EÖTVÖS LORÁND University
Faculty of Informatics
Department of Programming Languages and Compiler

Interactive Query Language for Code Comprehension

Supervisor:
Zoltán Porkoláb
Associate Professor

Author:
Patrik Vas
Computer Science MSc

Budapest, 2019

Contents

Chapter 1: Introduction

Understanding large code bases is a significant advantage in many computer science and engineering fields. Some of the advantages are the following: better integration, maintainability, re-usability, development cost reduction and faster development. For these reasons we have many code extraction tools for our disposal to do code analysis. By parsing the code these tools produce a significant quantity of data, sometimes even more than the original source code.

The size of the data makes efficient interpretation and navigation of the information challenging. The size of the data is not the only concern, as variety of data is also something that we need to handle.

Code comprehension tools usually provide a parser and some kind of user interface. The user interface may offer a diverse list of functionalities like: diagrams, searching for usages and declaration of certain components, code browsing. But even a well-design graphical interface can't fully utilize the stored data. Finding something may require tedious manual work. The information retrieval interface may be too rigid to express certain queries. The information might be there but accessing it may prove to be a complex task. Not having easy data access hinders further development of tools that could potentially add more features and functionality to the code comprehension system. What we really need is an expressive and efficient way to do code search utilizing the parsed data.

Code search is vital component of software analysis and development. Applications of code search can be found but not limited to software architecture analysis, reverse engineering, consistency checking, coding conventions enforcing, documentation and library/programming language design. The search mechanisms have evolved from simple "grep"-like tools to tools that can understand the syntax as well. One of the most expressive ways to perform code search is with a code query language.

In this thesis we going to analyze and discus how a code query language should look and what kind of features should have. We will be going to examine how developers search code and what other code query languages are out there. We will to try to answer questions like: What kind of features can we add to the language? What features do we need? How are

we planning to use it? What programming languages should we support? How should the language grammar look? How can we show the results?

After answering the above questions, we going to present our own implementation of a code query language named CodeQL We going to discuss what options we have when creating a tool like this. We will present the different parts of the architecture as well.

The query tool will leverage and enhance the functionality of the CodeCompass code comprehension tool.

Chapter 2: Program comprehension

Most software development related activity involve code understanding in one way or another. A significant portion of time required to maintain, debug and reuse existing code is spent on understanding code. When we want to add or modify a feature, we also need to understand the software system. Code understanding can also be useful when managers are doing cost or time estimation of a new task.

The users will use our code query language when they are trying to understand an existing system. For an effective code query language design, we need to analyze how the human users understand code. But this task proves to be difficult as code understanding is highly subjective and use case dependent. The background, experience and domain specific knowledge of an individual may highly affect how he approaches the code understanding.

Frederick Brooks[1] observed that software engineering is made difficult to understand by the following 4 properties: complexity, conformity changeability and invisibility.

- Complexity: Software is remarkably complex. It has a diverse set of artifacts that interact with each other in an unpredictable way. The software components are unique and have a large amount of states. These complex interactions my hide unexpected behaviors hard to predict. Also, systems size grows over time becoming even more incomprehensible. Problems caused by complexity: software bugs, hard to use software, communication issues in developer teams, difficult project estimations.
- Conformity: Software systems need to communicate with each other. To communicate they need to conform to each other's interfaces. These interfaces are created by different programmers/organizations, each one has their own ideas and design. Programmers need to invest extra energy to make components like hardware, libraries, operation system, applications work together.

- Changeability: Used software is frequently changed. Reason for change may be continuously evolving harder or changing requirements. Software is relatively cheap to modify, so changes will happen all the time. The difficulty here is added by the fact that changes can make the knowledge about state of software outdated.
- Invisibility: Software is hard to visualize for humans. It doesn't have a clear geometric structure. Certain parts like class, name-space, component relationships can be modeled with graphs, flow chart diagrams, but still it requires work to create those and it may be less descriptive than the code itself.

2.1 Cognitive models of program comprehension

During code exploration the programmer builds a mental comprehension model of the software system. As this mental model becomes more exact and inclusive the programmers understanding of the code becomes better. Multiple psychological studies were performed to understand this process, resulting in development of multiple theoretical mental models worth exploring. A mental model is the software program representation inside the programmer's mind.

2.1.1 Bottom-up program comprehension

With bottom-up model the programmer starts with lower level source code information and then they create more abstract concepts step by step. They interpret the code line by line and chunk together control flow elements like if-else, loops, switches. A level is understood when the programmer aggregates the individual groups to see the bigger picture. After that the programmer moves to the next level and uses the previously comprehended structures as low level input. This can be done till a full understanding of the program is achieved

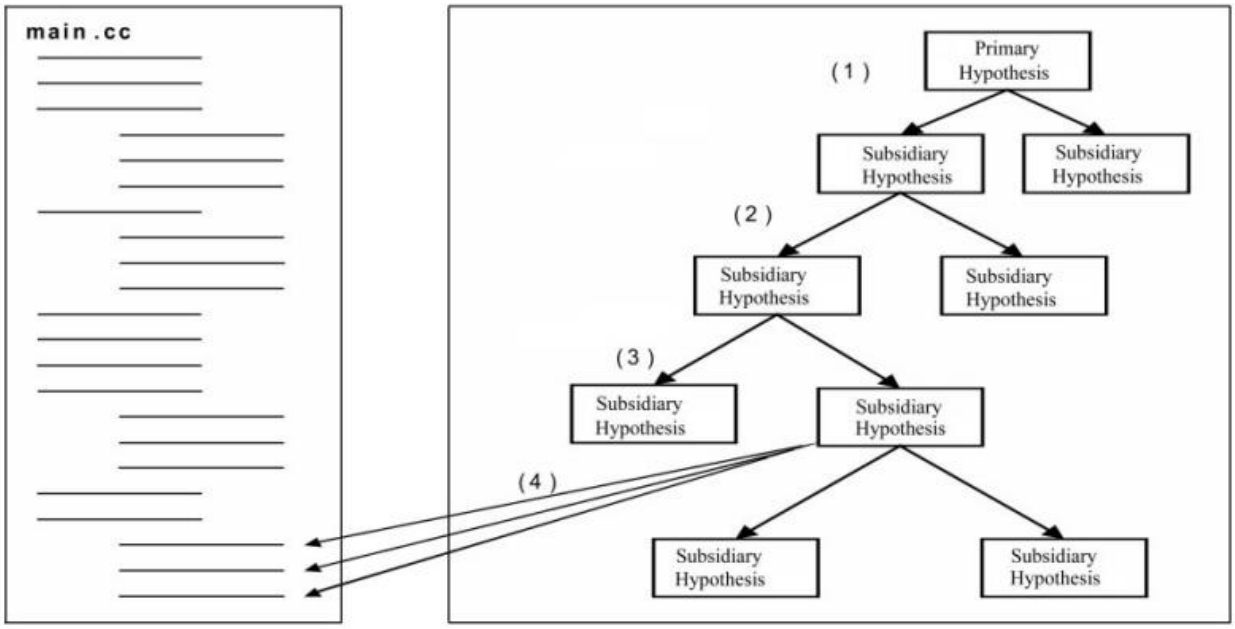
This approach is chosen when the programmer's application domain knowledge is not comprehensive enough. By application domain we mean the context of the problem that the program is trying to solve. Usually the programmer is trying to answer questions like why and what. This is how programmers approach an unfamiliar code base.

2.1.2 Top-down program comprehension

For this model to work the programmer needs to understand the application domain well. With this approach the programmer has some kind higher level functionality in mind. We call this a hypothesis. Then on base of a set of code features he tries to confirm or reject this hypothesis. With each iteration the programmer fine-tunes the hypothesis. The process can be described as a creation of a mapping between application knowledge and code. For this to be usable the programmer needs a higher level of application domain knowledge.

This is the approach that we usually use when we are doing debugging. We know what the expected output should be and we try to find what code that is responsible for generating the output. We start with a higher-level entry point like the program starting point or the implementation of a faulty feature if we know where it is. Through the process we go deeper and deeper till we reach to source of the bug.

The application domain knowledge incorporate knowledge of requirements, system architecture, programming best practices, algorithm and data structure used in program.



Brook's program comprehension model [2]

2.1.3 Knowledge-base understanding model

Letovsky[3] views the programmer as an opportunistic processor. It is able to change its strategy base on the goal, knowledge or situation. Most programmers will use a strategy that is something between top-down and bottom-up.

Letovsky's model has 3 components:

1. Knowledge-base: the programmer's application and programming knowledge. This base can be enriched by reading code, documentation and analyzing code metrics and diagrams.
2. Mental model: this represents the current understanding. Initially this model contains only the specification of the goals, but as the programmer understands the code this model will contain the implementation details as well.
3. Assimilation process: defines how the mental model evolves using the knowledge base. This can be a bottom up or a top down.

2.1.4 Systematic and as-needed program understanding strategies

Littman[4] observed that programmers read code in systematic way in order to gain some global knowledge of the system; or they focused only on code parts that were needed to accomplish a certain goal.

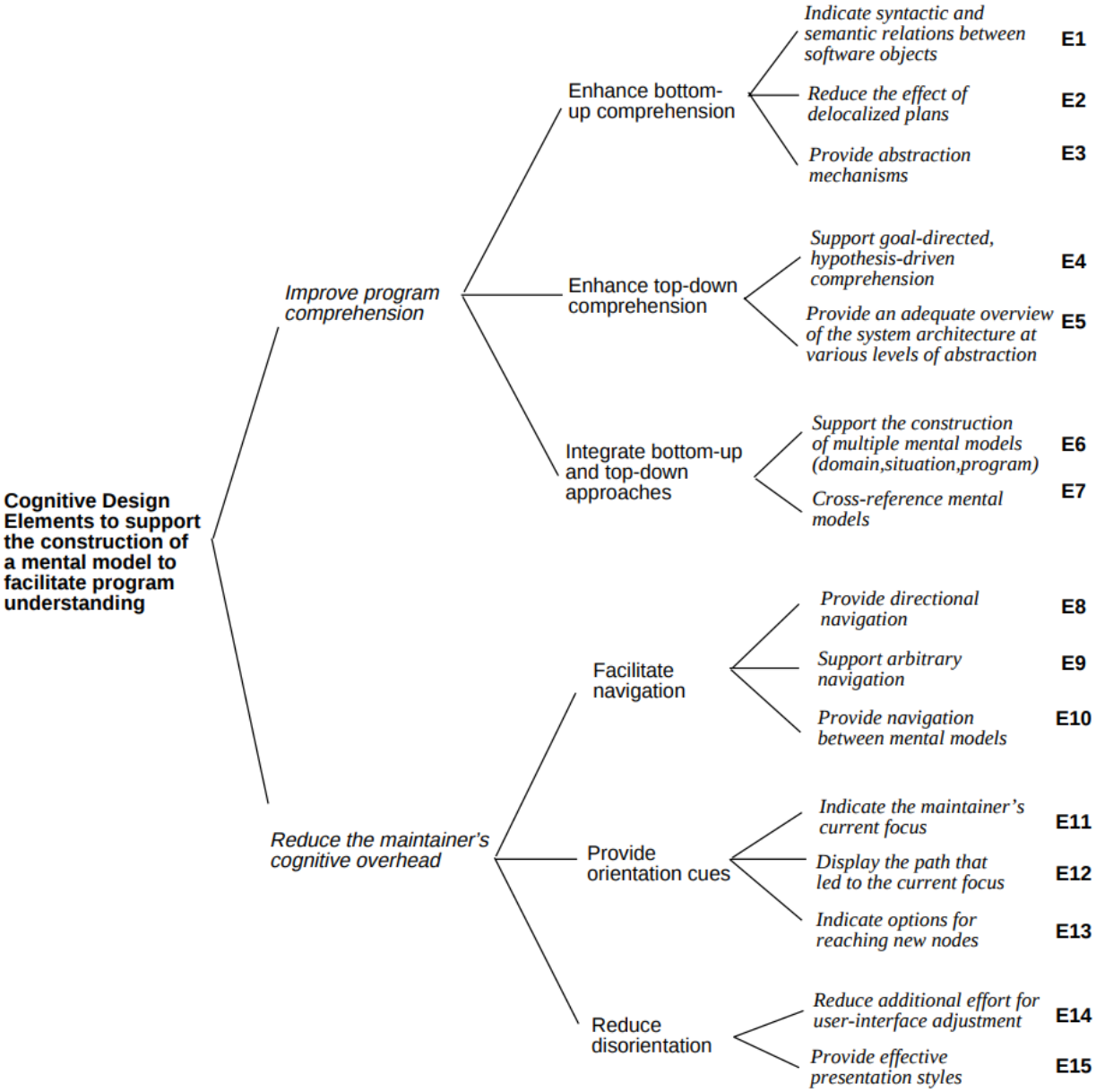
Soloway[5] merges the above two approaches into a single model:

- Micro-strategies: this includes exploration phases that consist of read, question, conjecture and search cycles. Programmers need these phases because the code mostly is located in non-contiguous parts of the program.
- Macro-strategies: the goal of this strategy is to gain knowledge on a global level.
    - Systematic macro-strategies: The programmer reads all the code and the documentation and this way he traces the entire program flow. This strategy results in more correct work. But in the real world it's impractical for a single programmer to read the entire code-base of a large project that may contain millions of lines of code.

- As-needed macro-strategies: Programmers investigate only parts of the code that they assume is related to the task they are working on.

2.2 Cognitive design elements

There are numerous design decisions that we need to consider when we are creating a software exploration tool. Some decisions help the support of bottom up compression while others help top down. [6] article describes 15 cognitive design elements to guide the development of software exploration and comprehension systems. We going to take a closer look at each point and decide if it's relevant for a search tool or not.



Cognitive design elements for Software Exploration [6]

E1: Indicate syntactic and semantic relations between software objects

This point expresses that the relevant source code should be easily accessible from the software exploration tool. For example, if our tool is displaying a class inheritance diagram, we should have the option of opening the source code file by clicking on the class representation figure.

In case of our search tool the query should map to a list of file, line number pair.

E2: Reduce the effect of delocalized plans

The syntactic elements my not be grouped together in the source code, but rather scattered between multiple lines, files or even repositories. For a tool to be useful it needs to include all the pieces and show the connection between them.

E3: Provide Abstraction Mechanisms

The amount of details in the code may be overwhelming for the user. The software exploration tool could provide a mechanism to aggregate the lower level software elements.

We can provide some abstraction on the result display section. We going to discuss about abstraction on the code query language level in later sections.

E4: Support goal-directed, hypothesis-driven comprehension

The hypothesis consists of finding certain software patterns, like finding a function call. Not many graphical software comprehension tools support top down exploration approach. Here our code query search tool could really have an advantage.

If we find a way to express the hypothesis in our query, we could easily support this comprehension method.

E5: Provide overviews of the system architecture at various levels of abstraction

For top down exploration being able to view the system architecture is a great advantage. But this is not the goal of a search tool. A better suitable approach would be to use the search tool with a diagram generator tool. After consulting the diagrams, the user could use our search tool to look up specific elements in the source code.

E6: Support the construction of multiple mental models

A code search tool could be used with both bottom up or top down models. Searching is essential for both directions. For example, when using bottom up model, after aggregating lower level code elements to a new higher-level element it would be useful to search for the occurrences of the new high level element, especially when using as-needed approach. When using top down model, the programmer could search for the hypothesis itself.

E7: Cross-reference mental models

As mentioned previously programmers often switch between mental models and this may cause the loss of previous data when the switching is happening, This is not a problem in case of search, as the search results should be same in both cases.

E8: Provide directional navigation

Directional navigation could be achieved with multiple search queries. But the task would be much easier if a code browser tool were used. Clicking on graphical elements to jump to the source would be preferred rather than typing a new query each time.

E9: Support arbitrary navigation

Arbitrary navigation means that the user can navigate to a location that is not directly reachable by following an application or user define link. Our search tool main purpose is to support this kind of navigation.

E10: Provide navigation between mental models

A possibility to navigate between mental models would be to navigate between search results. But the goal of our code search tool is not to present the entire mental model but to help build it. It’s up to the programmer to remember the different models and navigate between them.

We could create some visualizations from the search results to help the users navigate the mental model, but this topic is out of the thesis scope.

E11: Indicate the maintainer's current focus

The query itself reflects the maintainer's focus. For example, the maintainer's focus could be functions, classes, definitions, declaration etc. all these are revealed by the query itself.

E12: Show the path that led to the current focus

This point is represented by supporting search query history feature.

E13: Indicate the options for reaching new nodes

Query recommendation could be a useful feature (like google auto-complete), but we won't cover it in this thesis.

E14: Reduce additional effort for user-interface adjustment

The user will interact with the search tool using a query language, not a graphical user-interface. Here we should pay attention to make the query language as user friendly as possible. Search intentions should be easy to express.

E15: Provide effective presentation styles

Search tool results are relatively simple. We can represent results with a list of pairs consist of a file and line number. On top of the raw search result data users can build more custom visualization systems.

Chapter 3: How developers search code

We would also like to explore the code search problem from a practical viewpoint. For this we take a look at a case study performed at Google[7], using a combination of survey and log-analysis methodologies.

3.1 Why do developers search?

Usually programmers search code to answer the following questions about the project: How? What? Where? Why? Who and when?

The main reason of developers searching code is they are looking for code examples. In some cases, they try to identify the correct tools for certain task. In other cases they know the api class and method names but they still need more information about parameters and code context. In software engineering there is a common pattern to start with an example and modifying it to solve a new problem.

Another reason developers search code is to read it. They may be curious about implementation details. The software element names can be complicated, and developers may only remember parts of the it. To find these elements they resort to code search. They also read code to check code standards and conventions of the repository.

Code can be delocalized, and some software elements can be spread through multiple files or even repositories. Even if they are in a single file, programmers have a hard time remembering all the locations. For this reason, they use code search to locate code.

Developers also search code to determine the impact of a change. When they get some unexpected behavior, they search the code to answer why does the code act certain way. They are trying to discover the dependencies between components and what are the hidden side effects of changing an element.

Code search also helps discovering meta data of the code. This meta data can contain the author of certain code parts, time when the code was added, change history.

3.2 In what contexts is search used?

According to the survey most searches are performed when working on code changes. Developers also use code search when triaging a problem and doing code reviews.

Programmers mostly search familiar code, this means that code search is used as navigation mechanisms. When doing the search their main goal is to understand how code work, a secondary goal is to learn how to use it. Another scope of search is to find out how code changed.

3.3 What are the properties of the search queries?

The average query contained only 1.8 keyword, which is pretty low. This is thanks to the improvement of search engines. The examined search queries were indicating that the developers were building more complex search queries iteratively. Also, they were doing this fast, meaning they were reformulation queries every 8 seconds.

Chapter 4: Code Query Languages

Over the past years many code query languages were developed each with their own advantages and disadvantages. As code search is such a necessity in software engineering many studies were conducted over comparing these query languages[8].

Code query follows the extract-abstract-present paradigm:

- Extract: It processes the source code and creates a mapping between the code and some intermediate structure that is easier to traverse. This structure usually is a graph.
- Abstract: Queries and operations are executed on the intermediate state to produce results.
- Present: Display results in an user friendly way

4.1 Database Query languages

Before diving into code query languages it's worth exploring query languages in general. Most query language research was done in the database querying context. We can leverage this research as the code querying is a subset of database querying.

Looking at the theory behind the query languages we observe that they usually are declarative. Declarative means the user specifies the proprieties of the searched data and it leaves the system to compute the results. In this manner user can construct queries easier and more natural.

Prolog is highly expressive language as it is Turing complete. Every programming language can be expressed with Prolog. But this expressibility has its own downsides. Bounds are hard to set for this language and for larger datasets the performance can suffer. Also some properties like termination are theoretical undecidable.

Most database query languages are based on Tarski's relational algebra (RA) or Codd's relational calculus (RC). Both are based on first order logic which is less expressive, but more manageable form of logic. Many database query languages use some combination of these logics, one of the most known such language is SQL.

Datalog represent a syntactic subset of Prolog, which has bounds and can be evaluated in polynomial time, but it is sacrificing Turing completeness. It had the advantage over SQL that it could express recursive queries. However, the SQL:1999 standard included recursive queries as well.

4.2 Code Query Language proprieties

The [8] paper provides a comparison of query languages based on a set of properties. These properties are useful to understand what other languages can provide.

Paradigm

This defines the paradigm of the code query language. Paradigms represent a modality to classify languages based on their characteristics. Like programming languages query languages can be classified into multiple paradigms.

Some example of code query language paradigms is declarative, imperative, object oriented, relational, SQL-like, first-order logic.

Types

A query language can support several data types, just like usual programming languages. Some example types are integer, bool, string.

Parametrization

This property means that the functionality of the query can be influenced with parameters. Parametrization can fine-tune the query languages to certain use cases.

Polymorphism

This property defines if a query supports polymorphic types. Polymorphism means that it provides a single interface to support multiple types. The two type of polymorphism that we going to discuss is subtype polymorphism and parametric polymorphism.

Subtype polymorphism means that there is a sub-typing relationship is defined between a super-type and sub-type. This relationship means that the sub-type term can be safely used in any context where a super-type term is expected. In C++ and Java this type of polymorphism is accomplish using inheritance.

A query language supports parametric polymorphism if queries can be written in a generic way so that it's not dependent on the data type. The basis of this paradigm is generic programming and for example in C++ this can be represented by use of templates.

Modularity

This property specifies how much can queries be separated and reconnected. If the query language is modular than we can separate queries into implementation and interface. A goal of modularity is to hide the implementation details of queries from the users.

Libraries

Modules can be grouped together into libraries. Libraries are a collection of similar queries. Users can create, share and use libraries. This further improves software re-usability.

4.3 Tool proprieties

Output formats

Specifies how is the result encoded. Examples of output formats are : XML, JSON, plain text, charts.

Interactive interface

Include the possible ways to interact with the query technology. Query technologies can provide command line interface, graphical user interface or development environment plug-in.

API support

This property specifies how other programs can interact with the query technologies. With a well-defined API, programmers can write their own applications that use or extend the functionality of the query technology.

Interchange format

This propriety describes what kind of file formats can the query technology understand and if they can exchange this data with other query technologies. Each query technology can extract information from certain types of programming languages. Then it saves the extracted data as another type format.

If query technologies can understand each other's extracted data formats they can interchange data and make use of each other's functionality. For example, if A language can parse Java, B language can parse C++ and both query languages save their parsing data as same format, then we can use A language for C++ and B language for Java. Even though they may be optimized for their original language.

Extraction support

Before using queries, projects need to be parsed for information extraction. Extraction support propriety determines what kind of programming language can be parsed by the query technology.

4.4 Code Query language examples

In this section we going to look at few code query technologies. We going to present what they have to offer and what are their advantages and disadvantages.

4.4.1.1 Implementation examples

The comparative study [8] uses package instability [9] implementation as query example for the languages. Package instability is a good example to demonstrate what the query language can achieve. It also gives a flavor of the query languages style.

The package instability metrics are a set of indicators that can help software engineers measure the quality of a Object Oriented design. Whit these metrics we can measure how maintainable, reusable and robust our code is. As many industry leading technologies are OO, these metrics are a good representation of what kind of queries programmers write during their work.

The metrics that we going to use are the following:

- Afferent Couplings (CA): The number of classes outside the declarative region (package/namespace) that depend upon classes within the declarative region.
- Efferent Coupling (CE): The number of classes inside the declarative region that depend upon classes outside the declarative region.
- Instability (I = Ce ÷ (Ca+Ce)) : This metric has a value in [0,1] interval. I=0 indicates a maximally stable declarative region. I=1 indicates a maximally instable declarative. Region.

Grok

The initial version was created in 1995 by Ric Holt at the University of Toronto, Canada. It was written in the Turing Programming Language which was also developed by Holt. Grok was re-implemented in Java under JGrok name. The two languages share the same principles but while JGrok is slower it compensates with a richer set of features. JGrok is no longer officially maintained.

This language is based on relational calculus. Grok is not Turing complete language so it cannot express every programming language. But what it can express it does it in a very concise way. This conciseness is achieved by using a low number of operators and symbols. The downside of this approach is that it can be difficult to understand as is not very communicative language.

Types are not supported by the language; this design decision makes faulty query debugging more difficult. Neither parametrization nor polymorphism are supported by Grok. Also there is no API available and the output cannot be customized.

For interchange format it supports RSF (Rigid File Format). It uses SWAG kit's C++ extractor to parse C++.

***Package instability metric implementation in Grok** [Appendix 1]*

```
1  PackageDep := PackageOf o ClassDep o ( inv PackageOf )
2  PackgDepInterPackg := PackageDep - ( id dom PackageOf )
3  ClassFowardRel := ( inv PackageOf )
4                    o PackgDepInterPackgo PackageOf
5
6  ClassDepInterPackg := ClassForwardRel ^ ClassDep
7  AffCoupling := PackageOf o ( inv ClassDepInterPackg )
8
9  AfferentCoupling := ( dom AffCoupling ) outdegree AffCoupling
```

Rscript

Rscript[11] was developed in 2002 by Paul Klint at the Centrum voor Wiskunde & Informatica in Amsterdam, Netherlands. For implementation technology it uses ASF+SDF.

The language's goal is to provide a tool for exploring the design space of software analysis. It is trying to achieve this using relational calculus. It does support static type checking. Beside the common int, string, bool types it also supports special types designed to helps code analysis. Parametrized polymorphism is supported through user defined functions that can take polymorphic types as parameters.

It doesn't provide any extraction support nor API. The extraction can be solved using another language's extractor like Grok. The extracted data than can be converted to a format understood by Rscript.

User can interact with Rscipt using CLI and GUI as well.

Package instability metric implementation in Rscript [Appendix 2]

```
1  rel [ str , str ] PackageOf
2  rel [ str , str ] ClassOf
3  rel [ str , str ] MethodCall
4  rel [& T1 , int ] outdegree ( rel [& T1 ,& T2 ] R )
5        = { <D , # R [ D ] > | <& T1 D , & T2 U > : R }
6
```

```
7  rel [str,str] ClassDepInterPackg
8        = { <C1 , C2 > | < str C1 , str C2 > : ClassDep
9        , PackageOf [ - , C1 ] != PackageOf [ - , C2 ] }
10
11 rel [ str , str ] AffCoupling
12       = PackageOf o inv ( ClassDepInterPackg )
13
14 rel [ str , int ] AfferentCoupling = outdegree ( AffCoupling )
1  rel [ str , int ] PackageInstability
2        = { <P1 , (100* N1 )/( N1 + N2 ) >
3        | < str P1 , int N1 > : EfferentCoupling
4        , < str P2 , int N2 > : AfferentCoupling
5        , P1 == P2 }
```

SemmleCode

SemmleCode is a query technology developed in 2006 by a company named Semmle Ltd. The project was guided by Oege de Moor, the implementation was written in Java. The main use of the tool is to do code analysis.

The language is based on relation algebra and also has a strong object-oriented influence. The object-oriented style makes it suitable for programmers as they are familiarized with OO programming languages. SemmleCode heavy borrows from Java and SQL As fact extraction is concerned it can parse Java and XML.

It is a statically typed language; the typing system also provide subtyping polymorphism. It supports a diverse set of output formats like text, charts, maps and graphs. The user can interact with the tool using command line interface or using it as an Eclipse IDE plugin.

Package instability SemmleCode implementation **[Appendix 3]**

```
1  predicate classDepInterPackg(Class c1, Class c2) {
2        c1.getPackage() != c2.getPackage () and
3        classDep (c1,c2)
4  }
5  class MyPackage extends Package {
```

```
6       MyPackage (){this.fromSource ()
7  }
8
9  predicate affCoupling(Class c) {
10      exists(Class c1 | this . contains(c1) and
11              classDepInterPackg(c1, c))
12 }
13
14 int afferentC o u p l i n g () {
15      result = count(Class c | this.affCoupling(c))
16 }
17 float packageInstability() {
18      result = (1.0 * this.efferentCoupling()) /
19                  (this.afferentCoupling() +
                    this.afferentCoupling())
20      }
21 }
```

JgraLab

This query technology was initially developed in 2006 as the diploma thesis of Steffen Kahle. Since than it had several maintainers and contributors. As the name suggests it is implemented in Java.

The JgraLab language is based on relational algebra and graph theory. Nevertheless, it has an advantage when handling graph-based data. It has incorporated an extensive graph algorithm library. It supports fact extraction from Java and C programming languages. It has types like int, string boolean, real but also graph data types like Edge and Node. It includes parametric polymorphism support for its function parameters. The language syntax looks like inverted SQL.

The modularity is not well defined, and it also lacks in library support. But it has an API. Users can interact with this query technology using its command line interface. The query result output is displayed as plain/text or HTML For interchange format it supports its own data encoding called TGraph.

The uniqueness of this technology is that it transforms the source code into a graph, and then the user can use a set of graph algorithm to perform code analysis. The language is also interesting as it combines relation algebra and SQL with graph theory.

Package instability JGraLab implementation **[Appendix 4]**

```
1  from p : V{JavaPackage}
2  reportMap p ,
3        from outerClass : V{JavaClass}
4        with
5               (not p -->{PackageOf} outerClass) and
6               (p -->{PackageOf} <--{ClassDep} outerClass)
7        report outerClass end
8  end store as AffCoupling
9
10 using AffCoupling :
11 from p : V{JavaPackage}
12 reportMap p, count (get (AffCoupling, p)) end
13 store as AfferentCoupling
14
15 using AfferentCoupling, EfferentCoupling:
16 from p : V{JavaPackage}
17 reportMap p, get (EfferentCoupling, p) /
18             ( get (EfferentCoupling, p) +
19              get (AfferentCoupling, p))
20 end store as PackageInstability
```

CrocoPat

CrocoPat was developed in 2002 by Dirk Beyer at university of Cottbus, Germany. The implementation is written in C.

The interesting part of this technology is the attempt to combine predictive logic with imperative code elements. The result is a concise and simple language that is easy to use but in this process, it sacrifices some expressivity. It can manipulate relational data, including graphs as edges can be expressed as binary relationships.

The language has no polymorphism nor library support. The user can interact with CrocoPat through the command line interface. Output formats include text and RSF.

This query technology doesn't provide any language extractor, but as it uses RSF as interchange format the source code parsing can be done by other third-party software, like for example Grok.

Package instability CrocoPat implementation **[Appendix 5]**

```
21  ClassDepInterPackg(c1, c2)
22        := EX(p1, PackageOf (p1, c1)
23              & EX(p2, PackageOf(p2, c2) & !=(p1, p2)
24                    & ClassDep(c1, c2)));
25
26  AffCoupling (p, c) := EX(c1, PackageOf(p, c1) &
27              ClassDepInterPackg(c,c1));
28
29 Package(x) := PackageOf (x, _);
30 FOR p IN Package(x) {
31        ca := #(AffCoupling (p, c));
32        PRINT " AfferentCoupling ", p, " ", ca, ENDL;
33        ce := #(EffCoupling (p, c));
34        PRINT "EfferentCoupling ", p, " ", ce, ENDL;
35        i := ce / (ca + ce);
36        PRINT " Instability ", p, " ", i, ENDL;
37 }
```

JTransformer

The initial tool was developed in 2002 by Tobias Rho and Uwe Bardey under the guidance of Günter Kniesel at University of Bonn, Germany.

The paradigm of the language is based on first order predicate logic. It was implemented using SWI-Prolog. The language is compact and concise, but sometimes can be hard to write thanks to parameter ordering This query technology was mostly used for software analysis in academia.

The only interchange format supported is Prolog fact file. The users can interact with JTansformer as an Eclipse plugin. It also can extract facts from Java language. The only supported output format is text, but it provides multiple APIs.

Package instability JTransformer implementation **[Appendix 6]**

```
38 classDepInterPackg(C1, C2): -
39       packageOf (P1, C1), packageOf (P2, C2),
40       not (P1 = P2), classDep (C1, C2).
41
42 affCoupling (P, C): -
43       packageOf (P, C1), classDepInterPackg(C, C1).
44
45 afferentCoupling(P, N): -
46       setof (C, affCoupling (P, C), AffClasses),
47       length (AffClasses, N).
48
49 packageInstability(P, I): -
50       efferentCoupling(P, Ec), afferentCoupling(P, Ac),
51       I is Ec /(Ec + Ac).
```

CQLinq

CQLinq[12] (Code Query Linq) is part of Ndepend and CPPDepend static analysis tools . This tool provides Linq query facility for code using the CodeModel namespace interface types as data sources.

Linq[13] means Language Integrated Query and it adds native querying possibility to the .NET framework. It does work with a variety data structures like XML, enumerables types, databases and many more data sources. The language syntax looks similar to SemmleCode, but in this case it combines C# and SQL. CQLinq, however is not only for .NET as it also has raw C++ project support.

With the CodeModel namespace Linq's capabilities were extended to code as well. This namespace contains interfaces like IMethod, IType, IField and more classes that help describing code. With this tool users can query code, calculate code metrics and create notification alert events for code changes. CQLinq is a strongly typed language with support of polymorphism.

CQLinq is part of the Depend Visual Studio Extension. NDepend can extract facts from .NET projects while CPPDepend can parse C++ projects. The output of the queries can be view in Visual Studio and also exported as HTML. The Depend tool also provides an API.

This is a proprietary software, but it provides a free trial.

The first query returns the methods with more than 30 lines of code.

CQLinq query1 **[Appendix 7]**

```
from m in Methods where m.NbLinesOfCode > 30 select m
```

The second query returns the methods that are declared in source files named '*.designer.cs'

CQLinq **query2 [Appendix 8]**

```
notmycode from m in Methods where
  m.SourceFileDeclAvailable &&
  m.SourceDecls.First()
.SourceFile.FileName.ToLower().EndsWith(".designer.cs")
select m
```

PQL

Program Query Language (PQL) was created in 2005 by Michael Martin, Benjamin Livshits Monica S. Lam at the Standford University, USA,

This technology is different from the other ones as it is not quite a static code query language. But nevertheless, its syntax is interesting. This tool is for finding application errors and security flaws in the program. For this it is using a query language to describe the faulty code patterns.

Another major difference compared to the previous technologies is that PQL is not extracting fact from the source code, but it is using the abstract execution trace. The abstract execution trace is a kind of program operation log. When the program executes an instruction, it will log it out as an event ID, event type and parameters. This execution trace generation happens runtime. PQL will try to find patterns in the abstract execution trace.

The disadvantage of doing dynamic code analyses is that it is may leave out some code paths. For example, the program may not enter some if branches, the execution depends on the parameters and control flow. The program also may have randomness incorporated like random data, time, thread scheduling that affect to control flow as well. The creators of this tool tried to fix these shortcomings by providing a way to do static code analysis as well.

They have an extractor that can extracts information from Java jar files and store it in a bddbddb database. But this database can be queried using Datalog not PQL. To overcome this obstacle, they provided an algorithm that translates PQL into Datalog. The static analyzer converts PQL into Datalog and it queries the bddbddb using the translated query.

Additional feature of the PQL technology is that it can replace code runtime, if a match occurs. User can specify callbacks to handle errors and security faults this way, the problem is stopped before some serious escalation could happen. For example, users can write patterns to match SQL injections and specify a code replacement if the injection is about to happen.

The interesting feature about PQL language is its code patterns definition syntax. The matching pattern looks like a programming language which makes usage very intuitive for developers. Even those developers who just encountered the syntax for the first time, can make a good guess on what the query is doing.

PQL is working with the Java bytecode and it supports Java types in the queries. It provides an API and command line user interface.

SQL injection PQL matcher implementation **[Appendix 9]**

```
5  query simpleSQLInjection()
6  uses
7        object HttpServletRequest r;
8        object Connection c;
9        object String p;
10 matches { p = r.getParameter(_); }
11 replaces c.execute(p)
12       with Util.CheckedSQL(c, p);
```

GREP

The initial version of grep was created in 1974 by Ken Thompson. It was included first in the Version 4 of the Unix operation system. Currently is supported by all Unix-like operating systems. Over time it proved to be such a popular tool that it even become a word on it's own synonym with searching.

‘grep’ is a plain text search command line tool that matches regular expression with text lines. It’s common for developers who work on Linux like systems to use this as their default code search tool. Its success is credited to being simple, easy to understand and doing nothing more than it has to do[16]. Grep provides fast results to its user without the need to understand the structure of the data.

However, text base source-code search has some limitations as consequence of missing syntactic information. Searching for a method of a class with a standard/common name can lead to many false positive results, example searching for methods like init, open, close, start, setId, getId, clear etc.

Also, there is no way to search for information that does not appear in the source code, for example implicit function call in C++.

Chapter 5: CodeQL

In this section we going to design our own query language.

In the following sections we going to discuss:

- what use-cases we want to cover?
- what features we want in the language?
- what tools are we going to use to implement it?

5.1 Goals

Code search plays a crucial role in software engineering. The main goal of this tool is to make this activity more straightforward. However, this is a complex goal so we going to break it up to multiple smaller and more measurable goals.

Easy to learn and use for software developers

In a software development project, the code search technology does not occupy the center of attention. Developers have many other problems to worry about like the project requirements, business logic implementation technologies, testing, maintenance. They may not be willing to learn a complex language just to perform code search even if it would benefit them it in the long term.

The survey [7] shows that developers search a 5.3 times per workday. These sessions are short but frequent. This is one more reason to focus on usability, because developers are not investing to much energy into writing one query than reusing it, but they write many unrelated queries that cover independent pieces of code.

‘Search bar’ usage

We want to be able to incorporate querying into existing code comprehension tools like CodeCompass. This will make the query technology more accessible to developers who prefer the use of graphical user interfaces. Front-end developers may prefer this approach as their terminal scripting skills may not be up to date.

Another advantage of adding a query bar is that users can use the query functionality together with the features of the code comprehension tool. The developers could search for

some classes and then view their inheritance diagram. Choosing this approach, we don't need to add output formats like graphs, maps and charts to the query tool itself, but developers can use it from the code comprehension tool.

Focus on object-oriented programming

Programmers usually need to search code in large code bases. These large code bases are somewhat maintainable because they are following the object-oriented paradigm. Also programming languages that support the object-oriented paradigm are the most popular among software developers. Because of these justifications the CodeQL language will focus on querying object-oriented languages.

Expressiveness

When searching for code, developers want to express a wide variety of construct elements. There are many interest points in a programming language and for a query language to be useful it must cover at least the most common occurring ones.

Concise

Developers write search queries frequently through their work day and it's important for them to be able to do this as quickly as possible.

Each time developers write a query they take away their attention from their main objective. This mental context switching makes developers less productive. The objective of making the language concise is to make developers return to their initial work as fast as possible.

Another benefit is the limited number of keywords that the user needs to remember. This takes away stress from the user's long-term memory, further improving productivity.

Command line interfaces

Over the year the Unix terminal showed how versatile command line search tools can be. Linux developers are accustomed to the power of command line information retrieval tools like grep, awk, wc etc. In conclusion we set the goal for CodeQL to be accessible from CLI.

Extendable

The targeted users for this tool are software developers. Having a competent user base means that users will want to add functionality to the tool. One way is to contribute to the project. Another way is to build software that leverages the results provided by CodeQL. In this section we are referring to the second options.

Scalable

Software project size can be very large, especially in the enterprise sector. CodeQL needs to work on repositories with millions of lines of code. These are the type of projects on which a code search tool would provide the most value.

A firm may have global mono repositories, where all the code is stored in the same place and they build everything together. If the infrastructure underneath is working well this can be an effortless experience for the developers. In such a case our tool should be able to search through the firms entire code-base.

Output configuration

The result of queries can be consumed by different entities. These entities could be human users or other software systems. The output should be configurable for the receiving side.

Focus on code search

The grep tool proves that doing less is sometimes more. Developers prefer working with modular components, that perform one single task, but they do it correctly, reliable and predictable. The developer will than decide how these modules will work together.

We don't want to add to much unrelated functionality to the tool as that over complicates the usage and may cause some unpredictable behavior. Rather we provide a way for the user to create modules with extra features.

The Linux piping mechanism is the perfect example of how effective this concept can be.

5.2 Syntax of CodeQL

This section presents the initial syntax of the CodeQL query language. From the design point of view, it tries to accomplish the previously described goals as much as possible. We try to balance all objectives as they are influencing one other. For example, if we chose a very concise approach, we need to sacrifice expressibility on some level.

Over the history of object-oriented programming language C++ was one of the most influential language. Many other languages borrowed ideas and even keywords from C++. It is a complete language and offers plenty of freedom to the user. For this reason, the first language supported by CodeQL is C++. This fact does have an impact on query syntax, but the query language itself is language agnostic.

Another reason why C++ was chosen is that we use CodeCompass as parser and this tool has a more mature C++ support.

Operations

The first objective is to categorize the actions of the user. We observe that in C++ developers are interested in three type of programming terms. These are declaration, definition and function usage.

The declaration is a statement that describes a code element like a class, function or variable. The job of the declaration is to inform the compiler that such an identifier exists and what is its type. An identifier can be defined multiple times.

Example C++ declarations:

```
1) class Bar;

2) void foo(int);

3) extern char s[100];
```

The definition is the implementation/instantiation of the previously declared identifier. An element can have only one definition per translation unit.

Example C++ definitions:

```
1) class Bar {};

2) void foo(int x){

     std::cout << x;

   }

3) char s[100];
```

Function usage means finding the location in source code where a function is called. Finding these locations with text search is a challenge. Sometimes it is impossible for implicit function calls.

Example:

```
1  class Foo {
13 public:
14    void bar(); // we want to see who is calling this
15 }
16 ...
17 Foo foo;
18 foo.bar(); // we are interested in this line
```

As result we came up with the following structure for our query language:

`<action>: code description`

where `<action>` can be `use, dec, def` representing usage, declaration and definition.

Describing code

One of the most expressive way to describe code is with code. This solution is inspired by PQL language. We saw how intuitive is the PQL usage for developers who are already familiar with Java. We are using a similar approach for CodeQL as well, to make usage intuitive for developers who are familiar with C++.

The main code terms that we want to focus are classes and functions. Doing search on classes we find data types, while doing search on functions we find operations. Data and operations are the cornerstones of every program. If we can describe these two terms, we have a very good source code coverage.

The idea is to use code to do search, so if we want to search for class Foo declaration, we would write a query like:

```
dec: class Foo
```

For definition:

```
def: class Foo
```

Usage is more bound to functions. Developers could be interested in finding all the bar function calls where bar's declarations would look like: `void bar()`. We could use the declaration itself as query:

```
use: void bar();
```

Methods of a class are just function that have the this pointer passed as an extra argument. The this pointer points to the address of the object on which the method was called. In some languages like Python developers need to explicitly pass the this pointer in the methods parameter list.

We design the query syntax for method description same way as C++ would refer to a method during definition:

```
<action>: <return type> <class>::<method name>(<param list>)
```

Example query method bar of class Foo, that accepts an int:

```
dec: void Foo::bar(int)
```

We can further enhance the query by adding namespaces to the name if necesarry:

```
use: size_t std::string::length()
```

This approach is highly expressive as we can query a wide variety of callable elements.

Constructors:

```
1  use: Foo::Foo() //default constructor
2  use: Foo::Foo(const Foo&) //copy constructor
3  use: Foo::Foo(Foo&&) //move constructor
4  use: Foo::Foo(int, const char*) //custom constructor
```

Operators:

```
19       dec: std::ostream& operator<<(std::ostream, Foo)
20       use: int operator+(int, int)
21       use: Foo Foo::operator+(int)
22       use: Foo Foo::operator+(Foo)
```

Another aspect worth paying attention to is the parameter list. This is a list of comma separated types. These types can be more than just class types, they can be references and pointers as well. In addition, we also can have the const type qualifier, example `cons char*`. The same type rules are valid for the return types as well.

An observation is that we don't want always to specify all the elements in a query. We may not care about the return type. Or we may want all the methods of a class/namespace. For this functionality we introduce the wildcard feature. This is a frequently used technic in pattern searching. The wildcard is a placeholder defined with a single symbol that matches everything.

Examples for wildcard usage:

We want to query all the method definitions of class Foo that accept an int and return a bool:

```
def: bool Foo::?(int)
```

We want to query all the methods definitions of class Foo regardless of return type and parameters:

```
def: ? Foo::?(?)
```

Get all the create method definitions that accept a string and return a bool.

```
def: bool ?::create(std::string)
```

Another important topic that we need to address is inheritance, as want to focus on object-oriented programming. Inheritance is a vital part of many programs and it can seriously complicate text base search tools. Also, inheritance relationships can be complex, so there is a real need for code search tools that can search through this inheritance hierarchy.

Library and programming language developers frequently encounter code search uses cases where inheritance complicates understanding. For example, they want to update or remove a function that other users may depend on. They want to measure the impact of this change, however using text base code search they might get the wrong idea about how that code is used. Let's say they want to modify the *bar virtual* function of *Foo* class.

```
class Foo {
    virtual void bar();
}
```

But the call of this function can be masked by a complicated inheritance hierarchy.

```
1  class FooDer1 : public Foo { ... }
2  class FooDer2: public FooDer1 { ... }
3  ...
4  std::shared_ptr<FooDer1> fooDerPtr =
5                 std::make_share<FooDer2>();
6  fooDerPtr->bar(); //the user is interested in this line
```

Somehow, we need to specify in the query that we are interested in the derived classes as well. Again, we take inspiration from the C++ language and we provide the following syntax:

`(?: public Foo)-` types that inherit from Foo

With this syntax our query would look like this:

```
use: void (?: public Foo)::bar()
```

This will match only on direct base-derived relationships. However, we would like this query to work on indefinite inheritance relationship chains. For this we provide the wildcard feature (‘*’):

```
use: void (?: public *Foo)::bar()
```

One more important use-case can be observed, when programmers want to find where a member functions is overridden. This is an important knowledge as it defines how the program is working. With virtual functions, library users can inject code. The maintainer developers than have a hard time finding out from where this code came.

```
7  class FooDer1 {
8        void bar() { ... } // overridden
9  }
10 std::shared_ptr<Foo> fooPtr = std::make_shar<FooDer2>()
11 fooPtr->bar(); //we don’t know from where this is comming
```

The override query looks same way as the virtual function usage query, but it uses the definition (*def*) action:

```
def: void (?: public *: public Foo)::bar()
```

In summary the syntax is influenced by C++ and regular expressions. Combining these two technologies gives a very powerful tool, that is also simple, concise and easy to understand by the programmers. Most programmers are already familiar with grep and C++ like syntax, for this reason adopting CodeQL tool should not rise much difficulty for them.

5.3 CodeCompass

CodeCompass is a open-source code comprehension tool, that has the purpose to help developers understand unfamiliar code. It is also an open framework that users can extend and create their own parsers and visualizations using the plug-in mechanism. Besides the extensibility, it already provides a feature rich platform by default.

It achieves a deep analysis of the source code by using the build information. This mean that it gives different parsing for different binary versions. This is necessary because the final software is not dependent only on the source code, but on the build parameters and the environment as well. Build options can completely change the behavior and the structure of the software. For example, a cross platform software can have features that are supported on Linux, but not supported on Windows and vice-versa. The compiled source code can be branched using `#ifdefine` and the included binaries are also different.

```
#ifdef WINDOWS_BUILD_OPTION
  DoSomethingOnWindows();
#else
  DoSomethingOnOtherPlatforms();
#endif
```

Some example of build parameters that have effect on the compiled software behavior are platform-options, turning debug on, setting compiler optimization levels (gcc -O, -O2, -O3), architecture options (x86,x64), define setting.

It provides architectural information about the relationships between the header files, source files and object files. Users can easily track down which header files were included by which source file, which source files were compiled to which object, which objects were linked to which libraries and which libraries were used by which executables.

CodeCompass helps code comprehension with textual and visual summaries of different source code elements like types, variables, functions and macros. Some of the generated visual representations are interactive further enhancing code exploration. The user also can locate efficiently files using text based regexp search.

This tool is scalable and usable even for larger code bases. The response time is fast, and this is needed as programmers are easily distracted and their short-term memory fades away fast. The parsing time is around 2-3 times longer than the build time, this is acceptable because parsing doesn't need to be done too frequently. The parsing will generate a database that usually is 30 or 50 times the size of the source code.

It has a web user interface on which the user can explore the code using features like info tree, metrics and diagrams. It can work together with CodeChercker to provide static code analysis as well.

CodeCompass parsers will create a relational database using the build logs. Users can create their own parser like cpp parser, git parser, search parser. After the creation of the database by the parsers the CodeCompass web server is able to query this database. On this level users can create their own services that can make use of the new parsing data or create new visual tools. The services are accessed through a web client that displays the charts, graphs and other visualization forms of the data.

5.4 Implementation

CodeQL follows the extract-abstract-present pattern. In the scope of this thesis we implemented the abstract and present components, for extract we reused CodeCompass's functionality. At the time of writing this thesis, the implementation is working with only C++ programming language.

5.4.1 Extract

For the code parsing part, we used CodeCompass's clang abstract syntax tree (ast) based extractor. The clang ast dump proved to be enormous for larger projects, with a rate of 1:500 between source code and the extracted data. This wasn't scaling well, for this reason CodeCompass filters this ast data and stores only the necessary information, this way it reduces the database size to a rate of 1:30-50 between source and CodeCompass database. While parsing a project CodeCompass stores the data in a relational database.

The CodeCompass relational database contains all the data that we would need to perform code search. The advantage of this solution is that other applications can make use of the data, by connecting directly to the database.

5.4.2 Abstract

Programmers could write SQL queries to gather the information about certain code elements. But this is a tedious task and some SQL queries can be complicated. Programmers would rather use simple text search, than start writing and debugging complicated SQL queries. Our CodeQL is much simpler to use and provides the basic queries that a developer would need.

We created an interpreter for this purpose that accepts CodeQL queries and outputs SQL queries. The CodeQL interpreter converts the CodeQL query to an SQL query which queries the CodeCompass database. For this purpose we used the flex and bison tools[17].

With flex we define the language tokens and keywords. Flex is a free and open source program that creates lexical analyzers. The main keywords of CodeQL are `use`, `def`, `dec` and c++ keywords like `class`, `public`, `const` etc. After defining the tokens, flex will tokenize the input, and will provide these tokens to Bison. With Bison we express the relationships between these tokens and we define the logic of parsing.

Bison code that defines the high-level structure of CodeQL:

```
query: qoperation ':' regexcpp;

qoperation: USE | DEF | DEC
```

After implementing the parser we have a tool that can generate SQL from CodeQL:

```
> CodeQL_to_SQL 'usage: void Foo:foo2()'
```

Generates:

```
12 select path, location_range_start_line,
13        location_range_end_line,
14        location_range_start_column,
15        location_range_end_column
16 from "CppAstNode", "File"
17 where "astValue" ~'void Foo::foo2()' and
18       "File".id=location_file and
19       "CppAstNode"."astType" = 5  and
20       "CppAstNode"."symbolType" = 1;
```

If this query is executed on the CodeCompass database, it will give back the expected results:

Path	start_line,	end_line,	start_coll,	end_coll
/home/path/to/file/Foo.h	164	164	57	63
/home/path/to/file/Main.cpp	50	50	32	38

This query will return the path of the files with the beginning and end line/column of the pattern.

To provide full functionality we need another component that can executes the query on the CodeCompas database. For this we make use of the Linux piping mechanism :

```
21 > CodeQL_to_SQL ‘usage: void Foo:foo2()’ | CodeQL_SQL_Executer
   db.propriety
```

This will produce the same output as running the SQL command.

5.4.3 Present

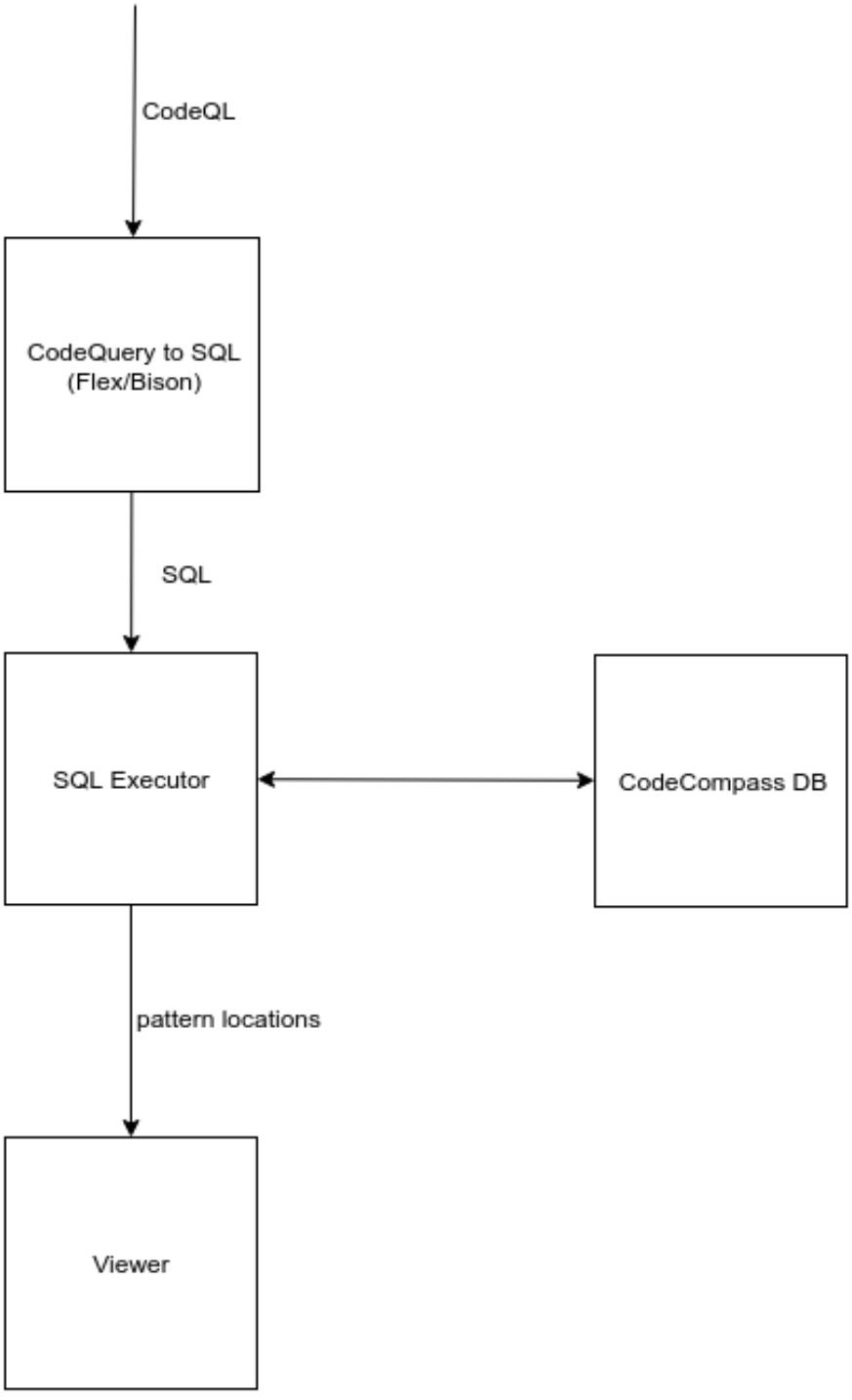
The last challenge of the tool is to show the results in a meaningful and simple way. The presentation component reads the executors output as input, each row represents one result. We provide a simple default shell script implementation that displays the 5 parameters together with the code snippet.

```
> CodeQL_to_SQL 'usage: void Foo:foo2()' |
  CodeQL_SQL_Executer db.propriety |
  CodeQL_display
```

Output:

```
============================================================

/home/path/to/file/Foo.h, line: 164-164, column:57-63

foo2();

============================================================

/home/path/to/file/Main.cpp, line: 50-50, column: 32-38

fooObject.foo2();

============================================================
```

If this viewer is not enough for the user's needs, they can easily create their own viewer. For example, the user may want to view git commits information with the matches, or they want the data in a different format like xml or json.



CodeQL architecture

Chapter 6: Conclusion

CodeQL acomplishes the goals that we put for our query language. For programmers familiar with object-oriented programming, CodeQL is an easy to lean and use. The queries can be written fast and express common OOP patterns. The tool usage is like grep, but the queries are enhanced with syntactic knowledge.

As the queries are so concise, they are ideal for search bar or command line use. Programmers don't need to write complicated queries to a file, they can just use the command lines or search bar to execute their query. This is very important as programmers tend to search code frequently and iteratively.

This tool was designed for command line use, for this reason is optimized for Linux piping usage. CodeQl can be piped with other costume components created by the users.

CodeQL leverages the scalability and versatility of CodeCompass as it uses its database for search.

6.1 Further development

The further development possibilities are endless. As users will use the tool a priority of what else should be included will be formed. Some ideas of new features include: api support, templates, Java support, keyword search, more metrics, multiline patter expression. Another important problem that we are going need to address is better debug capabilities and better protection against SQL injection. For some further development ideas, CodeCompass also will need to be modified.

Chapter 7: Appendix

Appendix 1: Package instability Grok implementation

```
1  PackageDep := PackageOf o ClassDep o ( inv PackageOf )
2  PackgDepInterPackg := PackageDep - ( id dom PackageOf )
3  ClassFowardRel := ( inv PackageOf ) o
4                    PackgDepInterPackgo PackageOf
5
6  ClassDepInterPackg := ClassForwardRel ^ ClassDep
7  AffCoupling := PackageOf o ( inv ClassDepInterPackg )
8
9  AfferentCoupling := ( dom AffCoupling )
10                 outdegree AffCoupling
```

Appendix 2: Package instability Rscript implementation

```
11 rel [ str , str ] PackageOf
12 rel [ str , str ] ClassOf
13 rel [ str , str ] MethodCall
14 rel [& T1 , int ] outdegree ( rel [& T1 ,& T2 ] R )
15       = { <D , # R [ D ] > | <& T1 D , & T2 U > : R }
6
16 rel [str,str] ClassDepInterPackg
17       = { <C1 , C2 > | < str C1 , str C2 > : ClassDep
18       , PackageOf [ - , C1 ] != PackageOf [ - , C2 ] }
19
20 rel [ str , str ] AffCoupling
21       = PackageOf o inv ( ClassDepInterPackg )
22
23 rel [ str , int ] AfferentCoupling = outdegree ( AffCoupling )
24 rel [ str , int ] PackageInstability
25       = { <P1 , (100* N1 )/( N1 + N2 ) >
26       | < str P1 , int N1 > : EfferentCoupling
27       , < str P2 , int N2 > : AfferentCoupling
28       , P1 == P2 }
```

Appendix 3: Package instability SemmleCode implementation

```
29 predicate classDepInterPackg(Class c1, Class c2) {
30       c1.getPackage() != c2.getPackage () and
31       classDep (c1,c2)
32 }
33 class MyPackage extends Package {
34       MyPackage (){this.fromSource ()
35 }
36
37 predicate affCoupling(Class c) {
38       exists(Class c1 | this . contains(c1) and
39               classDepInterPackg(c1, c))
40 }
41
42 int afferentC o u p l i n g () {
43       result = count(Class c | this.affCoupling(c))
44 }
45 float packageInstability() {
46       result = (1.0 * this.efferentCoupling()) /
47                   (this.afferentCoupling() +
                    this.afferentCoupling())
48       }
49 }
```

Appendix 4: Package instability JGraLab implementation

```
1  from p : V{JavaPackage}
2  reportMap p ,
3        from outerClass : V{JavaClass}
4        with
5               (not p -->{PackageOf} outerClass) and
6               (p -->{PackageOf} <--{ClassDep} outerClass)
7        report outerClass end
8  end store as AffCoupling
9
10 using AffCoupling :
```

```
11 from p : V{JavaPackage}
12 reportMap p, count (get (AffCoupling, p)) end
13 store as AfferentCoupling
14
15 using AfferentCoupling, EfferentCoupling:
16 from p : V{JavaPackage}
17 reportMap p, get (EfferentCoupling, p) /
18             ( get (EfferentCoupling, p) +
19              get (AfferentCoupling, p))
20 end store as PackageInstability
```

Appendix 5: Package instability CrocoPat implementation

```
1   ClassDepInterPackg(c1, c2)
2         := EX(p1, PackageOf (p1, c1)
3               & EX(p2, PackageOf(p2, c2) & !=(p1, p2)
4                     & ClassDep(c1, c2)));
5
6   AffCoupling (p, c) := EX(c1, PackageOf(p, c1) &
7               ClassDepInterPackg(c,c1));
8
9  Package(x) := PackageOf (x, _);
10 FOR p IN Package(x) {
11       ca := #(AffCoupling (p, c));
12       PRINT " AfferentCoupling ", p, " ", ca, ENDL;
13       ce := #(EffCoupling (p, c));
14       PRINT "EfferentCoupling ", p, " ", ce, ENDL;
15       i := ce / (ca + ce);
16       PRINT " Instability ", p, " ", i, ENDL;
17 }
```

Appendix 6: Package instability JTransformer implementation

```
1  classDepInterPackg(C1, C2): -
2        packageOf (P1, C1), packageOf (P2, C2),
3        not (P1 = P2), classDep (C1, C2).
```

```
4
5  affCoupling (P, C): -
6        packageOf (P, C1), classDepInterPackg(C, C1).
7
8  afferentCoupling(P, N): -
9        setof (C, affCoupling (P, C), AffClasses),
10       length (AffClasses, N).
11 packageInstability(P, I): -
12       efferentCoupling(P, Ec), afferentCoupling(P, Ac),
13       I is Ec /(Ec + Ac).
```

CQLinq query1 [Appendix 7]

```
  from m in Methods where m.NbLinesOfCode > 30 select m
```

CQLinq query2 [Appendix 8]

```
1  notmycode from m in Methods where
2    m.SourceFileDeclAvailable &&
3    m.SourceDecls.First()
4      .SourceFile.FileName.ToLower().EndsWith(".designer.cs")
5  select m
```

SQL injection PQL matcher implementation [Appendix 9]

```
1  query simpleSQLInjection()
2  uses
3        object HttpServletRequest r;
4        object Connection c;
5        object String p;
6  matches { p = r.getParameter(_); }
7  replaces c.execute(p)
8        with Util.CheckedSQL(c, p);
```

Chapter 8: Table of figures

Bibliography

[1] Brooks, F. P. "No silver bullet: Essence and accidents of software engineering", IEEE Computer, pp. 10-19. 1987.

[2] Brooks, R. Towards a theory of the comprehension of computer programs. International Journal of Man–Machine Studies, pp. 543–554. 1983.

[3] Letovsky, S. Cognitive processes in program comprehension. Empirical Studies of Programmers: Proceedings of the 1st Workshop. Ablex: Norwood NJ. 1986.

[4] Littman, D, Pinto J, Letovsky S, Soloway E. Mental models and software maintenance. Empirical Studies of Programmers: Proceedings of the 1st Workshop. Ablex: Norwood NJ, pp. 80–98 1986.

[5] Soloway, E, Ehrlich K, Empirical studies of programming knowledge. IEEE Transactions on Software Engineering, pp. 595–609, 1984.

[6] Storey , M.-A.D, Fracchia , F.D, Müller, H.A.k Cognitive design elements to support the construction of a mental model during software exploration. Journal of Systems and Software, Volume 44, Issue 3, 1999

[7] Sadowski, C., Stolee, K. T , Elbaum S. How Developers Search for Code: A Case Study, Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)}, 2015

[8] Tiago, L. A., Jurriaan H., Peter R., A Comparative Study of Code Query Technologies, Working Conference on Source Code Analysis and Manipulation, 2011

[9] Martin, R. C. OO design quality metrics — an analysis of dependencies. Technical report, Object Mentor, October 1994

[10] Holt, R. C. Structural manipulations of software architecture using Tarski relational algebra. In WCRE'98, page 210. IEEE ComputerSociety, 1998.

[11] Paul, K., Using RSCRIPT for Software Analysis, In Working Session on Query Technologies and Applications for Program Comprehension, 2008

[12] NDepend https://www.ndepend.com/docs/cqlinq-syntax

[13] LINQ https://msdn.microsoft.com/en-us/library/bb308959.aspx

[14] Lämmel, R , Lämmel, V, João, J., João S. Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, Revised Papers., 2007

[15] Martin, M., Livshits, M. , Lam, M. S. Finding application errors and security flaws using PQL: a program query language, Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, October 16-20, 2005, San Diego, CA, USA

[16] Singer, J, Lethbridge, T. What's so great about 'grep'? Implications for program comprehension tools, Technical report, National Research Council, Canada 1997.

[17] Levine, J., flex & bison: Text Processing Tools, O'Reilly Media Inc., 2009