several pruning techniques to deal with such scenarios. The techniques that they propose are based on a graph visualization of the trace. It may be interesting to study if these techniques can be adopted to a tree-map layout. For example, by pruning the joinpoints that belong to a *for-loop* into a single tree-map rectangle, which contains a label with an informative message such that the programmer is aware of the pruning.

### 7.2.6 Extending our approach to support concrete AD language

There are two challenges that need to be addressed to support a concrete AD programming language, e.g. AspectJ. Firstly, the current implementation of our approach

debugs AD activity in terms of the intermediate representation, i.e. the LIAM entities, and is therefore unaware of the source abstractions that defined the AD activity. ALIA-TBD can therefore not refer to the actual source abstractions when inspecting joinpoints that are related to AD activity.

The second challenge is related to the query-based debugging approach. Currently, the query templates are based on the terminology and concepts of ALIA. In effect, the query-based debugging approach requires programmers to be familiar with this terminology, and have detailed knowledge about the LIAM constructs which represent the actual source abstractions. Otherwise, it becomes difficult for programmers to express a query that was based on the actual source code. For example, each pointcut-advice pair in an AspectJ aspect corresponds to an *Attachment* in LIAM. When the programmer wants to locate all program activity that is related to an aspect, she must find out which attachments can be associated with the aspect, and query the activity for each such attachment in the trace. This example showed that there exists a mismatch between the debugging questions that arise from the source code, and the way they have to be expressed by the query-based debugging approach.

**Suggested solution to address the first challenge**
The work of Yin et al. [61] have described the implementation of a breakpoint-based debugger, which is also based on the ALIA4J framework, and which offers debugging in terms of the actual source abstractions of AspectJ. Their work generates an XML file, at compile time, that reflects the relation between the LIAM entities and the AspectJ source code. In effect, when AD activity is inspected at a breakpoint suspension, the debugger can refer to the actual source abstraction by consulting this XML file. When developing an extension to our approach to support AspectJ programs, it seems promising to reuse the strategy of Yin et al.

**Suggested solution to address the second challenge**
To address this challenge, the query-based debugging approach must be extended to express queries which closely resemble the debugging questions that arise when using a certain AD language. This can be accomplished by providing an XQuery library module for a specific AD programming language. This library can contain functions that correspond to the concepts and terminology of the AD programming language. For example, the library can contain an XQuery function called `AspectJ:locateAspectActivity($aspectName)` to locate the activity of an aspect in AspectJ, where `$aspectName` represents the aspect's name that is defined in the source code.

### 7.2.7 Conducting a user experiment

The evaluation of the front-end of our approach is based on a case study, which may not indicate strongly if ALIA-TBD outperforms existing debugging approaches for the debugging scenarios that were illustrated during this evaluation. An user experiment can produce more conclusive evaluation results, give an indication of the usability of ALIA-TBD based on feedback, and offer insight into the strategies that programmers utilize to resolve bugs.

# Bibliography


[1] World wide web consortium. extensible markup language (xml) 1.0 (fifth edition). http://www.w3.org/TR/REC-xml, 2008.

[2] Ken Arnold and James Gosling. *The Java programming language (2nd ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[3] Benjamin B. Bederson, Ben Shneiderman, and Martin Wattenberg. Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies. *ACM Trans. Graph.*, 21(4):833–854, October 2002.

[4] C. M. Bockisch and A. Sewe. Generic ide support for dispatch-based composition. In *Proceedings of the First International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines, Rennes, France*, volume 564 of *CEUR Workshop Proceedings*, Aachen, 2010. M. Jeusfeld c/o Redaktion Sun SITE, Informatik V, RWTH Aachen.

[5] Christoph Bockisch and Mira Mezini. A flexible architecture for pointcut-advice language implementations. In *Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, VMIL '07, New York, NY, USA, 2007. ACM.

[6] Christoph Bockisch, Andreas Sewe, Haihan Yin, Mira Mezini, and Mehmet Aksit. An in-depth look at alia4j. *Journal of Object Technology*, 11(1):1–28, April 2012.

[7] Christoph-Matthias Bockisch. *An Efficient and Flexible Implementation of Aspect-Oriented Languages*. PhD thesis, Technische Universität Darmstadt, Germany, July 2008. This thesis is accredited by the European Network of Excellence on Aspect-Oriented Software Development.

[8] J. Bohnet, M. Koeleman, and J. Doellner. Visualizing massively pruned execution traces to facilitate trace exploration. In *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on*, pages 57–64, 2009.

[9] Bob Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 299–310, New York, NY, USA, 2000. ACM.

[10] D. Chamberlin. Xquery: An xml query language. *IBM Syst. J.*, 41(4):597–615, October 2002.

[11] Jeffrey K. Czyz and Bharat Jayaraman. Declarative and visual debugging in eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, eclipse '07, pages 31–35, New York, NY, USA, 2007. ACM.

[12] Marc Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, April 1997.

[13] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 186–211, London, UK, UK, 1998. Springer-Verlag.

[14] Stuart I. Feldman and Channing B. Brown. Igor: a system for program debugging via reversible execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, PADD '88, pages 112–123, New York, NY, USA, 1988. ACM.

[15] Fabiano Ferrari, Rachel Burrows, Otávio Lemos, Alessandro Garcia, Eduardo Figueiredo, Nelio Cacho, Frederico Lopes, Nathalia Temudo, Liana Silva, Sergio Soares, Awais Rashid, Paulo Masiero, Thais Batista, and José Maldonado. An exploratory study of fault-proneness in evolving aspect-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 65–74, New York, NY, USA, 2010. ACM.

[16] Mark A. Foltz. Designing Navigable Information Spaces, 1998.

[17] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, October 2007.

[18] H. Z. Girgis and B. Jayaraman. Javadd: a declarative debugger for java. Tech. Report 2006-7, Department of Computer Science and Engineering, University at Buffalo, 2006.

[19] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. *SIGPLAN Not.*, 40(10):385–402, October 2005.

[20] Christian Grün. *Storing and querying large XML instances*. PhD thesis, 2010.

[21] B. Hailpern and P. Santhanam. Software debugging, testing, and verification.

*IBM Syst. J.*, 41(1):4–12, January 2002.

[22] Elliotte Rusty Harold. *Effective XML: 50 Specific Ways to Improve Your XML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[23] Christian Hermanns and Herbert Kuchen. Hybrid debugging of java programs. In MaraJos Escalona, Jos Cordeiro, and Boris Shishkov, editors, *Software and Data Technologies*, volume 303 of *Communications in Computer and Information Science*, pages 91–107. Springer Berlin Heidelberg, 2013.

[24] Christoph Hofer. Implementing a backward-in-time debugger, 2006.

[25] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In Robert Hirschfeld, Andreas Polze, and Ryszard Kowalczyk, editors, *NODe/GSEM*, volume 88 of *LNI*, pages 17–32. GI, 2006.

[26] Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. Expositor: scriptable time-travel debugging with first-class traces. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 352–361, Piscataway, NJ, USA, 2013. IEEE Press.

[27] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[28] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[29] Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[30] Raimondas Lencevicius. On-the-fly query-based debugging with examples. In *AADEBUG*, 2000.

[31] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *OOPSLA*, pages 304–317, 1997.

[32] Bil Lewis. Debugging backwards in time. *CoRR*, cs.SE/0310016, 2003.

[33] Henry Lieberman. Reversible object-oriented interpreters. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '87, pages 11–19, London, UK, UK, 1987. Springer-Verlag.

[34] Henry Lieberman. The debugging scandal and what to do about it (introduction to the special section). *Commun. ACM*, 40(4):26–29, 1997.

[35] Henry Lieberman and Christopher Fry. *ZStep95: A reversible, animated source code stepper.*, pages 277–292. 1998.

[36] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 592–615, Berlin, Heidelberg, 2008. Springer-Verlag.

[37] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. *SIGPLAN Not.*, 40(10):365–383, October 2005.

[38] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. pages 2–28. Springer-Verlag, 2003.

[39] Microsoft. Debug your app by recording code execution with intellitrace. `http://msdn.microsoft.com/en-us/library/dd264915.aspx`.

[40] Todd Millstein, Christopher Frost, Jason Ryder, and Alessandro Warth. Expressive and modular predicate dispatch for java. *ACM Trans. Program. Lang. Syst.*, 31(2):7:1–7:54, February 2009.

[41] Salman Mirghasemi. Query-point debugging. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 763–764, New York, NY, USA, 2009. ACM.

[42] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John M. Vlissides, and Jeaha Yang. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, London, UK, UK, 2002. Springer-Verlag.

[43] J.-Hendrik Pfeiffer and John R. Gurd. Visualisation-based tool support for the development of aspect-oriented programs. In *Proceedings of the 5th international*

*conference on Aspect-oriented software development*, AOSD '06, pages 146–157, New York, NY, USA, 2006. ACM.

[44] Guillaume Pothier. Benchmarks of cots database management systems. Technical report tr/dcc-2006-16, University of Chile, October 2006.

[45] Guillaume Pothier. *Towards Practical Omniscient Debugging*. PhD thesis, University of Chile, June 2011.

[46] Guillaume Pothier and Éric Tanter. Extending omniscient debugging to support aspect-oriented programming. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 266–270, New York, NY, USA, 2008. ACM.

[47] Guillaume Pothier and Éric Tanter. Summarized trace indexing and querying for scalable back-in-time debugging. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 558–582, Berlin, Heidelberg, 2011. Springer-Verlag.

[48] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *SIGPLAN Not.*, 42(10):535–552, October 2007.

[49] A. Sewe, C. M. Bockisch, and M. Mezini. Redundancy-free residual dispatch: using ordered binary decision diagrams for efficient dispatch. In *Proceedings of the 7th Workshop on Foundations of Aspect-Oriented Languages, Brussels, Belgium*, pages 1–7, New York, April 2008. ACM.

[50] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11(1):92–99, January 1992.

[51] Chronon Systems. Chronon, a dvr for java. `http://chrononsystems.com/`.

[52] Éric Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 37–48, New York, NY, USA, 2010. ACM.

[53] Marnix van't Riet, Haihan Yin, and Christoph Bockisch. Research topics: Omniscient debugging for advanced-dispatching programming languages. Technical report, University of Twente, 2012.

[54] Marnix van't Riet, Haihan Yin, and Christoph Bockisch. The potential of omniscient debugging for aspect-oriented programming languages. In *Proceedings of the 1st workshop on Comprehension of complex systems*, CoCoS '13, pages 13–16,

New York, NY, USA, 2013. ACM.

[55] Darren Willis, Supervisors James Noble, and David J. Pearce. Squirrel: A query based debugger for java, 2005.

[56] W. Eric Wong and Vidroha Debroy. Software fault localization. *IEEE Reliability Society 2009 Annual Technology Report*, 59(3), September 2010.

[57] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 85–94, New York, NY, USA, 2007. ACM.

[58] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, Boris Weissman, and Vmware Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*, 2007.

[59] H. Yin, C. M. Bockisch, M. Akşit, W. De Borger, B. Lagaisse, and W. Joosen. Debugging scandal: The next generation. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE 2011), Zurich, Switzerland*, page 1, York, UK, June 2011. University of York.

[60] Haihan Yin. Developing a generic debugger for advanced-dispatching languages, August 2010.

[61] Haihan Yin, Christoph Bockisch, and Mehmet Aksit. A fine-grained debugger for aspect-oriented programming. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 59–70, New York, NY, USA, 2012. ACM.

[62] Sai Zhang and Jianjun Zhao. On identifying bug patterns in aspect-oriented programs. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '07, pages 431–438, Washington, DC, USA, 2007. IEEE Computer Society.