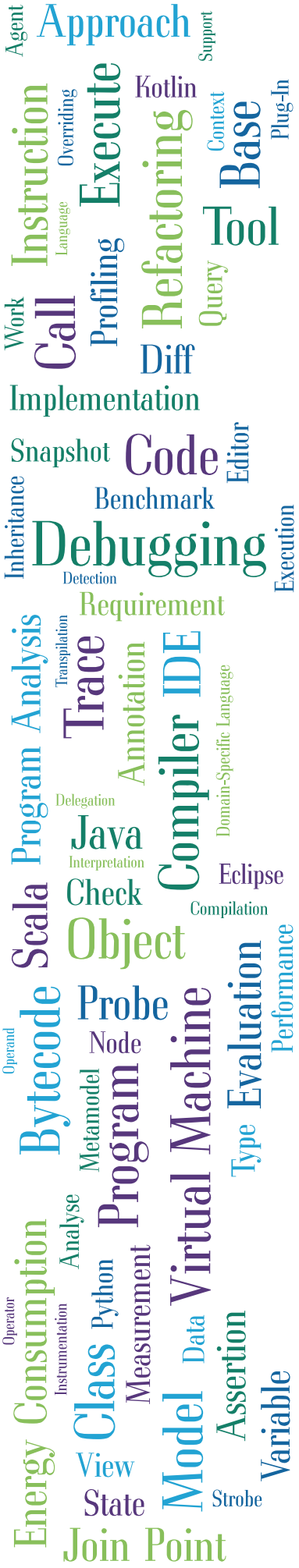

Philipps-Universität Marburg

Faculty of Mathematics and Computer Science

An Efficient Domain-Specific Language For Breakpoints

Freya Dorn

Bachelor Thesis

December 23, 2021

Supervisor:
Prof. Dr.-Ing. C. Bockisch
Programming Languages and Tools Group

Advisor:
M.Sc. Stefan Schulz
Programming Languages and Tools Group

Institution

Philipps-Universität Marburg
Faculty of Mathematics and Computer Science
Programming Languages and Tools Group
Hans-Meerwein-Str. 6
35043 Marburg
Deutschland

About the author

Freya Dorn is a computer science student with an interest in linguistics, programming language design, and bioinformatics. You can reach her under freya.siv.dorn@gmail.com.

License

Classification (ACM CCS 2012)

- **Applied computing** → **Education**; **Document preparation**;
- **Software and its engineering** → *Software notations and tools*;

Keywords

Breakpoints, Debugging, Domain Specific Languages, Java programming language, Xtext

Acknowledgment

I would like to thank my supervisor Prof. Dr.-Ing. Christoph-Matthias Bockisch for his continuous encouragement during the many roadblocks I encountered while writing this bachelor thesis. His helpful feedback kept me on track when my work schedule was suffering the most.

I would also like to thank Anaïs and Elena for their ceaseless support and love. Thank you for giving me stability and acceptance that I've never had in my life before.

Finally, I want to acknowledge Rasputin for helping me through many long coding sessions.

Though she'd heard the things he'd done
She believed he was a holy healer
Who would heal her son

Rasputin, Boney M. (1978)

Abstract

Breakpoints are a powerful tool for debugging programs. Real-world debuggers support breakpoints through a wide range of divergent and non-composable commands with no agreed-upon shared notation across independent debugging tools.

We developed a unified domain-specific language for breakpoints together with an efficient backend implementation.

We followed a set of design principles to develop an expressive breakpoint language for the Java programming language. We designed an ergonomic language for all commonly used types of breakpoints in a way that closely follows the language conventions of Java.

Furthermore, we implemented an efficient automaton and backend for the breakpoint language as an extension of the instrumentation framework used by Bockisch et al. in their unified specification of breakpoints.

The breakpoint language and its efficient implementation largely meet our initial goals in terms of expressive power, ergonomics, and practical performance.

Zusammenfassung

Haltepunkte (Breakpoints) sind ein wichtiger Bestandteil des Debuggens von Software. In der Praxis unterstützen Debugger Haltepunkte durch ein Sammelsurium unterschiedlicher und nicht kombinierbarer Befehle, denen zudem eine gemeinsame Notation zur Nutzung in voneinander unabhängigen Debugging-Werkzeugen fehlt.

Das Ziel dieser Arbeit war es, eine einheitliche domänenspezifische Sprache für Haltepunkte in Kombination mit einer effizienten Backend-Implementierung zu entwickeln.

Anhand von ausgewählten Designprinzipien wurde eine ausdrucksmächtige Haltepunktsprache für die Programmiersprache Java entwurfen. Diese ergonomische Sprache deckt alle gängigen Arten von Haltepunkten ab und orientiert sich dabei eng an den in Java üblichen sprachlichen Konventionen.

Darüber hinaus wurden ein effizienter Automat swoie ein Backend für die Haltepunktsprache als Erweiterung eines Instrumentierungsframeworks implementiert, welches bereits von Bockisch u. a. in deren vereinheitlichter Spezifikation von Haltepunkten verwendet wurde.

Die Haltepunktsprache und ihre effiziente Implementierung erfüllen weitestgehend die ursprüngliche Zielsetzung in Hinblick auf Ausdrucksmacht, Ergonomie und Laufzeitverhalten.

Contents

1 Introduction

Debugging is one of the core aspects of developing reliable software. It is the task of finding and understanding bugs so that they can be fixed. To facilitate that task, dedicated debugging tools are used to control the execution of programs, to inspect and modify their state during runtime, and to collect and analyze additional information about programs.

Popular debugging tools are standalone debuggers, such as GDB[gdb] and WinDbg[win], and integrated debuggers in sophisticated IDEs, such as in Eclipse[ecl] or Visual Studio Code[vsc]).

In order to inspect a program, programmers can use a debugger to pause the execution of the program at certain points. The debugger will also expose the current state of the program, such as the values of variables or the current function call stack.

Breakpoints are specific conditions that define points during the execution of a program when the execution should be suspended. Breakpoints are commonly specified by stating a position in the source code, typically in the form of a file name and line number. The task of the debugger is then to pause execution whenever the instruction that corresponds to that source code position is being executed. Modern IDEs often provide a graphical interface to select lines and mark them as breakpoints for the debugger.

There are many related concepts in common use today, such as watchpoints[wat] and tracepoints[tra]. Watchpoints are breakpoints that monitor the current value of a given variable or field and suspend execution if the value matches a given condition. Tracepoints do not interrupt execution, but log execution points and values during the execution when a condition is fulfilled, creating a specific execution trace.

For the purpose of this paper, watchpoints, tracepoints, and similar concepts can all be considered special cases of breakpoints. They differ either in their specific condition (eg. the current source line number, value assignments to certain variables, the type of an exception, etc), or in what specific action the debugger is to take when the condition is met. Typically, the debugger would either suspend execution or log the condition.

Currently, there are no unified interfaces or protocols for debuggers. Each editing environment and each debugging tool implements its own set of tools and conventions. Common forms of breakpoints are implemented with disparate interfaces. For example, marking a line as a breakpoint and triggering a breakpoint based on the value of a global variable are two completely separate tasks in most common debuggers that use different commands and can't be easily combined. One might involve clicking on a line with the cursor and selecting a breakpoint option, the other might require entering a variable name and conditional value into a separate debugging pane.

Additionally, there are no established standards to exchange breakpoint specifications between different tools or often even any means to share them between programmers using the same set of tools. Debuggers with textual interfaces such as GDB allow sharing commands between users in an ad-hoc fashion, but graphical debuggers rarely allow reading and writing specific breakpoint conditions to a shared file, for example.

Standardizing common programming tools is very useful, as it concentrates development efforts into a single project and makes it easier for programmers to collaborate across different development environments. It also unifies the concepts involved and may lead to more powerful tools through abstraction that have a significantly lower learning curve because of the smaller number of necessary concepts.

Two prominent examples are the standardization of data exchange formats, such as with XML[BPSM+00] and JSON[Bra17], and the development of a standardized language server protocol (LSP[lsp]) to provide semantic information to the IDE in the form of better type information, navigation for function definitions, improved semantic markup, and similar tools.

These standardization efforts are generally agnostic of the specific tools that consume or implement them, so that they can be employed in a wide range of tooling. This also allows programmers to use the same standardized approach in different domains possibly written in different languages and developed with different editing environments.

Our goal in this thesis is to develop a unified domain-specific language (DSL) to specify breakpoints and to implement a backend for this language that can be used by a debugger.

Our breakpoint DSL builds on the unified approach to breakpoints introduced by [BSWK21]. Our backend is implemented as an extension to [BSWK21]'s existing instrumentation framework for the Java programming language and Java Virtual Machine (JVM). Because of that, we focus our efforts only on developing a breakpoint DSL for programs written in Java.

We first decided on design principles to aid the development of the DSL.

We then developed a grammar using the Xtext language workbench[EB10] and decided on specific breakpoint semantics according to the design principles. We used the textbooks «DSL Engineering» [VBD+13] and «Implementing Domain Specific Languages with Xtext and Xtend» [Bet16] as our guides for the design and implementation of the Query DSL.

Finally, we implemented an automaton and backend on top of the instrumentation framework that consumes our breakpoint DSL.

The backend significantly outperforms the XML-based implementation in [BSWK21] and provides a powerful, human-readable language for breakpoints.

2 Design Principles

Before designing the domain-specific language, we assembled several principles to guide the design process. We used the requirements implied by the goal of developing an efficient backend for an existing framework as a major guideline as to what principles to focus on.

2.1 Event-based

Breakpoints can be understood as the combination of an event, such as executing an instruction corresponding to a particular line number, changing the value of a variable, or throwing an exception, plus a condition related to the event attributes, the program state and the execution history that determines whether the given event should trigger a breakpoint or not. The same breakdown into a base event and a condition was taken by [BSWK21].

Because we agree with their core analysis of the breakpoint domain, and we are extending the authors' existing instrumentation framework, we followed their decision to design breakpoints around base events and conditions.

[BSWK21] proposes the following seven base events:

Table 1 Base Events used by [BSWK21]

kind	parameters
line	file name, line number
change	field or local variable
read	field or local variable
method enter	method
method exit	method
class load	class
exception	type

Conditions can refer to any of the parameters of the base event, and any component of the program state, such as the value of a variable, the name of the current method call, or the type of a value. They can also refer to any past or present event in the execution history.

[BSWK21] modeled the execution of a program as an event stream and breakpoints as search queries over that stream. We followed the same approach and therefore continue to refer to the combination of a base event and a condition as a query.

2.2 Runtime-based

The base implementation we are aiming to extend uses the instrument framework DiSL[MVZ+12] to generate the instrumentation. DiSL is an «AOP-inspired domain-specific language for Java bytecode instrumentation»[DiS]. The existing implementation generates events in the form of callbacks that our backend will later use to implement the automaton for the Query DSL.

Some debugging approaches, such as the macro-based facilities in Common Lisp[CLB], modify the target source code. That way, the debugging tools are integrated into the compilation of the program and can influence the resulting program directly. For example, the programmer might place a call to the **break** macro in any place they want to suspend execution.

However, that approach requires the recompilation of the program whenever the programmer wants to set new breakpoints or modify old ones. This slows down the debugging experience and makes it considerably more difficult to debug existing programs without access to the source code or an iterative compiler.

Because of the design of the instrumentation framework and the disadvantages of a compilation-based debugger, we chose an approach that does not modify the program and instead relies purely on runtime data.

In order to allowing debugging a running program without modifying its source code, the debugger must be able to access certain information about the execution flow and the program state, such as the names and values of variables, the current call stack, and a mapping of instructions to locations in the source code. Modern programming languages commonly use managed runtimes, such as the Java Virtual Machine (JVM)[SBS01] or Common Language Runtime (CLR)[MWG01], which expose that information to the program, or can at least be compiled in a debug mode to do so.

As we restricted our scope for this thesis entirely to the Java programming language, we can take the JVM and its means of reflection for granted. Additionally, the instrumentation framework already provides nearly all the remaining necessary information such as a mapping of events to source code line numbers.

Because the Query DSL only uses runtime information and does not directly interfere with the existing program, it is also possible to enable or disable queries, to add new ones during debugging[1], and to replace existing ones.

As we rely on runtime reflection to access the state of the program, in particular the values associated with variable names, we chose to limit the Query DSL to static fields. We chose not to support instance fields for reasons we discuss in ‹Variables› (section 3.2).

[1] If the programmer defines new breakpoints during the execution of the program, our implementation will currently not be able to make the past execution history available to the query. A query starts from scratch the moment it is defined. Otherwise, a full execution history would need to be kept and that would be a violation of the efficiency principle we will introduce later.

The DiSL framework provides events corresponding to local variables, but it does not expose the full lexical scope to distinguish multiple local variables of the same name reliably. Additionally, there is no agreed-upon or intuitive way to refer to a particular local block scope in a textual language[2] like Java, so it is unclear how to explicitly refer to a local variable by name. This problem can often be worked around, however, by using the line number in the query. ‹Variables› (section 3.2) also discusses the problems with local scope in more detail.

2.3 Ergonomic for Programmers

The primary use case of the Query DSL is to serve as a language for a debugging backend. However, that language should still be usable directly by a programmer to define breakpoints by specifying a condition based on the program state and execution flow.

We do not want to design the Query DSL exclusively for the internal use of a programming environment or as a machine-readable exchange format. Otherwise, the programmer would not be able to specify custom breakpoints that go beyond the commands provided by the IDE. We explicitly want to empower the programmer directly, even though we would still expect common debugging tasks to use a graphical interface in addition to explicit queries written in the Query DSL.

Furthermore, if the Query DSL were not easy for a programmer to read, then the ability to share breakpoint queries between programmers would be much more limited. The programmer would not be able to easily infer the behavior of a shared breakpoint.

Therefore the DSL should be human-readable and ergonomic to write. While IDEs will surely want to provide additional graphical means to specify breakpoints, such as right-clicking on a line and selecting a **breakpoint** option, all the provided functionality should nonetheless by independent of the specific tooling and unified in its presentation.

To reduce the effort of implementing a domain-specific language, we used an existing language workbench. We chose a textual DSL implemented in the Xtext framework[EB10]. Xtext is a popular language workbench for the Eclipse Project[ecl]. One major advantage of using Xtext is that we can use Xtext's parser generator based on ANTLR[PQ95] to generate a parser for the grammar specification discussed in ‹Query DSL› (section 3). Additionally, the integrated Eclipse Modeling Framework[SBPM09] is a good foundation for our data model.

By using a textual DSL, all breakpoints are specified in the same format and can be saved and exchanged in files or through text-based channels. The same approach is used by the popular language-server protocol[lsp]. Additionally, Xtext provides full editor integration for syntax highlighting, error checking, semantic completion, and means for further

[2] Structural editing environments such as JetBrains MPS[Vö11] and Lamdu[lam] provide more elegant ways to reliable refer to particular variables, such as through direct references to nodes in the structural source code.

integration into the graphical programming environment. Most of those capabilities are beyond the scope of this thesis, but will hopefully be useful for future extensions of the breakpoint backend.

In order to be most ergonomic for the programmer, we aimed to minimize the amount of new and unfamiliar syntax that must be learned. We also followed established industry and language conventions whenever possible.

‹Query DSL› (section 3) will provide a detailed breakdown of the Query DSL, but to illustrate the basic ergonomic principle, consider the following queries and their representation in the DSL:

Table 2 Simple Queries

Plain Text	Query DSL
break when the variable X is larger than 10	x > 10
break when obj is of type SubClass	obj instanceof @SubClass
break when the method print is called	print()
break when a FileNotFound exception is thrown	throw @FileNotFound

2.4 Abstraction

Like pattern matching for conditional expressions, we want to provide additional abstractions to allow the programmer to specify more generic patterns of queries.

In particular, we wanted to be able to express breakpoints that use more than just literal values and variable names. We were inspired by the declarative approach used in pattern-matching[Tur76] to express conditionals and to destructure data (‹Simple Queries› (table 3)).

We solved the problem by introducing query variables, analogue to binding in pattern matching[BMS80]. We will discuss their syntax and design in ‹Query Variables› (section 3.8).

Table 3 Simple Queries

Plain Text	Query DSL
break when the method foo is called with three arguments of the same value	foo(?=x, ?x, ?x)
break when the method g is called with the same value as the method f before it	foo(f(?=x) then g(?x)

We also looked for ways to organize and reuse breakpoint queries to enhance the debugging interface and to help the exchange of breakpoints between different environments. ‹Query

Organization› (section 3.6) will introduce two tools for that purpose – named queries and a query file.

Lastly, we also wanted to be able to use the unified breakpoint DSL to express more complex breakpoints that are currently often beyond the scope of debugging tools. Most importantly, we needed a way to express conditions that depend on the execution flow and refer to past events. We also needed a way to combine queries into larger compound queries.

The Query DSL models this aspect by logically combining simple queries through connectives (‹Compound Queries› (table 4)).

Connectives are the subject of ‹Connectives› (section 3.7).

Table 4 Compound Queries

Plain Text	Query DSL
break when x has the value 5 and y has the value 10	x == 5 and y == 10
break when one of the methods foo or bar is called	foo() or bar()
break when foo is called, and then bar is called	foo() then bar()

2.5 Efficiency

The previous implementation developed by [BSWK21] uses XQuery and XML to encode queries and the execution stream. That approach leads to a large performance overhead that makes it impractical for most debugging purposes.

As outlined in [BSWK21], there are three main flaws to using XQuery and XML:

1. It saves the entire execution history regardless of what information is actually needed by the queries. That way, the memory use of the debugger grows linearly with execution time, and the evaluation of a query often has to search through the entire execution trace for any event that might potentially trigger a breakpoint.

2. It encodes the execution history as XML, which is both wasteful in terms of additional markup overhead, and does not efficiently provide the information needed by the queries.

3. It encodes queries through generic XQuery expressions, which makes it difficult to determine bounds for the resources used by a query as generic XQuery expressions can be arbitrarily powerful.

Our aim is to avoid the high memory cost of having to keep a full execution history. Additionally, the implementation of the automaton should have an efficient design that scales well with long execution times.

We chose a new automaton design that is significantly more efficient and only captures the information that is strictly necessary for a given query. Additionally, the Query DSL encodes most conditions declaratively and so allows for a broader use of optimizations. The implementation is discussed in ‹Automaton Design And Implementation› (section 4).

By representing sequences of events through the logical combination of simple queries with connectives, we are also explicitly aware what amount of nesting and history is needed. For example, the previously used query `foo() then bar()` would only require us to keep a call history for at most two methods. ‹Connectives› (section 3.7) discusses multiple relevant implementation choices and optimizations.

3 Query DSL

3.1 Design

The main goal of this thesis is the development of an efficient domain-specific language as an interface for a breakpoint backend. We used the Xtext language workbench[EB10] to specify the grammar for the DSL and to automatically generate a parser.

A breakpoint is a particular point during the execution of a program where the debugger should interrupt the execution so that the programmer can closely inspect the program state. Therefore the purpose of the breakpoint DSL is to specify exactly which points are breakpoints.

The instrumentation framework segments the execution of the program into particular events that correspond to instructions. Each type of event has additional attributes for the data associated with the instruction.

Each of these events is a candidate breakpoint. We categorized these events into eight base events that define the interface of the backend (‹Base Event Overview› (table 5)). This simplifies the complexity of the implementation significantly. Furthermore, we limited the types of events exposed by the framework to those that have a clear correspondence to statements in the source code of the program as that is the level of abstraction on which the programmer wants to understand their program.

Table 5 Base Event Overview

Event	Attributes
variable change	variable name, new value
variable access	variable name, current value
source code line change	line number
class file loading	class name
method call	method name, argument values
method return	method name, argument values, return value
exception throw	exception
exception catch	exception

To identify if a base event is a breakpoint, we use the breakpoint DSL to specify queries. If a query matches an event, it is a breakpoint. Each query corresponds to a specific debugging situation defined by the programmer.

We designed a range of simple queries that each correspond to a single type of event. Each simple query has a single condition that needs to be evaluated. The syntax of simple queries closely follows the conventional syntax of the Java programming language in order to create an intuitive mapping to statements.

Our classification leads to four types of base events – those relating to variables, method calls, exceptions, and source code context. We defined a syntax for simple queries that correspond directly to a single base event of one of those types. These simple queries will be discussed in full in the following four sections.

In addition, we added several constructs to organize queries and simplify their specification. These constructs will be introduced in ‹Query Organization› (section 3.6).

To support more complex breakpoints, we also provide connective operators to join multiple simple queries into larger compound queries. That way, many natural breakpoint queries can be expressed in a concise and readable format while limiting the number of necessary constructs in the language. The four supported ways to form compound queries are addressed in ‹Connectives› (section 3.7).

Lastly, we provide query variables as a way to express more generic queries. The debugger can also expose query variables to the programmer as additional context for a breakpoint. We discuss their syntax and semantics in ‹Query Variables› (section 3.8).

We restricted the values supported by our implementation to the primitive types of integer, float, boolean, and string. Objects are also supported as opaque references that can be compared for equality (as discussed in ‹Variables› (section 3.2)). These restrictions were chosen to limit the scope of this thesis.

In order to implement the base event that corresponds to a source line change, we introduced a new synthetic event. The DiSL-based instrumentation framework adds the source code context to each of its events as an additional attribute. We extract the source line in all supported contexts and introduce a new standalone event. That way, source line changes can be treated in isolation as their own type of breakpoint, independent of the actual instruction.

We chose to introduce special variables in the breakpoint DSL to represent the current line and the current class file that is being executed by the JVM runtime. That way, both of those source code attributes can be treated similar to other variables in the syntax, reducing the number of necessary concepts in the DSL. Both source code context base events – source line change and class file loading – are treated as a variation on the base events belonging to variables, and so will be discussed directly after them in ‹Special Variables› (section 3.3). We do not expose the current source code file that corresponds to the class file as the instrumentation framework does not currently retain that information.

We unified local variables and fields into a single variable concept. However, we currently only support static fields because they have a well-defined canonical name. Furthermore, the lack of advanced support for objects as a value type prevents us from supporting instance fields.

The instrumentation framework supports events for local variables, but it does not currently expose an unambiguous scope context for local variables. As such, different local variables with the same name but different block scopes can’t be told apart. Our backend supports events relating to local variables, but because we found no clear way to name them, we chose not to support them in our DSL. Otherwise, breakpoints could easily refer to different

local variables or even fields the programmer did not intend, violating the principle that the DSL should be easy to understand. We discuss the semantic complexities of trying to refer to local variables in detail in ‹Variables› (section 3.2).

Each of the following sections on the Query DSL will illustrate the syntactic constructs with excerpts from the full Xtext grammar. The complete grammar is included in ‹Query DSL Grammar› (appendix A).

3.2 Variables

Variable Base Events	
Event	**Attributes**
variable change	variable name, new value
variable access	variable name, current value

As mentioned in the introduction to ‹Query DSL› (section 3), we unified local variables and fields into a single variable concept. Variables have two corresponding base events, either the reading of a value or the assignment of a new value. The attributes of each event are the name of the variable and its value.

In order to unambiguously connect the variable name and the correct variable in the running program, we canonize the name into its fully qualified form. That way, we can successfully handle any static field. For readability's sake, most of the upcoming DSL examples will use unqualified names. We will introduce a quality-of-life construct in ‹Query Organization› (section 3.6) to simplify the handling of qualified names.

There is no established convention to refer to a particular block scope in the Java programming language or similar languages in the ALGOL lineage. Consider the following example:

```
1  public int f() {
2      int a = 1;
3      {
4          int a = 2;
5          {
6              int a = 3;
7          }
8      int a = 4;
9      }
10     return a;
11 }
```

There are four different local variables named **a** in **f()**. If we simply used the name **a** in our query, it would be unclear which local variable we should associate it with.

We considered two naming scheme for block scope.

One option would be to number scopes in the order in which they appear in a method. For example, in order to refer to the local variable **a** that is initialized with the value 3, the numbering scheme might call that variable **a:3** (in the method **f()**).

Alternatively, one might associate the variable declaration with its line number. In the line number scheme, the same local variable would be **a:6**.

Both systems would be hard to use for the programmer if the syntactic scope is complex or the enclosing method is large. A graphical IDE might provide scope annotations for the programmer, but that would create a tool dependency on certain graphical displays to easily understand what variable a query refers to.

These systems would also be fragile if the source code is ever changed after defining a breakpoint, as block scopes can move around easily during refactoring.[3]

Without being able to unambiguously refer to local variables, we decided to not support them in the Query DSL.

Assignment

```
Assigned:
    var=VarName '<-' right=Element;

Element:
    Value | VarName;
```

```
// break if x is assigned the value 5
x <- 5;

// break if skip is assigned the value true
skip <- true;

// break if the fully qualified variable Author.name is assigned the value "Freya"
Author.name <- "Freya";

// break if x is assigned the value of the variable y
x <- y;
```

Assignments are the most basic kind of variable base event. An assignment query is true if the base event has a matching name and value. If the right-hand side element of the query is itself a variable name, then the backend will look up its current value and substitute it. If the variable is still undefined, then the query does not match.

The provided examples demonstrate several of the supported value types, namely integers, booleans, and strings.

Assignments are not meant to be the typical way to specify queries concerning variables in the Query DSL, so we decided to give them an explicit syntax that makes them stand out. We introduced a new assignment operator **<-** not supported by the Java programming language, but common in many programming languages and pseudocode conventions.

[3] Structural programming environments such as MPS[Vö11] or Lamdu[lam] would provide unambiguous references for all kinds of variables. Similarly, explicit annotations inside the source code would also provide a possible solution as the compiler could resolve the correct name reference for the debugger and the annotation would be locally connected to the source code.

By using a dedicated operator, the conventional = assignment operator remains available for equality comparison, as discussed in the following paragraphs.

Access

```
Accessed:
    var=VarName;
```

```
// break if the value of x is read
x;
```

The simplest kind of **variable read** base event is independent of the actual value of the variable. This simple query would allow the programmer to trace all uses of a variable during execution. This use case fits easily into the general syntax of our DSL at no real cost and so we decided to support it.

Comparison

```
Comparison:
    left=Element pred=PredOP right=Element;

enum PredOP:
    EQUALS  = '='  |
    EQ      = '==' | NEQ       = '!=' |
    LESS    = '<'  | LESSEQ    = '<=' |
    GREATER = '>'  | GREATEREQ = '>=' ;
```

```
// break if the value of x is greater than 5
x > 5;

// break if the value of x is less than or equal to 0.96
x <= 0.96;

// break if the value of "needed" is identical to the value of "available"
needed == available;
```

More commonly, the programmer is interested in the concrete value of a variable that is being accessed. We support all common comparison operators with the expected Java semantics for this purpose. Notably, the operators == and != also test for equality of reference and so can be used to opaquely compare objects.

However, that still leaves the problem of how to compare strings or other objects that have structural equality. Java normally uses the method **equals()** for this purpose.

We decided to introduce a new operator as a shorthand for **equals()**. As we used <- for assignments, the symbol = is still available for this purpose. It is also very compact and therefore matches the ergonomic principle.

We considered using a textual operator such as **eq** or **equals**. Most notably, the Perl programming language uses **eq** for string equality. However, we decided that = was a more natural fit for generic equality.

Comparing values with **equals()** is typically the desired behavior in the context of breakpoints, as it is the default comparison method for most objects in Java and has the same semantics as == for numerical types. By using = for the default case, our syntax follows the principle that the most common use case should have a simple, short form of expression.

But what about comparisons using other instance methods, such as **length()**? While arbitrary instance methods could be added, we decided against it to limit the scope of this thesis and because arbitrary methods would violate our goal to implement an efficient backend. The performance of such a method could be arbitrarily bad and our system would be required to evaluate the query many times during the execution. This would drastically affect the overall performance. Similarly, the system can provide no guarantee that methods are free of side-effects.

Type Change

```
InstanceOf:
    obj=Var 'instanceof' type=TypeName;

TypeName:
    '@'(name=QName);
```

The final built-in comparison operator in the Java programming language is the type check operator **instanceof**. However, in order to implement that operator, we have to consider the syntax used for types. Java makes no inherent syntactic distinction between the names of variables and types. would It be unambiguous to make the same decision for the **instanceof** query, but it would create a confusing situation for method calls, which will be discussed in ‹Method Calls› (section 3.4). Therefore, we decided to explicitly mark types with the prefix @.

```
// break if obj is of the type Student
x instanceof @Student;
```

3.3 Special Variables

Source Code Context Base Events	
Event	**Attributes**
source code line change	line number
class file loading	class name

```
LineChanged:
    '$line' pred=PredOP val=Integer;

ClassChanged:
    '$class' ('=' | '==') val=Text;
```

```
// break if the source line is 72
$line = 72;

```

```
// break if the current class file is Debug(.class)
$class = "Debug";
```

The source code context is modeled through the two base events **source line change** and **class file loading**. The programmer would certainly be interested in an additional base event for the source file that corresponds to the class file, but unfortunately the current instrumentation framework does not support attributes for source file names. Trying to reconstruct the correct source file from just the class file name is too unreliable in complex projects, so we decided against supporting a **source file loading** base event until the instrumentation framework can be improved.

The Query DSL exposes the source code context through the special variables **$line** and **$class**, using the **$** prefix. That convention is already well established in many programming languages such as Mathematica, Perl, and Ruby, and has no prior meaning in the Java programming language and syntactically related languages.

By using special variables, the source code context can be treated like other variables. The structure and implementation of queries for comparing a numerical value of a variable and for comparing the current numerical source line is very similar and simplifies the automaton design.

In order to ease the writing of queries for the class file name, we support a glob value type[TR75]. Globs are a regular expression language that is widely used in Unix shell environments for matching file names. Because globs are a subtype of more general regular expressions that only differs slightly in their syntax, we also implemented a regular expression value type using the familiar slash notation supported by the **java.util.regex** library and many other standard libraries.

```
// break if any class file with the name prefix "de.umr.plt" is loaded
$class = [de.umr.plt.*]

// break if the class file contains the substring "debug"
$class = /debug/
```

3.4 Method Calls

Method Base Events	
Event	**Attributes**
method call	method name, argument values
method return	method name, argument values, return value

Method base events represent either calling a method or returning from a method. In both cases, the attributes include the method name and the argument values, even though **return** statements in the Java programming language do not explicitly annotate the method they return from. By including the information in the **method return** base event, we can design a richer query syntax, as queries cannot rely on the implicit lexical scope of an enclosing method definition to resolve which method a **return** belongs to.

Calls

```
MethodCall:
    (method=MethodName)'(' (args=MethodArgs)? ')';

MethodArg:
    var=Element               |
                type=TypeName |
    var=Element type=TypeName ;

MethodArgs:
    void?='void' | (args+=MethodArg (',' args+=MethodArg)*);
```

Because queries should follow the syntax of the source programming language whenever possible, we chose the same core syntax as that of a method call. However, the programmer should also have tho option to define breakpoints conditional on the arguments to the method.

We carefully considered three situations:

1. break, no matter what arguments (or lack thereof) are passed
2. break if no arguments are passed
3. break if one or more argument are passed

We consider the first case the most common. When debugging, the programmer is likely to first investigate the execution of a method, irrespective of the arguments, simply to understand the overall behavior of the method. Only later might they want to provide constraints on the arguments to narrow down their investigation.

Additionally, method queries will be used later to provide a context for other queries, using the **then** connective in ‹Then, :› (section 3.7). In that case, simply narrowing the debugging scope down to a particular method is the most typical situation. The use of compound queries will be introduced in ‹Connectives› (section 3.7).

Because of those two common use cases, we wanted the shortest default syntax to reflect the first type of situation where the concrete arguments are ignored. However, the most straightforward syntax that would still superficially follow the Java convention would be a call without explicit arguments, eg. **print()**. That syntax would clash with the way to write a method call with no arguments.

We considered two possible solutions.

1. We could introduce a special token to represent “any number and type of argument”, for example **print(*)** or **print(...)**. That solution would be fully unambiguous and explicit, but it would introduce a new syntactic construct for the most common case and would make method call queries more complicated to write. That violates our principle that the Query DSL should still be ergonomic for the programmer.

2. We could use the empty call as the default, and introduce a new token to explicitly mark the case where we want to match a call with no arguments. The token **void** is already used by the Java programming language to mark the absence of a return value in method signatures and therefore well established. Additionally, the use of the syntax **print(void)** to mean a method with no arguments is widely used in function declarations of the C and C++ programming languages.

We decided to use the second solution because we considered the ergonomic trade-off more important. Explicit queries to match the absence of arguments is clearly the special case and therefore it can tolerate a more verbose syntax. Furthermore, referring to a method or function merely by adding empty parentheses in the style of **print()** is widely used in the programming community and a familiar ambiguity for many programmers.

Arguments in a method call base event may be matched either by their value or their type. As we already introduced an explicit syntactic type prefix for type change base events in ‹Variables› (section 3.2), there is no ambiguity in queries whether a name refers to a variable or a type.

In total, we get the following possibilities to match a method call base event:

```
// break if the method print() is called with any arguments (including none)
print();

// break if the method print() is called with the two arguments 5 and 10
print(5, 10);

// break if the method print() is called with a String argument
print(@String);

// break if the method print() is called with a String argument and the argument "
    ↪ false"
print(@String, false);

// break if the method print() is called the value of the variable "unsafe"
print(unsafe);

// break if the method print() is called with the float value 10, and an int
    ↪ argument
print(10 @float, @int);

// break if the method print() is called without arguments
print(void);
```

Returns

```
Return:
    'return' {Return} (retval=Value)? ('from' method=MethodCall)?;
```

return base events follow the same design logic as **method call**s.

We also wanted a compact way to connect the **return** statement in the query with the corresponding method call. Most importantly, it is likely unusual for a breakpoint to be merely about the return value independent of the method, such as in the query `return 5`,

which would interrupt the execution whenever the value 5 is returned by any method in the programming. There are certainly some situations where that might be the intended breakpoint, for example if the programmer wants to trace a particular notable value as it passes from method to method in the program.

Nonetheless, we decided that the more typical situation would be the debugging of a particular method and a breakpoint when said method returns a noteworthy value. It was therefore essential that **return** queries could be tied to a method call in a compact way. Compound statements as described in ‹Connectives› (section 3.7) cover a similar ground, but are not completely sufficient for this purpose. That led us to enhance the syntax of **return** queries with a **from** keyword to specify the corresponding method call in the same way as for a **method call** base event.

```
// break if any method returns the string "password"
return "password";

// break if the authenticate method returns the null value
return null from authenticate();
```

3.5 Exceptions

Exception Base Events	
Event	**Attributes**
exception throw	exception
exception catch	exception

```
Throw:
    'throw' (type=TypeName);

Catch:
    'catch' (types+=TypeName('|' types+=TypeName)*);
```

```
// break when FileNotFoundException is thrown
throw @FileNotFoundException;

// break when FileNotFoundException is caught
catch @FileNotFoundException;

// break when FileNotFoundException or NullException are caught
catch @FileNotFoundException | @NullException;
```

Exception base events differ only in whether the exception was thrown or caught. We decided to only match against the exception type because many exceptions do not expose inner state beyond their error message. In the case of uncaught errors, debuggers already interrupt the execution and so they don't fall in the domain of breakpoints.

However, query variables and compound queries nonetheless provide ways to treat exceptions as values in queries. The extent of that support is however limited by the current restriction on instance methods, as explained in ‹Variables› (section 3.2).

The type prefix syntax, as discussed in ‹Variables› (section 3.2), is used consistently for all types in the Query DSL. It would be possible to modify the DSL such that the type prefix is optional in contexts that only allow type arguments, such as exception queries. That way, these queries would look closer to the Java statement they are modelled after. In order to keep the syntax consistent and avoid possible ambiguities, we decided against optional prefixes.

3.6 Query Organization

```
QueryFile:
    (entries+=Entry)+;

Entry:
    Query | Import;

Query:
    (name=QueryName '=>')? condition=Condition ';'+;

QueryCall:
    '<' name=[QueryName] '>';

Import:
    'package' pkg=PkgName ';'+;
```

```
// break if x is too big, or if it is too small
too_big   => x > 60;
too_small => x < 10;
```

To help manage breakpoint queries for more complex debugging situations, we organized queries into a larger query file[4]that can continue any number of queries. Every query corresponds to a different breakpoint scenario. Note that our example queries have already implicitly been multiple entries in a larger file.

In order to further organize the queries, we also introduced a way to name them. That way we also gain a useful form of abstraction, as we can reuse queries in the definition of other queries once we add compound queries in ‹Connectives› (section 3.7). Our current implementation does not fully support nested queries, however, as we considered the correct implementation of mutually recursive definitions beyond the scope of this thesis.

Query names are also useful as a visual aid in the debugger to quickly communicate to the programmer which breakpoint query the current breakpoint belongs to.

Lastly, we mentioned in the introduction of the current chapter that variable names have to be fully qualified in order to establish the correct correspondence. It is clear that having to write fully-qualified names in all breakpoints would be very impractical for the programmer, so we needed a way to simplify the task. We added a way to declare an active package prefix that is used to qualify any unqualified names used afterwards.

[4] Naturally, a **query file** may simply be a buffer in the editing environment and does not have to correspond to an actual file in the file system.

```
// break when the fully qualified variable filename has the value "methods.query"
io.bitbucket.umrplt.extra.query.Tracer.filename = "methods.query"

// ditto, but more readable, especially for multiple queries in the same namespace
package io.bitbucket.umrplt.extra.query.Tracer;
filename = "methods.query";
```

As a further improvement, the editing environment might automatically associate breakpoint queries with a particular source code file and add a package statement for the programmer so that most queries can be written without having to manually qualify names.

3.7 Connectives

```
Condition:
    left=SimpleCondition (=> connective=ConnectiveOP right=Condition)?;
```

Breakpoints do not have to map directly to only one base event, but may refer to multiple events in the execution history. Note that because we implemented the source code context as a synthetic standalone event, writing a query that refers to an instruction and the line or class file it belongs to will need to connect two base events as well.

We implemented operators to logically connect the previously introduced four types of simple queries into compound queries. We considered the basic case that the compound query consists of two constituent queries, which we will call its **branches**. Based on the following two questions, we constructed a matrix (‹Connective Matrix› (table 6)) of the logical possibilities:

1. Should both branches be true, or is it enough for one to match?

2. Do the branches have to match in a particular order during the execution, or is any order (ie. **A then B**, or **B then A**) acceptable?

Table 6 Connective Matrix

		both true?	
		no	yes
order important?	no	**Or**	**And**
	yes	*n/a*	**Then**

If only one of the two branches has to match in order for the whole compound query to match, then obviously there is no meaningful order of events to consider, so only three of the four possible permutations have a useful interpretation.

Our analysis yields three fundamental connective operators. We found that they cover all non-contrived breakpoint scenarios we tested. However, we further split the **Then** operator into two versions which we will discuss in ‹Connectives› (section 3.7).

We introduced four new keywords, **or**, **and**, **Then** and **:** for the connectives.

```
enum ConnectiveOP:
    OR        = 'or'   |
    AND       = 'and'  |
    THEN_SEQ  = 'then' |
    THEN_TRUE = ':'    ;
```

Or

```
// break if either x is 10 or y is 100
x = 10 or y = 100;

// break if mode is assigned the value "next" or the value "skip"
mode = "next" or mode = "skip";

// break if the method error() is called or a ParseException is raised
error() or throw @ParseException;
```

The simplest connective operator is **or**. An **or** query matches if either of its two branches matches the current base event.

As the **or** query only requires that one of its branches matches, it does not need to refer back to previous events in the execution. That simplifies the semantics of **or** queries as we do not need to consider the semantics of matching over the execution history yet. The next section on the **and** operator will address that problem.

An **or** query is semantically equivalent to testing the branches one after another as if they were two independent queries, with the only exception that they still form a single logical breakpoint and therefore should not generate two interrupts in the debugger.

It would be tempting to skip the evaluation of the second branch if the first branch already matches, and for all the stateless query constructs we have introduced so far, that optimization would be valid. However, the introduction of query variables in ‹Query Variables› (section 3.8) will include query state and so lazy evaluation would no longer have the same semantics as eager evaluation.

However, all stateful query constructs are explicit in the Query DSL, so the automaton can perform lazy evaluation and branch reordering as an optimization for any query that is free of side-effects, which is any query that does not use query variables.

And

```
// break if x is 10 and y is 100
x = 10 and y = 100;

// break if x is greater than 100 in line 72
x > 100 and $line = 72;
```

```

// break if f() is called before line 100
f() and $line < 100;
```

The **and** connective requires that both of its branches match in any order. There are several possible ways to interpret that requirement. We need to develop consistent semantics for the **and** operator and what it means for a query to depend on the execution history.

We assumed so far that a breakpoint should always trigger at any point in the execution where a relevant base event occurs and the breakpoint condition holds true. However, we have not generally considered that a breakpoint might refer back in the execution history and that it might be conditional on not just the current base event, but also previous base events.

To decide on the semantics of an **and** query and what it would mean that both of its branches have to match, we considered three possible interpretations:

1. Both branches match the same base event. (**Same Event**)
2. Each branch matches a base event, but not necessarily the same event. (**Any Prior Events**)
3. Each branch matches a base event, and both branches are still true. (**Both True**)

The **Same Event** interpretation would be highly restrictive. Most importantly, an **and** query could only ever match if both of its branches match the same type of base event. For example, the query `f() and x <- 10` could never match, as it would require a base event that is both a method call and a variable assignment.

To illustrate the difference between the other two interpretations, consider the following Java code example and two queries:

```
int a, b;

a =  10; // (1)
b =   0; // (2)
b =  10; // (3)
b = 100; // (4)
```

```
q1 => b > 0 and a = b;

q2 => b < 10 and b > 10;
```

The two branches of query **q1** are **b > 0** and **a = b**. The first branch, **b > 0**, would clearly match statements 3 and 4, while the second branch, **a = b**, would only match statement 3.

If we merely require that there is a base event that matches each of the two branches, than during the execution, that requirement would be fulfilled at point 3 and at point 4. In other words, if we search backwards through the execution history starting at point 4, we would find that the current base event matches one of the branches, and that there is fitting past

base event for the other branch. According to the **Any Prior Events** interpretations we considered earlier, point 4 would therefore qualify as a breakpoint.

However, at that point in the execution, the condition implied by the branch **a = b** is no longer true, as **a** and **b** have different values now. In the **Both True** interpretation, that contradiction would rule out point 4 as a breakpoint. If a branch encounters a fitting base event, but the event contradicts the condition of the branch, then we consider that branch as no longer matching, even if there is a prior event that matches it.

That difference becomes even more clear with query **q2**. Point 4 would be a breakpoint under **Any Prior Events**, as there is a base event where **b < 10** and one where **b > 10**, but of course there is no point in the execution where both of those conditions are true at the same time.

One can consider **Any Prior Events** as a way to express queries about a sequence of events, whereas **Both True** looks for an intersection of multiple conditions.

We rejected the highly restrictive **Same Event** interpretation as it was only of very limited use and either of the other two interpretations could easily express those situations as well.

The implicit sequential nature of the **Any Prior Events** interpretation clashes with the design decision that the **and** connective should not care about the order of the branches. For example, if the programmer is interested in a sequence of values that a variable takes on, they would most likely also be interested in a particular order, not just a set of values.

However, the **Both True** interpretation gives us a way to combine queries in order to narrow down a context for a breakpoint. For example, `f() and unsafe = true` can work as a guard to restrict an overly generic breakpoint for the method **f** to only trigger if **unsafe** is also true.

Because of these considerations, we decided that only the **Both True** interpretation is semantically sensible for the **and** connective.

Then, :

The **then** connective follows the same logic as the **and** connective, except that it defines a correct order in which the branches need to match.

As we discussed for **and**, there are two possible interpretation when considering the execution history, **Any Prior Events** and **Both True**. We rejected **Any Prior Events** for **and** because of the lack of ordering, but **then** provides just that missing requirement.

With well-defined ordering, both interpretations are useful for breakpoints. Consider the previous example from **and** again, with two new queries:

```
int a, b;

a =  10; // (1)
b =   0; // (2)
b =  10; // (3)
b = 100; // (4)
```

```
q1 => a = b then b > 0;

q2 => b < 10 then b > 10;
```

Under **Any Prior Events**, points 3 and 4 would be breakpoints for query **q1**. Similarly, point 4 would satisfy query **q2**, as there was just such a sequence of base events, even though at no point in the execution were both branches true at the same time.

Analogously, **Both True** would yield point 3 as a breakpoint for the query **q1**, the intersection where both branches are both true, but not point 4. As expected, the query **q2** would never match.

Both of those interpretations are useful for debugging. **Any Prior Events** lets the programmer explore a sequence of events, and **Both True** narrows down the scope of a query to find intersections. Therefore, we decided to support both semantics with different keywords, using **then** for the sequential **Any Prior Events**, and **:** for **Both True** to mimic the nested structure of certain constructs in well-established programming languages like Python. That suggestive syntax becomes particularly clear with the use of (syntactically irrelevant) indentation:

```
// break when x is greater than 10,
// and later y is greater than 10 (while x is still greater than 10)
x > 10: y > 10

// (with indentation)
x > 10:
    y > 10

// break when f is called while x is greater than 10
x > 10:
    f()
```

So far, we have assumed an intuitive sense of what it means for a query to still be true after it has matched a base event for the first time. In order to properly clarify the semantics of the **:** connective, we analyzed all base event types.

The variable base events have a clear condition that depends on the program state. For any base event, we can decide whether it either fits the condition, is independent of it (for example because it refers to a different variable), or directly contradicts it. There is an obvious and unambiguous sense of when a variable query is no longer true, namely when its condition is contradicted by the current base event. The same applies to special variables.

Exceptions do not normally have a persistent effect on the execution beyond being thrown or caught. Whereas variables have state that spans multiple events, exception base events can be considered point events. As such, **then** and **:** would behave identically for exceptions.

The main complication arises with method calls. Consider these two different examples:

```
// (1) sequential
{
    public void f() {}
    public void g() {}

    f(); g();
}

// (2) nested
{
    public void f() { g(); }
    public void g() {}

    f();
}
```

In both cases, there are two method call base events for the methods **f** and **g** in the same order, but they represent very different situations – one sequential, one nested. To be explicit, due to the nesting, the full sequence of events for **1** is **call f, return f, call g, return g**, whereas for **2**, the sequence is **call f, call g, return g, return f**.

If the Query DSL did not provide a way to distinguish between these two situations, we could not use **then** and **:** to properly provide a method context for queries.

The following example illustrates the problem. Say we have a method **init()** that initializes some data structure, and a method **use()** that accesses the initialized data structure. The programmer is concerned that the method **use()** might erroneously get called before the call to **init()** is completed, so they want to define several breakpoints to explore that possibility.

```
void init() { /* initializes the data */ }
void use()  { /* uses the initialized data *) }
```

There are two possible error scenarios – calling **use()** during the execution of **init()** (either directly, or indirectly through other methods used in **init()**), and calling **use()** before the call to **init()**.

Using the Query DSL constructs available so far, these two scenarios might be represented with the following two queries:

```
during => init() then use();
before => use()  then init();
```

The query **before** works as intended,[5] but **during** is too permissive. Because **then** only requires that the two base events occur one after another, **init()** may have returned or not before the call to **use()**.

The programmer could express the intended behavior of the two methods by using a query like return from init() then use(), but there is no way yet to state that another query has match during a method call.

There are three useful kinds of breakpoints concerning method calls:

1. The method is called. (**Agnostic**)
2. The method is called and has not yet returned. In other words, it is still active on the call stack. (**During**)
3. The method has successfully returned and is no longer on the call stack. (**After**)

The simplest kind is **Agnostic**, and when a query contains no connectives, it is the primary use case. After all, in a simple query method calls can't provide context for further elements of the query.

Using the same principle that the most common and least specific case should get the least complex syntax that we applied for method signatures in ‹Method Calls› (section 3.4), we decided that unmarked method calls would be **Agnostic**. We added the optional **call** keyword to offer an explicit way to mark the programmer's intention regardless, but it adds no additional semantic information.

```
// two identical queries:
     init() then      use();
call init() then call use();
```

Because the **Agnostic** case only indicates that a method was called at all, it can't be contradicted by any future base event, and so the semantics for **then** and **:** are identical as well.

To mark the more specific situations **During** and **After**, we added the keywords **in** and **after** as follows:

```
during  => in init():    use(); // the actual intended earlier breakpoint
correct => after init(): use(); // the correct program behavior
```

A method call query using the keyword **in** is only true until the corresponding method call returns, analogue to how a variable query is only true while its condition holds, as discussed earlier. Clearly **in** provides no additional information using the **then** connective as the current truth state has no effect on its semantics, so it is only useful for **:** queries.

[5] A possible exception might be that **use()** calls **init()** itself before actually accessing the data when it notices that the data isn't yet initialized. Slightly more complex queries could cover that possibility too.

Similarly, a method call query using **after** only matches once the method returns. Note that it is not possible to use a combination of a call and return event to achieve the same result as **after**, as there is no way to correctly link the return to the same method, and not just any other previous method call fitting the same specification. Both **then** and**:** behave the same for **after** as it is not possible to contradict an **after** query again, but the additional specificity it allows is nonetheless useful for the ordered connectives.

We now have a full understanding of what it means for a simple query to still be true after it matches for the first time. The semantics of the **Both True** connective **:** are now clear and allow us to write queries with well-behaved ways to narrow down the context in which a condition holds.

```
// break if loadClasses() accesses the Database.students field
//(directly or indirectly)
in load():
    Database.students;

// break if x is greater than 10 while y is less than 10
x > 10:
    y < 10;

// break if x takes the values 1, 2, 3 in order
x = 1 then
x = 2 then
x = 3;
```

Nonetheless, the **then** connective can only declare the mere sequence of events, but there is no fine-grained control over exactly how these events are spread out or what events might or might not occur between them. That task would be better served by applications for the detailed analysis of time streams and not a general-purpose debugger. Additionally, the analysis is likely to be highly domain-dependant and will require proper data sets, not the short queries typical for breakpoints.

3.8 Query Variables

```
Var:     VarName | QVar;
SetVar:  Var     | SetQVar;

Element:
    Value | Var;

SetElement:
    Element | SetQVar;

QVar:    '?'(name=QName);
SetQVar: '?='(name=QName);
```

In ‹Method Calls› (section 3.4), we introduced a syntax for matching method signatures. So far, we've seen ways to match a method argument by comparing it against values, either specified explicitly as literals or implicitly by reference to variables. Additionally, arguments can be matched against concrete types, and can be narrowed down through the use of connectives.

As we explored various plausible breakpoint scenarios during the design process of the Query DSL, we encountered situations that could not be adequately represented by the existing constructs. In particular, there was no way to match a method signature purely based on the number of arguments, for example, or to match a situation where two variables had the same (unspecified) type.

In order to represent these queries, we added query variables as a form of abstraction, similar to the use of bindings in pattern-matching[BMS80]. As a preliminary design, we marked query variables with the **?** prefix. Consider the following three candidate queries:

```
// match if the method f() is called with the same three values
f(?x, ?x, ?x)

// match if the method f() is called with three arguments (of any value)
f(?x, ?y, ?z)

// match if the method g is called with the same value as the method f before it
f(?x) then g(?x)
```

That sketch of query variables would seem sufficient, but it leaves open one main problem. All usages of a particular query variable are equivalent throughout the query, so for compound queries, it is unclear what usages would bind the query variable and what would read the binding. That problem is particularly severe for the **and** connective because does not impose an ordering:

```
// match if a and b are assigned the same value
a <- ?x and b <- ?x
```

We considered several solutions to the problem.

It would be possible to create the binding the first time a simple query otherwise matches a base event and then keep the binding constant. That approach has multiple important disadvantages. It would be unpredictable what places in the query would be binding, which adds a sense of non-determinism to the Query DSL, which would be highly undesirable. Additionally, only one single event during the entire execution could set a query variable, so queries would lose a lot of their generic power that we are trying to add to the DSL in the first place. Breakpoints should not just describe one particular point in the execution, but potentially multiple ones that all follow the same condition.

Another approach would be to impose a syntactic order on queries, such that the first usage of a query variable is the binding site, and all other usages afterwards treat it as a value. The most natural way to create such an order would be to take the syntax tree of a query, descend depth-first, and treat the first encounter of a query variable name as the binding site. Intuitively, that matches the left-most position in the query expression. This lexical approach does not lead to a sensible interpretation for the **and** connective as it would treat the left branch as definitive, even though both branches are otherwise semantically on equal footing. That way, **and** branches would have a hidden ordering after all, a strong violation of their design principles.

We solved the problem by splitting the binding and the use of a query variable into two different constructs, marked with the prefixes **?=** and **?**. The corrected version of our previous queries are:

```
// match if the method f() is called with the same three values
f(?=x, ?x, ?x)

// match if the method f() is called with three arguments (of any value)
f(?=x, ?=y, ?=z)

// match if the method g is called with the same value as the method f before it
f(?=x) then g(?x)
```

The main disadvantage of the explicit distinction between setting and using a query variable is the syntactic overhead.

It might be tempting to introduce an inference rule (maybe based on the previously mentioned depth-first ordering) to automatically reinterpret a query that only uses **?** constructs, but then the Query DSL would contain elements whose interpretation depends on a potentially much larger context. Additionally, programmers would have to learn about the **?=** syntax anyway whenever they want to use **and** connectives together with query variables, so the total amount of concepts necessary to learn would in fact be greater, as they would now also have to understand the inference rule.

As a backend language, the Query DSL should always be as explicit and unambiguous to parse as possible, so we decided against implementing any kind of syntactic shortcut that would contradict that.

One additional advantage of query variables is that they offer a way for the debugger to track and display named values that aren't directly part of the original program. That way, the programmer can name certain parts of the query and have them available during debugging without having to modify the original program.

```
Type:    TypeName | QType;
SetType: Type     | SetQType;

QType:    '?@'(name=QName);
SetQType: '?@='(name=QName);
```

One last extension is necessary to fully implement our initial goal. So far, query variables can match values, but they cannot bind types. As discussed in ‹Variables› (section 3.2), types are also syntactically distinct from variables, so we analogously added query type variables using the prefixes **?@=** and **?@**.

```
// match if the method f() is called with the same three types
f(?@=x, ?@x, ?@x)

// match if the method f() is called with three arguments (of any type)
f(?@=x, ?@=y, ?@=z)

// match if the method g is called with the same type as the method f before it
f(?@=x) then g(?@x)
```

We extended the grammar of the Query DSL to add query variables to all other constructs in various value and type places. See the full grammar in ‹Query DSL Grammar› (appendix A) for the complete version.

4 Automaton Design And Implementation

4.1 Design

We implemented a backend for the Query DSL based on the instrumentation framework used in [BSWK21].

First, we parse the provided breakpoint queries using the Xtext-generated parser for the Query DSL. We then construct an automaton for each query to match base events.

The breakpoint backend receives events from the instrumentation framework through the use of callbacks. We grouped these events into the eight base events we introduced in ‹Design› (section 3.1).

We then pass these events to the matching automaton. If the automaton signals a match, we trigger a breakpoint. Lastly, the backend also performs some necessary bookkeeping for the automata.

Beyond base events and their associated attributes, queries may require information about the current program state and its call history. A query may reference variables, so we make a mapping of variable names to values available to the automata.

We implemented the variable bookkeeping by adding a call to the following method **updateStaticField(String name, Object value)** to each field event supported by the instrumentation framework.

```
1 private final Map<String, Optional<Object>> statics = new HashMap<>();
2
3 private void updateStaticField(String name, Object value) {
4     statics.replace(name, Optional.ofNullable(value));
5 }
```

Additionally, method call queries may track when their respective methods return, so we also provide that information based on the call stack. Furthermore, the instrumentation framework does not currently include the method name and signature in its return events, so we also need to use the same call stack to recreate that information.[6]

We added similar calls to the method call and return events to maintain our own call stack.

[6] The current implementation does not correctly process the unrolling of the call stack in the case of an exception. Exception control flow can still disrupt the correct mapping of method calls to returns. We decided that correcting this limitation should wait until the instrumentation framework is able to handle the call stack portion of our bookkeeping system.

```
private final Stack<Method> callstack = new Stack<Method>();
private Method current_method;

public class Method {
    public String name;
    public List<Object> args;
    public Optional<Object> retval;
}

@Override
public void beginMethodCall(String classFqn, String methodName, String methodDesc,
    ↪ int line, boolean isDynamic, boolean isInterface, boolean isSpecial, boolean
    ↪ isStatic, boolean isVirtual) {
    // maintain a call stack
    current_method = new Method(classFqn, methodName, methodDesc);
    callstack.push(current_method);
}

@Override
public void addArgument(Object val) {
    // add method arguments
    current_method.args.add(val);
}

@Override
public void endMethodCall() {
    // maintain a call stack
    Method m = callstack.pop();
    current_method = callstack.lastElement();
    calledMethod(m, true); // true -> return event, not call event
}
```

Before execution, the backend traverses all the queries to find the variable names that are actually of interest to the queries, and which methods may need to be associated with return events. We then reconstruct that highly limited subset of the program state from the event stream. As an optimization, the backend performs all the bookkeeping for the active queries in a single location.

Ideally, the backend would access the program state and call stack directly through the runtime. However, the current instrumentation framework does not have that functionality and extending it in that way was beyond the scope of this thesis. In the performance evaluation in ‹Evaluation› (section 5) we measure the bookkeeping overhead. As we only track the bare minimum of the information that is necessary for matching the queries, the overhead remains small.

The instrumentation framework currently generates events of all types and for the entire scope of the instrumented program. A significant gain in performance would be possible by restricting the events to those that are potentially relevant to the active queries.

Similarly, it would be possible to restrict the instrumentation to only a subset of the program. For example, static analysis of the queries could easily yield a comprehensive list of all methods used in the active queries. Any other method in the instrumented program would not need to contain instrumentation to generate method call and return events.

Both of those optimizations were outside the scope of this thesis and so we did not pursue them. However, the backend already performs some the the necessary static analysis for its own optimizations and as such could easily pass it to the instrumentation framework.

We originally considered using the Xtext framework to compile each query into specialized code for the automaton. That would have the advantage that there would be no indirection through multiple method calls or nested queries because the compiler could flatten the entire code. It would also make it possible to skip any code for a particular Query DSL construct that is not necessary for a given query.

However, we encountered several major problems with that approach.

Xtext is not specifically designed for runtime code generation and typically compiles DSL input into separate **.java** files. That additional complexity does not fit the use case of a debugger that would have to handle many new breakpoint queries at runtime.

We also found it much more complex to implement the logic correctly, as we could not easily inspect the automaton during the compilation process or after flattening.

Finally, we decided that the potential gains from compilation likely weren't great enough to justify the additional complexity or the switch to a different code generation framework, so we implemented the automaton as an interpreter with nested **Matcher** instances that would handle the matching of the Query DSL constructs.

We recursively construct the Matcher instances by traversing the query syntax tree. As our automaton and the Query DSL correspond very closely, the entire construction process is fairly straightforward.[7]

```
1  private Matcher constructMatcher() {
2      return parse(query.getCondition());
3  }
4
5  private Matcher parse(Condition c) {
6      if (c.getRight() == null) { // just a simple condition
7          return parse(c.getLeft());
8
9      } else { // connective
10         Matcher left  = parse(c.getLeft());
11         Matcher right = parse(c.getRight());
12
13         switch (c.getConnective()) {
14             case AND:       return new AndMatcher(left, right);
15             case OR:        return new OrMatcher(left, right);
16             case THEN_SEQ:  return new ThenSeqMatcher(left, right);
17             case THEN_TRUE: return new ThenTrueMatcher(left, right);
18         }
19     }
20 }
21 // (continued on next page)
```

[7] The code excerpts used in this section are primarily illustrative and contain some minor simplifications. The full source code of the automaton implementation is available under https://segit.mathematik.uni-marburg.de/ExTra/extra-base as part of the Programming Languages and Programming Tools research group at the Philipps-Universität Marburg.

```
22 private Matcher parse(SimpleCondition c) {
23     if      (c instanceof VariableChanged) return parse((VariableChanged) c);
24     else if (c instanceof MethodCalled)    return parse((MethodCalled) c);
25     else if (c instanceof Exception)       return parse((Exception) c);
26     else if (c instanceof Return)          return parse((Return) c);
27 }
28
29 private Matcher parse(VariableChanged v) {
30     if (v instanceof Comparison) {
31         Comparison c = (Comparison) v;
32         return new VarMatcher(c.getLeft(), c.getRight(), c.getPred());
33
34     } else if (v instanceof Accessed) {
35         Accessed a = (Accessed) v;
36         return new AccessMatcher(a.getVar().getName());
37
38     } /* other variable matchers... */
39 }
40
41 private Matcher parse(MethodCalled m) {
42     MethodCall      call  = m.getCall();
43     MethodArgs      margs = call.getArgs();
44     List<MethodArg> args;
45
46     if (margs == null) { // any argument is allowed
47         args = null;
48     }  else if (margs.isVoid()) { // explicitly o args
49         args = new ArrayList<MethodArg>();
50     } else { // explicity some args
51         args = margs.getArgs();
52     }
53
54     return new MethodMatcher(m.getModifier(), call.getMethod().getName(), args);
55 }
56
57 private Matcher parse(Return r) {
58     /* nearly identical to MethodCalled */
59 }
60
61 private Matcher parse(Exception ex) {
62     if (ex instanceof Catch) {
63         Catch c = (Catch) ex;
64         return new ExceptionMatcher(c.getTypes(), c.getQtype());
65
66     } else if (ex instanceof Throw) {
67         Throw t = (Throw) ex;
68         return new ExceptionMatcher(t.getType(), t.getQtype());
69     }
70 }
```

A query automaton has the following data structure:

```
public class QueryAutomaton {
    public  String  name;    // the query name
    private Query   query;   // the parsed syntax tree of the query
    private Matcher matcher; // the generated automaton used for matching

    private final Map<String, Optional<Object>> statics; // the variable mapping
    public  final Map<String, Optional<Object>> qvars;   // the query variables
    public  final Map<String, String>           qtypes;  // the query type variables
}
```

Note that we wrap all values in the mappings with the **Optional** type to handle uninitialized variables. The Java programming language and other memory-safe languages do not allow the use of uninitialized variables, but the debugger is in a special position as queries don't exist in the same lexical environment and so can attempt to reference variables outside their normal scope.

The automaton receives base events through a small set of methods that it passes to its matcher for evaluation.

```
public enum Match { yes, no, skip }

private class Matcher {
  // Variable Read and Write
  public Match changedVariable(String name)                 { return Match.skip; }

  // Method Call (returned == false) and Method Return (returned == true)
  public Match calledMethod(Method method, boolean returned) { return Match.skip; }

  // Exception Throw (caught == false) and Catch (caught == true)
  public Match threwException(Throwable ex, boolean caught)  { return Match.skip; }

  // Source Line Change
  public Match changedLine(int line)                        { return Match.skip; }

  // Class File Load
  public Match changedClass(Class<?> cls)                   { return Match.skip; }
}
```

We encoded matching with three possible states using the **Match** enum.

If a method returns **yes**, the base event matches and should trigger a breakpoint.

However, if the base event does not match, we make a distinction between two possible situations. Either the query represented by the matcher contains a simple query related to the current base event but the condition does not hold true, or the base event is unrelated. If the condition is false, we return **no**, otherwise we return **skip**.

The use of **skip** is necessary to properly support connectives and will be discussed in detail in ‹Connectives› (section 4.7). Using a default method for each base event that just returns **skip** encodes that the matcher does not handle that particular type of event. More specific subclasses override the appropriate methods for the type of query they represent.

Following the same structure as ‹Query DSL› (section 3), we will now discuss the specific matcher subclasses for each Query DSL construct. As discussed in ‹Design Principles›

(section 2), the backend was designed to be efficient, so we will focus on the space and time requirements in particular.

4.2 Variables

```
private class VarMatcher extends Matcher {
    protected PredOP  op;
    protected Element left;
    protected Element right;

    protected final List<String> vars = new ArrayList<String>();

    public Match changedVariable(String name) { /* ... */ }
}
```

Variable queries can be seen as having a predicate operator and two arguments, which we call the left and right side. The only exception is the **Access** query which only compares the variable name, independent of any values.

In order to decide whether a base event matches, the matcher first needs to determine whether the variable name in the base event occurs in the query. If it does, then both sides need to be evaluated and then compared according to the predicate operator. The structure of the different variable matchers differs little beyond the predicate operator and what evaluation method is applied to the arguments.

As both sides may contain variables that need to be evaluated, the matcher needs access to the program state or previous base events. The bookkeeping system maintains just that mapping.

Note that the base event does not pass the variable value attribute itself in **changedVariable(String name)**. The bookkeeping system of the backend registers the current value in the **statics** map as part of the variable mapping. The event only pass the variable name to allow the distinction between **no** and **skip**.

In order to be more efficient, we didn't want to traverse the full query for every base event just to decide whether the variable name is used by the query. We added a preprocessing pass during instantiating that extracts all variable names first, so that we can perform a fast lookup.

```
protected List<String> findVars(SetElement el) {
    Set<String> names= new LinkedHashSet<>(); // avoid duplicates

    if (el instanceof VarName) {
        names.add(((VarName) el).getName());

    } else { // try to descend
        TreeIterator<EObject> it = el.eAllContents();
        while (it.hasNext()) {
            EObject o = it.next();

            if (o instanceof VarName)
                names.add(((VarName) o).getName());
        }
```

```
    }
    return new ArrayList<>(names);
}
```

We store the names in a simple array as queries typically contain very few names (often just a single one) and so a hash-based lookup would be slower than a direct comparison.

At the beginning of **changedVariable(String name)**, the name is looked up in the list of names and if it is not found, the matcher returns **skip**.

Otherwise, it attempts to evaluate both sides of the query:

```
Optional<Object> lvar = eval(this.left);
Optional<Object> rvar = eval(this.right);
```

The evaluation method currently only supports literal values, variables, and query variables. The syntax of the Query DSL can easily accommodate full expressions with binary operators, multiple variables, and parentheses. In fact, an earlier version of the Query DSL supported such expressions, but we decided to disable them for the time being until the evaluation method can be extended.

Evaluation may fail because some variable might still uninitialized at this point in the execution. In that case, the matcher returns **skip**.

With both sides reduced to a known value, all that is left to do is to apply the appropriate predicate, as shown here for **Comparison**:

```
public Match changedVariable(String name) {
    // ...
    boolean m = false;
    switch (this.op) {
    case EQ:        m = (lvar.get() == rvar.get());       break;
    case EQUALS:    m = (lvar.get().equals(rvar.get()));  break;
    case NEQ:       m = (!lvar.get().equals(rvar.get())); break;
    default:
        int cmp;
        try {
            cmp = ((Comparable<Object>) lvar.get())
                  .compareTo((Comparable<Object>) rvar.get());
        } catch(ClassCastException e) {
            throw new ParseException("invalid type comparison: "+e);
        }

        switch (this.op) {
        case GREATER:   m = (cmp > 0);  break;
        case LESS:      m = (cmp < 0);  break;
        case GREATEREQ: m = (cmp >= 0); break;
        case LESSEQ:    m = (cmp <= 0); break;
        default:                        break; // see above
        }
    }
    return m ? Match.yes : Match.no;
}
```

For variable queries that support binding query variables, the matcher also sets the correct value before evaluating the right side.

```
private class AssignMatcher extends VarMatcher {
    public Match changedVariable(String name) {
        // ...
        if (this.right instanceof SetQVar) {
            Optional<Object> lvar = eval(this.left);
            if (lvar.isEmpty()) return Match.skip;

            String qvar = ((SetQVar) this.right).getName();
            qvars.put(qvar, lvar);
        }
        // ...
    }
}
```

For type comparisons for **instanceof** queries, we compare the fully qualified type names using **equals()**, as that is the information that the instrumentation framework passes in its events.

Variable matchers only save the (unevaluated) query expression for each side of the variable query. During comparison, variable references are substituted with their values (except for **Access**). The size of the query expression is only dependent on the size of the query itself, which is expected to be very small, even with full expressions.

Evaluating a side is only necessary if it contains variables, otherwise it can be reduced to a single value during instantiation once and then reused.

There are no additional space requirements for variable matchers. The only computation is the necessary evaluation of both sides of the predicate and the time needed for the predicate itself, which are both fixed requirements for any variable query. Therefore, the variable matcher fits our efficiency goal.

4.3 Special Variables

The matchers for special variables have a very similar design to the previously discussed variable matchers. After all, that is why we chose to represent the source code context events as variable-like base events.

Unlike with variables, we pass the line number or class file name value directly as a value, as special variable queries cannot contain other variable names. The **source line change** base event can be handled very straightforwardly:

```
private class LineMatcher extends Matcher {
    private PredOP op;
    private int    val;

    public Match changedLine(int line) {
        switch (this.op) {
        case EQ:         return (line == val) ? Match.yes : Match.no;
        case EQUALS:     return (line == val) ? Match.yes : Match.no;

// (continued on next page)
```

```
11          case GREATER:    return (line >  val) ? Match.yes : Match.no;
12          case LESS:       return (line <  val) ? Match.yes : Match.no;
13          case GREATEREQ:  return (line >= val) ? Match.yes : Match.no;
14          case LESSEQ:     return (line <= val) ? Match.yes : Match.no;
15          case NEQ:        return (line != val) ? Match.yes : Match.no;
16          default:         return Match.skip;
17          }
18      }
19  }
```

The **class file load** matcher works analogously. The only difference is that it only performs a string equality test against the class file name.

```
1  private class ClassMatcher extends Matcher {
2      private String  val;
3      private Pattern rx;
4  }
```

If the value used by the query is a glob or a regular expression, we use the **java.util.regex** library for matching. We compile globs into regular expressions to simplify the implementation.

As with variable matchers, the special variable matchers fulfill our performance requirement. There is no additional space requirement beyond the expected value of the query. The **source line change** and **class file load** matcher perform only the single necessary comparison.

4.4 Method Calls

```
1  // passed as part of the base event
2  public class Method {
3      public String           name;   // method name
4      public List<Object>     args;   // method arguments
5      public Optional<Object> retval; // return value
6  }
```

The backend represents all method attributes as a **Method** object. The return value is **Optional.empty** during method call base events. The bookkeeping system links **method return** events to the correct method by maintaining its own call stack.

Both the **method call** and **method return** base events are handled by the same matcher class. The base event represents the difference with a single boolean flag **returned** that is set to true for returns. To decide whether the method query should return **skip**, the matcher compares the method name first.

```
1  // default method call and method return base event implementation
2
3  private class Matcher {
4    // ...
5    // Method Call (returned == false) and Method Return (returned == true)
6    public Match calledMethod(Method method, boolean returned) { return Match.skip; }
7    // ...
8  }
```

```
private class MethodMatcher extends Matcher {
    private String          method;      // expected method name
    private List<MethodArg> args;        // expected method arguments
    private Object          retval;      // expected return value

    private CallModifier    modifier;    // query type: Agnostic, During, or After
    private int             matches = 0; // state for During and After semantics

    public Match calledMethod(Method method, boolean returned) { /* ... */ }
}
```

The method matcher encodes the difference between **Agnostic**, **During**, and **After** using a **CallModifier** enum in the **modifier** field. We previously discussed the semantic differences in detail in ‹Connectives› (section 3.7).

In order to correctly decide whether a matched method is still true, the matcher needs the internal state **matches** to count how often the method was called and how often it returned. See the upcoming section ‹Connectives› (section 4.7) for details.

The only remaining step in matching a method base event is to compare each actual argument against the expected argument. In order to simplify and speed up the comparison, we first check the arity of the method and whether a comparison is necessary at all:

```
protected boolean matchSignature(Method m) {
    if (args == null) return true; // any signature allowed

    if (args.size() != m.args.size()) return false; // wrong arity

    for (int i=0; i<args.size(); i++) {
        if (! matchArgument(args.get(i), m.args.get(i))) return false;
    }
    return true;
}
```

One might suppose at first that a failure to match the signature should return **no** from the matcher, as the query did not match. The Java programming language allows overloaded methods with different signatures, however, and calls to different method variants should not be a failure to match, just as a base event for an entirely different method with a different method name would return **skip**, not **no**.

After all, the following two queries should be treated the same way:

```
// different method names
f(x):
    g(x, y):
        x = y;

// different method signatures
f(x):
    f(x, y):
        x = y;
```

The evaluation and comparison of each method argument is performed just as with variables matchers (see ‹Variables› (section 4.2)) and has the same overall efficiency. The

only additional overhead of the method matcher is the internal **matches** state, which is a single integer per method call query. As we will see in ‹Connectives› (section 4.7), the additional logic is minimal.

4.5 Exceptions

```
private class ExceptionMatcher extends Matcher {
    private final List<Type> caughts = new ArrayList<>();
    private Type thrown;

    public Match threwException(Throwable ex, boolean caught) {
        // analogue to VarMatcher
    }
}
```

We use a single matcher type to handle both **exception throw** and **exception catch** base events. The base event contains a boolean flag to communicate whether the exception was caught or thrown.

As with **instanceof** variable matchers, the exception matcher performs type comparison based on string equality. Its performance is similarly just a function of the query size and necessary comparison and therefore acceptable.

4.6 Query Organization

Each query is represented by its own (possibly nested) matcher instance. That way, all breakpoints are kept logically separate with their own automaton.

The backend maintains a list of active queries, so that the programmer can enable or disable them as necessary. The backend passes base events only to the active query automata, as shown here for the variable change base event:

```
for (QueryAutomaton qa : qa_active) {
    if (qa.changedVariable(name)) breakpoint(qa);
}
```

In order to handle fully-qualified names consistently, we decided to qualify all names used in a query during the instantiation of the automaton:

```
// ...
if (e instanceof Import) {
    pkg = ((Import) e).getPkg().getName();
}
// ...

if (e instanceof VarName) {
    if (! varname.contains(".")) {
        // not qualified yet, so let's qualify it against the active package
        varname = pkg + "." + varname;
    }
}
```

That way, all names used inside a matcher have the same format and can be compared using a simple string comparison.

The current automaton does not support nesting queries through the use of query names, as the correct implementation of mutually recursive calls was beyond the scope of this thesis.

4.7 Connectives

In order to implement the connective operators, we first broke down the problem into two parts – the evaluation of its branches and a defining predicate.

Connectives are composed of two branches which are themselves queries, so the connective matcher would need to contain references to the matchers corresponding to those branches. The connective matcher needs to pass the base event to one or both of its branches (depending on the specific operator) and receive the results.

In a second step, the matcher applies the correct predicate for the operator to decide whether the base event matches the connective or not.

All connective operators follow this basic structure. We noticed that the connectives differ only in whether they enforce an ordering, and what predicate they use to combine their two branches.

Therefore, we decided to implement an abstract ConnectiveMatcher class that generalizes the shared behavior of passing the base events to its branches, and defines an abstract **match(left, right)** method that represents the specific predicate of the connective.

That way we could use the same simple base event implementations for all connectives. Each connective operator would only need a single method implementation to define its full behavior.

It is important that the **match()** method does not receive the evaluated results of its branches directly as **Match** values, as that would require us to always evaluate both branches. In particular, that would force us to evaluate the right branch even when that is not the desired behavior, as with the **then** operator should the left branch fail to match, as required by the Query DSL design in ‹Connectives› (section 3.7).

We considered using two interfaces, one in which **match()** receives the results directly, and a separate implementation for **then** and **:**. However, that would lead to a significant amount of code duplication, especially for passing the base events to the branches. That way, only two matchers – **and** and **or** – would be able to share the simpler interface. Furthermore, this design would also preclude the possibility of optimizing the connectives later to avoid evaluating branches if we can statically determine that it is safe to do so.

Instead, we used a design for **match()** that receives the the unevaluated branches as lambda arguments.

```
private abstract class ConnectiveMatcher extends Matcher {
    public Matcher left_branch, right_branch;

    protected abstract Match match(Supplier<Match> left,
                                   Supplier<Match> right);

    public Match changedVariable(String name) {
        return match(() -> left_branch.changedVariable(name),
                     () -> right_branch.changedVariable(name));
    }
    // ditto for remaining base events
}
```

We used lambda expressions to wrap the left and right branch argument to control the evaluation order and let the predicate evaluate the branches as needed.

If we had implemented the automaton with a code generator the way we considered in ‹Design› (section 4.1), then the additional level of indirection of using lambda expressions would not be necessary. The code generator could insert the unevaluated source code of the right branch in the correct position instead.

Except for the **or** connective, the other connectives need to match at least two separate base events, ie. one for each branch. Because the automaton should be efficient, it should not be necessary to perform a possibly open-ended backwards search of the execution history, so we decided to add state to the matcher so it can track which branches have already matched a previous base event.

To properly support the **During** and **After** semantics of method calls in connectives, we needed to extend the method call matcher with an additional return state variable. We will discuss this extension after we have presented the connective matchers.

Or

The **or** connective matches a base event if either of its branches matches it. It only ever needs to match a single branch to match a base event, so our implementation needed no state and just a single conditional.

```
private class OrMatcher extends ConnectiveMatcher {
    @Override
    protected Match match(Supplier<Match> left, Supplier<Match> right) {
        Match ml = left.get();
        Match mr = right.get();

        // nb: the ternary operator takes care of our Match logic,
        // it does not represent logical short-circuiting
        return (ml == Match.yes) ? Match.yes : mr;
    }
}
```

As discussed in ‹Connectives› (section 3.7), it is not possible to skip the evaluation of a branch even when it is already known that the other branch matches because query variables in particular can have side-effects.

And

The **and** connective needs to match both of its branches, so it is necessary to remember whether a branch has matched for a previous base event. We added two **Match** fields to the matcher to represent that state.

Additionally, the **and** connective needs to unset that state for a branch if it is no longer true, as discussed in ‹Connectives› (section 3.7).

To support invalidating the saved state of a branch, we added the **yes** / **no** / **skip** distinction introduced in ‹Design› (section 4.1).

Otherwise, if **skip** and **no** were collapsed into one **no** state, a branch that already found a matching base event might be set to **yes**, and then upon encountering a base event unrelated to its query could not tell the difference to its original query no longer matching. The branch would end up falsely set to **no** and the **and** matcher would not behave as expected.

```
private class AndMatcher extends ConnectiveMatcher {
    private Match m_left  = Match.skip;
    private Match m_right = Match.skip;

    @Override
    protected Match match(Match ml, Match mr) {
        // save the current truth state of the left branch
        switch (ml) {
        case yes:  m_left = Match.yes; break;
        case no:   m_left = Match.no;  break;
        case skip: // unchanged
        }

        // save the current truth state of the right branch
        switch (mr) {
        case yes:  m_right = Match.yes; break;
        case no:   m_right = Match.no;  break;
        case skip: // unchanged
        }

        // if both branches skipped, then the base event is unrelated to this query
        if (ml == Match.skip && mr == Match.skip) return Match.skip;

        // match if both branches are true
        if (m_left == Match.yes && m_right == Match.yes) return Match.yes;
        else return Match.no;
    }
}
```

Then, :

We implemented two separate matchers for the **then** and **:** connective. We will discuss **:** first, as it is the more important case.

The core functionality of the **:** matcher is the same as the **and** matcher. However, we do not need two state fields. The **:** connective matches its branches in order, so we only need to remember the state of the left branch.

However, we have the additional complication that we should only evaluate the right branch if the left branch has matched successfully before, otherwise we might end up corrupting the state of the right matcher with a spurious base event.

Similarly, we must also ensure that a single base event cannot match both branches by itself. Consider the following example:

```
// match if f() is called three times (before returning)
f(): f(): f();
```

Clearly a call to **f()** would match by branches of the first **:** connective. If we passed the base event to the right branch right away when encountering the first method call, we would in effect collapse the query to just matching f();. We would have no way to encode a nested breakpoint like in our example.

To control the evaluation of the right branch, we used an abstract **match()** method that receives lambda expressions representing each branch. That way, the **match()** implementation can evaluate only the branches it requires.

The invalidation of the saved state for the left branch works the same way as for the **and** operator.

```
private class ThenTrueMatcher extends ConnectiveMatcher {
    private Match m_left = Match.skip;

    protected Match match(Supplier<Match> left, Supplier<Match> right) {
        // definitely evaluate the left branch,
        // but only evaluate the right one if the left one matched before
        Match ml = left.get();

        switch (ml) {
        case skip: // saved state unchanged
            if (m_left == Match.yes) return right.get();
            else                     return m_left;

        case no: // update state
            m_left = Match.no;
            return Match.no;

        case yes: // update state
            if (m_left == Match.yes) {
                return right.get();
            }    else {
                // update, but stop here
                m_left = Match.yes;
                return Match.no;
            }
        }
    }
}
```

The defining difference between **:** and **then** is that **then** only looks for multiple sequential events that match its branches, but is not concerned with the question whether the condition expressed by those branch queries still holds or not.

Therefore, the matcher for **then** does not need the logic to invalidate the saved branch state. The matcher for the **then** connective still needs to track the state of left branch, but failing to match a similar base event later does not invalidate that state once it has been set to **yes**.

Note that as with **or**, we cannot skip the evaluation of the left branch in general, as queries can have side-effects due to the presence of query variables. See ‹Connectives› (section 3.7) for the semantic argument. As with **or** would be possible, it would be possible to perform static analysis of the left branch and mark it as skippable once it is set to **yes** if it contains no stateful matchers.

```
// match if x has the values 1, 2, 3, in order
x = 1 then x = 2 then x = 3;
```

```
private class ThenSeqMatcher extends ThenTrueMatcher {
    private Match m_left = Match.skip;

    protected Match match(Supplier<Match> left, Supplier<Match> right) {
        // definitely evaluate the left branch,
        // but only evaluate the right one if the left one matched before
        Match ml = left.get();

        // update state if the left branch matches
        if (ml == Match.yes) m_left = Match.yes;

        // nb: the ternary operator takes care of our Match logic,
        // it does not represent logical short-circuiting
        return m_left == Match.yes ? right.get() : ml;
    }
}
```

Method Call Semantics

Finally, as mentioned in ‹Method Calls› (section 4.4), we needed to extend the method matcher to support the **During** and **After** variants. In order to know whether a method has already returned or not, the automaton needs a similar kind of state as the connectives.

More specifically, the method matcher needed to be able to connect a **method return** base event to the corresponding previous **method call** base event.

We considered two possible implementations for the method matcher:

1. Record the entire call history and search backwards for the matching **method call** base events.

2. Use a call stack to connect **method return** base events with **method call** base events, and add a counter how often each method has been called and returned from so far.

Recording the call history would be fully general and support any operator we might want. If we supported a language with unlimited continuations that ability would be very desirable, but for the more restricted Java programming language, we considered that generality excessive.

The call history would also require a potentially very large amount of memory because it will continue to grow indefinitely during execution. It would be possible to restrict the call history to only the method calls that are referenced in a query, but the unboundedness would remain.

Instead, using a call stack would be sufficient to map **method return** base events to their correct **method call** base events. If the instrumentation framework exposed either the actual call stack of the runtime or if its return events were to be extended to add the missing method attributes, then the backend call stack would be unnecessary.

As the Java runtime has a fixed-size call stack, our call stack will have the same size restriction.

However, it is not enough to just know which method a **method return** base event belongs to, as we also need to be able to handle nested calls to the same function (eg. `f(): f(): f();`).

We solved that problem by adding a counter to each method matcher that gets incremented by 1 each time it matches a **method call** and is decremented by 1 each time it matches a **method return**. That way, we only need a single counter and we can still distinguish **Agnostic**, **During**, and **After** by the difference.

```
call  f(); // -> match on call event, no count needed
after f(); // -> count == 0
in    f(); // -> count >  0
```

Therefore the only new overhead for the method matcher is a single integer to track the count, and an increment or decrement operation for each successful match. There is no unbounded space requirement, and only a fairly minimal runtime cost, so we considered this option acceptable.

```
private class MethodMatcher extends Matcher {
    public Match calledMethod(Method method, boolean returned) {
        // ..
        switch (this.modifier) { // type of call query

        case ANY: // Agnostic
            // count does not matter
            return returned ? Match.skip : Match.yes;

        case CLOSED: // After
            // check that we had at least one call before
            if (!returned) {
                // method call
                this.matches = 1;
                return Match.skip;
            } else {
                // method return; match if all open calls have been closed
                return (this.matches > 0) ? Match.yes : Match.skip;
            }

// (continued on next page)
```

```
22          case OPEN: // During
23              // count the number of closed calls
24              if (!returned) this.matches += 1;
25              else           this.matches -= 1;
26
27              // the method is active as long as there are open calls
28              if (returned) return Match.skip;
29              else          return this.matches > 0 ? Match.yes : Match.no;
30          }
31          // ...
32      }
33  }
```

4.8 Query Variables

```
1  public class QueryAutomaton {
2      // ...
3      public  Map<String, Optional<Object>> qvars;
4      public  Map<String, String>           qtypes;
5  }
```

Query variables can be divided into the evaluation of a query variable, and setting a binding.

Using query variables as values (or types) is completely analogous to the other variables supported throughout the Query DSL, so all that was necessary was a separate mapping for query variables and query type variables. The additional space requirement depends only on the number of query variables used and is generally minimal.

The extension to the **Assignment** matcher is a typical example:

```
1  private class AssignMatcher extends VarMatcher {
2      // Variable Change base event
3      public Match changedVariable(String name) {
4          // skip if the variable is not used in the query
5          if (!vars.contains(name)) return Match.skip;
6
7          // is the right side a query variable binding?
8
9          if (this.right instanceof SetQVar) { // set qvar
10             Optional<Object> lvar = eval(this.left);
11
12             // skip if the left side is still uninitialized
13             if (lvar.isEmpty()) return Match.skip;
14
15             // update query variable
16             String qvar = ((SetQVar) this.right).getName();
17             qvars.put(qvar, lvar);
18
19             return Match.yes;
20
21 // (continued on next page)
```

```
22          } else { // no query variable
23
24              Optional<Object> lvar = eval(this.left);
25              Optional<Object> rvar = eval((Element) this.right);
26
27              // skip if either side is still uninitialized
28              if (lvar.isEmpty() || rvar.isEmpty()) return Match.skip;
29
30              return (lvar.get().equals(rvar.get())) ? Match.yes : Match.no;
31          }
32      }
33  }
```

In order to allow the binding of a query variable in a limited number of places in the Query DSL, we extended the evaluation method to also update the mapping as needed. Note that this modification makes some matchers stateful, as they can modify the query variable mapping. It is therefore very important that matchers only actually set a binding once they have ensured that the query does in fact match, or otherwise an unmatched query could corrupt the query variable mapping. As discussed in ‹Connectives› (section 4.7), the connective matchers also need to ensure that they pass on base events correctly.

The performance overhead of query variables is overall minimal. In particular, it does not involve any constraint solving or other sophisticated forms of abstraction that might have unacceptable performance, but it still enhances the expressive power of the Query DSL.

5 Evaluation

5.1 Principles

How well do the Query DSL and its automaton implementation live up to the design principles we laid out in ‹Design Principles› (section 2)?

The entire implementation and the semantics of connectives are designed fully around base events. The event-based design is a natural fit for the runtime principle and dictated by the existent instrumentation framework, which enforces an event-based interface using callbacks for our implementation.

The only significant deviation is our simplification of the supported instrumentation events into eight base events, but we only do so to create a simpler interface for our automaton and design analysis. The functionality is not otherwise affected.

Our backend only communicates with the instrumentation framework through events and does not interact with the running program except to access necessary information about values such as their types. The backend does not influence the instrumentation or execution, except of course to signal a breakpoint, so the principle that the breakpoint backend should be runtime-based is also met. Additionally, we support enabling and disabling breakpoints during execution, a major advantage of the runtime approach.

We met the limitation that we needed to perform some redundant bookkeeping for the automaton, namely the mapping of variable names to values, and the call stack. We will discuss the performance overhead in the next section.

We also found the lack of information about local scopes by the instrumentation framework problematic for our initial goal to support local variables. However, even if the correct scope information was available to us or we reconstructed it correctly, we would've still not been able to refer to local variables in the textual queries.

When considering the principle that the Query DSL should be ergonomic for the programmer, we found it useful to focus on two aspects:

1. How intuitive is the Query DSL to read and write for a Java programmer?

2. How well does the Query DSL fit the existing conventions for Java source code?

We achieved an acceptable trade-off between unambiguously specifying queries and the number of additional constructs. While the use of unfamiliar prefixes to distinguish types and special variables is unfortunate, their syntactic cost is small overall.

We found that all breakpoints we wanted to express, many examples of which we used as examples in the thesis, had a compact and straightforward representation in the Query DSL. We consider the first requirement, that the DSL should be easy to read and write, successfully met.

We also don't stray far from Java's syntactic conventions. Most of our innovations are themselves widely used in other popular programming languages or pseudocode, so the Query DSL fulfills this requirement as well.

The main syntactic limitation in the Query DSL is the lack of first-class support for objects. That is also a limitation of the existing instrumentation framework, which is similarly restricted to primitive types and opaque objects. We consider this a serious shortcoming of the Query DSL, but an analysis of the appropriate semantics for supporting instances and instance methods as values was beyond the scope of this thesis and would likely be of comparable difficulty to the entirety of the rest of the Query DSL.[8]

The Query DSL and automaton support query names and query variables as a means to increase the expressive power of the language. The previous chapters have shown that their implementation cost is well worth the increased generality and expressiveness.

One might bemoan that query variables introduce side-effects and hinder some optimizations, but as we saw in ‹Connectives› (section 4.7), method calls are also internally stateful in order to implement the **During** and **After** semantics.

We found it too difficult to attempt an automaton implementation that is internally stateless, so the inherent state in the Query DSL did not create any additional problems for the implementation or available optimizations.

Additionally, in the case of connectives, the programmer might be interested in knowing why the query matched, so potentially only presenting half the reason by skipping the evaluation of a redundant branch would be misleading.

Lastly, the goal was to develop a more efficient implementation of a debugging backend compared to the XQuery-based design in [BSWK21]. ‹Automaton Design And Implementation› (section 4) has demonstrated that queries introduce very limited memory overhead that is independent[9]of the runtime of the program. Additionally, the amount of additional computations is generally very small and does not depend on more than one base event at a time.

From a design perspective, the Query DSL and automaton meet the original performance goal. We will consider a simple set of benchmarks to validate that initial judgment.

[8] A programming language with stronger guarantees about side-effects and structural comparisons such as Standard ML[MTM97] might be a better starting point.

[9] Strictly speaking, the performance of the automaton could degenerate over time because calls to **equals()** might scale badly over time, for example when comparing growing tree structures. However, that performance penalty would be inherent in the breakpoint specification and would not be caused by the backend itself.

5.2 Performance Evaluation

The breakpoint backend builds on the previous DiSL-based approach from [BSWK21]. As such, the overhead introduced by the instrumentation framework or similar details of the underlying implementation are not part of the scope of this work.

The instrumentation framework supports a dummy backend called the **Null Tracer** that implements the callback interface in the same way our backend does. As the name indicates, the Null Tracer does no additional work and is therefore the ideal comparison for our backend. Any cost incured by the Null Tracer can be considered outside our scope.

Additionally, we are going to evaluate the performance of the Query DSL by splitting the backend into two parts, the bookkeeping backend and the automaton itself.

As we discussed in ‹Automaton Design And Implementation› (section 4), the instrumentation framework does not easily expose the current values bound to variables or the full call stack. We considered extending the framework, but focused our efforts on the automaton instead. Because of that limitation, our backend needs to perform its own bookkeeping for variables and methods. By extending the instrumentation framework, that same information could be accessed with little additional cost, as it is already an aspect of the normal execution of a program inside the Java Virtual Machine.

We assembled a set of 17 short benchmarks that represent all constructs of the Query DSL and cover the range of examples shown in other chapters.[10]

As expected, we found that the performance of the benchmarks does not change after repeated executions,[11] so the goal to not have the performance degrade over time was clearly met. As the automaton does not keep a full execution trace, that result is not surprising.

We measured repeated executions of the benchmark set (with $N = 10,000$) and took the median execution time (in milliseconds) for each benchmark to reduce measurement noise. As many of our measurements are of the size of one millisecond or less, noise introduced by JIT optimizations or garbage collection was a significant factor, so the whole measurement should be taken only as a rough approximation. (‹Back-End Benchmarks› (figure 1))

Notably, the bookkeeping system outperforms the Null Tracer in two benchmarks by less than $1ms$, which is almost certainly just a measurement artifact. The goal of the performance evaluation is to confirm the plausibility of the claim that our design is

[10] See ‹Query DSL Grammar› (appendix B) for the queries we evaluated. The minimal test cases are part of the source code repository available under https://segit.mathematik.uni-marburg.de/ExTra/extra-base.

[11] The only difference we observed is that the first few measurements are typically outliers, which we strongly suspect is due the execution of cold code in the JVM. We correct for this artifact by performing two full measurements for each benchmark after another and discarding the first one.

suitable for a practical backend implementation, not to assess the exact overhead of our implementation.

The bookkeeping overhead for most benchmarks is on average a factor of 4 to 5, still well inside the millisecond range. The overhead introduced by most queries is a factor of 2 or less, but more complex or costly queries can produce a slowdown of up to a factor of 20. Overall, the total performance overhead of our implementation is on average a factor of 8.

Additionally, the overhead of the automaton compared to the redundant bookkeeping is generally small except for significantly nested queries queries involving multiple **Then** or **:** connectives. (‹Bookkeeping vs Automaton› (figure 2))

The XQuery implementation in [BSWK21] noted overheads of up to 300 and more in their preliminary benchmarks, in addition to the growing memory cost of keeping a full execution trace.

We consider a total factor of about 8 an acceptable overhead for the first implementation of our debugging backend, but many low-level optimizations would still be possible. Using code generation as discussed in ‹Design› (section 4.1) would also likely remove many redundant checks and layers of indirection. Most importantly, the entire bookkeeping system could be removed by extending the instrumentation framework at little additional cost.

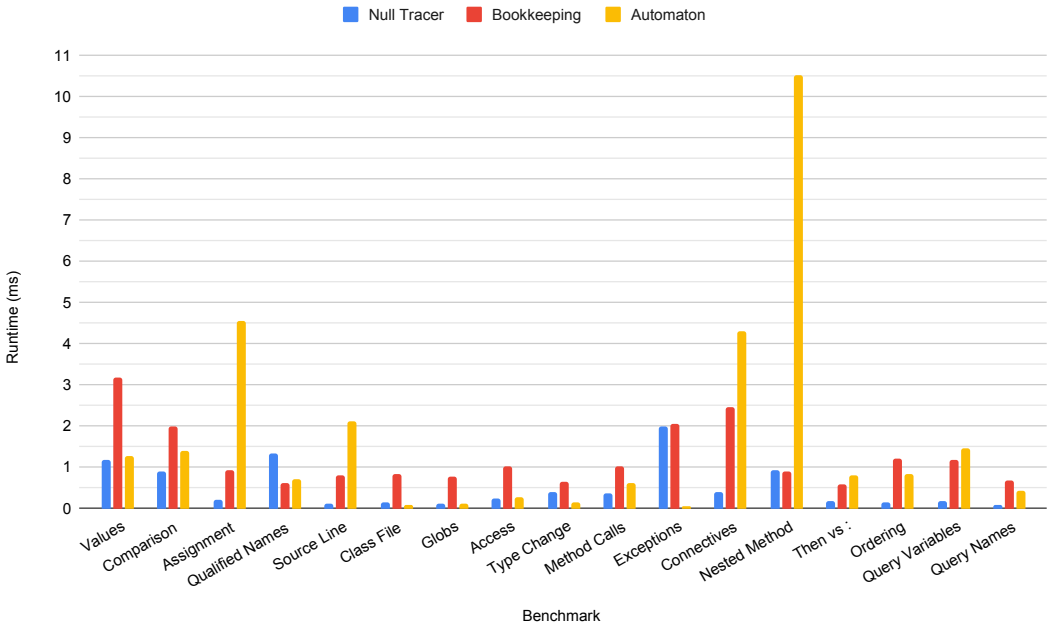

Figure 1 Back-End Benchmarks

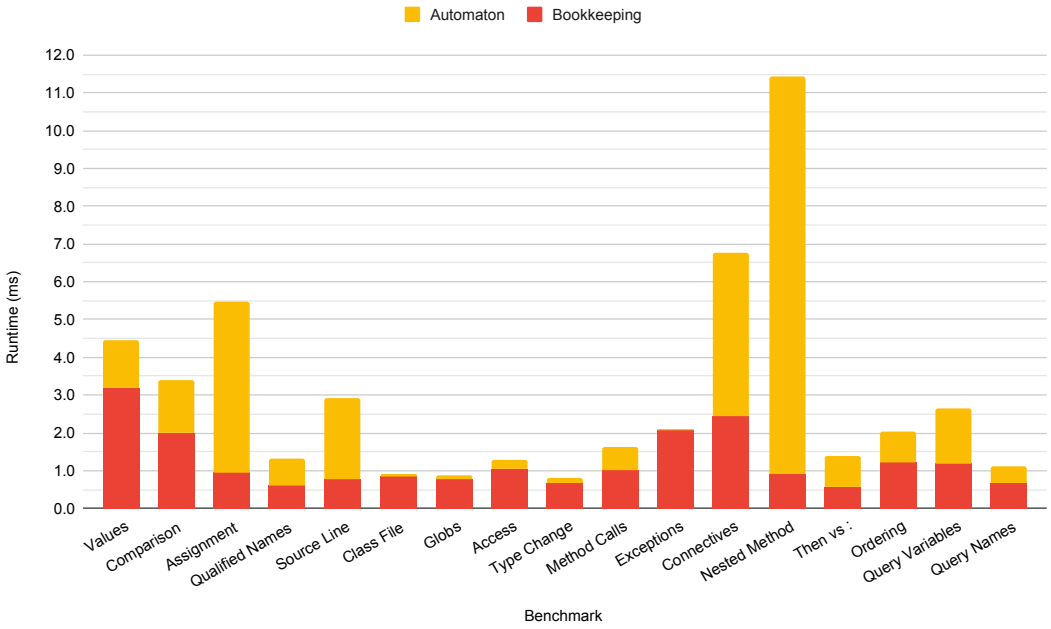

Figure 2 Bookkeeping vs Automaton

6 Related Work

Bockisch et al.[BSWK21] introduce a unified specification for breakpoints. They model execution as a sequence of events represented as an XML stream. Each event is categorized into one of seven classes of base events comparable to the base events used in ‹Design› (section 3.1). A breakpoint is defined by its base event and a condition over its attributes, the dynamic program state, and the execution history.

Breakpoints are expressed as search queries for the event stream using XQuery. They map the base events and a number of composition operators to XQuery and discuss the possibility of using XQuery's functional language features as a means of abstraction for breakpoints.

In addition to the specification, they implement a proof-of-concept framework for XQuery search queries on an XML event stream generated from bytecode instrumentation. The implementation suffers severe performance penalties do to the overhead of encoding events as an XML character stream, the need to maintain and search through a full execution history, and the large number of instrumentation sites due to the indiscriminate generation of all base events, independent of what events are actually relevant to a given search query.

De Volder[DV06] develops a declarative configuration language for a generic code browser. In order to improve the customizability of IDEs, the author analyzes the design space of various navigation tools offered by the Eclipse Java Development tools[ecl], such as view panels for type hierarchies, Java packages, and statically known call sites. Next, they model all such code browsers as search queries over the static structure of the program source code, and declarative rules on how to display the search results.

Of particular interest are the JQuery predicates that define the search queries, which have a close syntactic similarity to Prolog[SS94]. Predicates select static entities such as methods or types from the source project and bind the results to variable names in a similar style to the query variables used in this thesis. Afterwards, the results are displayed in a tree view based on a hierarchy generated from declarative rules. That way, all code navigation tools in the IDE use a unified design space and configuration language, and the programmer can easily add customized views according to their specific needs.

Because JQuery is implemented as an extension to an inference engine in the Prolog family, it offers significant flexibility and forms of abstraction. The author also significantly reduces the size of the design space from over 3,000 original types and methods in the Eclipse plugin API to just 13 JQuery types and 53 predicates.

7 Conclusion

Breakpoints are a powerful tool for debugging programs. Real-world debuggers currently include breakpoints through a wide range of divergent and non-composable commands.

We designed a unified domain-specific language for queries to express breakpoints in an ergonomic and general way. We used a range of design principles to guide the design of the syntax and semantics of the Query DSL.

We used a runtime-based design built on a categorization of program instructions into a set of eight base events. The Query DSL closely follows the conventions and semantics of the Java programming language and extends the expressive power of conventional breakpoints through query variables and connectives.

Additionally, we implemented an efficient automaton and backend for the Query DSL, using the existing instrumentation framework used by [BSWK21]. The backend consists of a bookkeeping system to complete the necessary runtime information otherwise provided by the instrumentation framework, a parser and compiler for the Query DSL, and an efficient automaton generated by the query compiler.

We met our design goals with only a small number of design restrictions and trade-offs, such as the use of type prefixes, and the lack of support for local variables. The primary limitation of the current design is the lack of first-class objects and instance methods. Otherwise, we have achieved our initial goal to design and implement a unified breakpoint language and a corresponding backend for an instrumentation framework.

A cursory performance evaluation of the backend confirmed the efficiency of our design and justifies its validity for a practical debugging tool. The backend has good, effectively constant memory usage during execution and a total linear runtime overhead of less than a factor of 10, split between a factor of 4-5 for the bookkeeping system and 2 or less for most query automata.

Our Query DSL with its efficient backend implementation constitutes a powerful, human-readable language for breakpoints.

8 References

[Bet16] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend*. Packt Publishing, 2nd edition, 2016.

[BMS80] Rod M Burstall, David B MacQueen, and Donald T Sannella. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 136–143, 1980.

[BPSM+00] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (XML) 1.0, 2000.

[Bra17] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017. URL: https://rfc-editor.org/rfc/rfc8259.txt, doi:10.17487/RFC8259.

[BSWK21] Christoph Bockisch, Stefan Schulz, Viola Wenz, and Arno Kesper. A Unifying Approach to Breakpoint Specification. In *24th Iberoamerican Conference on Software Engineering*. CIbSE Steering Committee, aug 2021.

[CLB] Common Lisp HyperSpec, function BREAK. http://clhs.lisp.se/Body/f_break.htm.

[DiS] DiSL source code repository. https://gitlab.ow2.org/disl/disl.

[DV06] Kris De Volder. JQuery: A Generic Code Browser with a Declarative Configuration Language. In Pascal Van Hentenryck, editor, *Practical Aspects of Declarative Languages*, pages 88–102, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309, 2010.

[ecl] Eclipse IDE. https://www.eclipse.org/eclipseide/.

[gdb] The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[lam] Lamdu project overview. https://www.lamdu.org/.

[lsp] Language Server Protocol Specification. https://microsoft.github.io/language-server-protocol/specifications/specification-current/.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[MVZ+12] Lukáš Marek, Alex Villazon, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. DiSL: A domain-specific language for bytecode instrumentation. *AOSD'12 - Proceedings of the 11th Annual International Conference on Aspect Oriented Software Development*, 03 2012. doi:10.1145/2162049.2162077.

[MWG01] Erik Meijer, Redmond Wa, and John Gough. Technical Overview of the Common Language Runtime. *Microsoft Research*, 11 2001.

[PQ95] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250705, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250705, doi:10.1002/spe.4380250705.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley, 2 edition, 2009.

[SBS01] Robert F. Stark, E. Borger, and Joachim Schmid. *Java and the Java Virtual Machine: Definition, Verification, Validation with Cdrom*. Springer-Verlag, Berlin, Heidelberg, 2001.

[SS94] Leon Sterling and Ehud Y Shapiro. *The art of Prolog: advanced programming techniques*. MIT press, 1994.

[TR75] Ken Thompson and Dennis M Ritchie. *unix Programmer's Manual*. Bell Telephone Laboratories, 1975.

[tra] Tracepoints, an introduction on the official Microsoft DevOps Blog. https://web.archive.org/web/20190109221722/https://blogs.msdn.microsoft.com/devops/2013/10/10/tracepoints/.

[Tur76] DA Turner. SASL language manual, St. *Andrews University, Fife, Scotland*, 1976.

[VBD+13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[vsc] Visual Studio Code. https://code.visualstudio.com/.

[Vö11] Markus Völter. Language and IDE Modularization, Extension and Composition with MPS. *GTTSE 2011*, 7680, 07 2011. doi:10.1007/978-3-642-35992-7_11.

[wat] Setting watchpoints, GNUD Debugger manual. https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_29.html.

[win] Debugging Tools for Windows (WinDbg, KD, CDB, NTSD). https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/.

Appendix

A Query DSL Grammar

```
grammar io.bitbucket.umrplt.extra.Query hidden(WS, ML_COMMENT, SL_COMMENT)
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate query "http://www.bitbucket.io/umrplt/extra/Query"

// top-level

QueryFile:
    (entries+=Entry)+;

Entry:
    Section | Query | Import;

Section:
    '{' note=QName '}';

Import:
    'package' pkg=PkgName ';'+;

Query:
    (name=QueryName '=>')? condition=Condition ';'+;

QueryCall:
    '<' name=[QueryName] '>';

Condition:
    left=SimpleCondition (=> connective=ConnectiveOP right=Condition)?;

SimpleCondition:
    VariableChanged | MethodCalled | Return | Exception | QueryCall;

// variables

VariableChanged:
    Comparison | LineChanged | ClassChanged | Assigned | Accessed | InstanceOf;

Accessed:
    var=VarName;

Comparison:
    left=Element pred=PredOP right=Element;

LineChanged:
    '$line' pred=PredOP val=Integer;

ClassChanged:
    '$class' ('=' | '==') val=Text;

Assigned:
    var=VarName '<-' right=SetElement;

InstanceOf:
    obj=Var 'instanceof' type=SetType;

// (continued on next page)
```

```
// methods

MethodCalled:
    (modifier=CallModifier)? call=MethodCall;

MethodCall:
    (method=MethodName)'(' (args=MethodArgs)? ')';

// nb: ANY is the default case
enum CallModifier:
    ANY    = 'call' |
    OPEN   = 'in'   |
    CLOSED = 'after';

MethodArg:
    var=SetElement              |
                  type=SetType  |
    var=SetElement type=SetType ;

MethodArgs:
    void?='void' | (args+=MethodArg (',' args+=MethodArg)*);

Return:
    'return' {Return} (retval=(Value | SetVar))? ('from' method=MethodCall)?;

// exceptions

Exception:
    Catch | Throw;

Catch:
    'catch' (types+=Type('|' types+=Type)*) (qtype=SetQType)?;

Throw:
    'throw' (type=Type) (qtype=SetQType)?;

// names and values

enum PredOP:
    EQUALS  = '='  | EQ        = '==' | NEQ = '!=' |
    LESS    = '<'  | LESSEQ    = '<=' |
    GREATER = '>'  | GREATEREQ = '>=' ;

enum ConnectiveOP:
    AND       = 'and'  |
    OR        = 'or'   |
    THEN_SEQ  = 'then' |
    THEN_TRUE = ':'    ;

terminal ID: ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
QName:   ID ('.' ID)*;

// (continued on next page)
```

```
VarName:              (name=QName);
MethodName:           (name=QName);
PkgName:              (name=QName);
QueryName:            (name=QName);
TypeName:           '@'(name=QName);
QVar:               '?'(name=QName);
QType:             '?@'(name=QName);
SetQVar:           '?='(name=QName);
SetQType:         '?@='(name=QName);

Var:  VarName  | QVar;
Type: TypeName | QType;

SetVar:  Var  | SetQVar;
SetType: Type | SetQType;

Element:
    Value | Var;

SetElement:
    Element | SetQVar;

Value:
    {Null}    val=Null    |
    {Integer} val=Integer |
    {Float}   val=Float   |
    {STRING}  val=STRING  |
    {GLOB}    val=GLOB    |
    {REGEX}   val=REGEX   |
    {Bool}    val=Bool    ;

Text:
    {STRING} val=STRING |
    {GLOB}   val=GLOB   |
    {REGEX}  val=REGEX  ;

Integer returns ecore::EInt:
    ('+' | '-')? DIGIT+;

Float returns ecore::EFloat:
    ('+' | '-')? DIGIT'.'DIGIT;

Bool returns ecore::EBoolean:
    'true' | 'false';

Null: 'null';

terminal DIGIT returns ecore::EInt: ('0'..'9')+;
terminal STRING:
    '"' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|'"') )* '"';

terminal GLOB:
    '[' ( '\\'('*'|'?'|'.'|'['|']'|'\\') | !('\\'|']') )* ']';

terminal REGEX:
    '/' ( '\\'('/'|'\\') | !('\\'|'/') )* '/';

terminal ML_COMMENT : '/*' -> '*/';
terminal SL_COMMENT : '//' !('\n'|'\r')* ('\r'? '\n')?;
terminal WS         : (' '|'\t'|'\r'|'\n')+;
```

B Query DSL Grammar

```
// overview of all supported query features, used for benchmarks

package QueryTest;

////////////////

// variables

{ testValues }

// basic values
x      =  10;
y      = -10;
a      =  20.0;
name   = "freya";
valid  = true;
ret    = null;

{ testComparisons }

y == 10;
y <  10;
y <= 10;
y >  10;
y != 10;

{ testAssign }

y <- 10;

{ testName }

QueryTest.name = "freya";

{ testLine }

$line = 145;
$line > 145;
$line < 150;
$line == 145;

{ testClass }

$class = "QueryTest";

{ testGlob }

$class = [debug.*];
$class = [*.debug.*];

{ testAccess }

x;

// (continued on next page)
```

```
{ testInstanceOf }

ret instanceof @String;

{ testMethods }

// any arguments, the sensible default
f();
QueryTest.f();

// overloaded methods
f(void);   // explicit 0-ary call
g(@String); //
g(@String, @int, @String);

// simple arguments
f(3);
f(@int);

f("Freya");
f(@String);

f(@String, 10);
f("Freya", @int);
f(@String, @int);

// return values
return 10;          // break if a specific value is returned (from any method)
return    from h(); // break if the method f returns (with any value)
return 10 from h(); // break if the method f returns the value 10;

{ testExceptions }

throw @NullException;

catch @NullException;
catch @NullException | @DivError;

{ testConnectives }

// or

x = 10 or y = 20;
f() or g();
f() or throw @NullPointerException;

x = 10 or x = 20 or x = 30;

// and

x = 10 and y = 20;
y = 20 and x = 10;

f() and g() and h();

// in-order

x = 10:
  y = 20;

// (continued on next page)
```

```
{ testNested}

/* void outer() { inner();   }
 * void inner() { x = foo(); }
 */

outer():
  inner():
    x = 10;

{ testInOrder }

x = 10:
  y = 20;

/* ie, this wouldn't cause a break:
 * int f() {
 *   int y = 20;
 *   int x = 10;
 *   return x;
 * }
 *
 * but this would:
 * int f() {
 *   int x = 10;
 *   int y = 20;
 *   return x;
 * }
 */

{ testUsed }

// difference between "in order" and "while still active"

/* void init{ ... }; => initializes some data
 * void use{ ... };  => uses that data, expects it to be initialized
 *
 * so there's two possible errors: calling use() before init(),
 * and calling use() during init();
 */

init():       use();
use():        init();
in init():    use();
after init(): use();

{ testOrdering }
in f():
  QueryTest.name;

f():
  Student.name;

after f():
  Student.name;

return from f(): // (equivalent)
  Student.name;

// (continued on next page)
```

```
// sequence vs both true
x > 10:     x < 10; // contradiction
x > 10 then x < 10; // sequence

{ testQVars }

x <- ?=a;

a = b;
a <- ?=x and b <- ?=x;
a <- ?=x: b = ?x;

f(?@=T) and g(?@=T);
f(?@=T): g(?@T);
f(?@=T): ret instanceof ?@T;

g(?@=T, ?@T, ?@T);
g(?@=a, ?@=b, ?@=c);
// f(?@_, ?@_, ?@_);
f(?=x, @String);

f(?@=a): x instanceof ?@a;
f(?=a):  x = ?a;

{ testQNames }

name_1 => x = 1;
y = 1 or <name_1>;
```