XQuery (scripting) debugging
IDE and engine support

Master Thesis

Author(s):
Petrovay, Gabriel

Publication date:
2008

Permanent link:
https://doi.org/10.3929/ethz-a-005575230

Rights / license:
In Copyright - Non-Commercial Use Permitted

XQuery (Scripting) Debugging

IDE and engine support

Gabriel Petrovay

Master Thesis

Databases and Information Systems Group
Department of Computer Science
ETH Zurich

http://www.dbis.ethz.ch/

16th October 2007 – 15th April 2008

Supervisor:
Dr. **Peter Fischer**

Mentor:
Prof. Dr. **Donald Kossmann**

Databases and Information Systems Group

inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

This document discusses the problem of debugging functional languages in general, and XQuery-based languages in particular. The task of debugging an XQuery program is complex, because of the functional nature of the language and because of the rewrites and optimizations performed by execution engines that prevent the usual "stepping-through" style of inspection.

The goal of this work is to create the necessary infrastructure to write, run and debug XQuery-based programs using the Eclipse Platform and the open source XQuery execution engines, MXQuery and Zorba. The instrumentations needed in order to add debugging support to one of these engines is also described in this document.

This thesis also contains a detailed description of the engine debug protocol that we developed in order to be able to link any two debugger client and an execution engine that implement our specified debug interface.

Together with the engine debug interface implementation we also provide a description of a prototype debugger client user interface based on the Eclipse Platform that supports our execution engine debugger.

Contents

Chapter 1

Introduction

1.1 Motivation

XQuery [1] language holds a lot of promise not only as a query language for XML [2] data, but also as an application development language for XML-oriented applications, including most web service technologies. XQuery by itself is already a Turing-complete functional language with a strong type system. A small scripting extension called XQuery Scripting [4] allows imperative-style constructs, which makes working with the language easier.

One of the reasons functional languages are not extensively used nowadays is the lack of tools to support programming in such languages. XQuery and its new follower XQuery Scripting are partly falling in this category of languages. There are good reasons why such support tools are not available on a large scale. The main one is that providing such tools is not only an engineering task. Debugging a language that performs a lazy-evaluation execution – unlike the case of the imperative languages – is known to be a difficult problem. There are no agreed upon techniques in how to solve this. In this setting we have defined the semantics of what debugging an XQuery-based language means. Our implementation also tries to make a tradeoff between the lazy nature of the languages and the usability and predictability a user expects from a debugger.

Although a young language[1], XQuery programmers can already benefit from just a few available language debuggers. Our experience with some of them showed us that there are still things that can be improved in order to meet the expectance level of a common debugger user having an imperative language debug experience. On the other hand, we have no single example of a debugger that supports either the XQuery Update Facility [3] or the new XQuery Scripting extensions. All these motivate the implementation of our debugger engine and the debugger client.

[1] As W3C standard, the XQuery language went public in the beginning of 2007. As a Working Draft, though, the language had been pending since early 2001.

1.2 Objectives

Our main goal was to provide an Eclipse [8] based debugger client for the XQuery family of languages. The targeted execution engines were the following two, since these are the only ones we know having or adding support for the XQuery Update and XQuery Scripting language extensions:
Zorba [5] – a C++-based, embeddable and high-performance XQuery engine;
MXQuery [6] – a Java-based, low-footprint, extensible XQuery engine.

In order to enable debugging for the above execution engines additional functionality had to be added to them. This functionality, which will be described in the *Architecture* chapter, includes:

- the internal engine instrumentation;
- the TCP socket-based *Engine Debug Interface* (EDI);
- the Engine Debug Protocol (EDP).

The Engine Debug Protocol (EDP) and the API provided with it were developed independent of the execution engine or the Eclipse Platform in order to enable programmers to link any client to any XQuery execution engine, provided both parties implement our specified EDP. The protocol will be detailed in a dedicated section in *Chapter 4*.

In this work we also define the semantics of what breakpoints and stepping procedure mean in the context of debugging an XQuery-based language. The engine execution restrictions, like optimizations, code rewritings etc. are discussed from a debugging perspective. All these will be addressed both in *Chapter 3* from a conceptual solution point of view, but also in chapter *Chapter 4* from an architectural point of view.

This thesis wraps up the discussion by listing the limitations of our work and suggesting possible enhancements and new paradigms that could be developed in order to provide a better user experience in debugging functional languages like XQuery.

1.3 Outline

This document has the following structure. In *Chapter 2* we introduce the main concepts and build a theoretical foundation of the problem. We also present the problem statement, state of the art in this field and related work. Chapter 3 will give a conceptual solution to our problem. *Chapter 4* details some aspects of our implementation like: an architectural overview of the debugger client, the engine debug protocol and the execution engine instrumentation. *Chapter 5* wraps up this work by presenting what are the limitations of the current implementation and what are the next steps that have to be performed in order to address some of these limitations or to enhance the usability of the existing implementation.

Chapter 2

Fundamentals

2.1 XQuery and extensions

In our work we will not only discuss about the XQuery [1] language but also about other extensions that emerged from it: XQuery Update Facility [3] and XQuery Scripting Extensions [4]. Before starting to define our problem statement we find it useful to provide a short description of these languages.

2.1.1 XQuery

XQuery [1] is a language used for querying XML [2] data sources. The language is very close to functional languages, but also has some high-level SQL-like constructs and some imperative behavior. The language syntax embeds both XML and XPath [8] syntax and adds to them the XQuery language constructs. It also defines its own data model: XQuery 1.0 and XPath 2.0 Data Model (XDM) [9]. This data model is shared between XSLT 2.0 [10], XPath 2.0 [11], and XQuery.

A possible way to describe XQuery as a language is by saying that it is a declarative, fully composable, high level, strongly typed, read only language. Every query is formed out of a single expression that can be nested with full generality. Also there is no way in XQuery 1.0 to modify the existing data handled in the query. Hence, regarding the data, XQuery 1.0 is a side effect free language.

A prominent XQuery expression is the FLOWR language construct. This has a similar syntax to the `SELECT-FROM-WHERE-ORDER` SQL expression. The name “FLWOR” is an acronym built from the list of clauses this expression can contain in XQuery: `for`, `let`, `where`, `order by`, and `return` clauses. An example of such expression is provided in the following code listing:

```xquery
for $b in doc("books.xml")//book
let $a := $b/author[1]
where data($a) eq "Stephen King"
order by data($b/title) ascending
return $b
```

The above FLWOR expression retrieves all the `book` XML elements from the `books.xml` file and performs one iteration for each of these elements, binding in each iteration one of the elements to the `$b` variable. The difference to the for-loop construct the we know from imperative languages is that the order in which the bindings of the variable `$b` to each of the elements in the sequence is execution engine implementation specific. The bindings are also non-mutable. Therefore, there is no data dependency between different iterations and engines can also evaluate then in parallel. During each one of the iterations the rest of the clauses are executed in a sequential manner. The `let` clause binds to the variable `$a` the first `author` element in document order inside the `book` element bound to the variable `$b`. The `where` clause performs a filtering operation. Only the iterations in which the expression inside this clause evaluates to a true Boolean result will return a value by evaluating the expression in the `return` cause. There is an optional `order by` clause that allows the results to be returned in an ordered manner. The above query will return a list of `book`'s that have Stephen King as the `author`, ordered ascending by the `title` of the `book`.

The previous example shows also other constructs available in XQuery: path expressions and function calls. The language defines also conditional branching expressions like the `if` expression. XML constructs are directly supported by the language and they can be arbitrarily nested with XQuery code. The fragment below displays this powerful feature of XQuery:

```
<elem> element content {
    (: embedded XQuery code :)
    for $e in (1, 2, 3)
    return <a attr="{ $e }"/>
}</elem>
```

All the language constructs together with large operator set and built-in function collection make XQuery a very strong XML processing language.

2.1.2 XQuery Update

But the read-only, almost side effect free nature of XQuery led to the appearance of extensions. The Update extensions [3], already a standard W3C recommendation, provide the additional language expressions to perform modifications of existing XML data. Even XQuery, because it is a Turing-complete language, could achieve the same kind of results, but with much more effort, through an approach where modification is seen as rewriting everything from scratch. But constructing an XML document in this manner only looks like the modified original XML document. In fact we have a new document. The internal XML node IDs show this. Therefore the Update extensions are necessary. XQuery Update Facility is a superset of the XQuery language.

The XDM was modified to allow the following operations on XML nodes: insert nodes, delete nodes, replace a node with other nodes, and rename nodes. These operations are implemented by the new simple updating expressions:

- `insert` expression;

- `delete` expression;
- `replace` expression;
- `rename` expression

and the transform-`copy` expression which provides an easy way to copy nodes of an XDM instance.

2.1.3 XQuery Scripting

The way modifications are applied is a limitation for XQuery Update extensions. All updates performed are gathered in a Pending Update List (PUL) and they are applied only after the evaluation of the expression. Since an XQuery allows only one single root expression per query, the updates will be applied at the end of the execution. Therefore the update extensions do not add any notion of state that can be used, for example, for multiple transformations of the same node in an XDM instance.

This feature of state, together with sequential execution constrains have been added through the XQuery Scripting extensions [4], which recently became a working draft at W3C. The main differences to the XQuery Update Facility, which is a subset of the scripting extensions, are:

- for certain expressions an evaluation ordering has been imposed. For example:
    - function calls: the argument expressions are evaluated before the function body
    - comma concatenation operator: left to right evaluation of the operands;
- the expressions can have side-effects and these side-effects are visible to subsequent expressions;
- several new expressions have been introduced:
    - Assignment expressions;
    - Blocks;
    - Exit expressions;
    - While expressions;
    - Continue expressions;
    - Break expressions.

The Scripting extensions are providing a good candidate language that addresses the *impedance mismatch*. This term was coined for the set of all problems raised by the change of the data model and the programming language when multiple tiers of an application are developed. For example, most of the 3-tier applications developed for the Web use one relational database for the data storage with the corresponding SQL language, perform the application logic in a high-level, probably object-oriented language (Java, C#, etc) and display information to the user through another language (JSP, ASP, etc) that produces the markup language consumed by the browsers. In many of the cases XML is used as the intermediate data representation. In such scenarios, XQuery Scripting seems to be the best match and does not only eliminate the impedance mismatch, but also reduces the complexity and allows for global optimizations that span multiple tiers of the systems.

2.2 Problem statement

Due to the wide spread use of imperative languages, debugging any language is often associated by users with the debugging of an imperative language. For this reason, even in functional language environment, there might be a high expectance from the programmer used to debugging imperative languages, to use debug primitives common to a strict, imperative debugger. The imperative debug model includes primitives like breakpoints, code stepping, and code inspections and expression evaluation during a suspended execution. But, as many researchers pointed in their works [12, 13, 14, 15], these debug primitives are simply not suitable in the context of functional languages, especially if they are also implementing lazy evaluation, like many XQuery execution engine.[2] Thus, a vast amount of effort has been spent by researches to find the right primitives for such languages.

Due to the large debate on the topic, until today we haven't seen the emergence of a dominant approach. Therefore most, if not all, of the proposed techniques are used only on a small scale. Maybe the divergences between all these approaches were the cause the weak adoption of the functional language.

But, also the users themselves, and not the language tool developers, might be the ones causing the low acceptance of the functional languages. The users don't know that their expectances are barely suitable for this kind of languages. If this is true, one approach would be to reeducate the users on how functional languages must be debugged. This approach does not only require a lot of time, but there is factor of a much greater importance that makes this approach a very hard task. Programmers tend to think in a sequential, imperative manner while in functional languages parallelized execution is a common technique. Therefore, with such an approach the users are not only forced to learn something that they are not familiar with, but also something that doesn't suit their way of thinking.

While many tried this approach by developing new debugger prototypes with new debug primitives, we don't know many works that try to achieve the same goal the other way round. Having the user expectance to debug a functional language like an imperative one, one should try to map the debug primitives that the user is aware of to the new ones for the functional language. Of course, this mapping is not a straightforward task due to the many intricacies that lie under a functional language in general, and XQuery in particular. Also, this mapping is not always possible. In such cases, a tradeoff between the debugger user experience and the code execution has to be made. An overview of our approach to debugging XQuery will be detailed in the next section of this chapter while the full conceptual solution will be detailed in the next chapter. We believe that due to these tradeoffs we make, we impose some limitations on our debugger. Therefore, in the last section of this chapter we present other techniques and approaches proposed or implemented for functional languages debuggers and compare them with ours.

[2] The XQuery semantics was designed to make lazy evaluation possible. Therefore, many XQuery execution engines implement this.

There are a few existing tools on the market providing support for debugging XQuery. In our opinion they emerged as quick response to the appearance of the language, but are not well supported by a theoretical foundation and therefore they are confusing sometimes the debugger user. For this reason, we try to first fundament our approach and provide an exact semantics of our view on debugging XQuery. Also, we provide a debugger client that meets our specifications of an XQuery debugger.

We believe that XQuery is not just another programming language, but the start of a new direction in the Web development world. This discussion is beyond the topic of this thesis, but it is worth mentioning that because the usage rate of XML technologies has significantly increased in the last years, a similar trend can also be seen for the XML processing languages in general and XQuery in particular. This explains, maybe, the new wave of standards and proposals coming from W3C [16]: XQuery Update Facility, XQuery Full-Text, XQuery Scripting Extensions. Our approach takes into consideration also these language extensions. This, we believe, makes our work unique.

2.3 Requirements

In this section we will first present what particular problems can one encounter while debugging an XQuery-based language. Our solution to these problems will be given in *Chapter 3* where a high level conceptual solution of our approach will be presented. Further details about the implementation of this solution will be given in *Chapter 4*.

The XQuery language is categorized as a functional language, but there is one non-obvious case of side effects which makes the language a non-pure functional one: XML node constructors. The following code snippet illustrates an example of such side effects:

```
declare function local:new-element($data as xs:string) as item() {
    <element>{ $data }</element>
};

let $x := local:new-element("abc")
let $y := local:new-element("abc")
return $x is $y
```

In the above example we have the function `local:new-element` that accepts as input a parameter `$data` of type `xs:string`. The function returns an XML element having the data passed through the function parameter as element content. We than call this function two times with the same input data and compare the two results. Trying to guess what the result of the query above is, one could be misled to say that the equality between the two results yields a true answer. According to the definition of pure functionality, a pure function must always return the same result given the same argument values. But in XQuery this is not true. Similar to object IDs in an object-oriented programming language, XML nodes are also assigned IDs. This, together with the `is` operator that checks for ID equality, make the above query return a false result. Furthermore, the semantics of several other operators and expressions, like `union` and `intersect`, for example, is based on the node ID comparison. Therefore the node IDs, which are hidden from the programmer, influence the result of expressions. This breaks the pure-functionality of XQuery.

The Update Facility and Scripting extensions add more elements to XQuery that drift it even further away form the class of functional language. The Update Facility adds the possibility to modify existing data used in the query. These side effects are gathered in a pending update list until the end of execution and only then made visible. XQuery Scripting adds the general notion of side effects, preservation of state and execution order. One can even see XQuery Scripting as an imperative language with functional constructs, but the purpose of this extension is, of course, not to transform XQuery in an imperative language, but rather to give the programmer some of the strengths of an imperative style of programming.

As we can see, the context of our work is not only a non-pure functional language, but, in the case of extensions, a language that also has imperative constructs. This gives a good reason for our work to implement the debugger by finding the mapping between the imperative debug primitives and, where necessary, the functional language constructs.

In the remaining of this section, we present a few of the problems that could make an imperative style of debugging harder in the context of an XQuery-based language. Most of these problems are generated by aspects inherited from the functional languages.

2.3.1 Lazy evaluation

By the way the function arguments are passed to a function when an expression is being evaluated, one can split functional languages in languages with either strict or non-strict evaluation. Lazy evaluation is one type of non-strict evaluation. This technique is employed by many XQuery engines in order to improve the performance of code execution. With lazy evaluation, expressions are not evaluated until they are actually needed. This has a very interesting consequence in some cases. For example, if a certain expression is used in the declaration of a variable that is never used after its declaration point, the code inside this expression will never be executed. But what if the unexecuted code contains a run-time error? According to the lazy evaluation, this error will now appear when executing the code. The following code listing illustrates this in an example:

```
declare function local:div-by-zero() as item() {
    5 div 0
};

let $x := local:div-by-zero()
let $y := <a/>
return $y
```

Since the variable `$x` is not used in the result returned by the query, the expression bound to this variable is not evaluated. Therefore, the function `local:div-by-zero()` will not be called and the division by zero error will not appear when this query is run.

In an imperative style of debugging, the user might have the choice to break the execution after the first binding, the one for the variable `$x`, and to inspect the value bound to this variable. This will not only break the lazy evaluation strategy of the engine but also cause an error that in the normal execution environment, without a debugger, wouldn't have been raised.

2.3.2 Unordered execution

Unordered execution is another issue of interest when trying to debug XQuery code. The language specification itself leaves the execution order for many of its expressions open to a specific implementation. This is, for example, the case of function parameters that don't have a specified evaluation order, or the individual expression used in a sequence expression with the comma concatenate operator. Having no ordering restrictions for such expressions, and letting the control solely to the execution engine can give a very hard time to debug XQuery. During debugging, the programmer is concerned with the logic of his query and not the inner workings of a particular XQuery engine. Moreover, using different engines would also make a debugger behave different for each engine. From our experience this brings a lot of confusion to the debugger user.

XQuery adds even more to this pain through the `unordered` expression type. For such an expression the order of the returned results is a free choice of the execution engine. This is, of course, reflected on the execution order of the expressions contained in the `unordered` expression. Therefore, debugging such expressions might be beyond the understanding of the normal debugger user.

2.3.3 Normalization

XQuery normalization is one of the steps described in the XQuery processing model. Its purpose is to reduce the complexity of the XQuery language by mapping each expression to an equivalent one in a non redundant, minimal language grammar called XQuery Core [23]. This normalization step raises other problems for a potential language debugger implementation. A language debugger relies mainly on the textual representation of the user code. Normalization, performed through code rewriting, can alter the textual representation of certain expressions and thus cause a mismatch between the code the user provides and the runtime information a debugger handles. This problem has to be taken into consideration when providing a debugger for the XQuery language.

2.3.4 Optimizations

Many aspects of the XQuery language were especially designed to allow optimizations. Therefore, after the normalization process, many engines use optimization to achieve a better performance of the execution. Debugging in an environment that performs optimizations is a very challenging task and sometimes impossible. This is partly because optimization deals with code rewritings, which can cause expressions in the original code to be transformed, merged or eliminated.

Optimization also means that an engine can choose a parallel or an unordered execution of certain expressions if such an execution is a better candidate in terms of performance.

From the point of view of the user debugging XQuery code, the optimizations can create a lot of problems since he might not know anything about how these are performed and what to expect from an optimized code.

2.3.5 Debugger client

In spite of all the problems mentioned above that are raised by the XQuery language, together with the mismatch between the functional and the imperative debugging model, we want to provide users an imperative-like debugger client that uses most of the primitives such debuggers use. They include breakpoints, code stepping and code inspection.

2.3.6 The scope of this work

In the following chapter we will address most of the problem above. We will impose some conditions on lazy evaluation and unordered execution when the execution will be performed in the debug mode. We will also try to enable users to debug queries independent of the normalization process. Regarding the debugger client, we will try to match the imperative debug model with an XQuery-based language.

Regarding XQuery code optimizations, due to time constraints and the complexity of the problem, we did not address this topic. As a side note, we believe that a debugging environment is not a profiling one and therefore performance is not a critical issue for the debugger user. We are, though, aware of the fact that there are cases when optimized code yields other results than the non-optimized one. In our solution, we have dealt with code optimizations by only not allowing them during the execution in debug mode. Even the imperative language debuggers turn off or limit the scope of optimizations during debugging since some code transformations are not reversible and therefore impossible to debug while still providing meaningful debug information to the user.

2.4 Related work. State of the art

2.4.1 Debugging lazy-evaluation functional languages

Debugging lazy-evaluation functional languages has been a much debated topic in the last decades but without a consensus reached in this direction. The main cause of this is the big discrepancy between such languages and the imperative style of debugging. As a consequence, many alternative debugging models have been proposed or developed for lazy-evaluation functional languages. Tracing [17, 19, 20], cost centre stacks [18], or time travel [29] are just a few of them. Some surveys on these models have already been written in [14, 15]. We believe that all these approaches have diluted the effort to provide programmers the tool they would use.

On the other hand, some argue [21] that lazy evaluation is not as efficient as it might seem at the first sight. To quote the Ennals et al. in [21], the extra memory traffic caused by the call-by-need argument-passing mechanism, makes lazy programs perform noticeably worse than their strict counterparts, which implement a pass-by-value argument-passing mechanism. According to the authors, both time and space performances suffer. Therefore they introduce the notion of optimistic evaluation which decides at run-time what should be eagerly evaluated and provides an abortion mechanism that stops eager computation when this runs for too much time, preventing in this way possible infinite loops. Except the abortion mechanism, the idea of optimistic evaluation is similar to our notion of *eager-on-demand evaluation* which will be described later in this document.

The published work that that we found to be the closest to our debugger implementation is HsDebug [14]. The authors used the approach of not being lazy to debug Haskell [22], a lazy-evaluation functional language. HsDebug also provides a “stop, examine, continue” debugger interface to Haskell, which is their way to describe what we call an imperative-style debugger. Our goal is to implement a similar debugger interface for XQuery-based languages, task which raises also some problems that are specific to our targeted languages. Further more, we want to provide a complete set of tools for XQuery development and debugging, not just a conceptual solution or a proof of concept.

In the XQuery world, while there are a bunch of execution engines, just a few of them also provide debugging support. One of them is Saxon [28], a proprietary XQuery processor. This debugger is widely spread in the existing debugger clients, one of them being the one provided by Stylus Studio®, which will be described in the next section.

Another interesting approach for debugging XQuery is the Rover [19] implementation. This instruments the code at runtime with the `fn:trace()` function implemented to populate a relational database with XQuery item sequences. The main assumption made by this implementation is that the function `fn:trace()` is backed by a database to store the runtime data and thus allowing the user to perform post-mortem forward and backward interactive debugging. This is an assumption that we don’t want to make. Also the debugger is not a breakpoint oriented one, but an expression oriented one. Their debugger client provides new debug primitives like XML viewers for the trace tables associated to different expressions. But, adding such new debug paradigms is also something we want to avoid as much as possible.

2.4.2 XQuery debugger clients

A significant roadblock in the adoption of XQuery is the lack of good development tools. Most existing tools are either small extensions of text editors or general-purpose XML editors, neither of which providing the necessary support to efficiently develop programs in XQuery. One reason might be because they are XML-driven and not XQuery-driven, XQuery not the focus of these tools. Our experience showed us that, being XQuery-driven, has indeed its pitfalls. For example, the XQuery extensions that we are also targeting, like XQuery Scripting Extensions, are not standardized and therefore not stable. Adding support for these extensions requires indeed a very big commitment due to the changes both in the syntax and the semantics that can appear in the targeted language. We believe that since these changes are not major ones, the support tool development can be performed along with the evolution of the working drafts of the standards. This approach has a major advantage. Having ready-to-use language support tools by the time a standard is issued maximized the adoption rate and minimizes the adoption delay. But, besides not being dedicated XQuery-dedicated support tools, the existing tools also don't have our approach. As a consequence, 1-year old standards, like XQuery Update Facility, have no support in any of the up-to-date known tools.

Nowadays, the most used implemented XQuery editors and debugger clients are the ones provided together with large XML development environments. This makes it harder be accepted by the programmers that want to use only the XQuery support features. This happens for two reasons: usability – they only want XQuery support tools – and price – they don't want to pay for tools they don't use. This is the case of Altova's XMLSpy®, DataDirect's Stylus Studio® or SyncRO Soft's <oXygen> XML Editor®. Therefore a free, lightweight version of XQuery support tools is needed. We are trying to provide such a package.

But, even if all these existing environments' support is limited only to XQuery 1.0, we have also found several problems that developers are common faced with.

For all these XQuery Editors we have use cases in which syntax highlighting has major problems. We consider major, a syntax highlighting error that persists even when the query has a correct syntax. There are two main reasons these problems appear. Firstly, XQuery is not a standard keyword based language by the fact that keywords are not language reserved words. Secondly, a correct syntax highlighting can only be performed when the query is complete, having a correct syntax. The existing editors rely only on static syntax highlighting rules that do not check if query tokens in the editor correspond exactly to the ones in the query's Abstract Syntax Tree. This as-you-type syntax highlighting is indeed very useful for the programmers, but this eager approach leads many times to errors. Our implementation has such static rules, but also implements a syntax highlighting reconciliation strategy which corrects possible errors of the static-only approach. We have also found that full language composability and deep nesting are also causing problems for two of the previous mentioned editors.

Code completion is an important feature that language editors must provide. In this category fall both static code completion like keyword completion, built-in function completion and code templates, but also dynamic code completion like, variable and user defined function completion. Out of all there code completion features, only a small set of them are provided by each of the above mentioned editors. With our implementation we try to provide all these features.

Due to the much debated topic on debugging lazy-evaluation functional languages like XQuery, and the lack of a formal stepping semantics defined for such a language, the notion of stepping is a fuzzy notion for the programmer because of the different approaches each editor takes. Moreover, a lazy debugging behavior, like the one used by the Stylus Studio environment and provided by the Saxon execution engine, is very confusing for the programmer. We try to address these problems by implementing an imperative-like style of debugging, for reasons explained in the previous sections of this chapter.

Chapter 3

Conceptual Solution

This chapter will present the solutions that we found to the problems mentioned in the previous chapter and will provide a conceptual view on our debugging model.

3.1 Stoppable expressions

One reason why debugging an XQuery-based language is not a straight forward task, is the full generality allowed for nesting expressions. Consider a programmer with an imperative language debug experience, where primitives like “step in”, “step over” and “step return” actions are available. Keeping the same stepping semantics for these actions in the context of XQuery debugging would have, in many cases, an unexpected behavior. A very often case might be the termination of the query when a step over action is performed. This happens because in simple XQuery a query is always formed out of a single expression[3]. There are also other examples. One of them is presented given the following XQuery query:

```
let $x := <result> {
    for $i in 1 to 10
    where $i mod 2 eq 0
    return
        <elem>{ $i }</ elem>
} </result>
return $x
```

The previous code example contains a variable binding, `$x` – similar to the variable assignment in Java for example – and returns, in the last line, the bound variable as the result of the query. The imperative languages taught us that the “step in” action can be performed only on function calls. Keeping this debugger rule will give an XQuery programmer a hard time debugging his code. In the previous example assume that the debugger stopped the execution at the `let` clause before the variable binding. Since the expression that is bound to the variable `x` does not contain any function call, and sticking

[3] XQuery Scripting extensions add the concept of sequence of expressions, and, in this case, the stepping behavior would be more close the semantics in the imperative languages.

to the previous rule, one can not step in this expression. But it is quite obvious that this is not the behavior one would expect form an XQuery debugger.

In order to allow a more intuitive debugging experience, we have mapped the imperative style stepping primitives to the expression tree traversal operations. Allowing users to navigate no only through function calls, feature already provided by the imperative language debuggers, but also through the nodes of the expression tree. This makes the task of programmers easier when debugging a functional language like XQuery.

The above mentioned mapping was realized with the help of the *stoppable* paradigm that we introduced in the context of XQuery debugging. A *stoppable expression* is any language expression the user can set a breakpoint on and where the engine can suspend the execution – either due to a breakpoint, a user request to suspend the execution or through code-stepping actions. At the break locations, the user can perform debugging actions, like code stepping or expression inspection. The rules how stoppable expressions are chosen is based on the language grammar and the semantics of individual expressions. These rules of choosing stoppable expressions along with the exact breakpoint and stepping semantics for each of these expressions will be given in the next section. In this section we will only give the general rules of how debugging actions are mapped to XQuery debugging through stoppable expressions.

The expression tree is a conceptual XQuery representation used by execution engines for processing. This is obtained from the query abstract syntax tree (AST) the language parser generates. Even if conceptually all engines rely on the expression tree, only some implement it as a physical structure. The expression tree is the basis for the engine's XQuery processing model. Operations like normalization and optimization are performed on this tree.

We will focus in our discussion on the conceptual idea of expression tree obtained through a 1-to-1 mapping from the AST. For our model, we consider that for each language expression defined in the targeted XQuery specification or extension specifications there is one corresponding node in the expression tree. We will not make any other assumption about the representation of this tree other than the annotations we use for the purpose of the debugger.

In order to have a better understanding of the stoppable expressions and the expression tree annotations, we will first provide a short example. Consider the following XQuery fragment:

```
let $x :=
    <result>{
        (1 + local:foo()) + local:goo()
    }</result>
return $x
```

According to the standard XQuery grammar, this query generates the AST in *Fig. 1*. For simplicity we have pruned out some leaf nodes that did not present any interest in our discussion. This is the case for the `QName` nodes or the language token alias nodes.

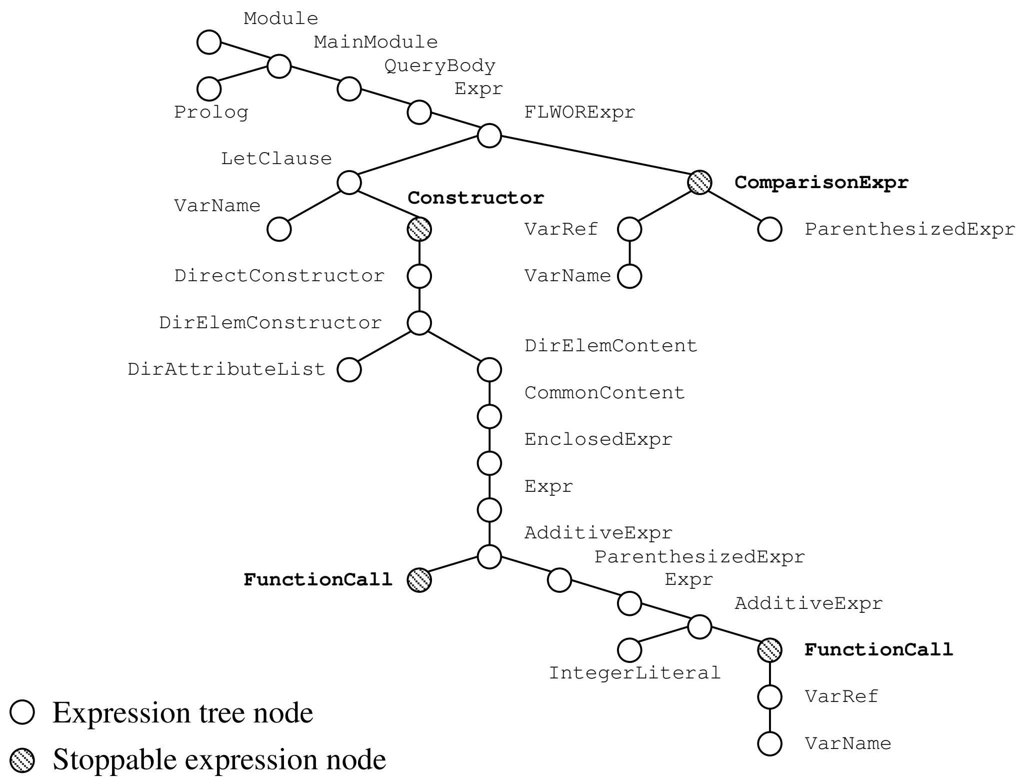


Figure 1 – The query expression tree with outlined stoppable expressions

An engine would use the above tree to obtain a similar expression tree. For this reason, we will consider this one as the expression tree. Given the simplicity of the given query, one can easily remark the verbosity of the generated tree – this version in the diagram being as well a simplification of the complete tree. Defining the debug primitives based on such a structure would certainly generate difficulties for a programmer debugging the previous simple query. Imagine what the debugging activity would become if the user would need to know the exact structure of this tree and the exact composition of each expression. This, we believe, does not only create complexity in debugging, but also drifts the programmer away from the task at hand, which is, to follow and debug the query logic. In order to simplify the user view on the query during the debugging, we annotate only some of the expression tree nodes as stoppable expressions. This stoppable annotation is assigned in such a way that the code stepping, which means navigating between the annotated nodes, provides a meaningful debugging semantics to the user. Assume that we want to provide the user the following behavior when debugging the above query:

- The user can break the execution at every expression that forms a variable binding in the FLWOR expressions is evaluated, as well as at the expressions in the `where` and return clauses. Furthermore the user sees all those expressions on the same level from a stepping behavior point of view. This means that using only the "step over" action is enough to navigate through all these expressions. Therefore the root node of every expression mentioned are considered stoppable expressions;

- The execution can be broken before a function call. Therefore, a `FunctionCall` is marked as a stoppable expression.

These rules determine the stoppable expressions in our expression tree. These are outlined in the diagram in *Fig. 1*. The complete set of rules for selecting stoppable expressions will be given in the *Section 3.3*.

3.2 Default stepping semantics

A stoppable annotation can be either marked by setting a flag in the corresponding node or by inserting an additional node in the expression tree between the node that must be annotated and its parent. We favor this latter approach, which will be used in this conceptual solution because it avoids changing the tree node structure. Only a single new node type must be introduced.[4] The new node type is called a *debug node*. The expression tree with inserted debug nodes or with annotated stoppable nodes will be named the *debug expression tree*. *Fig. 2* displays the debug expression tree generated from the expression tree in *Fig. 1* by inserting the debug nodes.

The semantics of breaking and stepping will be defined based on tree traversal operations on the debug expression tree. In the following sections we will call *source node* the debug node where the execution was last broken, and *target node* the debug node chosen by the engine as the next point to break the execution. In case the suspend debug action is performed the next debug node encountered during the execution is considered the target node.

The semantics of stepping is equivalent to the selection of the target node, given a step action and a source node. After the target node has been chosen, the execution continues until this node is encountered, provided no other breakpoint is set in-between. Each of the stepping actions will be performed by the debug node associated with a certain stoppable expression. Every debug node will compute the target node for a particular step action with a step function having the following prototype:

DebugNode *step_action* (DebugNode *sourceNode*)

where the *action* stands for one of the three possible step types: *in*, *over* and *return*. The step functions have one parameter, the source node, and return the target node chosen according to the step action type and the stoppable node the debug node is associated with. The data types used in the pseudo-code definitions are the following:

[4] From an implementation point of view, the direct tree node annotation scheme benefits of a better performance than the new inserted node approach. Our implementation, described in the *Architecture* chapter, implements the node annotation scheme.

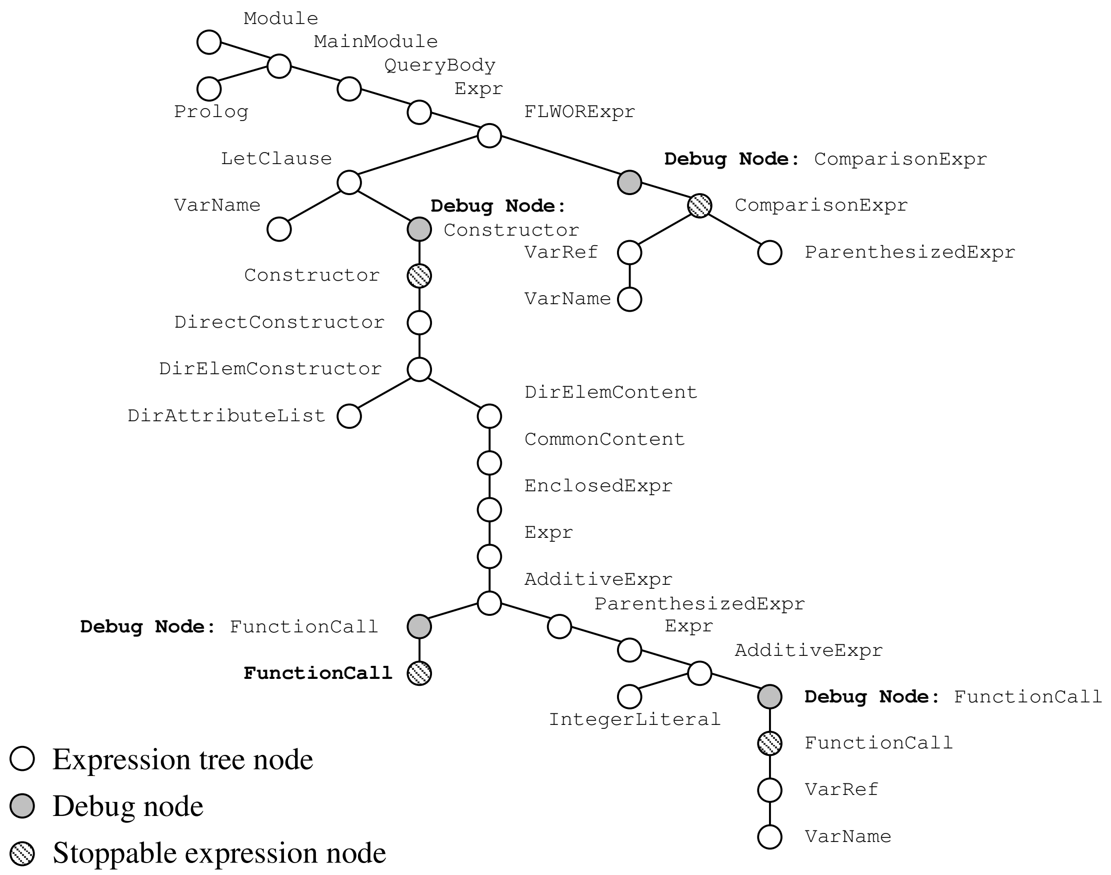


Figure 2 – The debug expression tree

- Node – represents any node in the tree;
- DebugNode – represents only the debug nodes in the tree. This type is compatible with the Node type. A DebugNode is a Node.

The following step function descriptions provide the default implementation for all the debug nodes. Some debug nodes, though, have to override this default implementation with one to meet the semantics of the expression this node belongs to. This is the case for all the expressions that control, in a way or another, the execution flow. Such expressions are the FLWOR, `if`, `typeswitch` and quantified expressions in simple XQuery and `while`, `break`, `continue`, and `exit` expressions in XQuery Scripting. The control flow alters the tree navigation behavior implemented in the default step functions by performing jumps between the nodes of the debug expression tree. The overridden step functions are meant to implement jump behavior dictated by different expression semantics.

3.2.1 Breakpoints and execution breaking

The execution will be broken only when a debug node is encountered if the debug node has an associated breakpoint or if the node is the target node. This rule assures that the execution can only be broken above stoppable expression nodes, meaning, before a stoppable expression is evaluated.

3.2.2 Step in

The “step in” debug action chooses as target node the first debug node found through a preorder, left-first tree search, starting from the source node. If no debug node is found, the “step over” action is performed. The *step_in* function performs the following logic:

```
DebugNode step_in (DebugNode sourceNode) {
    DebugNode result = preOrderLeftSearch (child (sourceNode))
    if (result is null) {
        result = step_over(sourceNode)
    }
    return result
}
```

where the *preOrderLeftSearch* function has the following prototype:

```
DebugNode preOrderLeftSearch (Node node)
```

and implements a preorder left-first tree search algorithm that starts from the node given as parameter and returns the first debug node found. If no such node exists, a null result is returned. The *child* function returns the child of a debug node. Debug nodes will always have one single child which is not a debug node.

3.2.3 Step over

The “step over” debug action target node assignment is implemented by the *step_over* function:

```
DebugNode step_over (DebugNode sourceNode) {
    Node nodeAtLeft = sourceNode
    Node inParent = parent (sourceNode)
    while (inParent not instance of DebugNode and inParent is not null) {
        Node result = findTargetAtRight (inParent, fromNode)
        if (result is not null)
            return result

        fromNode = inParent
        inParent = parent (sourceNode)
    }
    return null
}
```

where the *parent* function returns the parent of a node and the *findTargetAtRight* function finds the first debug node in the list of children of a node after a certain child. This function is defined as:

```
DebugNode findTargetAtRight (Node node, Node fromChild) {
        foreach (Node sibling in getRightSiblings (fromCihld)) {
                DebugNode result = preOrderLeftSearch (sibling)
                if (result is not null)
                        return result
        }
        return null
}
```

The *getRightSiblings* function returns the list of children of *node* at the left of the given child *fromChild.*

3.2.4 Step return

The "step return" debug action chooses as target node the node found through a "step over" action starting from the innermost debug node ancestor of the source node. The *step_return* function has the following definition:

```
DebugNode step_return (DebugNode sourceNode) {
        sourceNode = innermostDebugNodeAncestor (sourceNode)
        if (sourceNode is not null) {
                return step_over (sourceNode)
        }
        return null
}
```

where the *innermostDebugNodeAncestor* function has the following prototype:

```
DebugNode innermostDebugNodeAncestor (Node node)
```

and finds the first debug node found on the path from the source node to the root of the tree. If no such parent exists, a null result is returned.

3.3 Stoppable expressions selection rules

In the previous and current sections, the semantics of "step out" and "step return" were considered in the absence of breakpoints set in the debug expression tree fragments evaluated during that step execution[5]. In the case breakpoints are met during the evaluation, the execution engine will stop accordingly.

In this section we provide the complete set of rules used to select the stoppable expressions together with a description of why this is necessary. As mentioned in the

[5] The "step in" semantics is not changed by the presence of breakpoints since the corresponding execution will always stop at the next stoppable expression met during the evaluation, therefore there is no other valid breakpoint location between the source and the target node.

previous section, the stepping semantics is given not only by the structure of the debug expression tree, but also by the semantics of different expressions. These expressions introduce a jump behavior in the debug tree navigation. For each of theses expressions we provide also the description of the jump behavior.

In XQuery 1.0, we have the following stoppable expressions selection rules:

- `FunctionDecl` – the user must be able to suspend the execution engine before the evaluation of the function body begins in order to inspect the values bound to the function parameters. One solution would be to make the expression that forms the body of the function stoppable. But because the external functions don't have a body, and we want to treat all function declarations in a uniform way, we mark all function declarations as stoppable.
  Semantics for:
  - breakpoint: the execution stops before the engine starts to evaluate the body of the function.
  - step in: Default.
  - step over: Overridden. The default *step_in* implementation with the change that, when no target node is found, the *step_return* function is called. This assures that if there is a stoppable expression in the body of the function, no matter how deep is this nested, the corresponding debug node will be the target.
  - step return: Overridden. The default *step_over* implementation having as source node the debug node associated with the `FunctionCall` that caused the jump to this `FunctionDecl`.
- `FLWORExpr` – the user must be able to step through all clauses of a `FLWORExpr` in a linear way – only using the "step over" action. This means that he should see all the bindings in the `for` and `let` clauses, and the `where` and the `return` expressions on the same level.[6] Therefore each `ExprSingle` child of the `ForClause`, `LetClause` and `WhereClause` as well as the one of the `FLWORExpr`, which corresponds to the return expression, will be marked as stoppable.
  Semantics for:
  - breakpoint: for bindings, the execution stops before the binding of the positional variable is made, if this is present, and before engine starts evaluating the expressions in the binding; for the rest of the expressions, the execution is stopped before the evaluation of the expression starts.
  - step in: Default.
  - step over: Overridden. The default *step_over* implementation except for the debug node associated with the return expression. This will select as target node the debug node associated with the expression in the next `for` binding to be made. If no more bindings have to be made, the default *step_over* implementation is performed.
  - step return: Default.
- `QuantifiedExpr` – this expression has a similar loop behavior as the `FLWORExpr`. The user must be able to stop the execution for each binding and at the `satisfies` expression. Therefore, all direct `ExprSingle` children will be marked as stoppable.

[6] This is the case because the return expression is one level above the expressions in the other clauses.

Semantics for:

- breakpoint: the execution stops before the engine starts evaluating the expression in the binding or the `satisfies` expression.
- step in: Default.
- step over: Overridden. The default implementation except for the last `ExprSingle` child that corresponds to the `satisfies` expression. This will select as the target node, the debug node associated to the expression of next binding to be made. If no more bindings have to be made, the default *step_over* implementation is performed.
- step return: Default.

- `TypeswitchExpr` – the user must be able to break the execution at the `typeswitch` expression and at the chosen `case` clause. This allows him to see which execution path is chosen by the engine. Therefore, the `ExprSingle` child of each `CaseClause` and the `Expr` and the `ExprSingle` children of the `TypeswitchExpr` must be marked as stoppable.
  Semantics for:
  - breakpoint: for all the expressions, the engine stops the execution before the evaluation of the expression starts; for the ones in the `case` or `default` clauses, the optional binding, if this is present, is performed before stopping the execution.
  - step in: Default.
  - step over: Overridden. The debug node associated with the `typeswitch` expression will choose as target node the debug node above the expression in the case clause chosen by the runtime. The debug nodes associated with the other expressions will have the default *step_over* implementation, but using as source node, the debug node associated to the default clause return expression.
  - step return: Default.
- `IfExpr` – the user must be able to stop the execution at the conditional expression and also to see which branch the execution follows. For this reason, all the three children of the `IfExpr` must be marked as stoppable.
  Semantics for:
  - breakpoint: the same semantics for all child expressions of the `IfExpr`. The Engine stops the execution before the evaluation of the expression starts.
  - step in: Default.
  - step over: Overridden. The debug node of the conditional expression chooses as target node the debug node associated with the expression in the branch to be followed.
  - step return: Default.
- `FunctionCall` – a user must be able to break the execution at a function call. This allows him to inspect the values of and step in the expressions used as arguments to the function. Also this break point allows the user to step in the declaration if the function if the function is a user defined one. For this reason, function calls are also marked as stoppable.
  Semantics for:
  - breakpoint: the engine stops the execution before starting to evaluate any of the expressions passed as arguments to the function, or, if no such expression is available, before the context is switched to the function declaration.

  - step in: Overriden. The default *step-in* implementation with the following modification. The debug expression of the `FunctionDecl` corresponding to this call is seen as the last child of the the `FunctionCall` node.
  - step over: Default.
  - step return: Default.

There might be a need for debugger users to debug the expressions used in the `OrderByClause`. We are currently not supporting this stoppable expression but we keep it as candidate for the next revision of the stoppable selection rules. Allowing a user to debug a FLWOR expression containing an `OrderByClause` might generate an unexpected behavior of the debugger due to either different engine implementations of the ordering predicate in a FLWOR expression or the need for materializing all the results before the first one can be generated. In this context, we introduced the following convention: when debugging a FLWOR expression that contains an `OrderByClause`, the engine will not stop and will not honor the breakpoints set for any of the bindings or the `WhereClause` in the same FLWOR expression. This rule does not apply to the sub-expressions used for bindings, in the where-expression or in the ordering predicate. We make no assumption about the order in which the engine evaluates these sub-expressions.

The rules for selecting the stoppable expressions in XQuery Update extensions are:

- for all the basic updating expressions (`InsertExpre`, `DeleteExpr`, `ReplaceExpr` and `RenameExpr`) – since all these expressions use `SourceExpr` and/or `TargetExpr` and we want to allow users to stop the execution at these expressions, all the basic updating expressions can be covered by the same rule.
  Semantics for:
  - breakpoint: the engine stops the execution before starting to evaluate any of the `SourceExpr` and `TargetExpr` child.
  - step in: Default.
  - step over: Default.
  - step return: Default.
- `TransformExpr` – the user must be able to break the execution for every binding in the `TransformExpr` and for both of the `modify` and `return` expressions. Therefore all the `ExprSingle` children are marked as stoppable.
  Semantics for:
  - breakpoint: for any of the bindings or the `modify` and `return` expressions, the engine stops the execution before starting to evaluate the `ExprSingle`.
  - step in: Default.
  - step over: Default.
  - step return: Default.

In XQuery Scripting, the following rules are applied for selecting the stoppable expressions:

- `AssignmentExpr` – the user must be able to step through variable assignment expressions. This allows him to perform code inspections and to step in the

expressions that return the values to be assigned. Therefore the `ExprSingle` child will be marked as stoppable.
Semantics for:
- breakpoint: the engine will stop the execution before it starts the evaluation of the expression to assign.
- step in: Default.
- step over: Default.
- step return: Default.

- `BlockVarDecl` – the user must be step through the sequence of declare variables that also have an assigned a value. The user can perform code inspections of step in the expression returning the value to be assigned. Therefore all the direct `ExprSingle` children will be marked as stoppable.
  Semantics for:
  - breakpoint: the engine will stop the execution before it starts to evaluate the expression to be assigned to the variable. The engine will only stop for a variable declaration if this is also assigned a value.
  - step in: Default.
  - step over: Default.
  - step return: Default.
- `BlockBody` – the user must be able step through the sequence of expressions that form the block body. This allows him to inspect partial result of the program logic and to step in the nested expressions. Therefore all the direct `Expr` children of the `BlockBody` will be marked as stoppable.
  Semantics for:
  - breakpoint: for each expression, the engine will stop the execution before starting to evaluate the expression.
  - step in: Default.
  - step over: Default.
  - step return: Default.
- `ExitExpr` – the user must be able stop the execution when encountering an `ExitExpr` expression. This allows him to perform code inspections and to step in the return value expression. Therefore the `ExprSingle` child of the `ExitExpr` will be marked as stoppable.
  Semantics for:
  - breakpoint: for each expression, the engine will stop the execution before starting to evaluate the expression.
  - step in: Default.
  - step over: Overridden. The same as *step_return*.
  - step return: Overridden. The default *step_return* using as source node, the inner most function call in the call stack. If no such function call exists, the execution will terminate.

- `ContinueExpr` and `BreakExpr`[7] – the user must be able stop the execution when one of those expressions is encountered in order to perform code inspections before the control flow is changed according to the semantics of the expression. Therefore these expressions will be marked as stoppable.
  Semantics for:
  - breakpoint: the engine stops the execution when the expression is encountered.
  - step in: Default.
  - step over: Overridden. According to the semantics of the specific expression.
  - step return: Overridden. According to the semantics of the specific expression.

3.4 Query location

An important link between any language's execution engine and the corresponding debugger client is the textual representation of the user written code. The debugger client must provide information about the breakpoint locations to the engine and the engine must provide the location where the execution is suspended. Also, breakpoint management, for operations like setting or clearing breakpoints, needs some sort of identifiers in order to control the individual breakpoints. By using then notion *query location*, which is defined below, we solve both problems.

A *query location* is a range in the query file with the following properties:
- *startLine* – the number of the line in which the range starts;
- *startColumn* – the number of column in which the range starts;
- *endLine* – the number of the line in which the range ends;
- *endColumn* – the number of the column in which the range starts;
- *fileName* – the name of the file that contains the query

All the numbers are 1-based and the following relation holds:

$$startLine <= endLine \text{ OR } ((startLine = endLine) \rightarrow (startColumn < endColumn))$$

This relation enforces that ranges are not empty and the start of the range appears in the left-right, top-bottom document order before the end of the range.

By adding one more rule, we can use the query locations to uniquely identify the query expressions. The rule is the following: the query location of a certain expression starts with the first non-white character and ends with the last non-white character belonging to the expression.

The above specification of query location has also the property of being platform independent. To be more precise, it avoids the well known line terminator representation problem.

[7] The semantics for `ContinueExpr` and `BreakExpr` are very unstable compared to the other XQuery Scripting expressions. Because of the debate on this topic we will not make any assumptions about what is the jumping behavior of code stepping for these expressions.

3.5 Debug expression tree view

In this section we present a special property of the debug expression tree and based on it we give the solution to one of the problems identified in the previous chapter: the code rewriting problem caused by query normalization.

To illustrate the idea of this section we will use the following XQuery examples:

```
for $book in doc("books.xml")/books/book
let $authors := $book/author
where $book/title = "Ulysses"
return count($authors)
```

The query above returns as result, the number of authors of the book named “Ulysses”. The book is searched for in the list of books found in the `books.xml` XML file. This XQuery fragment generates the expression tree shown in the diagram in *Fig. 3*. The stoppable expressions were selected according to the rules presented in the previous section. For the topic of this section we will consider the annotated version of debug expression tree, therefore the debug nodes will not appear. This way we manage to simplify the explanations while keeping the same properties of the debug expression tree.

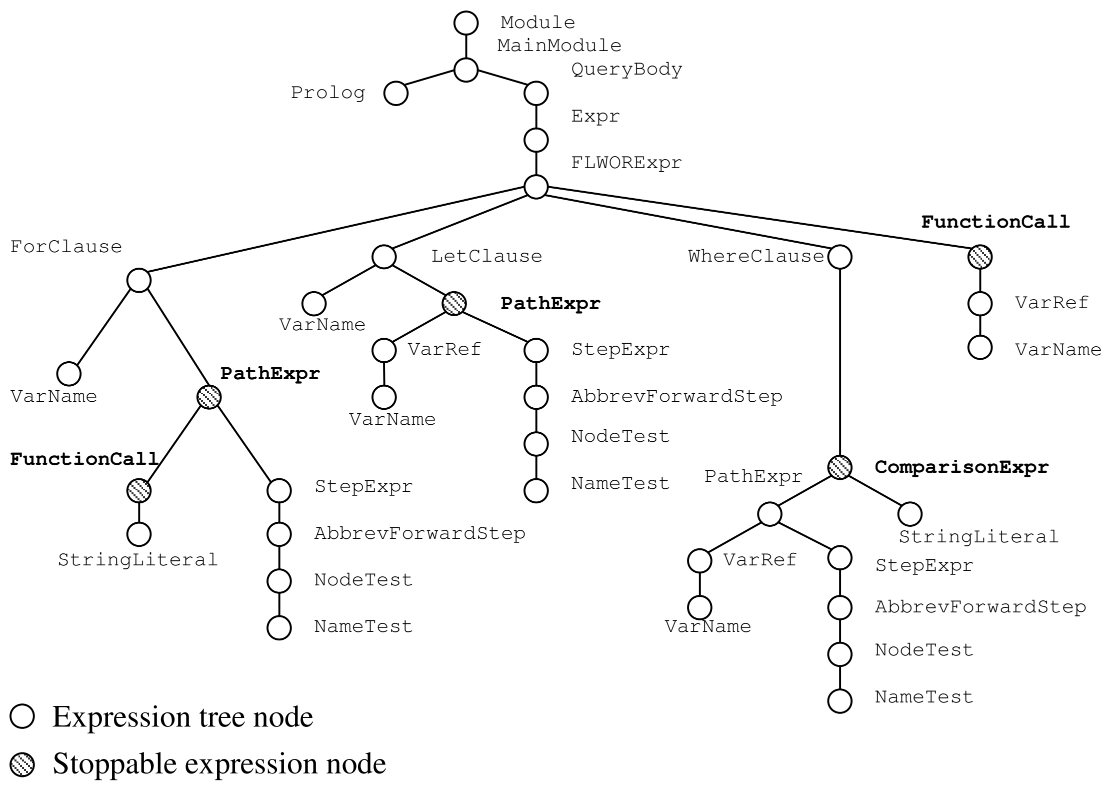


Figure 3 – The expression tree with outlined stoppable expressions

The stepping semantics defined in the previous sections, can also be defined in terms of the *debug expression tree view*, a tree structure that contains only the stoppable expressions in the tree. A recursive function can easily generate this view starting from the root of the tree, using only the following rule: if the root node is a stoppable node, this will be kept, otherwise, it will be replaced with the children nodes after this function has been applied on each of its direct sub-trees. It is easy to remark that all stoppable expressions that have no other stoppable ancestor will become roots of distinct trees. In this case we consider all these root nodes to have the same parent node which is not stoppable. This happens in our example. The expressions of the FLWOR expression clauses, after the filtering operation, remain with no parent. The debug tree view will look as in the diagram shown in *Fig. 4*.

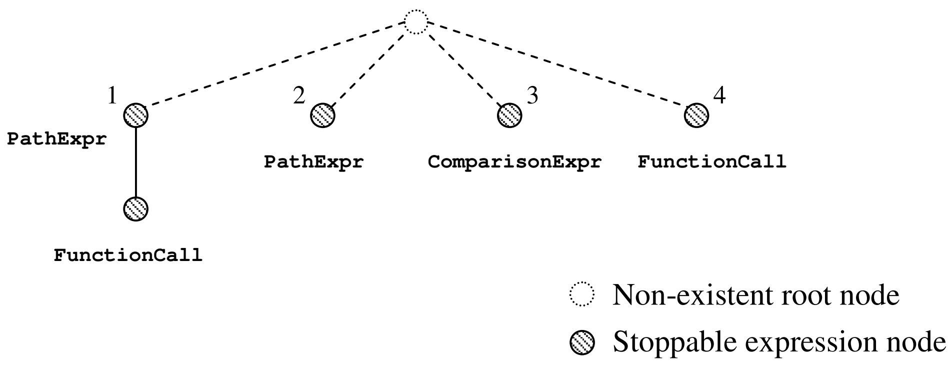


Figure 4 – The debug expression tree view

This debug tree view helps up show the properties of our conceptual solution that would have necessitated much more effort to understand or prove at the conceptual level using the complete debug expression tree. This filtering operation reduces the expression tree to the minimum necessary that is needed during the debugging session. As a side effect, all stoppable expressions having the same lowest common ancestor become siblings in the view. By adding the root node, also the stoppable nodes having no stoppable ancestor become siblings. Also, nodes initially placed at different levels in the tree, are brought on the same level in the tree view. The new sibling and parent-child relations between the tree view nodes give us the exact semantics of the three default code-stepping functions:

- step in – parent-child relation – the target node is the first child of the source node;
- step over – sibling relation – the target node is the next sibling of the source node;
- step return – child-parent and sibling relations - the target node is the next sibling of the parent of the source node.

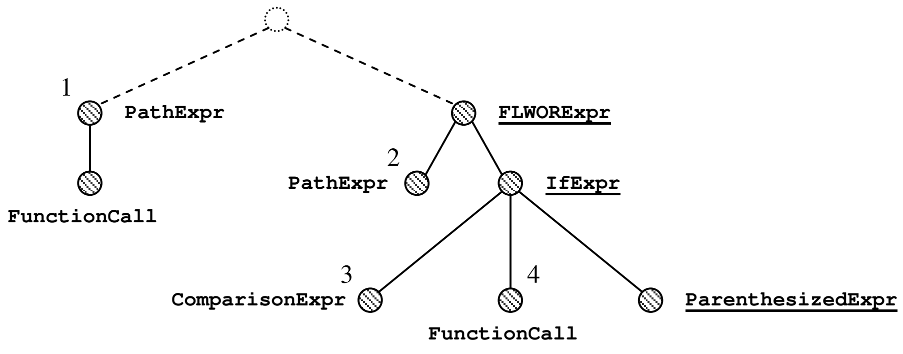
3.5.1 Code rewriting

In debugging there must be a strong correlation between execution and the textual position of expressions, which we model with the query location. All debug actions and events have to be reflected in the debugger client user interface and some of them directly in the query editor. This contains the query the way the user wrote it. With an XQuery engine that performs multiple code transformations during normalization and optimization, tracking the query location of the expressions in the original AST is not an easy task, if possible at all. This is because during code rewriting operations, the engine can decide to add, remove or transform expressions in the original code. Stoppable expressions initially in the code might be either removed or transformed into ones which are not stoppable. Also other stoppable expression can be added by the engine. Therefore we must find a way to preserve the relations between the stoppable expressions as they are in the original code in case transformations are performed.

For the XQuery code example above, according to the XQuery 1.0 formal semantics [23], after normalization of the FLWOR expression, the code looks like this:

```
for $book in doc("books.xml")/books/book
return
    let $authors := $book/author
    return
        if ($book/title = "Ulysses")
        then count($authors)
        else ()
```

Semantically, the code is equivalent. But, from the debugging point of view, between the two queries there is a major difference. This difference can be clearly seen if we compare the two debug expression tree views. The one of the normalized code is presented in the diagram below:



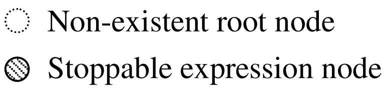


Figure 5 – The debug expression tree view of the normalized code

The underlined tree view nodes are stoppable expressions that the engine introduced during normalization. Also, the relation between the other nodes suffered major changes and this translates into a different behavior of the debugger because the stepping semantics remained the same. For example, the user can no longer step over from the first `PathExpr` to the second. In order to do this in the normalized tree, he would have to step in through the `FLWORExpr` node.

We find this kind of change in the debug behavior unacceptable. Therefore the new stoppable nodes must be either ignored or marked as non-stoppable. Furthermore, the stoppable annotation of the normalized tree view must contain the mapping to the node in the tree view before normalization.

In our example, the programmer sees and wants to debug the four clauses of the FLWOR expression in a linear way. The numbers in the diagram in *Fig. 4*, show the order in which the stoppable expression are followed if only the step over debug action is used, taking as initial source node, the leftmost `PathExpr`. In the tree view corresponding to the normalized code – see *Fig. 5* –, these numbers have to be mapped to reflect the same order. As we see, this ordering models the original sibling relation between the nodes. By defining the stepping semantics relative to this new ordering relation, we manage to adjust the debugger behavior when code normalization is performed. The same kind of ordering must also be defined for the parent-child relation between the nodes in the initial tree view. A possible implementation of this node ordering and mapping to the normalized tree view is presented in the *Architecture* chapter. The solution, based on the Dewey numbering scheme [24] manages to model the ordering for both relations: sibling and parent-child.

Another problem that might appear during code rewriting is the disappearance of some expressions from the debug expression tree. In our work we addressed this problem by only limiting this kind of transformations. More on this topic will be discussed in the *Limitations* section in *Chapter 5*.

3.6 Eager-on-demand evaluation

One important aspect of debugging is code inspection. But, as pointed out in the previous chapter, this breaks two properties of the functional languages. The first one is that breaking the execution and obtaining runtime information – like context or variables in XQuery – is considered a side effecting behavior, which is not allowed in a pure-functional language. But since, exactly for this reason, XQuery is not a pure functional language, adding this kind of side effecting is not a major change in the language.

The more complex issue raised by allowing code inspection is that it breaks the language's lazy-evaluation. In order not to completely change the language to a strict evaluation one, which in our opinion would be a much too drastic approach, we have introduced the *eager-on-demand evaluation* type. This is an enhanced version of lazy-evaluation that allows the engine to demand the evaluation of certain expressions that would have normally been lazily evaluated. These strict evaluation demands are driven by the user debugging needs. Whenever the user breaks the execution, everything that is in scope must be evaluated. This forces the engine to demand the evaluation of the in scope variables and the expression focus, which contains the context item, context position and context size.

We are aware that for an eager-on-demand evaluation implementation one has to make a tradeoff between the strictness of the evaluation and the jumping debugger behavior caused by lazy evaluation. We believe that both extremes are undesirable, the former from the language point of view and the latter from the user's point of view. The implementation has to be somewhere in between. we will shortly discuss this topic in the *Architecture* chapter.

Chapter 4

Architecture

This chapter will present an XQuery debugger implementation based on the conceptual solution presented in the previous chapter. This chapter is organized as follows. We start by presenting a minor modification of the XQuery and extension grammars that helps us better map the debugger client user interface to the debugging semantics. We describe how the Dewey numbering scheme works in order to correct the problems caused by code normalization. In the rest of the chapter we present our implementation and its architecture divided into three main parts: the debugger client and its Graphical User Interface (GUI); the engine debug protocol (EDP) used as the interface between the GUI and the execution engine; and the debug support implementation in the execution engine.

4.1 Language grammar enhancements

For the implementation of the stoppable expressions, we found very useful to slightly modify the original language EBNF grammars of the XQuery-based languages. The modification consists only of the introduction of alias expressions. These aliased are expressions defined on sequence of tokens in the already existing expressions. These aliases are only changing the granularity of the grammar and add more nesting in the query's abstract syntax tree (AST), but the semantics of the original expressions remains unchanged.

We will show how the grammar extensions were defined using the `LetClause` EBNF production as an example. In the standard grammar this production is the following:

```
LetClause    ::=   "let" "$" VarName TypeDeclaration? ":=" ExprSingle
                   ("," "$" VarName TypeDeclaration? ":=" ExprSingle)*
```

This was changed in our implementation by factoring out the sequence of tokens: `"$" VarName TypeDeclaration? ":=" ExprSingle`. The equivalent productions in the new form of the grammar are:

```
LetClause    ::=   "let" LetBinding (","LetBinding)*

LetBinding   ::=   "$" VarName TypeDeclaration? ":=" ExprSingle
```

The complete listing of the new grammars of XQuery and extensions are presented in *Appendix A*.

The advantage brought by these grammar extensions is a two-fold. First, it simplifies the implementation of the stoppable expression concept. All the new productions were introduced in the grammar at places where stoppable expressions appear according to the rules defined in the previous chapter. In this way, the stoppable annotation becomes for most of the expressions a static annotation. For example, `FunctionCall` or, for the new grammar, `LetBinding`, will always be stoppable expressions. For these expressions, this decision is not made at runtime anymore, because they are statically marked as stoppable.

The second enhancement made possible by the grammar extensions is regarding the XQuery editor during debugging. This can be better explained illustrated in a graphical way using the example below. For example, as defined in the stoppable expression selection rules for the FLWOR expression, the breakpoint semantics allows users to break the execution for each of the bindings. We will use again the previous XQuery example and assume the user stopped the execution at the binding in the `let` clause. The following code listing displays the highlighted query location corresponding to the stoppable expression where the execution was suspended:

```
for $book in doc("books.xml")/books/book
let $authors := $book/author
where $book/title = "Ulysses"
return count($authors)
```

But, the highlighted expression is only the one used in the binding and not the binding expression itself. Therefore, from the user's perspective, we find the highlighting in the next code listing more appropriate for this stoppable expression. Hence, our grammar extensions help us better represent the breakpoint semantics.

```
for $book in doc("books.xml")/books/book
let $authors := $book/author
where $book/title = "Ulysses"
return count($authors)
```

4.2 Dewey ordering annotations

For the implementation of the stoppable expressions we found an efficient annotation scheme that is based on the Dewey Order Encoding described by Tatarinov et al. in [24].

In order to show how the Dewey numbering is assigned to the stoppable nodes in the expression tree view, we will provide a short but more complex example. Consider the following XQuery fragment:

```
let $xml := doc("C:/books.xml")
for $book in $xml//book
return
    <book>{
        for $author in $book/author
        return <author>{ data($author/name) }</author>,
        <yeartitle>{
            concat(data($book/year), ": ", data($book/title))
        }</yeartitle>
    }</book>
```

The diagram in *Fig. 6* displays the tree view corresponding to the expression tree of the query above. The number sequence near each node is the Dewey number associated to the corresponding node.

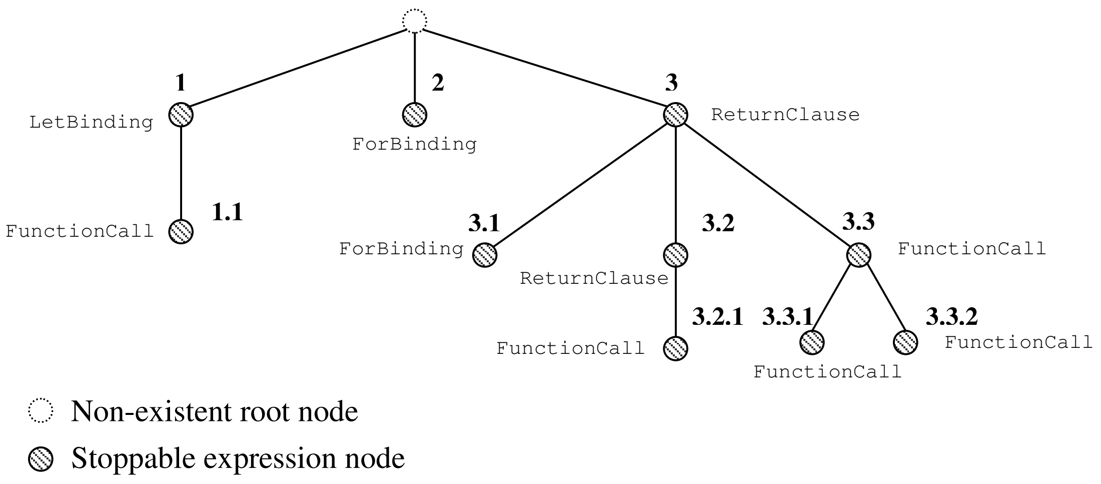


Figure 6 – Debug expression tree view with the Dewey numbering scheme

Associating the Dewey annotation to the stoppable nodes before code transformations helps during the execution to identify the stoppable expressions that were in the original user written code and their relative position to each other. The Dewey numbering scheme also aids an efficient index based implementation of the stepping functions. An easy and efficient node retrieval mechanism can be implemented by indexing the stoppable nodes based on their Dewey number.

4.3 Debugger client. Graphical user interface (GUI)

Due to the large popularity the Eclipse Platform has and the fact that this product already defines the basic user interface (UI) primitives to edit and debug a programming language, we have centered our debugger client development efforts on this platform. Our main concern was to provide a user experience that is as close as possible to the one in the Eclipse environment that the user is already accustomed with.

The XQuery support tools that we developed are packaged in several Eclipse plugins and deployed through the standard Eclipse installation and upgrade mechanisms. Among other utility components the plugins contain, the main ones are the XQuery Editor, XQuery Problem and Outline views, the XQuery edit and debug perspectives, XQuery Launcher, and the XQuery Debug Model.

The XQuery implements many of the features for which Eclipse provides extension points. The ones supported by our editor are the following: syntax color highlighting; error highlighting, and code completion and code templates. We have also developed some more user support features like: as-you-type syntax validation; variable, parameter and function scope checker.

The Outline and Problem Views help the user see the structure of the query written in the editor and the potential code problems, respectively. The Outline View displays the Abstract Syntax Tree (AST) of the most recent syntactically correct query in the editor. The expressions displayed in the AST are the ones in the extended grammar that we provide in *Appendix A*. The Problems View displays an as-you-type updated list of errors and warnings generated by the query in the editor. If the syntax of the query is incorrect, the Problems View displays only the first syntactical error. Otherwise, the AST is generated and several verifiers check the code for semantic inconsistencies. All the gathered semantic errors and warnings are displayed in the Problems View. For the semantic code analysis our current implementation provides the following checkers:

- variable scope validation – an error is displayed if a variable is used without being declared;
- function validation – an error is displayed if a function is used without being declared or the corresponding name does not belong to the list of built-in or constructor functions;
- function namespace prefix validation – for user defined function declarations the usage of the reserved `fn` namespace or the missing default `local` namespace for function calls will be signaled as errors;
- `fn:doc()` parameter validation – the user receives a feedback through a warning if the XML file indicated by the function parameter points to a non-existent XML file.

The XQuery perspectives are providing logical grouping and component layout for the two working environments: editing and debugging. The XQuery perspective is the default one when editing XQuery files having the XQuery Editor and the Outline and the Problems views as the main components. The *Appendix B.1.* shows a snapshot of the editor in the XQuery perspective. Several of the features available in our client can be seen in this image.

The XQuery Debugging perspective provides the necessary tools and for debugging XQuery code. A snapshot of this perspective is presented in the *Appendix B.2.* Among these are the Debug, Breakpoints, Variables, and Console views, editor breakpoint support, code highlighting for execution breaking. Also the necessary actions for controlling the execution during the debugging session are implemented: suspend, step in, step over, step return, resume, and terminate.

Another important and distinct package, crucial for the debugging support, is the Eclipse debug model. The next subsection will be dedicated to this topic.

4.3.1 Eclipse debug model

The diagram in *Fig. 7* displays the UML class diagram of the Eclipse debug model implemented in our plugin. For the aggregation relations both ends have an implicit cardinality of 1 unless else specified. Each of the classes in this diagram implements the corresponding interfaces or helper abstract class provided in the Eclipse debug model, but, for simplicity reasons, these were omitted in the diagram. For example, the XqueryDebugElement class inherits from the provided *DebugElement* abstract class while the XqueryDebugTarget class implements the given *IDebugTarget* interface.

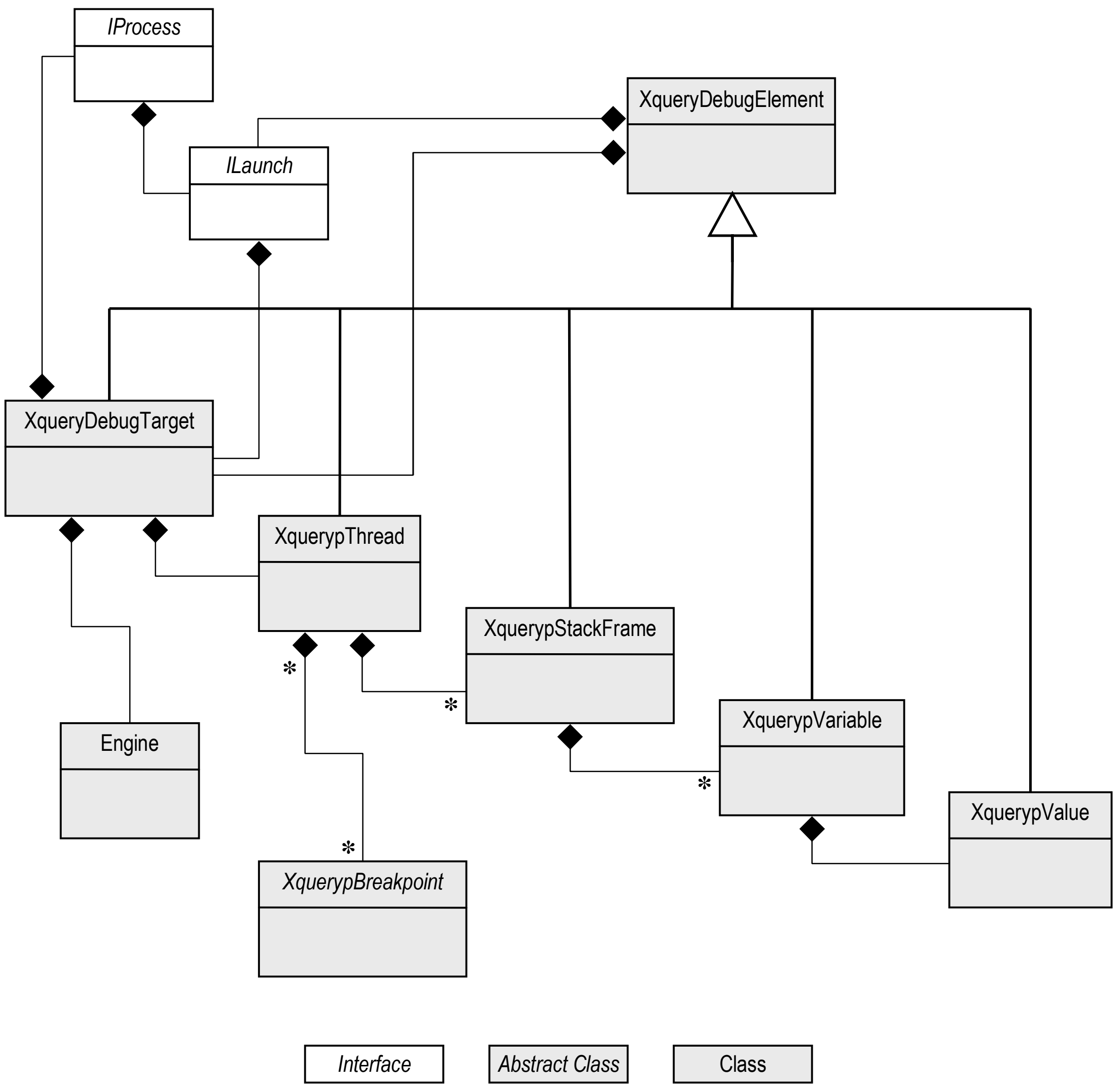


Figure 7 – The UML class diagram of the Eclipse debug model implementation

According to the Eclipse debug model documentation [27], the debug element represents different artifacts used while debugging a program. This class provides the common functionality for all these artifacts. Besides the default functionality implemented in the *DebugElement* class our XquerypDebugElement implementation provides also an access point to the Engine class that works as a proxy to the real execution engine in out plugin.

The debug target represents the execution context that is debugged and is associated with the operating system process of the execution engine running the query that is being debugged. The debug target implements the following actions to control the execution: suspend, resume and terminate. The breakpoint management is also performed at this level.

In the Eclipse debug model, the debug target can contain multiple thread objects. Since both execution engines that we use are performing a single threaded execution, in our Eclipse plugin implementation, a debug target contains only one thread. A thread provides the following execution actions: suspend and resume, stepping, and terminate. Because of the single threaded environment, the debug target's execution actions are always redirected to the thread. Once the thread has been terminated, both the debug target and the thread objects are destroyed and the execution engine process terminated.

In suspend mode, a thread contains one or more stack frames. A stack frame represents an execution context in the suspended thread. In the debug environment of XQuery we have slightly modified the standard concept of stack frames known from the imperative programming languages. In Java, for example, a stack frame is generated for each function call in the call stack up to the main method. It is very common in XQuery to have quite complex queries that don't even use user defined function. In this case the programmer debugging such query will only see one single stack frame regardless of the point where the execution is suspended. Since we implemented the "step in" action also for some other XQuery expressions besides the user defined function calls, these expressions will also generate stack frame objects in the debugger client. Seen from the AST point of view, when the execution is suspended at a certain expression in the tree, one stack frame will be generated for each stoppable expression on the tree path from the suspended expression to the root. In this list, the additional stack frames corresponding to the function calls, will be interleaved.

Each stack frame contains variables. These represent the visible data structures and the function parameters. Each such variable is associated with a value object that stores the data type and the variable value. In our implementation of the Eclipse debug model, the *XquerypValue* class does not provide a hierarchical variable structure. This could be the case of an object-oriented programming language where variables can be object instances with arbitrary structures. In our case all variables and parameters can easily be represented as string values, once the corresponding data type is known. Therefore we don't use the hierarchical variable model provided. The most complex types that we can encounter are XML instances of arbitrary XML schemas. This might necessitate only a better visualization tool, like an XML viewer, that converts the string value into a more user friendly representation.

Appendix B.2 provides a snapshot image of the debugger client showing how these concepts are mapped to the GUI of the Eclipse debugger client.

4.3.2 Challenges

Even though the Eclipse Platform helps with the rapid prototyping and development of language support tools, this was not always the case for the XQuery language family. There had been several components that required extra development to provide a suitable functionality for XQuery.

The main problem encountered was the fact that XQuery is not a typical keyword-based language like other programming languages, for example, C or Java. In XQuery the keywords are not reserved words. Therefore any keyword can appear in any expression as name identifiers – for variables, for functions, namespaces, XML elements and attributes, etc. – or as text in XML element content and attribute values. This invalidates some implicit assumption of the Eclipse platform when developing language support tools. Therefore the XQuery language needed a special treatment.

The following code listing displays a syntactically correct XQuery code fragment that uses the “return” word with multiple uses. The first and the last appearance are variable name identifiers; the second is a name test of a step expression as part of a path expression; the third occurrence is the “return” keyword; and the fourth instance is a function name.

```
for $return in doc("xmlfile")/return
return return($return)
```

In this case the tools provided in the Eclipse Platform for code syntax coloring help only to a limited extent. In order to better understand our approach for syntax coloring we will first describe how this functionality works in a keyword-based language with reserved keywords.

The whole process is a performed is several steps and triggered by any textual change in the editor. Initially, the source code is split into several partitions by a partition scanner. Each partition type is processed by a different set of rules. For example, in XQuery, we implemented the following code partitions types: one for XQuery comments, one for XML comments, and the rest of the code falling in the XQuery source code partition type.

Partitions are disjoint fragments of text and are identified by the partition scanner according to certain rules, for example, the sequence of characters that marks the beginning and the ending of a partition. Thus, we can always identify when, for example, an XQuery comment partition is starting by searching for the “`(:`” sequence. The following listing shows how the source code is split into partitions. The underlined character sequences are the partition delimiters. The text starts with an XQuery comment partition. In the 5th line we have an XML comment partition. The remaining text forms two XQuery source code partitions, one between the two comment partitions and one after the XML comment partition.

```
(: this is an XQuery comment partition :)
let $x := 123
return
      <result>
        <!-- This is an XML comment partition -->
        { $x }
      </result>
```

Other languages implement such partitions also for atomic types like strings and characters. This is not possible in XQuery because the string delimiters ‘"’ and ‘'’ are also used in other grammar constructs like XML attributes and hence, they don’t uniquely identify such a partition. Since partitions must be disjoint and XML attributes can contain an arbitrary XQuery enclosed expression, a partition types for such grammar constructs cannot be provided through the Eclipse Platform partitioning scheme.

Once the code partitions have been identified, each of them applies the associated rules and logic to process the enclosed tokens. This process generates the color highlighting of individual tokens inside the partition. If the language keywords were reserved, this would be the place where their coloring will be generated. Because in a code partition a reserved keyword cannot appear in another token, this can be safely given the associated keyword color.

In the previous process we identified at least two limitation of the Eclipse Platform for an XQuery-based language. On one hand there are a limited number of partition types that can be implemented through the classical Eclipse mechanisms. This is against the need to provide a better look-and-feel, and therefore we provide an enhancement to the partitioning mechanism. On the other hand, many language constructs cannot be identified until the source code is complete, in the sense of having a correct syntax.

For both limitations, we looked closer to the way users write source code in general. We observed that, while programming, users tend to always keep a correct syntactical source code. Thus, the process of writing a source code file is usually done in an incremental manner. In this situation, once the source code has a correct syntax, we can parse it and obtain the AST. This gives us the necessary information to correctly process the entire source code file.

For a better code partitioning, we have developed semantic partitions. These partitions have the same rules and logic as the ones described above, with the single difference that they are installed only after the AST is obtained. This approach enabled us to use partitions at a more granular level for: string literals, XML content and XML attribute values.

In order to support color highlighting for the source code partition we have implemented a semantic highlighting similar to the one used in Eclipse Java Development Toolkit

(JDT)[8]. Semantic highlighting works on the same principle as normal color highlighting with the difference that it are delayed by the so called code reconciliation. During this reconciliation we perform the parsing of the source file and obtain the AST. Based on the information in the AST we are able to complete the color highlighting with all the tokens that could not be colored during the classical partitioning and highlighting.

4.4 Debug protocol

The interfacing between debugger client and the execution engine is done through the debug protocol that we have defined.

4.4.1 Communication layer architecture

The diagram in *Fig. 8* shows the architecture of the communication layer that we implemented. On one side, we have the debugger client which, in our implementation, is the XQuery Eclipse Plugin, and, on the other, the execution engine. As a communication protocol between the two parties we chose the Transport Control Protocol (TCP). This protocol is fast enough when the two parties are on the same machine, and also allows for future extensions for remote debugging if this will be required.

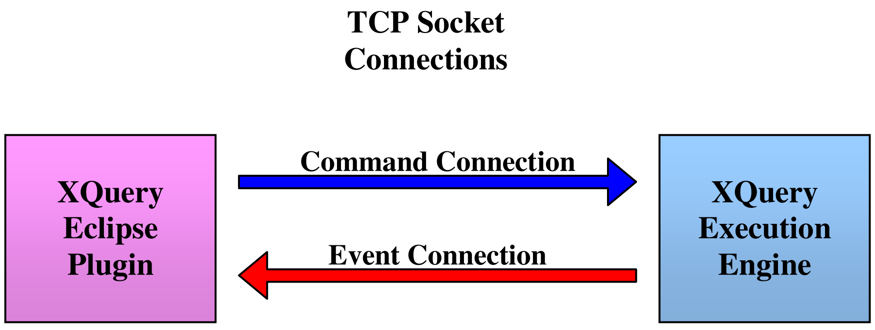


Figure 8 – The TCP communication layer

The communication structure follows the specification of the Java Debug Wire Protocol (JDWP) [25] taking benefit of the JDWP extensibility. This extension implementation would allow us to integrate our debugger implementation with other tools that have support for the JDWP.

Like in the JDWP, we use two communication channels: one for sending commands and requests to the execution engine and a second one to receive events and notifications from the engine. The Eclipse plugin is responsible for the command connection, having the server role, while the execution engine manages the event connection, in which case the engine plays the server role. The communication is considered established once each

[8] Because it has a limited usefulness in a language like Java, by the time we developed this feature, the JDT v.3.2.2 did not provide an API for it. We expect that later version of the Eclpse Platform will provide this API.

server has a connection established with the other party as client. The communication ports are chosen at runtime by the plugin and send to the engine through the execution engine command path arguments.

After the connection is established, a handshake message is sent by the plugin to the engine on the command connection. The role of this message is to test the functionality of the connection and the engine. The handshake message is a fixed length message that contains the debug protocol version that the plugin expects. The engine must reply with the version numbers of the protocol it implements. If the reply matches the initial message the communication can continue. Otherwise, the connections between the two parties are broken. In this situation the plugin can use the reply message to display the appropriate error message to the user.

4.4.2 Message formats

The handshake message is an 11 byte message having the following format:

```
XQDP vMM.mm
```

where:

MM – is the two digit major version number of the protocol;
mm – is the two digit minor version number of the protocol;
The remaining bytes are the corresponding 1-byte characters.

Messages sent on the command connection are request-reply messages. A command will always receive a reply; otherwise a protocol exception will be raised. The request will always be sent by the plugin and the reply by the execution engine. On the other hand, the messages sent over the event connection are only one way messages coming from the execution engine to the plugin. In the current implementation there are no commands requested by the engine, as well as no events raised by the plugin.

The commands and the evens have the following request message structure:

```
LLLLIIIIFKCDATA...
```

The first 11 byte of the message form the request message header where:

LLLL – is the 4-byte length of the entire packet. This gives an upper bound on the length of the packet sent;
IIII – is the 4-byte ID of the meaage;
F – 1-byte used for flags;
K – 1-byte identifying the command set;
C – 1-byte identifying the command.

The rest of the message is for the variable payload data which has a length of **LLLL** – 11.

The replies for commands have the following reply message structure:

```
LLLLIIIIFEEDATA...
```

The first 11 byte of the message form the reply message header where:

`LLLL`, **`IIII`** and **`F`** have the same meaning as for request messages;

`EE` – is a 2-byte error code.

The rest of the message is for the variable payload data which has a length of **`LLLL`** – 11.

We haven't assigned any additional flags other than what JWDP specifies. Therefore, the only reserved flag remains 0x80, the flag for a reply messages.

4.4.3 Message payload. Data serialization

For the message data payload we found JSON (JavaScript Object Notation) [26] to be a very convenient data representation. There are a few reasons for choosing JSON as the data serialization format.

Firstly, we avoided specifying or implementing other more costly versions of our debug protocol. On one hand, a binary data serialization would have meant more low level work to be done. On the other hand, an XML implementation wouldn't have been as lightweight as the JSON version, due to the higher complexity of XML compared to the JSON format. An XML format needs also special care when XML data has to be transmitted; and XML data is part of the XDM that has to be transported by the protocol.

Even if JSON itself is not a standard format[9], one can consider it as a de-facto standard due to its widespread use. The representation is a text, user oriented, language independent one. This suits our implementation environment where the debugger client and the execution engines are implemented in different programming languages.

Also, there are many ready-to-use free JSON implementations for more than 30 programming languages. This makes it easy to hook in most of the used programming languages.

4.4.4 Command sets and commands

We have followed the same kind of organization of commands into command sets as in JDWP. The constants assigned to our command sets and commands were from the ranges provided for extensibility in JDWP. Out protocol implements groups the commands in five command sets as follows:

- Execution Command Set – includes the following commands that are used to control the execution:
    - Run: starts the execution of the query. This command is sent by the debugger client after the "Started" event is received.
    - Suspend: suspends the execution of a running query.
    - Resume: resumes the execution of a suspended query.
    - Terminate: terminates the execution.
    - Step: performs one of the stepping actions: step in, step over or step return
- Breakpoints Command Set – groups the breakpoint related commands.

[9] JSON is a subset of the JavaScript programming language which is an ECMA standard since Dec. 1999.

  - Set: sets one ore more breakpoints at given query locations.
  - Clear: clears the breakpoints at the given query locations.
- Static Command Set – contains all the commands that are used to gather information about the static context during the XQuery execution.
  - Options: retrieves from the execution engine all the non-string – fixed length – static context options, like: ordering mode, empty sequence order, copy-namespace mode, etc.
  - Defaults: retrieves from the execution engine all the string value static context options, like: base URI, default collation, default function namespace, etc.
  - Sets: retrieves from the execution engine all the set-valued static options, like: in scope variable, function signatures, statically known collections, etc.
- Dynamic Command Set – contains all the commands that are used to gather information mainly from the dynamic context but also run-time information during the XQuery execution.
  - Data: used to perform arbitrary data evaluations.
  - Variables: retrieves the list of in scope variables together with their values.
  - Frames: retrieves the list of debug nodes in the debug expression tree up to the root. This list interleaves expression tree nodes with function calls stack frames.
  - Focus: retrieves the expression focus component of the dynamic context.
  - Time: retrieves the execution's implementation specific date and time
  - Documents: retrieves the available documents component of the dynamic context.
  - Collections: retrieves the available collations component of the dynamic context.
  - Collection: retrieves the default collection component of the dynamic context.
- Engine Events Command Set – includes all the events or notifications that the execution engine can send to the debugger client. These are asynchronously sent by the engine as responses to corresponding command when the engine state changes. The asynchronous communication is needed in case the processing takes more time.
  - Started: sent by the engine after it has initialized its state for the debug session – opening the communication ports, setting the query to debug and setting the breakpoints.
  - Terminated: sent asynchronously as a response to the terminate command.
  - Suspended: sent asynchronously as a response to the suspend command.
  - Resumed: sent asynchronously as a response to the resume command.

The larger description and the structure of each command are given in *Appendix C*.

4.4.5 Engine debug protocol communication pattern

There are a couple of communication patterns used for sending messages between the debugger client and the server implemented in XQuery engine.

The first pattern is a simple synchronous request-reply message. This pattern is used whenever the computation the engine needs to perform according to the received message is small. Such a message is the setting of breakpoints, which is displayed in *Fig. 9 (a)*.

The second pattern is the asynchronous request-reply message. This is used whenever the engine might need more time to perform the action specified in the message. This is the case for all the execution commands that change the execution state, since this operation might not only take a longer time to execute, but might also have to perform related computations. For example, the Suspend command, before switching the engine to the "suspended" state, might allow the engine to perform the necessary computation in order to be ready for providing internal state information to a debugger client. After the Suspend command is successfully performed, there is a high chance that a Variables command will be sent. In order to increase the debugger client responsiveness, the engine might make sure the needed debug information is ready by the time the engine state is changed to "suspended". This communication pattern is shown in *Fig. 9 (b).*

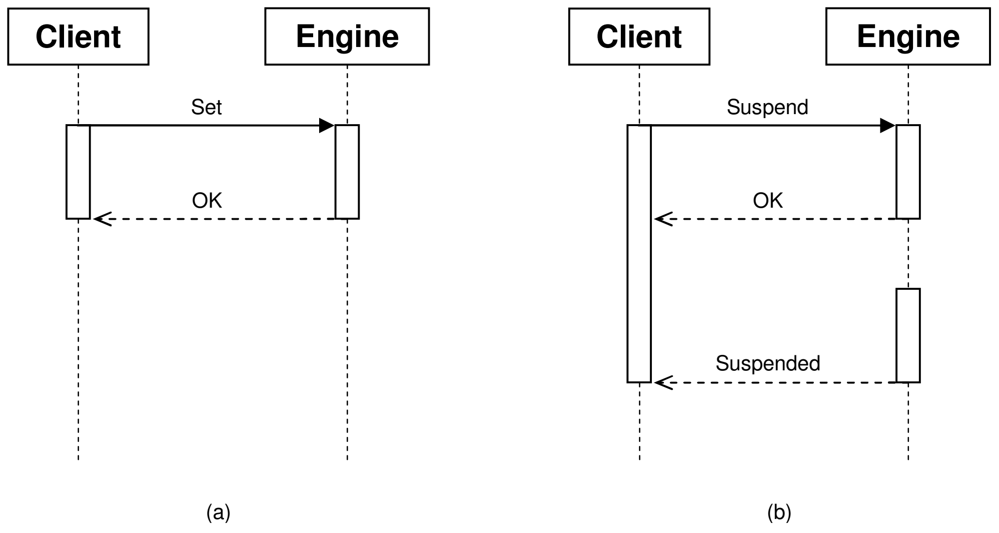


Figure 9 – (a) Synchronous request-reply communication pattern
(b) Asynchronous request-reply communication pattern

4.5 Execution engine instrumentation

Before starting to describe how the debugger functionality was introduced through execution engine instrumentations, a short introduction on how our engine processes the XQuery code might be helpful. *Fig. 10* displays the five different steps involved in the execution process. Initially, the XQuery source code is parsed and an Abstract Syntax Tree (AST) is generated. This tree is further transformed into an expression tree, which contains more information about each expression than the AST. During this step, code normalization is also performed. Another transformation of the expression tree is performed due to code optimizations. Once the normalized and optimized version of the expression tree is obtained, the code generation step is performed. This has as result an iterator tree which is interpreted by the runtime system of execution engine in order to compute the result of the query. The above process can be split into two phases: compilation – the first four steps and interpretation – the last step.

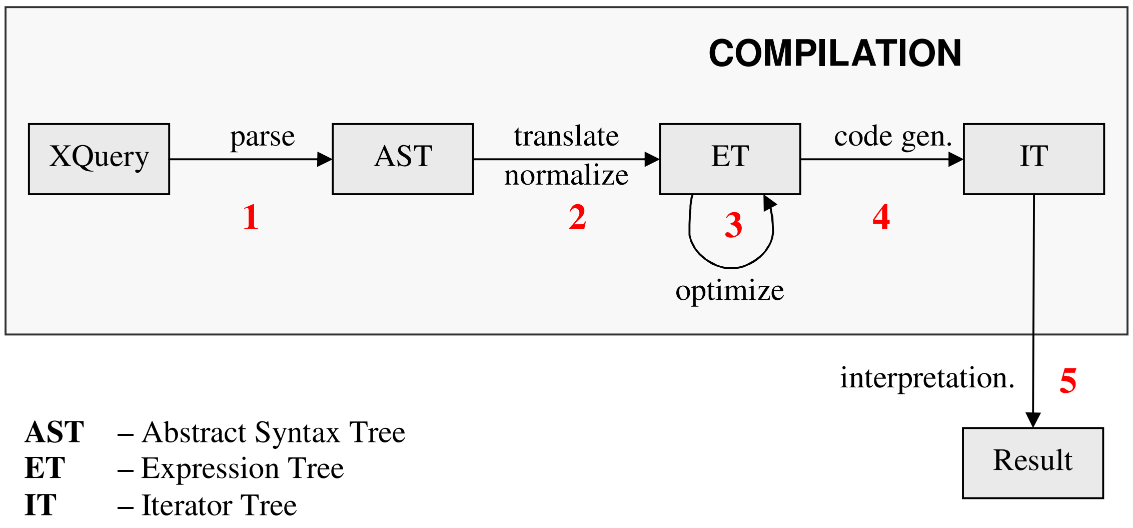


Figure 10 – XQuery execution stages

The iterator tree generation is the result of the last phase of the compilation. Similar to the SQL query processing technique, each iterator in the tree, has an open-next-close interface [30]. Such processing model enables modularity, has low memory requirements, and avoids the materialization of intermediate result. This is because iterators process the input one item at a time and only as much as it is necessary. This implementation allows the runtime to naturally exploit pipelining, or implement lazy evaluation.

4.5.1 Debug iterators

Like in all other programming languages, in order to gain control over execution compilation in debug mode has to be different than the one without debug support. On one hand, this allows disabling or simplifying the compilation steps that are not compatible with the debug option. This is the case of some code rewritings especially during the optimization step. On the other hand, the runtime must have more information available in debug mode, like the query location information for every expression in the code, the breakpoint information, and also the stoppable annotations described in the previous chapters. The best place to store all this information is the iterator tree which already stores the query locations for error reporting reasons.

Our approach to enabling this debug information was similar with the insertion of debug nodes in the expression tree for the implementation of stoppable expressions. In this case we introduce one more iterator type called *debug iterator*. Each debug iterator is corresponding to one debug node in the expression tree. Its purpose is to control the behavior of the child iterator which, of course, corresponds to a stoppable expression. The next listing displays a small XQuery frafment and the corresponding XML representation of the iterator tree in the debug mode. The debug iterators have been bolded in the tree listing. As we can see, there is one debug iterator for each of the binding in the `let` clause and one for the `return` clause.

```
Query:
------

let $x := 1, $y := 2
return $x


Iterator Tree:
--------------

<FLWORIterator loc="f.xq:2.2" addr="0x80...">
  <FnDebugIterator loc="f.xq:2.2" addr="0x80...">
    <LetVariable name="x" materialize="true">
      <SingletonIterator value="xs:integer(1)" loc="f.xq:2.2"
                 addr="0x80..."/>
    </LetVariable>
  </FnDebugIterator>
  <FnDebugIterator loc="f.xq:2.2" addr="0x80...">
    <LetVariable name="y" materialize="true">
      <SingletonIterator value="xs:integer(2)" loc="f.xq:2.2"
                 addr="0x80..."/>
    </LetVariable>
  </FnDebugIterator>
  <FnDebugIterator loc="f.xq:2.2" addr="0x80823f0">
    <ReturnClause>
      <LetVarIterator varname="x" loc="f.xq:2.2" addr="0x80..."/>
    </ReturnClause>
  </FnDebugIterator>
</FLWORIterator>
```

The debug iterators implement the stepping functions described in the conceptual solution chapter. They also enable the breakpoint behavior by locking and signaling the runtime thread. Also, in the "suspended" state of the engine, the debug iterator is responsible for providing the context information to the debugger client. This includes also in scope variables.

Having control over the iterators beneath in the tree, the debug iterators also implement the eager-on-demand evaluation. When information is requested by the debugger client, the debug iterator might need to consume items from the child iterators that, in the normal execution mode – without debugging or without the client requesting this information – wouldn't have been consumed due to the lazy-evaluation behavior.

Chapter 5

Conclusion

5.1 Limitations

The tradeoffs that we have made during the project did not only help us achieve most of our goals, but also introduced limitation in our approach. These will be described in this section.

5.1.1 Debugger client GUI limitations

Our XQuery editor relies on an external parser for syntax error checking. Such an approach limits our implementation to give the user only the first syntax error found in the code. Other editors use parsers with a recoverable state, allowing the parsing to continue after a syntax error. Due to time constraints we did not approach this problem, even though we have some possible solutions to the problem.

5.1.2 Execution engine limitations

Mapping the debugging of XQuery-based languages to imperative debugging primitives pays its price in the many tradeoffs that the engine has to make. Allowing code inspection at every stoppable expression location in the code, and thus a side effecting behavior, makes harder to verify, optimize, and parallelize programs. Also a debugger allowing some strict evaluation behavior will not provide the same query execution guarantees with a pure lazy-evaluation execution. This is, for example, the common case of code paths that generate dynamic errors but that would not be evaluated during a pure lazy-evaluation execution. Performing code inspections that force the evaluation of such code paths, the error will cause the execution to fail. Thus the debug mode execution will have a different result than the one not in debug mode. This is of course not acceptable, but we believe that, even if not always true, errors are one reason why programmers use debuggers, and their goal is to find the errors.

As mentioned in the *Chapter 2*, ordering is one of the main issues of debugging XQuery. On one side, there is the `unordered` language expression that frees the execution engine from any ordering constraints in processing data. On the other hand, the `order by` clause imposes the materialization of all the results in a FLWOR expression before the first result is retuned. Both these language features are not supported by out debug model. In these cases we cannot prevent the code stepping to have a jumping behavior which is dependent on a particular engine implementation.

5.2 Future work

There is not only a long, but also great road ahead of us. The few XQuery debugger implementations, the amount of effort put into standardization of the XQuery family of languages and the rising need for such a language give us good signs that the direction of our work is the right one.

Unfortunately, like in many real projects, time is limited, and so were we in providing more useful features to both our Eclipse debugger client and the execution engines. One of our main goals that fell out of scope during the project was providing debug support in the MXquery engine. Part of this support, which is the debug protocol implementation, was build for both debug clients and servers. Hence the future debugger support implementation in this engine can just use the current protocol API available in the Eclipse debugger client, both being Java implementations.

The current running version of the Eclipse plugin and the Zorba engine implement only part of the specified debug protocol. Suspending and resuming execution, breakpoints and variable retrieval command and related messages have been implemented. The messaging for the other features, like code stepping and data evaluation, are pending on a "to-do" list and were scheduled for the next development period, but they were not implemented during the timeframe of this work.

From the Eclipse debugger client point of view, there are many "nice-to-have" features pending. Many of these tools and features are meant to support XQuery development and some, even have usefulness form a debugger point of view. Since we had much more work to do to enable the core debugger functionality, these side features were postponed for later releases of our implementation. Just to enumerate some of these "nice-to-have" features, we can mention:

- providing XML editing support for the XQuery editor;
- enhancing the code completion feature with in scope variable, user defined function and imported function suggestions; also context sensitivity should be added to code completion;
- static error checking;
- enhanced syntax error checking, i.e. allowing the user to see more than the first syntax error by providing a syntax checker with a recoverable state;
- an extension of the breakpoint functionality to allow users to place breakpoint directly in the Outline View which displays the Abstract Syntax Tree (AST);
- adding conditional breakpoints;
- allowing variable updates from the debugger client interface during a suspended execution;
- providing the appropriate UI for data visualization, for example, an XDM viewer.

But the Eclipse Platform offers a great potential through the large codebase that has already been developed for it. Thus interaction with other tools is another possible future extension. There are multiple tools already implemented for features like schema editing or XML visualization. Therefore more benefits can be gained through an integration with such tools rather than implementing them from scratch.

Appendix A

EBNF Grammars

A.1 Modified XQuery EBNF

This section lists our modified version of the XQuery language EBNF grammar. The differences from the original grammar are outlined in the following listing. The newly introduced clauses are bolded. The underlined expressions or clauses are the ones that we treated as stoppable expressions.

```
 [1]  Module                   ::=  VersionDecl? (LibraryModule | MainModule)
 [2]  VersionDecl              ::=  xquery "version" StringLiteral ("encoding" StringLiteral)?
                                    Separator
 [3]  MainModule               ::=  Prolog QueryBody
 [4]  LibraryModule            ::=  ModuleDecl Prolog
 [5]  ModuleDecl               ::=  module "namespace" NCName "=" URILiteral Separator
 [6]  Prolog                   ::=  ((DefaultNamespaceDecl | Setter | NamespaceDecl | Import)
                                    Separator)* ((VarDecl | FunctionDecl | OptionDecl) Separator)*
 [7]  Setter                   ::=  BoundarySpaceDecl | DefaultCollationDecl | BaseURIDecl |
                                    ConstructionDecl | OrderingModeDecl | EmptyOrderDecl |
                                    CopyNamespacesDecl
 [8]  Import                   ::=  SchemaImport | ModuleImport
 [9]  Separator                ::=  ;
[10]  NamespaceDecl            ::=  declare "namespace" NCName "=" URILiteral
[11]  BoundarySpaceDecl        ::=  declare "boundary-space" ("preserve" | "strip")
[12]  DefaultNamespaceDecl     ::=  declare "default" ("element" | "function") "namespace" URILiteral
[13]  OptionDecl               ::=  declare "option" QName StringLiteral
[14]  OrderingModeDecl         ::=  declare "ordering" ("ordered" | "unordered")
[15]  EmptyOrderDecl           ::=  declare "default" "order" "empty" ("greatest" | "least")
[16]  CopyNamespacesDecl       ::=  declare "copy-namespaces" PreserveMode "," InheritMode
[17]  PreserveMode             ::=  preserve | "no-preserve"
[18]  InheritMode              ::=  inherit | "no-inherit"
[19]  DefaultCollationDecl     ::=  declare "default" "collation" URILiteral
[20]  BaseURIDecl              ::=  declare "base-uri" URILiteral
[21]  SchemaImport             ::=  import "schema" SchemaPrefix? URILiteral ("at" URILiteral (","
                                    URILiteral)*)?
[22]  SchemaPrefix             ::=  ("namespace" NCName "=") | ("default" "element" "namespace")
[23]  ModuleImport             ::=  import "module" ("namespace" NCName "=")? URILiteral ("at"
                                    URILiteral ("," URILiteral)*)?
[24]  VarDecl                  ::=  declare "variable" "$" QName TypeDeclaration? ((":=" ExprSingle) |
                                    "external")
[25]  ConstructionDecl         ::=  declare "construction" ("strip" | "preserve")
[26]  FunctionDecl             ::=  declare "function" QName "(" ParamList? ")" ("as" SequenceType)?
                                    (EnclosedExpr | "external")
[27]  ParamList                ::=  Param ("," Param)*
[28]  Param                    ::=  $ QName TypeDeclaration?
[29]  EnclosedExpr             ::=  { Expr "}"
[30]  QueryBody                ::=  Expr
[31]  Expr                     ::=  ExprSingle ("," ExprSingle)*
[32]  ExprSingle               ::=  FLWORExpr | QuantifiedExpr | TypeswitchExpr | IfExpr | OrExpr
```

```
[33]  FLWORExpr               ::=  (ForClause | LetClause)+ WhereClause? OrderByClause? ReturnClause
[34]  ForClause               ::=  for ForBinding ("," ForBinding)*
[35]  ForBinding              ::=  "$" VarName TypeDeclaration? PositionalVar? "in" ExprSingle
[36]  PositionalVar           ::=  at "$" VarName
[37]  LetClause               ::=  let LetBinding ("," LetBinding)*
[38]  LetBinding                   "$" VarName TypeDeclaration? ":=" ExprSingle
[39]  WhereClause             ::=  where ExprSingle
[40]  OrderByClause           ::=  (("order" "by") | ("stable" "order" "by")) OrderSpecList
[41]  OrderSpecList           ::=  OrderSpec ("," OrderSpec)*
[42]  OrderSpec               ::=  ExprSingle OrderModifier
[43]  OrderModifier           ::=  ("ascending" | "descending")? ("empty" ("greatest" | "least"))?
                                   ("collation" URILiteral)?
[44]  ReturnClause            ::=  return ExprSingle
[45]  QuantifiedExpr          ::=  ("some" | "every") QuantifiedBinding ("," QuantifiedBinding)*
                                   "satisfies" ExprSingle
[46]  QuantifiedBinding       ::=  "$" VarName TypeDeclaration? "in" ExprSingle
[47]  TypeswitchExpr          ::=  typeswitch "(" Expr ")" CaseClause+ DefaultClause
[48]  CaseClause              ::=  case ("$" VarName "as")? SequenceType "return" ExprSingle
[49]  DefaultClause           ::=  default ("$" VarName)? "return" ExprSingle
[50]  IfExpr                  ::=  if "(" Expr ")" ThenClause ElseClause
[51]  ThenClause              ::=  "then" ExprSingle
[52]  ElseClause              ::=  else ExprSingle
[53]  OrExpr                  ::=  AndExpr ( "or" AndExpr )*
[54]  AndExpr                 ::=  ComparisonExpr ( "and" ComparisonExpr )*
[55]  ComparisonExpr          ::=  RangeExpr ( (ValueComp | GeneralComp | NodeComp) RangeExpr )?
[56]  RangeExpr               ::=  AdditiveExpr ( "to" AdditiveExpr )?
[57]  AdditiveExpr            ::=  MultiplicativeExpr ( ("+" | "-") MultiplicativeExpr )*
[58]  MultiplicativeExpr      ::=  UnionExpr ( ("*" | "div" | "idiv" | "mod") UnionExpr )*
[59]  UnionExpr               ::=  IntersectExceptExpr ( ("union" | "|") IntersectExceptExpr )*
[60]  IntersectExceptExpr     ::=  InstanceofExpr ( ("intersect" | "except") InstanceofExpr )*
[61]  InstanceofExpr          ::=  TreatExpr ( "instance" "of" SequenceType )?
[62]  TreatExpr               ::=  CastableExpr ( "treat" "as" SequenceType )?
[63]  CastableExpr            ::=  CastExpr ( "castable" "as" SingleType )?
[64]  CastExpr                ::=  UnaryExpr ( "cast" "as" SingleType )?
[65]  UnaryExpr               ::=  ("-" | "+")* ValueExpr
[66]  ValueExpr               ::=  ValidateExpr | PathExpr | ExtensionExpr
[67]  GeneralComp             ::=  = | "!=" | "<" | "<=" | ">" | ">="
[68]  ValueComp               ::=  eq | "ne" | "lt" | "le" | "gt" | "ge"
[69]  NodeComp                ::=  is | "<<" | ">>"
[70]  ValidateExpr            ::=  validate ValidationMode? "{" Expr "}"
[71]  ValidationMode          ::=  lax | "strict"
[72]  ExtensionExpr           ::=  Pragma+ "{" Expr? "}"
[73]  Pragma                  ::=  (# S? QName (S PragmaContents)? "#)"
[74]  PragmaContents          ::=  (Char* - (Char* '#)' Char*))
[75]  PathExpr                ::=  ("/" RelativePathExpr?) | ("//" RelativePathExpr) |
                                   RelativePathExpr
[76]  RelativePathExpr        ::=  StepExpr (("/" | "//") StepExpr)*
[77]  StepExpr                ::=  FilterExpr | AxisStep
[78]  AxisStep                ::=  (ReverseStep | ForwardStep) PredicateList
[79]  ForwardStep             ::=  (ForwardAxis NodeTest) | AbbrevForwardStep
[80]  ForwardAxis             ::=  ("child" "::") | ("descendant" "::") | ("attribute" "::") | ("self"
                                   "::") | ("descendant-or-self" "::") | ("following-sibling" "::") |
                                   ("following" "::")
[81]  AbbrevForwardStep       ::=  @? NodeTest
[82]  ReverseStep             ::=  (ReverseAxis NodeTest) | AbbrevReverseStep
[83]  ReverseAxis             ::=  ("parent" "::") | ("ancestor" "::") | ("preceding-sibling" "::") |
                                   ("preceding" "::") | ("ancestor-or-self" "::")
[84]  AbbrevReverseStep       ::=  ..
[85]  NodeTest                ::=  KindTest | NameTest
[86]  NameTest                ::=  QName | Wildcard
[87]  Wildcard                ::=  * | (NCName ":" "*") | ("*" ":" NCName)
[88]  FilterExpr              ::=  PrimaryExpr PredicateList
[89]  PredicateList           ::=  Predicate*
[90]  Predicate               ::=  [ Expr "]"
[91]  PrimaryExpr             ::=  Literal | VarRef | ParenthesizedExpr | ContextItemExpr |
                                   FunctionCall | OrderedExpr | UnorderedExpr | Constructor
[92]  Literal                 ::=  NumericLiteral | StringLiteral
[93]  NumericLiteral          ::=  IntegerLiteral | DecimalLiteral | DoubleLiteral
[94]  VarRef                  ::=  $ VarName
[95]  VarName                 ::=  QName
[96]  ParenthesizedExpr       ::=  ( Expr? ")"
```

```
 [97]   ContextItemExpr              ::=   .
 [98]   OrderedExpr                  ::=   ordered "{" Expr "}"
 [99]   UnorderedExpr                ::=   unordered "{" Expr "}"
[100]   FunctionCall                 ::=   QName "(" (ExprSingle ("," ExprSingle)*)? ")"
[101]   Constructor                  ::=   DirectConstructor | ComputedConstructor
[102]   DirectConstructor            ::=   DirElemConstructor | DirCommentConstructor | DirPIConstructor
[103]   DirElemConstructor           ::=   < QName DirAttributeList ("/>" | (">" DirElemContent* "</" QName S?
                                           ">"))
[104]   DirAttributeList             ::=   (S (QName S? "=" S? DirAttributeValue)?)*
[105]   DirAttributeValue            ::=   ('"' (EscapeQuot | QuotAttrValueContent)* '"') | ("'" (EscapeApos |
                                           AposAttrValueContent)* "'")
[106]   QuotAttrValueContent         ::=   QuotAttrContentChar | CommonContent
[107]   AposAttrValueContent         ::=   AposAttrContentChar  | CommonContent
[108]   DirElemContent               ::=   DirectConstructor | CDataSection | CommonContent |
                                           ElementContentChar
[109]   CommonContent                ::=   PredefinedEntityRef | CharRef | "{{" | "}}" | EnclosedExpr
[110]   DirCommentConstructor        ::=   <!-- DirCommentContents "-->"
[111]   DirCommentContents           ::=   ((Char - '-') | ('-' (Char - '-')))*
[112]   DirPIConstructor             ::=   <? PITarget (S DirPIContents)? "?>"
[113]   DirPIContents                ::=   (Char* - (Char* '?>' Char*))
[114]   CDataSection                 ::=   <![CDATA[ CDataSectionContents "]]>"
[115]   CDataSectionContents         ::=   (Char* - (Char* ']]>' Char*))
[116]   ComputedConstructor          ::=   CompDocConstructor | CompElemConstructor | CompAttrConstructor |
                                           CompTextConstructor | CompCommentConstructor | CompPIConstructor
[117]   CompDocConstructor           ::=   document "{" Expr "}"
[118]   CompElemConstructor          ::=   element (QName | ("{" Expr "}")) "{" ContentExpr? "}"
[119]   ContentExpr                  ::=   Expr
[120]   CompAttrConstructor          ::=   attribute (QName | ("{" Expr "}")) "{" Expr? "}"
[121]   CompTextConstructor          ::=   text "{" Expr "}"
[122]   CompCommentConstructor       ::=   comment "{" Expr "}"
[123]   CompPIConstructor            ::=   processing-instruction (NCName | ("{" Expr "}")) "{" Expr? "}"
[124]   SingleType                   ::=   AtomicType "?"?
[125]   TypeDeclaration              ::=   as SequenceType
[126]   SequenceType                 ::=   ("empty-sequence" "(" ")") | (ItemType OccurrenceIndicator?)
[127]   OccurrenceIndicator          ::=   ? | "*" | "+"
[128]   ItemType                     ::=   KindTest | ("item" "(" ")") | AtomicType
[129]   AtomicType                   ::=   QName
[130]   KindTest                     ::=   DocumentTest | ElementTest | AttributeTest | SchemaElementTest |
                                           SchemaAttributeTest | PITest | CommentTest | TextTest | AnyKindTest
[131]   AnyKindTest                  ::=   node "(" ")"
[132]   DocumentTest                 ::=   document-node "(" (ElementTest | SchemaElementTest)? ")"
[133]   TextTest                     ::=   text "(" ")"
[134]   CommentTest                  ::=   comment "(" ")"
[135]   PITest                       ::=   processing-instruction "(" (NCName | StringLiteral)? ")"
[136]   AttributeTest                ::=   attribute "(" (AttribNameOrWildcard ("," TypeName)?)? ")"
[137]   AttribNameOrWildcard         ::=   AttributeName | "*"
[138]   SchemaAttributeTest          ::=   schema-attribute "(" AttributeDeclaration ")"
[139]   AttributeDeclaration         ::=   AttributeName
[140]   ElementTest                  ::=   element "(" (ElementNameOrWildcard ("," TypeName "?"?)?)? ")"
[141]   ElementNameOrWildcard        ::=   ElementName | "*"
[142]   SchemaElementTest            ::=   schema-element "(" ElementDeclaration ")"
[143]   ElementDeclaration           ::=   ElementName
[144]   AttributeName                ::=   QName
[145]   ElementName                  ::=   QName
[146]   TypeName                     ::=   Qname
[147]   URILiteral                   ::=   StringLiteral
```

A.2 XQuery Update Facility EBNF

The following EBNF grammar lists expression changed from the XQuery grammar presented in the above section. Even though no changes in the XQuery Update Facility grammar were necessary, we list the grammar here for completeness and because it is a prerequisite for the XQuery Scripting grammar presented afterwards. The EBNF is incremental from the one presented in the previous section. We only list the new or the changed expressions only. Like above, the stoppable expressions are underlined.

```
 [7]   Setter                        ::=   BoundarySpaceDecl | DefaultCollationDecl | BaseURIDecl |
                                           ConstructionDecl | OrderingModeDecl | EmptyOrderDecl |
                                           RevalidationDecl | CopyNamespacesDecl

       ...

[26]   FunctionDecl                  ::=   "declare" "simple"? "function" QName "(" ParamList? ")" ("as"
                                           SequenceType)? (EnclosedExpr | "external")
                                           | "declare" "updating" "function" QName "(" ParamList? ")"
                                           (EnclosedExpr | "external")
                                           | "declare" "sequential" "function" QName "(" ParamList? ")" ("as"
                                           SequenceType)? (Block | "external")

       ...

[32]   ExprSingle                    ::=   FLWORExpr | QuantifiedExpr | TypeswitchExpr | IfExpr | InsertExpr |
                                           DeleteExpr | RenameExpr | ReplaceExpr | TransformExpr | OrExpr

       ...

[148]  RevalidationDecl              ::=   "declare" "revalidation" ("strict" | "lax" | "skip")
[149]  InsertExprTargetChoice        ::=   (("as" ("first" | "last"))? "into") | "after" | "before"
[150]  InsertExpr                    ::=   "insert" ("node" | "nodes") SourceExpr InsertExprTargetChoice
                                           TargetExpr
[151]  DeleteExpr                    ::=   "delete" ("node" | "nodes") TargetExpr
[152]  ReplaceExpr                   ::=   "replace" ("value" "of")? "node" TargetExpr "with" ExprSingle
[153]  RenameExpr                    ::=   "rename" "node" TargetExpr "as" NewNameExpr
[154]  SourceExpr                    ::=   ExprSingle
[155]  TargetExpr                    ::=   ExprSingle
[156]  NewNameExpr                   ::=   ExprSingle
[157]  TransformExpr                 ::=   "copy" TransformBinding ("," TransformBinding)* "modify" ExprSingle
                                           "return" ExprSingle
[158]  TransformBinding              ::=   "$" VarName ":=" ExprSingle
```

A.2 Modified XQuery Scripting EBNF

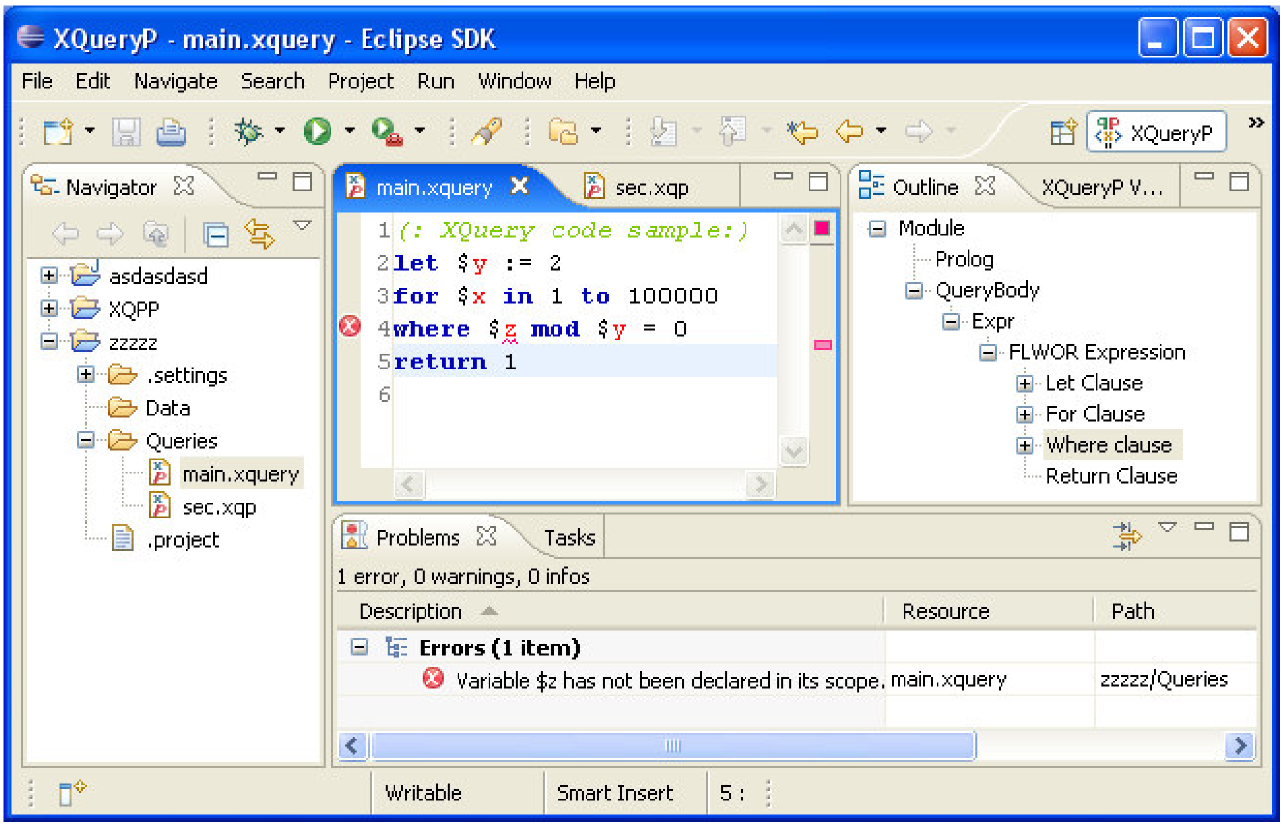
The following EBNF grammar is the modified grammar of the XQuery Scripting Extensions. Since the XQuery Update Facility is a requirement for this language most of the grammar listed in the previous section remains the same. We only list the modified expressions and the new ones that were added to this language. One more grammar modification was added, the expression being marked with bold font. The underlined expressions are the stoppable ones.

```
[24]   VarDecl                       ::=   "declare" ("variable" | "constant") "$" QName TypeDeclaration?
                                           ((":=" ExprSingle) | "external")

       ...

[26]   FunctionDecl                  ::=   "declare" "simple"? "function" QName "(" ParamList? ")" ("as"
                                           SequenceType)? (EnclosedExpr | "external")
                                           | "declare" "updating" "function" QName "(" ParamList? ")"
                                           (EnclosedExpr | "external")
                                           | "declare" "sequential" "function" QName "(" ParamList? ")" ("as"
                                           SequenceType)? (Block | "external")

       ...

[32]   ExprSingle                    ::=   FLWORExpr | QuantifiedExpr | TypeswitchExpr | IfExpr | InsertExpr |
                                                DeleteExpr | RenameExpr | ReplaceExpr | TransformExpr | OrExpr |
                                                AssignmentExpr | Block | ExitExpr | WhileExpr | ContinueExpr |
                                                BreakExpr

       ...

[159]  AssignmentExpr                     ::=   "set" "$" VarName ":=" ExprSingle
[160]  Block                              ::=   "{" BlockDecls BlockBody "}"
[161]  BlockDecls                         ::=   (BlockVarDecl ";")*
[162]  BlockVarDecl                       ::=   "declare" "$" VarName TypeDeclaration? (BlockVarBinding)? ("," "$"
                                                VarName TypeDeclaration? (BlockVarBinding)? )*
[163]  BlockVarBinding                    ::=   ":=" ExprSingle
[164]  BlockBody                          ::=   Expr (";" Expr)* (";")*
[165]  ExitExpr                           ::=   "exit" "with" ExprSingle
[166]  WhileExpr                          ::=   "while" "(" ExprSingle ")" Block
[167]  ContinueExpr                       ::=   "continue" "loop"
[168]  BreakExpr                          ::=   "break" "loop"
```

Appendix B

GUI Snapshots

B.1 XQuery Eclipse Plugin – XQuery Perspective

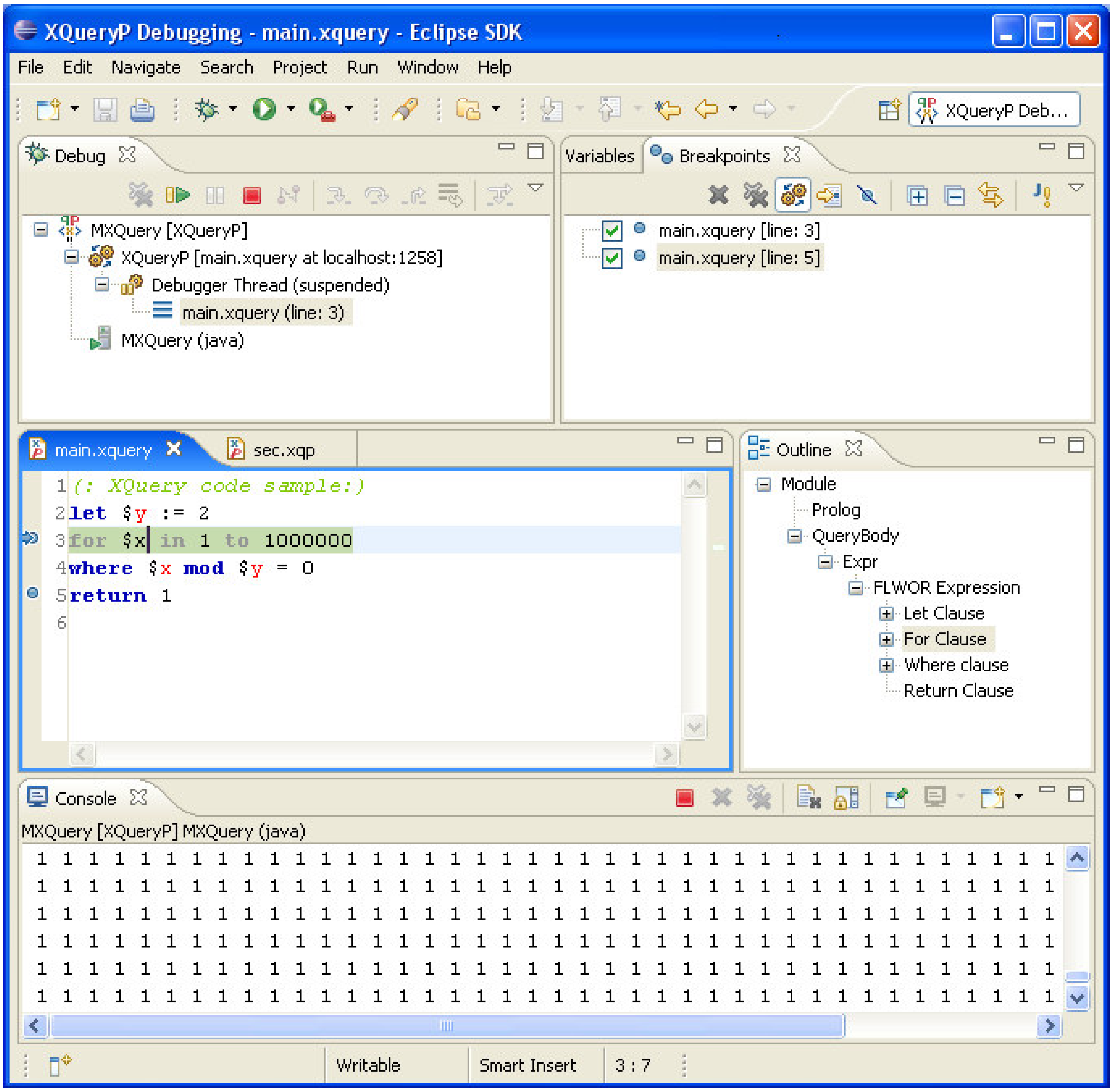


```
(: XQuery code sample:)
let $y := 2
for $x in 1 to 100000
where $z mod $y = 0
return 1
```

1 error, 0 warnings, 0 infos

Description	Resource	Path
Errors (1 item)		
Variable $z has not been declared in its scope.	main.xquery	zzzzz/Queries

B.2 XQuery Eclipse Plugin – XQuery Debugging Perspective



```
MXQuery [XQueryP]
  XQueryP [main.xquery at localhost:1258]
    Debugger Thread (suspended)
      main.xquery (line: 3)
  MXQuery (java)
```

```
main.xquery [line: 3]
main.xquery [line: 5]
```

```
(: XQuery code sample:)
let $y := 2
for $x in 1 to 1000000
where $x mod $y = 0
return 1
```

```
Module
  Prolog
  QueryBody
    Expr
      FLWOR Expression
        Let Clause
        For Clause
        Where clause
        Return Clause
```

```
MXQuery [XQueryP] MXQuery (java)
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Appendix C

XQuery Debug Protocol

C.1 XQuery Debug Protocol Specification

The following list contains the description and the payload format of each of the commands implemented by the XQuery Debug Protocol. K denotes the command set constant and C the command constant. Unless else specified, all the messages, except the engine events, have an acknowledgement reply message with error code 0 and no message data. Regarding the messages' data payload we only give a description of what the messages contain. Our data representation in the message payload is based on JSON and hence we avoided a binary data specification.

Execution CS (K = 0xF1)

- Run (C = 0x01)
  Description: starts the execution of the query. This command is sent by the debugger client after the "Started" event is received.
  Data: no payload data.
- Suspend (C = 0x02)
  Description: suspends the execution of a running query. This is an asynchronous command and expects a "Suspended" engine event to come when the state of the engine changed.
  Data: no payload data.
- Resume (C = 0x03)
  Description: resumes the execution of a suspended query. This is an asynchronous command and expects a "Resumed" engine event to come when the state of the engine changed.
  Data: no payload data.
- Terminate (C = 0x04)
  Description: terminates the execution. This is an asynchronous command and expects a "Terminated" engine event to come when the state of the engine changed.
  Data: no payload data.
- Step (C = 0x05)

Description: performs one of the stepping actions: step in, step over, or step return. This is an asynchronous command and expects a "Resumed" engine event when the engine resumed the execution. For the step to be complete, another "Suspended" or "Terminated" events expected.
Data: the step type according to the Step Types constants.

Breakpoints CS (K = 0xF2)

- Set (C = 0x01)
  Description: sets one ore more breakpoints at given query locations.
  Data: a list of query locations containing one location for each breakpoint to be set.
- Clear (C = 0x02)
  Description: clears the breakpoints at the given query locations.
  Data: a list of query locations containing one location for each breakpoint to be cleared
  .

Static CS (K = 0xF3)

- Options (C = 0x01)
  Description: retrieves from the execution engine all the non non-string – fixed length – static context options.
  Data: no payload data.
  Reply data: the information is coded into a one 2-bte word with the following structure:
  VCODBCCRRXXXXXXX
  where the X bits represent "don't care" values and:
  V is XPath 1.0 compatibility mode (0 = false, 1 = invalid value)
  C is Construction mode (0 = preserve; 1 = strip)
  O is Ordering mode (0 = ordered; 1 = unordered)
  D is Default order for empty sequences (0 = greatest; 1 = least)
  B is Boundary-space policy (0 = preserve; 1 = strip)
  CC is Copy-namespaces mode (0X = preserve; 1X = no-preserve; X0 = inherit; X1 = no-inherit)
  RR[10] is Revalidation mode (00 = strict; 01 = lax; 10 = skip; 11 – invalid value)
- Defaults (C = 0x02)
  Description: retrieves from the execution engine all the string valued static context options.
  Data: no payload data
  Reply data: one string value for each of the following static context settings:
  Default element/type namespace
  Default function namespace
  Context item static type (id, string ?)
  Default collation
  Base URI

[10] Only if the XQuery Update Facility is supported.

Statically known default collection type

- Sets (C = 0x03)
  Description: retrieves from the execution engine all the set valued static options
  Data: no payload data.
  Reply data: one set of string values for each of the following static context settings: Namespaces, Schemas, Variables, Functions, Collations, Documents, Collections.

Dynamic CS (K = 0xF4)

- Data (C = 0x01)
  Description: used to perform arbitrary data evaluations.
  Data: a list of expression to be evaluated.
  Reply data: the list of responses as XDM instances. If not all of the expressions can be evaluated, their result will be replaced with an appropriate error code and message.
- Variables (C = 0x02)
  Description: retrieves the list of in scope variables together with their values.
  Data: no payload data.
  Reply data: the list of name-value pairs for all the in scope variables.
- Focus (C = 0x03)
  Description: retrieves the expression focus component of the dynamic context.
  Data: no payload data.
  Reply data: the expression focus containing the three components: context item, context position and context size.
- Time (C = 0x04)
  Description: retrieves the execution engine's implementation specific point in time during the processing of a query.
  Data: no payload data.
  Reply data: returns the current date-time value.
- Documents (C = 0x05)
  Description: retrieves the available documents component of the dynamic context.
  Data: no payload data.
  Reply data: the list of available documents in the dynamic context.
- Collections (C = 0x06)
  Description: retrieves the available collations component of the dynamic context.
  Data: no payload data.
  Reply data: the list of available collection in the dynamic context.
- Collection (C = 0x07)
  Description: retrieves the default collection component of the dynamic context.
  Data: no payload data.
  Reply data: the default collection in the dynamic context.
- Frames (C = 0x08)
  Description: retrieves the list of debug nodes in the debug expression tree up to the root. This created frames for both function calls and nested expressions.
  Data: no payload data.
  Reply data: the list of query locations identifying the frames.

Dynamic CS (K = 0xF8)

- Started (C = 0x01)
  Description: sent by the engine after it has initialized its state for the debug session – opening the communication ports, setting the query to debug and setting the breakpoints.
  Data: no payload data.
- Terminated (C = 0x02)
  Description: sent by the execution engine asynchronously as a response to the "Terminate" command, or when the execution finishes, or when an error is encountered during execution and the engine had to terminate.
  Data: the cause of the termination: client request, finished execution or error. In case of error, the error message is also sent.
- Suspended (C = 0x03)
  Description: sent by the engine asynchronously as a response to the suspend command or when the engine encountered a breakpoint or when a stepping reached its target debug node.
  Data: the current frame where the execution was suspended.
- Resumed (C = 0x04)
  Description: sent by the engine asynchronously as a response to either a resume or step command.
  Data: the cause of resuming: client request or one of the step types.

Error codes:

- NO_ERROR: the default error code for successful messages.
- UNKNOWN: sent in a reply message when the execution engine encountered an error not falling in any of the other categories.
- INVALID_MESSAGE_FORMAT: when the format of the message sent to the execution engine has an invalid format.
- INVALID_COMMAND: when the message contains an invalid command set-command identifier
- COMMAND_NOT_IMPLEMNTED: when a command in the specification is requested but it is not implemented
- ENGINE_TERMINATED: when a command is received after the execution has been terminated.

Step Types:

- Step INTO
- Step OVER
- Step RETURN

Bibliography

[1] Scott Boag, Don Chamberlin, Mary Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. ***XQuery 1.0: An XML Query language***. Technical report, W3C, 2008. http://www.w3.org/TR/xquery/

[2] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau. ***Extensible Markup Language (XML) 1.0 (Fourth Edition)***. W3C Recommendation 16 August 2006. http://www.w3.org/TR/xml/

[3] Don Chamberlin, Daniela Florescu, Jim Melton, Jonathan Robie, Jérôme Siméon, ***XQuery Update Facility 1.0***, W3C Candidate Recommendation, 14 March 2008. http://www.w3.org/TR/xquery-update-10/

[4] Don Chamberlin, Daniel Engovatov, Daniela Florescu, Giorgio Ghelli, Jim Melton, Jerome Simeon. ***XQuery Scripting Extension 1.0***, W3C Working Draft 28 March 2008. http://www.w3.org/TR/xquery-sx-10/

[5] The FLWOR Foundation, Daniela Florescu, Donald Kossmann. ***Zorba XQuery Execution Engine***. http://flowrfound.ethz.ch/FLWOR1/zorba.shtml

[6] Donald Kossmann, Peter Fischer, Irina Botan, Rokas Tamosevicius, Matthias Braun, Tim Kraska, and David Graf. ***MXQuery project web page***, 2008. http://www.mxquery.org/

[7] Eclipse Foundation. ***Eclipse - an open development platform***. http://www.eclipse.org/

[8] James Clark, Steve DeRose. ***XML Path Language (XPath) Version 1.0***, W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath

[9] Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, Norman Walsh. ***XQuery 1.0 and XPath 2.0 Data Model (XDM)***, W3C Recommendation 23 January 2007. http://www.w3.org/TR/xpath-datamodel/

[10] Michael Kay. ***XSL Transformations (XSLT) Version 2.0***, W3C Recommendation 23 January 2007. http://www.w3.org/TR/xslt20/

[11] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, Jérôme Siméon. ***XML Path Language (XPath) 2.0***, W3C Recommendation 23 January 2007. http://www.w3.org/TR/xpath20/

[12] Guy LaPalme and Mario Latendresse. ***A debugging environment for lazy functional languages***. Lisp and Symbolic Computation, vol. 5, pag. 271-287, 1992.

[13] Bernard Pope, Lee Naish. ***Practical aspects of Declarative Debugging in Haskell 98***, 5th ACM SIGPLAN international conference on Principles and practice of declarative programming, Uppsala, Sweden, 2003.

[14] Robert Ennals, Simon Peyton Jones. ***HsDebug : Debugging Lazy Programs by Not Being Lazy***, The Haskell Workshop'03, ACM, Uppsala, Sweden, April 2003.

[15] Olaf Chitil, Colin Runciman, Malcolm Wallace. ***Tracing and Debugging of Lazy Functional Programs. A Comparative Evaluation of Three Systems***. Proceedings of the 12th International Workshop on Implementation of Functional Languages, Aachen, 2000.

[16] W3C. ***Specifications and Working Group Notes***. http://www.w3.org/XML/Query/#specs

[17] Henrik Nilsson. ***How to look busy while being as lazy as ever: the Implementation of a lazy functional debugger***. Journal of Functional Programming, vol. 11, issue 6, November 2001.

[18] R. Morgan and S. Jarvis. ***Profiling large-scale lazy functional programs***. Journal of Functional Programming, vol. 8, issue 3, May 1998.

[19] Torsten Grust, Jan Rittinger, Jens Teubner. ***Data-Intensive XQuery Debugging with Instant Reply***. 4th International Workshop on XQuery Implementation, Experiences and Perspectives, Beijing, China, June 2007.

[20] Richard Watson and Eric Salzman. ***Tracing the Evaluation of Lazy Functional Languages: a Model and its Implementation***. Asian Computing Science Conference, pag. 336-350, 1997.

[21] Robert Ennals and Simon Peyton Jones. ***Optimistic Evaluation: An Adaptive Evaluation Strategy for Non-Strict Programs***. 8th ACM SIGPLAN International Conference on Functional Programming, Uppsala, Sweden, August, 2003.

[22] S. Peyton Jones, R. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P.Wadler. ***Report on the programming language Haskell 98***, February 1999. http:/haskell.org

[23] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, Philip Wadler. ***XQuery 1.0 and XPath 2.0 Formal Semantics***, W3C Recommendation 23 January 2007. http://www.w3.org/TR/xquery-semantics

[24] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, Chun Zhang. ***Storing and querying ordered XML using a relational database system***. Proceedings of the 2002 ACM SIGMOD international conference on Management of data, 2002.

[25] ***Java Debug Wire Protocol***. http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdwp-spec.html

[26] **JavaScript Object Notation**. Subset of JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. http://www.json.org

[27] ***Eclipse Platform Plugin Development Guide***. http://help.eclipse.org

[28] Michael Kay. ***The Saxon XSLT and XQuery Processor***. http://www.saxonica.org

[29] Simon P. Booth, Simon B. Jones. ***Walk Backwards to Happiness | Debugging by Time Travel***. In Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG'97), Linkoping, Sweden, May 1997.

[30] Goetz Graefe. ***Query Evaluation Techniques for Large Databases***. ACM Computing Surveys, 25(2):73–170, 1993.