

Binary relational querying for structural source code analysis

Master's thesis

Peter Rademaker

Binary relational querying for structural source code analysis

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE TECHNOLOGY

by

Peter Rademaker
born in Baarn, the Netherlands



Universiteit Utrecht

Universiteit van Utrecht
Software Technology Group
Utrecht, the Netherlands
www.cs.uu.nl



Software Improvement Group
A.J. Ernststraat 595-H
Amsterdam, the Netherlands
www.sig.nl

Binary relational querying for structural source code analysis

Author: Peter Rademaker
Student id: 0111376
Email: peter.rademaker@gmail.com

Abstract

Software analysis involves the retrieval of large numbers of relations from the source code of the program. These include call, inheritance, inclusion, and database access relations. The querying of these relations for the purpose of obtaining high-level overviews of internal architecture and information flow is a non-trivial task.

The graph library used by the Software Improvement Group (SIG) for this task has limitations regarding its expressiveness, abstraction facilities, and offered functionality. This makes it difficult and time-consuming to write queries that retrieve the necessary information. The goal of this thesis project was to find a new solution that removes these limitations.

First, we have compared existing source code query tools. The tools were compared with respect to ten criteria concerning language and tool features. The language features were compared by implementing four archetypical source code queries in each of the tools. Although the compared solutions are promising, we found that only JRelCal offers the combination of good abstraction and extension facilities together with an API.

JRelCal is a prototype Java library that is based on Tarski's binary relational calculus. Binary relational calculus allows us to express source code analysis queries in a concise and declarative manner. JRelCal, however, has significant shortcomings: the performance of some of its operations is poor, it lacks some essential features, and its implementation contains several bugs. To overcome these shortcomings we have created a new implementation of JRelCal. The new implementation includes an optimised transitive closure operation, support for predicates, and an efficient representation for relations. We validated the new JRelCal in an extensive case study.

Thesis Committee:

Chair: Prof. Dr. S.D. Swierstra, Software Technology Group, Utrecht University
Supervisor: Dr. J. Hage, Software Technology Group, Utrecht University
Company supervisor: Dr. J. Visser, Software Improvement Group

Contents

Contents	iii
1 Introduction	1
1.1 Description of the company	1
1.2 Problem description	1
1.3 Research questions	2
1.4 Approach	2
1.5 Outline	3
2 Background and Related Work	5
2.1 Source code querying	5
2.1.1 Goals	5
2.1.2 Extract, abstract, present	6
2.2 Related work	7
2.2.1 Codd’s relational algebra	7
2.2.2 Tarski’s binary relational calculus	9
2.2.3 Logic programming languages	10
2.2.4 Other tools	10
2.3 Formal foundations of source code query languages	11
2.3.1 Codd’s relational calculus and algebra	11
2.3.2 Tarski’s binary relational calculus	12
2.3.3 Logic programming	13
2.4 Implementation of source code query languages	14
2.4.1 Main-memory implementations	14
2.4.2 Disk-based implementation	15
2.5 SIG’s software analysis toolkit	15
2.5.1 The SIG graph library	16
3 Comparison of code query technologies	17
3.1 Code query technologies	17
3.1.1 Crocopat	17
3.1.2 Rscript	18
3.1.3 JRelCal	18
3.1.4 GReQL 2	18
3.1.5 SemmleCode	19
3.1.6 Other technologies	19
3.2 Usage scenarios and criteria	20
3.2.1 Usage scenarios	20
3.2.2 Language criteria	21
3.2.3 Tool criteria	22

3.3	Language comparison	24
3.3.1	Lifting	25
3.3.2	Software Design Metric	27
3.3.3	Graph Pattern detection	31
3.3.4	Slicing	34
3.4	Summary of results	37
3.4.1	Language comparison results	37
3.4.2	Tool comparison results	38
3.5	Conclusions	38
3.5.1	General conclusion	38
3.5.2	Conclusion for the SIG	39
4	Improving JRelCal: from version 0.1 to 1.0	41
4.1	Introduction to JRelCal	41
4.1.1	Side-effect free methods	42
4.1.2	Generics	42
4.1.3	The new JRelCal	43
4.2	Predicates	44
4.3	Optimisation of reachability queries	45
4.3.1	Problem definition	46
4.3.2	Solution	46
4.3.3	Implementation	47
4.3.4	Run-time optimisation of reachability queries	49
4.3.5	Representation	51
4.3.6	Benchmarks	53
4.3.7	Discussion	56
4.4	Integration with SAT	57
5	Case study: Static estimation of test coverage	59
5.1	Introduction	59
5.2	Approach	60
5.3	Implementation	60
5.3.1	Distinguishing test code from production code	60
5.3.2	Identification of covered methods	61
5.4	Discussion	62
5.4.1	Validation	62
5.4.2	JRelCal compared to SIG Graph library	62
5.4.3	JRelCal compared to .QL	63
5.5	Conclusion	64
6	Conclusion	65
6.1	Research question	65
6.2	Contributions	65
6.3	Future work	66
6.3.1	Tool comparison	66
6.3.2	Optimisation by algebraic transformation	66
6.3.3	Optimisation by improving implementation	67
6.3.4	Supporting queries involving n -ary relations	67
6.3.5	Other language constructs	68
6.3.6	Creation of a DSL on top of JRelCal	68
6.4	Acknowledgments	68

1.1 Description of the company

The Software Improvement Group (SIG) is a consultancy company that provides management consulting services based on source code analysis. The main goal of the company is giving more insight in the technical quality of the software systems of its customers. SIG's services are mainly intended to be used at management level.

When management has to decide on large investments concerning software systems –like replacement or renovation of the system, large-scale changes, and in- or outsourcing– it needs objective and thorough information about the status and technical quality of the software system. Due to the large size and complexity of today's software, it is difficult for IT management to obtain such information. They often have to rely on information provided by the technical staff responsible for the system. This information may be incomplete or incorrect, often because written documentation is either unavailable or insufficient and the programmers lack a complete overview of the entire system.

To provide management with the correct information, SIG offers two services: software risk assessments [71] and software monitoring [48, 50]. Below, we will discuss these services in more detail.

A *Software risk assessment* is a one-time investigation of the technical quality of a system. It is based on partially automated source code analysis and information collected from documentation and interviews with system experts. The result is a report on the status and technical quality of the system. The report also includes recommendations that help management to make the right decisions.

The *Software monitor* is basically a software risk assessment repeated on a regular interval. These assessments are based on a standardised set of queries. The results are reported to the client in a web-based dashboard that provides an overview of the measurements. In addition, SIG consultants periodically give presentations to the management to summarize the finding. This way, an indication of the technical quality of the source code is provided without having to look into the source code itself. This gives the management continuous insight into the quality of the system, which can help maintaining the quality of a system over time.

Another service offered by SIG is the automatic documentation generator *DocGen* [25]. This service is aimed at the software engineer instead of the management. DocGen can generate technical documentation for a software system based on source code analysis. This is useful because often the technical documentation is insufficiently detailed or outdated. These documentation deficiencies will increase the risk and the cost of making changes to the system. Increased risk because software engineers have a less clear picture of what the effects of their changes will be, increased costs because it takes more time for the software engineers to comprehend the system before they can make the change.

The core of the software risk assessment and software monitor services is implemented in SIG's software analysis toolkit (SAT), which we will describe in more detail in Section 2.5.

1.2 Problem description

SIG's services are mainly based on information obtained by formulating software analysis queries using the SIG graph library. The SIG graph library is used to represent information extracted from the source code as

a graph. This graph is a directed and typed graph. At the nodes, results of metric computation are stored in a map. In addition, it offers functionality to perform queries on the information represented by the graph. We distinguish two use case scenarios for the SIG graph library:

1. The most common case is when it is used for the software monitor: periodically, a standard set of source code queries is run on the client's system to provide the information to be shown in the software monitor:
2. Sometimes, a client has a more specific problem. E.g., it wants SIG to do impact analysis on its code, in order to learn which code can be changed safely. Such situations require the analyst to investigate the system by formulating and running custom queries.

Currently, there is a problem with how queries are formulated in the SIG graph library. The library requires the programmer to specify queries as imperative graph algorithms, using (nested) loops to specify traversals over the graph structure. This is a complex and inconvenient way of specifying queries, which requires more effort to learn, and makes it difficult to understand queries. As a result, it is more likely that mistakes will be made, resulting in the retrieval of incorrect information.

Therefore, it is desirable to hide the implementations details beneath appropriate abstractions. It should be possible to specify a software analysis query more *declaratively*, using high level operations. This lets the analysts focus on specifying the properties of the desired result (the *what*), instead of specifying the algorithm to achieve this result (the *how*). An extra layer of abstraction makes it easier to specify sophisticated analyses. Additionally, a clear separation between implementation and specification of a query simplifies switching to other (more efficient) implementations.

1.3 Research questions

Based on the problem description we define the following research question:

Can we find a successor of the SIG graph library that allows the SIG to formulate source code queries concisely, declaratively and efficiently?

In the following section we describe our approach to answering these research questions.

1.4 Approach

There exist several tools that offer a domain specific language for source code querying. Therefore, we will first compare these existing tools to see whether one of these tools can serve as a successor of the SIG graph library. We will compare language and tool features with respect to ten criteria. To compare the languages of the tools we have selected four language criteria that we compare by implementing a set of four benchmark queries in each of the tools. The remaining six criteria are related to the features of the tools.

From the comparison we conclude that only one candidate meets the most important requirements of the SIG: only JRelCal offers the combination of sufficient abstraction facilities, extendability, and an API. Consequently, we have chosen JRelCal as successor of the SIG graph library.

JRelCal is Java library that implements Tarski's binary relational calculus. It is a research prototype that has some limitations: useful language features we have found in other tools are not all present in JRelCal, and the implementation can be improved upon. To address these shortcomings we have created a new version of JRelCal. In this version we have fixed bugs, added tests, separated the specification from the implementation, added predicates, and optimised the transitive closure operation. We have realised the integration of JRelCal with SIG's SAT by providing a mapping from SIG graphs to JRelCal relations.

Since there does not exist a benchmark suite for implementations of code query languages, we have carried out a case study to validate our work. In this case study we re-implement the work of Alves on the static estimation of test coverage. This is a representative, non-trivial analysis that SIG would like to perform.

1.5 Outline

Chapter 2 gives an overview of the foundations, goals, and different approaches to software analysis and source code querying. It also discusses formal concepts that are frequently used within the field of source code querying e.g., Tarski's binary relational calculus.

Chapter 3 reports on the evaluation of existing code query technologies. We will start with introducing the technologies we have evaluated. We then discuss the two typical usage scenarios we found and the criteria we use to compare the language and tools. Every criteria is then discussed in more detail. To compare the languages of each tool we have collected a set of four benchmark queries. We will show how these queries are implemented in the different tools. We will conclude with a generic conclusion, and a more specific conclusion tailored to SIG's requirements.

In Chapter 4 we will elaborate on the improvements and optimisations made to JRelCal. We start with a general description of JRelCal. Then, we will report on our implementation of predicates; a functional extension to JRelCal, followed by a discussion of our optimization of reachability queries. We have validated this optimization in a performance tests in which we compare the performance of the new JRelCal implementation to its old implementation.

The validation of this work is described in Chapter 5, where we discuss a case-study in which we use JRelCal to re-implement work of Alves on the static estimation of code coverage.

Chapter 6 summarises the work we have done. We formulate the answers to our research questions, and conclude with ideas on how this work can be further improved.

Chapter 2

Background and Related Work

We start with a general discussion of source code querying and its uses. Then, we explore related work on source code querying by providing an overview of the history of code querying. This overview is structured by the formal foundations of the code query languages: Codd's relational algebra, Tarski's binary relational calculus, and logic programming. We consider each of these foundations in detail, followed by a discussion of the implementation techniques that can be used to implement the query languages. We conclude this chapter with a discussion of the SIG tooling.

2.1 Source code querying

In general, today's software systems are very large and complex. The average size of the systems in SIG's software benchmark, which currently contains over 100 systems from industry, is about 200,000 lines of code, and 25% of the code units (e.g., methods or procedures, depending on the language) has a McCabe cyclomatic complexity [58] value higher than 10. McCabe indicates that when a unit's complexity value is higher than 10 it can be considered to be complex, which makes the unit more difficult to understand and therefore harder to maintain. As a consequence, software engineers spend a large amount of their time on understanding the system they are working on. Corbi reports that 50% of a software engineer's time is spent on gaining an understanding of the system he is working on [17]. It is necessary to understand the relevant parts of a software system before one can add functionality, improve the implementation, find and fix defects, etc.

To assist software engineers in gaining an understanding of the software, numerous researchers in the fields of program comprehension and software reengineering have worked on tools and techniques that support this task. Over the years, several tools for comprehending software have emerged from these fields of research. In fact, many modern IDEs already offer tools such as type and call hierarchy views that can help the developer in understanding the source code. A drawback, however, is that these tools are often limited to a specific task. They do not have the flexibility to answer questions specific to a particular situation.

Source code querying is a technique that addresses this issue by offering a domain specific language that allows software engineers to formulate queries about the source code specific to their particular situation. These languages are called *source code query languages*. A *query language* is a specialised language for asking questions or queries involving data stored in a database, a file, or any other form of data storage. The data that is queried by source code query languages are typically relations (e.g. call and inheritance relations) extracted from the source code by parsers or regular expressions.

2.1.1 Goals

Arguably, the most common use of source code query tools is for program comprehension. Nonetheless, this is certainly not the only use of code query tools. Hajiyev [33] lists several situations where code query tools can be of interest. We will discuss these situations and indicate how these relate to the goals SIG wants to use code querying for: software risk assessments and software monitoring.

- *Program comprehension*: code query tools can help software engineers to understand how software executes, or how software is structured. This can, for example, be done by performing control or data

flow analysis, or by creating high-level views of the software system. A better understanding of the system can help in making changes to or decisions about the system.

Program comprehension plays an important role in software risk assessments. To give valuable and correct advice to clients, SIG consultants have to attain a thorough understanding of the system they are assessing.

- *Software reengineering and refactoring*: users can detect "bad" software structures such as code duplication or poor modularisation with the help of code querying tools. The detected "flaws" can then be removed. Removing the flaws can either be done manually, or specified as formal transformations that can be carried out automatically.

Although SIG typically does not refactor or re-engineer software themselves, during a software risk assessment they may find weak spots in the software system of the client. These weak spots can then be remedied by the client itself, based on information provided by the SIG.

- *Program Metrics*: Source code query tools can also be used for calculating software metrics [30] such as McCabe's cyclomatic complexity, package instability and average method size.

SIG bases much of its advice and findings on several software metrics as a measure of the technical quality of a system. Software metrics are also used in the software monitor to continuously monitor the quality of the software.

- *Debugging*: Just like software querying tools can be used to find bad software structures, they can also be used for finding potential bugs, e.g., a wrong sequence of calls to API methods, or a Java class that implements the `compareTo` method without implementing the `equals` method. An advantage of code query tools over generic bug finding software such as FindBugs¹ is that code query tools offer a language that allows one to write queries that can detect project specific bugs or conventions.

As mentioned at *reengineering and refactoring*, SIG does not fix bugs in the systems of its clients themselves. However, if they detect potential bugs these are reported to the client, which enables the client to take appropriate measures.

Most of the previously mentioned goals can be performed at different levels of detail. Paul and Prakash [64] distinguish three levels of detail at which queries can be performed:

- Global structural information
- Statement-level structural information
- Flow information such as data-flow and control-flow

To run a query on a certain level, relations of that level of detail need to be extracted from the source code. The higher the level of detail, the more relations are extracted. The more relations involved in a query, the more time it takes to evaluate the query.

2.1.2 Extract, abstract, present

At the highest level, the process of source code querying can consist of three different phases: extract, abstract and present. This is shown in Figure 2.1. Below, we discuss each of the three phases in more detail.

1. *Extract*: In this phase, information is extracted from the source code. It depends on the tool what information is extracted and how it is stored. It either produces a file with extracted facts, or results in immediate population of a facts database. In the latter case, extracted facts are directly available for querying. In contrast, when producing a file the user might be required to transform it to an interchange format supported by the code query technology and then manually load it. The extraction phase is typically implemented using parsers and lexers.

¹<http://findbugs.sourceforge.net>

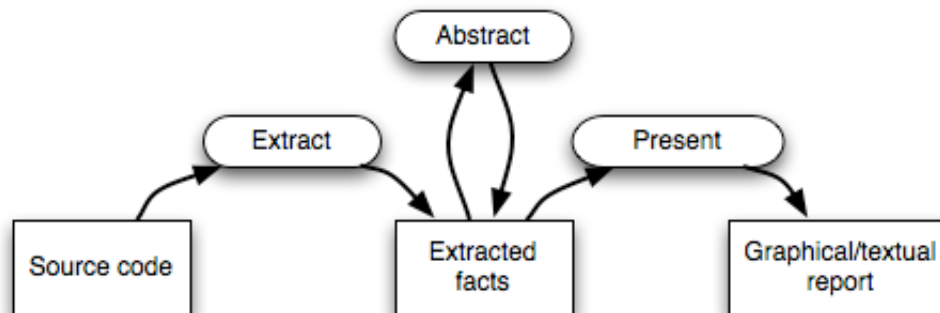


Figure 2.1: Extract - Abstract - Present

2. *Abstract*: In this phase the actual querying and analysis takes place; new information is derived from the extracted relations. This can be done by (repeatedly) executing queries on the extracted facts. For example, one might want to know which methods are called directly or indirectly from a given class. These queries can be specified in a domain specific language supported by a code query technology either in a IDE (when available) or in a text editor. This phase is where the focus of this thesis lies.
3. *Present*: In this phase the derived information is presented to the end user. Results can be presented in a textual, tabular, or graphical (e.g. graph or chart) format.

2.2 Related work

The idea that source code queries can be specified easier, more concisely, and more declaratively in a domain specific query language has been around for quite a long time. Over the years, a lot of technologies have emerged. In this section we give an overview of the different proposals that have appeared over time.

The earliest approaches to source code querying did not involve a query language at all. As early as the 70's people were using UNIX tools like `grep` and `awk` to query source code. These tools were used to perform regular expression matching on the programming language text. An `awk` script even made it possible to pair a regular expression with an action or procedure written in C-like code. When the regular expression is matched, the procedure is executed. The advantages of this approach are that it does not require parsing of the source code, it is fast, and it is relatively simple to write queries. The disadvantage is that it is impossible to express queries like "assignment to a variable of a certain data type".

In the 80's the first code query tools that offered a query language began to emerge. We discuss these tools categorised by the formal foundations of their query languages. First, we consider languages based on Codd's relational algebra (RA). This category contains all languages based on Codd's work, which are basically all SQL-like languages. Then, we explore languages that are based on Tarski's binary relational calculus (BRC). This is followed by a discussion of the query languages that are based on logic programming languages (LPL) such as Prolog and Datalog. We conclude with a short discussion of some program comprehension tools that do not offer a query language. A discussion of the formal foundations can be found in next section.

To put the tools into a historical perspective, we have created a timeline (Figure 2.2) that provides an overview of when the code query technologies emerged and how they developed over time.

2.2.1 Codd's relational algebra

The OMEGA system by Linton [52] is the first source querying tool in history. It uses a query language based on Codd's relational algebra: SQL. In 1984, Linton proposed to store information extracted from the source code as relations in a relational database. Once stored in a database, it is natural to use SQL to query this information. The information stored in OMEGA's database was very detailed to allow the reconstruction of the source code from the database. This level of detail significantly increased the size of the database, and as a result the query evaluation times.

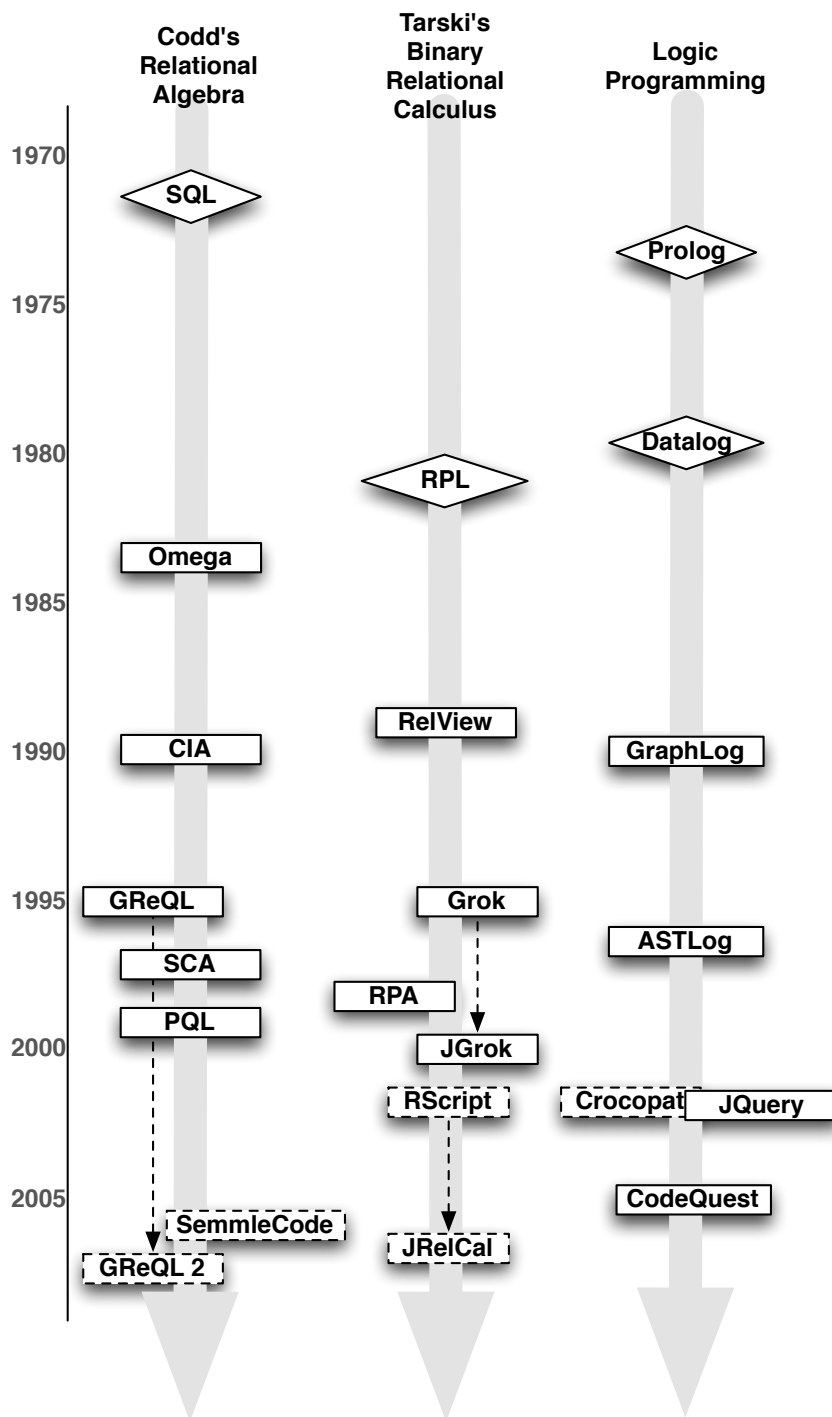


Figure 2.2: Timeline showing a historical overview of code query technologies, the relationships between them and the formalisms they are based on. A dashed arrows indicates a relation between a technology and its successor. A diamond indicates a general purpose programming or query language not specifically aimed at code querying. The code query technologies in a dashed box are included in the evaluation.

The C Information Abstraction (CIA) system [14] is another proposal that stores relations from the source code in a database. A difference with OMEGA is that CIA only stores a limited amount of information from the source code required to answer the most interesting queries. This way query evaluation times are significantly reduced. CIA is the first to suggest incremental update of the source code relations in the database. After a change to the source code, only the module containing the changed parts is parsed again and the new facts are added to the database. This is notably better than restructuring the entire database after a small local change.

The advantage of using SQL as a source code query language is that it removes the need for designing and implementing a new query language. Also, it is a language that it is known to a large part of the target users: programmers. Yet, an important disadvantage of SQL is that it does not allow recursive queries [51], which are essential for dealing with recursive constructs in software systems. A query such as "find all the methods that are directly or indirectly called from the main method" can not be expressed without a way to express recursion.

This problem is addressed in the Source Code Algebra (SCA) framework proposed by Paul and Prakash [64]. SCA is a formal framework that provides a formal data model and an algebraic query language for source code querying. The SCA query language combines a high level of abstraction with an expressive power equivalent to that of relational algebra plus transitive closure. In essence, the transitive closure operation encapsulates a limited form of recursion, which makes it possible to concisely express recursive queries. In addition to SCA, Paul and Prakash present ESCAPE, a prototype source code query system based on SCA.

PQL [43] is a more generic proposal for a SQL-like source code query language. Jarzabek focuses on flexibility and therefore does not make any assumptions about the implementation of the language, what relations to extract from the source code, and the representation of these facts.

GReQL [44, 20] and its successor GReQL 2 [11], also offer a SQL-like language. Both GReQLs use a graph representation of the source code. This is reflected in the query language: in GReQL it is possible to query both the nodes of the graph and the relations relating the nodes, and it offers regular path expressions which make it possible to search for paths in the graph.

A commercial spin-off of the academic CodeQuest project is SemmleCode [23]. Instead of using the logic query language Datalog (which we consider in Section 2.3.3), SemmleCode uses .QL, a query language very similar to SQL with two important extensions: a transitive closure operation, and object orientation. .QL is compiled to a Datalog variant, optimised, and further translated to SQL.

2.2.2 Tarski's binary relational calculus

Tarski's binary relational calculus allows concise, high level expression of binary relational queries. As opposed to RA, BRC allows one to think about relations, instead of the individual elements and tuples in a relation. Originally, a transitive closure operations was not included in BRC. However, it is included in all code query languages based on BRC to be able to express recursion. A disadvantage of BRC is that it is known to be less expressive than RA [39]. The basic reason for this is that BRC does not support n -ary relations. A practical example of this is that in BRC it is impossible to express pattern matching queries that search for n -ary patterns with $n > 2$.

The first languages based on BRC were general purpose languages called relational programming languages. Relational programming is a style of programming in which entire (binary) relations are manipulated instead of individual data. Relational programming is a generalisation of functional programming in the sense that anything that can be done with functional programming can be done with relational programming (a function is a relation with certain properties). Bruce MacLennan was one of the pioneers in the field of relational programming and published several papers [53, 54, 56, 55] on the subject. MacLennan designed a relational programming language RPL. To our knowledge, the language has never moved beyond the prototype stage.

RelView [1] is the first code query language that is based on BRC, appearing in 1989. RelView was mainly developed as a tool for prototyping graph algorithms. However, there are some reports of RelView being applied to software analysis tasks [31].

Relation Partition Algebra (RPA) by Feijs et al. [29], is basically BRC extended with a mathematical notion of partitioning. This makes it possible to elegantly express lifting queries. The disadvantage is that it does complicate the formalism with a partition concept. RPA is implemented as a set of UNIX command-line tools. Each tool represents a single RPA operation. Queries can be composed by piping outputs from tool to tool.

Around the same time, Grok [36] is born. Grok is a calculator for BRC expressions. It was mainly intended to be used as tool for analysing and manipulating software architecture, but has been applied to many other problems. JGrok is a Java implementation of Grok. To our knowledge, JGrok has never gotten beyond the prototype stage.

Rscript [47] is a small scripting language based on BRC. Distinctive features of Rscript are its (limited) support for n -ary relations, and the support for comprehensions. Van der Storm's JRelCal² is a prototype Java library that implements BRC. It was meant to be an fast implementation of Rscript.

2.2.3 Logic programming languages

Another approach to a code query language is to offer a language inspired by, or written on top of the logic programming language Prolog. Prolog is a declarative language with some imperative features that supports general recursion. However, not all features of Prolog are useful in a code querying application: its learning curve is quite steep, it is unfamiliar to a large group programmers, and large queries are often verbose.

Prolog has been used as fact database and query engine in the GraphLog tool [16]. GraphLog offers a visual query language for querying source code. ASTLog [19] uses a Prolog-like language for examining Abstract Syntax Trees. The JQuery query language [42] is a logic (Prolog-like) query language based on TyRuBa. TyRuBa is a logic programming language implemented in Java. The JQuery query language is defined as a set of TyRuBa predicates which operate on facts generated from the Eclipse JDT's abstract syntax tree.

Crocopat [9] is a relational calculator by Beyer et al. Crocopat's language RML is based on predicate logic and is therefore similar to Prolog. An important difference, however, is that RML does not follow the paradigm of logic programming. Instead of being declarative and inference-based, RML is imperative and executes the program statement by statement.

Hajiyev combines the advantages of using a Prolog-like language with those of using a RDBMS in his project CodeQuest [33]. This is done by compiling Datalog into SQL. Datalog [13] is a database query language that is like Prolog, but without control constructs and data structures. Recursive constructs in Datalog are not directly transferable to SQL, since SQL lacks a transitive closure operation. Hajiyev solves this by implementing transitive closure using extensions to standard SQL such as stored procedures and control structures. These extensions are offered by most modern RDBMS vendors.

2.2.4 Other tools

A different category of program comprehension and reengineering tools focusses on the interactive navigation through source code using a visual representation of the source code. Instead of querying source code using a domain specific language, the user can browse source code elements by navigating a visual representation of the source code. Three of such tools are Rigi [59], Sextant [26] and Moose [60].

Rigi is an interactive, visual tool designed to help better understand and navigate software. It was developed in 1988 at the University of Victoria. Rigi is capable of identifying subsystems based on certain criteria. Each identified subsystem is visualised as a separate window, together with an overview window that shows the subsystems in a hierarchy. Today, Rigi is probably most well-known for its RSF interchange format for interchanging relations. RSF stands for Rigi Standard Format, which was introduced with the Rigi tool, and is now used as an interchange format in many code querying tools.

Sextant is an interactive and visual software navigation and comprehension tool that stores source code facts in an XML database. In Sextant, XQuery queries are executed to query the source code based on the

²<http://sisyphus.meta-environment.org>

actions of the user in the user interface. Sextant is integrated with Eclipse which makes it possible to switch directly to the source code.

Moose is a general framework for program analysis. Moose tools are built around FAMIX [69] a language independent meta model for source code facts. Facts are saved to a custom file format that conforms to the FAMIX meta model. Moose offers a language that makes it possible to define new views and visualisations of the source code, and an imperative source code query language that is based on Smalltalk.

2.3 Formal foundations of source code query languages

As we have learned in the previous section, we can roughly divide the code query languages into three different categories based on the formalism the languages are based on. First, Codd's relational algebra, this category contains all languages based on Codd's work, which are basically all SQL-like languages. Second, language that are based on Tarski's binary relational calculus. And third, the query languages that are based on logic programming languages such as Prolog or Datalog.

Table 2.1 summarizes the related work discussion of the previous section. In addition, it includes the implementation of the languages, which we will consider in Section 2.4.

Language	Author	Year	Language roots	Implementation
Omega	Linton	1984	RA	DBMS
RelView	Berghammer et al.	1991	BRC	BDD
Grok	Holt	1996	BRC	Turing
RPA	Feijs et al.	1998	BRC	C & shell scripts
JGrok	Wu	2001	BRC	Java
RScript	Klint	2002	BRC	ASF+SDF
Crocopat	Beyer	2002	LPL	BDDs in C
JQuery	Volder et al.	2002	LPL	Java
CodeQuest	Hayijev et al.	2005	LPL	DBMS
JRelCal	Storm	2006	BRC	Java
SemmlCode	Moor et al.	2006	RA	DBMS
GReQL 2	Bildhauer et al.	2007	RA	Graphs

Table 2.1: Historical overview of code query technologies. Containing the year of their first publication, the formalism the language is based on, and how it is implemented.

2.3.1 Codd's relational calculus and algebra

Two formal query languages that heavily influenced commercial database languages such as SQL are the relational algebra (RA) and the relational calculus. Both are due to Codd [15]. The relational calculus comes in two variations: the tuple relational calculus (TRC), and the domain relational calculus (DRC). Both variations are very similar. The main difference is that in TRC variables take on tuple values, and in DRC variables take on field values.

In the Database Management Systems book of Ramakrishnan et al. [66] a clear distinction is made between RA and the two variants of relational calculus, TRC and DRC. The two relational calculi are *declarative*. A relational calculus expression only specifies the properties of the results, not how it should be computed. RA is *procedural* in that specifying a RA query also involves specifying in what order operations are performed. This is also reflected in the fact that relational database systems often use relational algebra expression to represent their query evaluation plans.

All RA queries can be expressed in relational calculus. If we restrict ourselves to *safe* queries in the calculus, the converse also holds. Safe queries are queries that return a finite result. An example of an *unsafe* query is $\{S \mid \neg(S \in R)\}$, it asks for all tuples S such that S is not in the given instance of R . The result set S

of such tuples is obviously infinite in the context of infinite domains (such as all integers). A query language that can express all queries that can be expressed in relational algebra is called *relationally complete*.

We will introduce the two formal query languages by means of a simple example query taken from the Database Management Systems book:

Find the names of the sailors who have reserved boat 103.

It queries the following relations: a relation *Sailors* with tuples $(sid, sname, rating, age)$ and a relation *Reserves* with tuples (sid, bid, day) , and a relation *Boat* with tuples $(bid, bname, color)$

Relational algebra is compositional, meaning that every operation in the algebra accepts one or two relations and returns a relation as result. This makes it easy to compose a complex query. RA operations include the standard operations from set theory (e.g., union, intersection) complemented with operations such as:

- *Projection* (π) and *Selection* (σ): both operations work on data in a single relation. Projection can be used to influence the rows of a relation, selection is used to select rows in a relation.
- Various kinds of *joins* (\bowtie): a join is used to combine information from two or more relations by taking the cross product of the relations followed by a selection based on the join condition. The most common join is the natural join in which the join condition consists solely of equalities on common fields of both relations.

$$\pi_{sname}((\sigma_{bid=103}Reserves) \bowtie Sailors)$$

Tuple relational calculus is essentially a restricted subset of first order logic. It is more declarative than RA. The most common logical constructs that build a TRC formula are the standard logical connectives and the logical quantifiers \exists and \forall .

$$\{P \mid \exists S \in Sailors \exists R \in Reserves (R.sid = S.sid \wedge R.bid = 103 \wedge P.sname = S.sname)\}$$

This query can be read as: "Retrieve all sailor tuples for which there exists a tuple in Reserves, having the same value in the *sid* field, and with *bid* = 103." That is, for each sailor tuple, we look for a tuple in Reserves that shows that this sailor has reserved boat 103. The answer tuple P contains just one field, *sname*.

Domain relational calculus is very similar to tuple relational calculus with the difference that free variables can range over field values:

$$\{\langle N \rangle \mid \exists I, T, A (\langle I, N, T, A \rangle \in Sailors \wedge \exists \langle Ir, Br, D \rangle \in Reserves (Ir = I \wedge Br = 103))\}$$

In this DRC expression only the *N* is a free variable.

2.3.2 Tarski's binary relational calculus

Although similar in name, Tarski's binary relational calculus [68] (BRC) is not the same as Codd's RA, TRC, and DRC. It is unclear what the precise relation between Codd's and Tarski's work is. Feferman and Bussche both investigated the history and influence of Tarski's work [28, 24]. Nonetheless, neither of them has succeeded in obtaining a clear view on the relation between Tarski's and Codd's work.

Just like RA, BRC is procedural and compositional. A BRC query specifies the order in which operations are performed, and every BRC operation can be composed with other BRC operations. Curiously, if we conform to the naming conventions used in the previous section, "binary relational *calculus*" should actually be called "binary relational *algebra*". Could it be that either Tarski or Codd used an incorrect name for their calculus or algebra? After a short survey in the literature and on the internet, we found that (as expected) both are right. There seems to be no consensus on the definition of algebra and calculus. In this thesis we will use the definition of the previous section: an algebra is procedural, a calculus is more declarative.

An important difference between RA and BRC is the arity of the relations: Codd's algebra supports n -ary relations, where Tarski's calculus only supports binary relations. The consequence is that BRC is less expressive than RA, which implies that BRC is not relational complete. An example of this is that in BRC it is not possible to express queries that search for graph patterns involving more than two nodes. An example of this can be found in Section 3.3.3 in which we show the implementation of the detection of the degenerate inheritance pattern (involving three nodes) in different tools.

In BRC, relations are defined as follows: Let X, Y be finite sets. A binary relation between X and Y is a finite set of pairs where the first element of the pair comes from the set X , and the second from Y . So, a binary relation R between X and Y is a relation such that $R \subseteq X \times Y$. We write xRy to denote that the pair $(x, y) \in R$. In the special case that $R \subseteq X \times X$, i.e., a relation for which domain and range coincide, we call a relation *homogeneous*. Homogeneous relations can be viewed as directed graphs. An arbitrary relation $R \subseteq X \times Y$, where $X \neq Y$, is called *heterogeneous*.

Tarski's gives an axiomatic specification of a set of operations for manipulating relations. These operations are limited to the standard set-theoretic operations (e.g. union, intersection) complemented with relational (or sequential) composition and inversion. Various researches have used this set as basis for their work. In many cases these researchers added extra operations. In RPL, the general purpose relational programming language by MacLennan, operations such as restriction, exclusion, and transitive closure are introduced. Some researchers (e.g., Holt, Klint) have used Tarski's works as basis for their code query languages. Just like MacLennan, they include transitive closure operations to be able to express recursive constructs. Below we list some of the operations, including their definition:

Name	Symbol	Definition
union	\cup	$x(R \cup S)y \Leftrightarrow xRy \vee xSy$
intersection	\cap	$x(R \cap S)y \Leftrightarrow xRy \wedge xSy$
difference	\setminus	$x(R \setminus S)y \Leftrightarrow xRy \wedge \neg(xSy)$
complement	c	$x(R^c)y \Leftrightarrow \neg(xRy)$
relational composition	\circ	$x(R \circ S)y \Leftrightarrow \exists z.xRz \wedge zSy$
inverse	$^{-1}$	$x(R^{-1})y \Leftrightarrow yRx$
domain restriction	\ll	$x(S \ll R)y \Leftrightarrow x \in S \wedge xRy$
range restriction	\gg	$x(R \gg S)y \Leftrightarrow y \in S \wedge xRy$
domain	dom	$x \in (dom R) \Leftrightarrow \exists y.xRy$
range	rng	$y \in (rng R) \Leftrightarrow \exists x.xRy$
carrier	$carrier$	$x \in (carrier R) \Leftrightarrow x \in dom R \vee x \in rng R$
cardinality	$\#$	$\#R \Leftrightarrow primitive\ to\ calculus$
transitive closure	$^+$	$x(R^+)y \Leftrightarrow xRy \vee \exists z.xRz \wedge zR^+y$
reflexive transitive closure	*	$x(R^*)y \Leftrightarrow x = y \wedge \exists z.xRz \wedge zR^*y$

Table 2.2: Overview of common relational operations

The above list is by no means complete, there are many more operations that are useful when using relational calculus in the context of software analysis. For example, the Rscript manual [47] lists over 30 operations and functions applicable to relations.

In [62] coreflexive relations (fragments of the identity relation, that is, $R \subseteq id$) are used to model predicates or sets. The meaning of a predicate p is the coreflexive relation $\llbracket p \rrbracket$ such that $b \llbracket p \rrbracket a \equiv (b = a) \wedge (p a)$. This is the relation that maps every a which satisfies p onto itself. The meaning of a set $S \subseteq A$ is the meaning of its characteristic predicate $\llbracket \lambda a.a \in S \rrbracket$, that is, $b \llbracket S \rrbracket a \equiv (b = a) \wedge a \in S$. We adopt the notation $\llbracket p \rrbracket$ for predicates, but with the distinction that here predicates are modelled as sets as opposed to coreflexive relations.

2.3.3 Logic programming

A program written in a logic programming language consists of a set of axioms and a goal statement called the query. In the logic programming model the programmer is responsible for specifying the basic logical

relationships and does typically not specify the manner in which the inference rules are applied.

One of the most well-known is Prolog. Prolog implements a subset of second-order logic (that is, it can deal with sets as well as atomic propositions), and the flexibility of the language permits propositions which lie well outside the boundaries of any classification of formal logical systems.

The following example is a classical logical programming example, written in Prolog:

```

1 sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).
2
3 parent_child(X, Y) :- father_child(X, Y).
4 parent_child(X, Y) :- mother_child(X, Y).
5
6 mother_child(trude, sally).
7
8 father_child(tom, sally).
9 father_child(tom, erica).
10 father_child(mike, tom).

```

Based on these axioms the following query is evaluated as true:

```

1 ?- sibling(sally, erica).
2 Yes

```

Datalog is a language for querying deductive databases that syntactically is a subset of Prolog. Deductive databases consist of an extensional and intensional part. The extensional part contains facts, and the intensional part contains logical rules from which new facts can be deduced by logical inference.

Datalog is relationally complete, and in addition it allows recursive queries which make it a more expressive language than RA. Query evaluation with Datalog can be done in polynomial time. In contrast to Prolog it disallows complex terms as arguments of predicates, e.g. $P(1, 2)$ is admissible but not $P(f1(1), 2)$, and it imposes certain restrictions on the use of negation and recursion.

2.4 Implementation of source code query languages

For the implementation of the code query languages there are two main approaches: main memory and disk based implementations. By disk based implementations we mean all the implementations that use a DMBS to store and query the relations. Main-memory implementations need to load the complete relation into memory to perform operations. Table 2.1 on page 11 shows for each of the languages the implementation it uses.

2.4.1 Main-memory implementations

The main memory implementation techniques differ in the data structures and algorithms that are used. Two examples of this are arrays and binary decision diagrams:

Grok uses arrays to stores the edges in a relation which is a relatively simple data structure. As Holt explains: "Grok uses three arrays, called *rel* (relation), *src* (source), and *trg* (target). Edge $R(A, B)$ is stored in some row i as $rel(i) = r$, $src(i) = a$, and $trg(i) = b$, where r , a , and b are 32-bit hashes of R , A , and B respectively." Holt states that the simplicity of this data structure simplifies maintenance and enhancement of Grok.

A more advanced data structure is a Binary Decision Diagram [12] (BDD). A BDD is a data structure that allows for compact representation and efficient manipulation of large relations. A additional advantage is that it can represent n -ary relations. It is used in RelView and Crocopat.

An advantage is that main-memory implementations are generally fast and relatively easy to implement. A disadvantage of main-memory implementations is that it is necessary to load complete relations into main-memory before operations can be performed on it. Because in source code querying this are often very large

relations, one runs the risk of running out of main-memory or suffer from low performance because the OS needs to resort to swapping.

2.4.2 Disk-based implementation

Several tools (e.g. CodeQuest, SemmleCode, OMEGA) are implemented using a DBMS. These tools either directly use SQL for querying the information in the database, or use a language that is translated to SQL. The advantage of this approach is that databases have been designed to work with large amounts of data with limited memory resources available. This makes them ideal for storing and querying the typically large amounts of data involved in source code querying. An additional advantage of using this approach is that one directly profits from all the research effort that has gone into performance optimisation of relational databases.

A disadvantage is that SQL does not support recursive queries. Moreover, it has proven to be difficult to efficiently implement recursive queries in a DBMS. Beyer compared the performance of the transitive closure operation in Crocopat and several other code querying tools, including SQL[10]. The results show that the transitive closure operation implemented in SQL performs poorly. However, note that this SQL query was not optimised.

SemmleCode does perform a substantial amount of advanced (proprietary) optimisations when compiling .QL to standard SQL. Currently, they are working on an implementation that is based on a hybrid approach combining DBMS and main-memory implementations. Unfortunately, there is no data available on the performance of SemmleCode. Due to licensing issues we are not allowed to publish any performance results concerning SemmleCode.

2.5 SIG's software analysis toolkit

To efficiently assess, monitor, and generate documentation for all kinds of software systems, SIG developed an in-house software analysis framework called the software analysis toolkit (SAT).

These were the most important requirements for the SAT:

- support multiple programming languages
- deal with incomplete source code
- scale to large systems (> 500.000 LOC)

SIG's software analysis toolkit conforms to extract-abstract-present paradigm discussed in Section 2.1.2:

For the *extraction* of relations from the source code SIG relies on several different parsers and lexers. Since SIG analyses systems implemented in many different programming languages, no single parser and lexer pair suffices to efficiently parse all the languages SIG encounters. Therefore, SIG currently uses SDF (Syntax Definition Formalism [34]) in combination with JJForester [49], and ANTLR (ANother Tool for Language Recognition [63]). The parsers are constructed so that they are able to cope with incomplete source code. When SIG encounters a programming language it has not analysed before, the SAT can be extended with parsers for the new language. These parser are implemented using SDF or ANTLR. During the parsing of the source code, both SDF+JJForester as well as ANTLR build a Java object representation of the abstract syntax tree (AST). In addition, visitor classes are generated that can be used for analysing the AST. For the complete system, the obtained information is represented as a graph. In this graph source files are vertices, and the AST of the files are attributes of the nodes.

When the parsing of the source is finished, we enter the *abstract* phase. A lot of analyses are performed on AST level. Examples of analyses are the computation of structural and complexity metrics, or the counting the number of lines of code in a unit. Results of the analyses are stored as attributes of the graph's vertices. Later iterations can use the information in the attributes to calculate new information, which in turn can be stored in the graph.

The results can be *presented* in a web-based monitor. This monitor provides textual and visual reports of the abstracted information as coloured text, tables, and charts.

2.5.1 The SIG graph library

SIG's graph library represents typed, directed graphs. That is, edges are directed, and can be of different types. The graph abstract data type is realised with a concrete data structure similar to the adjacency list data structure. The representation stores vertices explicitly as objects in a container. A vertex object v holds references to containers that store references to edges in-coming and out-going on v . An edge e holds references to the vertex objects associated with e , and contains a string that represents the type of the edge.

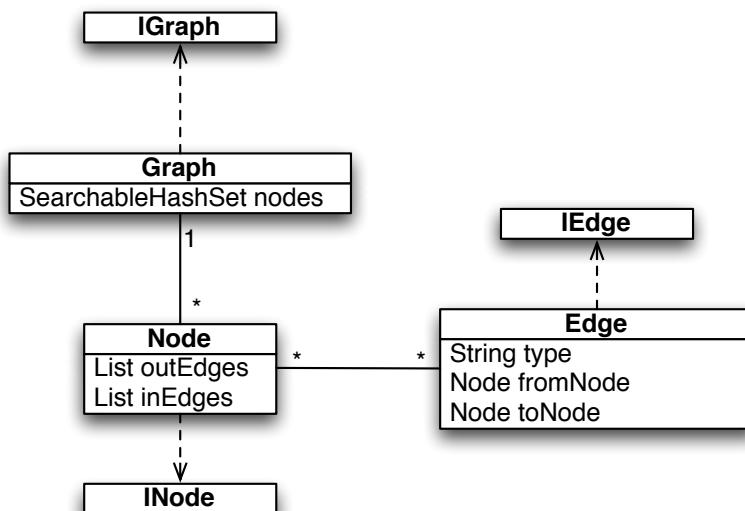


Figure 2.3: Simplified UML diagram of the SIG graph representation

The SIG graph library defines methods that implement graph operations like slicing and cycle detection. In addition, it provides abstract visitor classes that can be extended to conveniently implement new analysis. Figure 2.3 show a simplified UML diagram of the SIG graph representation.

Chapter 3

Comparison of code query technologies

Code query technologies can differ in many essential ways: some only provide means for querying code, but leave extraction and presentation to be provided by other tools, some support the whole paradigm. Some tools provide a separate language for writing the queries while others provide only a library to be used from a host programming language. The languages that are provided differ in their expressiveness, abstraction facilities, declarativeness, and conciseness. In this chapter we compare five alternative solutions for querying source code: CrocoPat [9], Rscript [46], JRelCal [70], SemmleCode [73], and GReQL 2 [45]. For that purpose, we compared the tools with respect to ten criteria focussing on language and tool features.

This comparison informs potential users of a code query technology on the pros and cons of the available alternatives. This assists the potential users in selecting the technology that most closely fits their requirements. Furthermore, we hope that our comparison will stimulate changes in the field. Ideally, the results of this comparison will be used by the developers of the tools to decide on the directions in which to further improve their tools.

Originally, we started this comparison with a specific goal in mind: find the best code query technology for use at the SIG as part of their in-house extendible framework for software analysis. Nonetheless, we have set up this comparison as a generic and objective study. This makes it valuable for people outside the SIG. To cater for our original goal we have included a separate conclusion in addition to the general conclusion. This conclusion describes which tool most closely meets the specific requirements of the SIG. To compare the SIG graph library with the other tools, we have included SIG graph library implementations of the benchmark queries.

This comparison of query technologies has been presented and published in a position paper [3] at the workshop "Query Technologies and Applications for Program Comprehension" (QTAPC) at the International Conference on Program Comprehension (ICPC 2008). In addition, there is currently work in progress on an extended version of the QTAPC paper. Note that work that has originally been done for this thesis has been incorporated in both papers, and vice versa: insights gained during writing the papers are incorporated in this thesis.

3.1 Code query technologies

In this section we give a detailed introduction of the tools that we have included in the comparison: CrocoPat, Rscript, JRelCal, SemmleCode, and GReQL 2. We have included specifically these tools because they either represent one of the latest developments in their approach, or they have a relevant feature that is not present in other tools. There are several reasons for excluding a tool from this comparison: the tool is not maintained anymore, there is a newer tool that uses the same approach, it is very similar to a tool that is already included in the comparison, no working version of the installation is available, or there are problems with the installation.

3.1.1 Crocopat

The implementation of Crocopat started in May 2002 by Dirk Beyer after submitting his PhD thesis. Done as his first postdoc project while at the University of Cottbus, Germany, Crocopat was meant to replace the SQL

front-end of the Crocodile reengineering tool developed in the same university. The first ideas of CrocoPat are described in a technical report from 2003 [7] and the first publication dates from that same year [9].

CrocoPat is an interpreter for programs written in the RML language. RML is an imperative relational programming language, the relational expressions in RML are based on predicate logic. Besides relational expressions in the form of predicate logic, the language includes control structures, numerical expressions, and some basic input and output facilities.

The main focus of the CrocoPat project was on generality and efficiency. In CrocoPat, efficiency (in terms of memory usage and runtime) is realized by internally representing relations as Binary Decision Diagrams [12], a data structure that allows for efficient representation and manipulation of large relations. Generality can be found in the fact that in RML it is possible to work with n -ary relations. According to the authors, the advantage of n -ary relations is that it allows the user to easily express queries that search for graph patterns of arbitrary size, e.g., cycles greater than length 4.

3.1.2 Rscript

Rscript is a small scripting language based on the relational calculus. It was developed in 2002 by Paul Klint at the Centrum voor Wiskunde & Informatica (National Research Institute for Mathematics and Computer Science), in Amsterdam, The Netherlands. It is developed as part of a framework for language development, source code analysis and source code transformation, but is currently also available as standalone tool. The first publication mentioning RScript dates from 2003 [46].

The most distinctive features of Rscript are the support for functions and the support for type variables in function declarations (parameteric polymorphism). Rscript also supports comprehensions which allow constructing sets and relations using generators (or enumerators) and boolean-valued expressions. Rscript uses sets and relations, and allows nested sets and relations. It is based on binary relations only, but has some syntactic support for n -ary relations. However, it does not support n -ary relations with labeled columns as in SQL.

Rscript is currently in development; the authors are working on the documentation and improvement of the performance of the tool.

3.1.3 JRelCal

The JRelCal project was started with the goal to make an efficient Java implementation of RScript datatypes and its operations. JRelCal was born in 2007 in the context of the PhD thesis of Tijs van der Storm, under supervision of Paul Klint. As opposed to JGrok which is a re-implementation of Grok in Java, JRelCal is not meant to be a re-implementation of the Rscript language but an API with the same functionality.

3.1.4 GReQL 2

In the context of Daniel Bildhauer's PhD, GReQL 2 started off in 2007 as the successor of the GReQL (Graph Repository Query Language) project. Both GReQL and GReQL 2 were developed at the University of Koblenz and Landau, Germany, in the research group of Jürgen Ebert. The initial ideas on GReQL were conceived during the summer of 1994 and the implementation started in 1995. It is based on the graph constraint language GRAL, developed in the same research group.

The most distinctive feature of GReQL 2 is regular path expressions. A path expression describes a path in the graph. Path expressions can be used to test if pairs of nodes are connected via the specified path. A path expression also denotes the set of nodes which can be reached from a starting node via the path.

The first references to GReQL can be found in a pair of technical reports dating from 1996 and 1997 respectively [44, 20]. The first publication on GReQL dates from 1998 [45]. GReQL 2 was first referred to in 2008 in [11].

3.1.5 SemmleCode

In contrast to the other compared code query technologies, SemmleCode is not an academic project, but being developed and commercialized by a company, Semmle Ltd. The development on SemmleCode started in December 2006, the same date when the company was founded by Oege de Moor. The ideas were presented for the first time in January 2007 [22]; the first publications referencing SemmleCode are from the same year [73, 23].

SemmleCode is mainly based on the ideas published in the dissertation of Elnar Hajiyeu: *CodeQuest - Source code querying with DataLog*[33]. Another difference with the other tools, is that it uses a relational database to store and query relations. It is provided as an Eclipse plugin that allows you to query the source code in a Java project using the query language .QL. It is similar to SQL, with the distinction that it adds a transitive closure operation and object orientation. Queries written in .QL are optimised, compiled into SQL and can then be executed on any major relational database management system.

3.1.6 Other technologies

In this section we explore some of the code query technologies we have considered for inclusion in our comparison, but for various reasons decided to exclude from the full comparison. We discuss these tools and the reasons for their exclusion.

RelView

A first implementation of RelView was made in 1989 by Gunther Schmidt, Hans Ziesrer and Rudolf Berghammer after moving to the University of German Federal Armed Forces, in Munich, Germany. RelView was based on previous work on relational algebra done by the authors at the University of Munich. The first reference to RelView appears in a 1989 technical report [1] and the first publication on RelView dates from 1991 [5]. Although RelView was intended to be a tool for the interactive creation and visualization of relations and the prototyping of graph algorithm, it has been used for a broad range of applications, including software analysis [31].

RelView is based on the KURE C library¹, a library written in C for the manipulation of graphs. KURE-Java is a port of KURE to the Java platform. It is a wrapper around the KURE library; it utilizes JNI (Java Native Interface) to call methods from the C library. Currently KURE-Java is only available for Windows. RelClipse² is a port of the original RelView program to an Eclipse version. RelClipse uses the KURE-Java library. Just like Crocopat, RelView uses BDD representations of graphs for efficiency.

We have decided to exclude RelView from the tool comparison for several reasons: it is not actively maintained, the tool is only available for Linux, and the supported interchange format is poorly documented.

Grok & JGrok

Grok is an interpreter for binary relational operators. It was developed as a research prototype in 1995 by Holt at the Computer Systems Research Institute, University of Toronto, Canada. Grok was implemented in the Turing Programming Language [40] also developed by Holt. The first use of Grok was to manipulate graph models to aid reverse engineering the software architecture implicit in the source code. Grok was first referenced in a 1996 technical report [37] and the first publication dates from 1998 [38].

Grok does not provide control constructs; it provides the basic operators of binary relational calculus such as relational composition and transitive closure. It supports reading from and writing to RSF (Rigi Standard Format³) and TA files. TA is the Tuple-Attribute language, a custom language by Holt. In TA, tuples can be attributed with graph drawing information. It can also serve as a plain data interchange format.

¹<http://www.informatik.uni-kiel.de/~progsys/relview/kure>

²<http://ls10-www.cs.uni-dortmund.de/index.php?id=137>

³<http://www.rigi.cs.uvic.ca/downloads/rigi/doc/node52.html>

Based on Grok, JGrok⁴ started in 2000 in the context of Jingwei Wu’s PhD project, under supervision of Ric Holt. JGrok supports a richer set of operations while Grok has better performance.

Both Grok and JGrok are research prototypes and are not actively maintained anymore. Therefore, we have decided to exclude them from the tool comparison.

3.2 Usage scenarios and criteria

We consider a total of ten criteria on which to compare the chosen tools. These criteria are largely derived from two typical usage scenarios: one in which the tool is used directly for interactive investigation, and one in which the tool is to be used from other software.

The criteria can be divided into two categories: the language criteria that are related to the query language, and the tool criteria that are related to the tool as a whole. The language related criteria are paradigm & characteristics, type system, abstraction facilities, and extendability. We shall compare the languages of the tools for these criteria based on a total of four benchmark queries. These benchmark queries are examples of queries that were specially chosen as diverse as possible, but still within the area of source code analysis. The tool criteria we address are user interface, API support, output formats, interchange formats, extraction support, and licensing.

3.2.1 Usage scenarios

We identify two usage scenarios: interactive use and tool integration.

In the former, the user interacts directly with the code query technology according to the extract-abstract-present paradigm. An example of this scenario is the situation where a user tries to comprehend a software system. In such a situation it is hard to predict beforehand the queries that are going to be needed to gain sufficient information from the system. That is why interaction is so important.

In the tool integration usage scenario, code query technologies are used in combination with other components and tools to develop new tools, e.g., those that deliver high value-added software services. An example of such a tool is a daily build system for source code monitoring which sends email messages whenever specific metrics exceed predefined thresholds. This scenario is best supported by providing an API for the programmers to execute queries and support for the construction of extractors.

Table 3.1 summarizes the criteria and their importance for the usage scenarios.

Table 3.1: The criteria and their importance in the usage scenarios. We use a + when the criterion is important for the scenario, a 0 when it is relevant but not important, and a – when it is not important.

<i>Scenarios vs. Criteria</i>	<i>Language criteria</i>				<i>Tool criteria</i>					
	<i>Paradigm & characteristics</i>	<i>Type system</i>	<i>Abstraction</i>	<i>Extendability</i>	<i>Output formats</i>	<i>User interface</i>	<i>API support</i>	<i>Interchange format</i>	<i>Extraction support</i>	<i>Licensing</i>
<i>Interactive use</i>	+	+	+	+	+	+	-	+	+	+
<i>Tool integration</i>	+	0	0	0	-	-	+	0	+	+

⁴<http://www.swag.uwaterloo.ca/jgrok/index.html>

3.2.2 Language criteria

In this section we will discuss the criteria on which the language comparison is based: paradigm & characteristics, type system, abstraction facilities, and extendability. We will also mention the importance of the criteria with respect to the two usage scenarios we have defined previously.

Paradigm & characteristics

This criterion covers the programming paradigm and other distinctive properties that characterize the query language.

A programming paradigm affects the way we design, organize, write, and think about programs⁵. Paradigms differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, etc.) and the steps that compose a computation (assignment, comparison, etc.). Note that many programming language support multiple paradigms, e.g., Java supports the imperative, object-oriented, and generic paradigm.

The characteristics we are interested in are the formalism(s) the language is based on, the conciseness of the expressed queries, and characteristic language constructs that are offered. In addition we would like to know whether a query language is easy to learn. We will refrain from making hard statements about this because such statements would require empirical evidence. Other language characteristics such as type system, abstraction, and extendability are so important that we have decided to make them into separate criteria. These criteria will be discussed in the following sections.

Which paradigm and characteristics are the best largely depends on personal taste and the context it is used in.

In both the interactive and tool integration scenario queries are written in the query language of the code query technology. Since the functionalities of a code query technology are exposed through its language, the "paradigm & characteristics" criterion is important in both scenarios.

Type system

There are three important advantages of a type system:

1. A type system can help verifying the partial *correctness* of a program by supplying information against which it is possible to check the correctness of expressions.
2. It can enhance the *readability* of a program. Typed language constructs such as parameters, functions, and variables can be seen as a form of documentation. This facilitates comprehending a program.
3. The *efficiency* of generated code can be improved by taking advantage of the data properties that becomes available through the type system.

For the query programmer only the first two advantages are relevant. The third advantage is a direct advantage to the compiler builder who wants to implement automatic performance optimisations. In the end, these optimisations are an advantage to the query programmer.

We would like to know whether the typing of the query language is strong or weak. Furthermore, we would like to know whether the type system supports primitive types, e.g., string, integer, and boolean, and other (non-primitive) types such as location types that facilitate navigating back to the original location in the source code.

In the user scenario the existence of a type system is very important since it will make some of the discussed advantages available to the query programmer. In the tool integration usage scenario, on the other hand, this is relevant but not very important. The reason for this is that it is possible to emulate a type system using the functionalities of the host programming language.

⁵http://en.wikibooks.org/wiki/C_Programming/Programming_Paradigms

Abstraction

The main advantage of abstraction is that it facilitates reuse of queries. We identify two types of abstraction: abstraction by parametrization and abstraction by polymorphism.

Abstraction by parametrization enable us to write queries whose behavior depend on the arguments. An example of this is a slicing algorithm that can be parametrised with the starting criteria.

Abstraction by polymorphism (or parametric polymorphism) goes even further in terms of possibilities for reuse. Parametric polymorphism is obtained when a function works uniformly on a range of types. This allows the user to write queries that can be reused for parameters of different types. E.g., a generic lifting function that can be used to lift call graphs from method to class level and from class to package level, or a generic slice function that can slice graphs of any type.

Just like the type system criterion, this criterion is more relevant for the interactive usage scenario than for the tool integration. For interactive usage, the absence of abstraction facilities will make it impossible to define functionality in such a way that it can be reused, and vice versa, make it impossible to reuse functionality that has been defined earlier. For the tool integration scenario, this can, in principle, be compensated for by the host programming language.

Extendability

In [27] extendability is defined as: *The ease with which a system or component can be modified to increase its storage or functional capacity.*

For code query technologies we distinguish two forms of extendability: model extendability and language extendability.

Users may want to add new operations or constructs to the query language. For this purpose it is desirable that it is possible (and relatively easy) to extend the language of the code query technology with new functionality. Not only it is important that extending the language is technically possible, it should also be legally possible, i.e., the license of the software should allow such modifications.

In model extendability, the user can build his own model or extend an existent model such that he can create abstractions close to his problem domain. Model extendability is only relevant for technologies that offer a model for the facts extracted from the source code.

The possibility to define functionalities in such a way that they can be reused later can also be considered to be an extension to the language. However we have already covered this under the header of abstraction. For the criterion extendability we are only interested in extensions to the language or model.

This criterion is only important to the user usage scenario. In the tool integration scenario, however, adding new functions can be easily done in the host programming language. However, if it is necessary to import many functions it could be handy to have available how to import a custom defined module automatically.

Note that there are several reasons that could make it impossible to extend the language or tool: Extending a language can be a non-trivial task that requires compiler technology knowledge, such knowledge may not be available in a commercial enterprise. Moreover, not all authors may allow the extension of their tool. And finally, the source code may not be available, so that extending the language is impossible.

3.2.3 Tool criteria

In this section we define the criteria we have used in our comparison of the tools. Clearly, there are many criteria we have not included, often because we could not come up with an objective measure for evaluating them. Still, certain tools have particular properties that we would like to point out. Even though we do not address such properties for all of the tools, we think mentioning them is in order. We have put these under the general heading of *Other issues*, in the tool comparison section.

Output formats

In the tool integration scenario, the host language can compensate for shortcomings in the way that results are presented to the user. Not so in the interactive scenario. Therefore, this criterion considers the output formats provided by the tools, e.g., charts, graphs, plain text (with or without Comma Separated Values), and XML. A special category for the technologies we consider is whether the tools provide explicit links to locations in the source code as part of the result of the query. Among the output formats we may also consider the file types that the tools use for storing their facts and relations, but we have decided to put these under the heading of Interchange formats below.

User interface

This particular criterion considers the availability and maturity of an interactive interface to the tool and is therefore very important for the interactive use scenario.

User interfaces come in a number of forms: a tool may be accessed through a command-line interface, it may provide a complete (graphical) IDE, or both. A graphical IDE makes it possible to offer user friendly features such as automatic syntax completion, and graphical presentation of results. Note that a graphical IDE does not have to be a standalone implementation: an alternative would be to provide a plugin for an existing IDE such as Eclipse.

API support

This criterion considers the effort a programmer needs to make to call the code query technology from other software, and thus contrasts with the User interface criterion discussed earlier. This criterion is particularly important for the tool integration usage scenario. We distinguish three different levels of API support:

At worst, there is only an *interactive interface* (GUI), and communication with other tools then typically has to be performed manually. If interchange support is also low, this will make the process of combining the technology with other tools even more problematic.

If a command-line *runtime* and *compiler* are provided, then queries can be executed via the command-line and results can, either directly through pipes or indirectly via files, be communicated to the invoking tool. A disadvantage of this approach is that queries must be passed as flat strings and results passed back need to be parsed by the invoking tool. Also, invoking queries by starting a new process can induce quite a bit of overhead on some operating systems.

The most advanced form of integration is possible when the code query technology is provided in the form of an *API*. In this case, there is no overhead for invoking queries. We do have to make a distinction here between an API that supports only a *call-level interface*, i.e., queries are passed to the API in unstructured form, or the case that the API is designed as an EDSL. In the latter case, the programmer can benefit from the typing facilities and other consistency checks provided by the host language. This enables compile-time checks on the code queries. However, this is only the case when the API is written in the host language it is used in. (Actually, this is already imposed by the definition of an EDSL.)

Interchange formats

Like any other field, code querying technologies can benefit from being able to interchange information. We distinguish two situations:

It may well be that a user wants to analyse the language *X* with his favourite tool *Y*. However, *Y* does not provide an extractor for *X* and building extractors is typically a lot of work. Now if it happens that a tool *Z* supplies such an extractor, and *Y* and *Z* share the same format for storing extracted facts, then the user does not have to program extractors for *X* in *Y*.

This is actually a specific case of a more general phenomenon, in which the user may want to alternate computations within *Y* and *Z*, for example, because *Y* is optimized for certain computations, *Z* is optimized for others, and the user needs to perform both kinds.

In the field of code querying technologies we have found two standard formats: the Rigi Standard Format (RSF) and the Graph Exchange Language (GXL) [41]. We note that converters between the two exist. A few tools provide their own, non-standard, format.

Extraction support

This criterion considers whether the tools comes with its own fact extractors, and if so, for which programming languages. Extraction support is related to interchange support, in the sense that a tool that does not come with any extractors often supports an interchange format, and vice versa. With respect to extraction support in combination with the interchange format criterion we distinguish four situations:

The tool does not come with any extractor. These tools all support an interchange format that makes it possible to import facts from every extractor that can export files to this format. The majority of the tools follow this approach. Although this is a flexible approach, we found that in practice there are very few good extractors publicly available.

The tool comes with its own extractors, but it does not allow third parties to provide their own fact extractors. The advantage is that this relieves the user from the task of finding a good fact extractor. The obvious disadvantage is that it limits the user to querying the languages for which fact extractors are available for the tool.

Another situation is that the tools comes with both extractors and support for an interchange format, in this comparison we have not encountered this. An impractical situation would be that the tool comes without extractors and without support for an interchange format.

Licensing

This criterion considers the license the technology is licensed under. Licensing can be an issue when adopting the technology in industry. Particularly if the user of the technology wants to be able to adapt the technology to his own needs, or wants to include the technology into proprietary software.

A technology is *proprietary* if the company has reserved some measure of control over the software. Otherwise, the technology is *free*; in the case of software, *open source* is often used as a synonym. One of the major implications of proprietary software is that the source code is not made available to its users. In this case, changing the software is typically not allowed, or even impossible.

Within the open-source community, various *licenses* exist. A *BSD license* is a very permissive license that allows the inclusion of the licensed material into proprietary software. The *GNU LGPL (GNU Lesser General Public License)* also allows inclusion into proprietary software, but with rather subtle restrictions. It is typically used for libraries. The difference with the less permissive *GNU GPL (GNU General Public License)*, is that software under GNU GPL may not be combined with proprietary software at all.

3.3 Language comparison

While tool feature comparison is in essence checking for the presence or absence of features, language comparison is a more subjective and subtle task. In order to prevent this comparison from becoming a subjective overview we have selected a set of code queries that cover a large spectrum of software analysis queries: lifting of a call graph [29], detection of the degenerate inheritance pattern [6], computation of the package instability metric [57], and forward graph slicing [74]. These benchmark queries allow us to objectively compare the expressiveness, abstraction, style/paradigm, and extendability of the languages.

In the following sections we will for each benchmark query introduce the query supported by a diagram, and then we will show and discuss the implementation of the query in each of the five tools. In addition, we also include code fragments that show how the query could be implemented using the SIG graph library.

3.3.1 Lifting

When analysing large software systems, views on different levels of abstraction are needed. The canonical example is that of a call graph on method level that is lifted to a higher level. The following diagram depicts the particular case where the method calls are lifted to the class level.

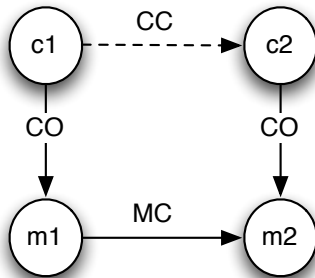


Figure 3.1: Call graph lifting

We take CC , CO , MC to denote the "class call", "class of", "method call" relations respectively. This can be expressed in pointfree binary relational calculus as follows:

$$CC = CO \circ MC \circ CO^{-1}$$

In the following sections we show how to express this query in the tools.

Crocopat

```

1 CC(c1, c2) :=
2   EX(m1, EX(m2, CO(c1, m1) & CO(c2, m2) & MC(m1, m2)));

```

Code Fragment 3.1: Lifting in Crocopat

Crocopat offers first order logic to express queries: EX denotes the existential quantifier and $\&$ denotes conjunction. The variables $c1$ and $c2$ are bound on the left hand side of the assignment, $m1$ and $m2$ are bound by their respective existential quantifier.

Rscript

In Rscript we can express a query in either pointfree binary relational calculus (Code Fragment 3.2) or using comprehensions (Code Fragment 3.3).

```

1 rel[class, class] lift(rel[method, method] MC, rel[class, method] CO) =
2   CO o MC o inv(CO)

```

Code Fragment 3.2: Rscript - relational calculus

In Code Fragment 3.2 the binary relational expression for lifting is defined in a function `lift` that takes two parameters: the original relation, and the relation that relates the original relation to the type it has to be lifted to. Both `class` and `method` are type synonyms (or alias types) for the string type. In Rscript, a type synonym can be declared as follows: `type class = str`. In the method body the relational expression that expresses the lifting can be found. As can be seen this is a literal translation of the expressions we have shown in the explanation of the lifting query in the beginning of this section. Here, the \circ operation denotes relational composition, and the `inv()` method inverses a relation.

```

1 rel[&T, &T] lift(rel[&S, &S] MC, rel[&T, &S] CO) =
2   {<C1, C2> | <&S M1, &S M2> : MC, <&T C1, &T C2> : CO[M1] x CO[M2]}

```

Code Fragment 3.3: Rscript - comprehension

Code Fragment 3.3 shows a polymorphic definition of lifting. In the function definition the parameters for the function are defined using type variables that are denoted as `&` followed by an identifier. In this way the functions is defined for arbitrary types. The function body is expressed using the comprehension construct: `CO[M1]` is the right image of the `CO` relation with respect to `M1`, `x` denotes cartesian product.

JRelCal

The JRelCal version of lifting defined in the body of the method `lift` is a direct translation of the binary relational calculus expression in Rscript, Code Fragment 3.2. The method is defined generically, making it applicable to relations of arbitrary Java types.

```

1 public static <T, S> Relation<T,T> lift(Relation<S, S> MC, Relation<T, S> CO) {
2   return CO.compose(MC).compose(CO.inverse());
3 }

```

Code Fragment 3.4: Lifting in JRelCal

SemmlCode

As can be seen from the code fragment below, `.QL` is very similar to SQL. The object orientation can be observed in the query below: `Class` is the type (or class) of the `c1` and `c2` objects. `getACallable()` is a method invocation, returning an object of the type `Callable`.

```

1 from Class c1, Class c2
2 where c1.getACallable().calls(c2.getACallable())
3 select c1, c2

```

Code Fragment 3.5: Lifting in SemmlCode

GReQL 2

Just like `.QL`, GReQL 2 uses syntax similar to SQL. The syntax used in the `with` clause (corresponding to `where` in SQL) uses an arrow like syntax. Note that the structure of the arrow expression in the `with` clause of this query resembles the diagram in Figure 3.1.

```

1 from c1, c2 : V{Class}
2 with c1 -->{CO} -->{MC} <--{CO} c2
3 report c1, c2
4 end

```

Code Fragment 3.6: Lifting in GReQL 2

The above query focusses on the elements (or vertices) of a relation, which can be seen from the `V{Class}` (where `V` denotes vertex) type declaration of the `c1` and `c2` variables. Since GReQL 2 is a graph query language it also allows the querying of edges in the graph. This can be done by declaring variables with the `E{...}` type.

SIG Graph library

As explained in the problem definition (Section 1.2), queries in the SIG graph library have to be programmed as traversals over a graph. This results in verbose queries, which can be observed in a fragment of the code that is part of the `GraphCollapser` class that lifts a graph. Notice that the `collapse` method takes a predicate as argument which is used to make a selection of the edges. This is necessary because different relations are now represented in a single graph, where each edge type represents a relation type.

```

1 private static void collapseNode(INode node, Predicate hierarchyEdges, IGraph graph,
2     boolean retainLoops) {
3     Collection<? extends IEdge> iEdges = node.getOutEdges(hierarchyEdges);
4     while (!iEdges.isEmpty()) {
5         for (IEdge iEdge : iEdges) {
6             collapseEdge(graph, iEdge, hierarchyEdges, retainLoops);
7             iEdge.getToNode().delete();
8         }
9         iEdges = node.getOutEdges(hierarchyEdges);
10    }
11 }
12
13 private static void collapseEdge(IGraph result, IEdge iEdge, Predicate edgeTypeToSkip,
14     boolean retainLoops) {
15     INode toNode = iEdge.getToNode();
16     INode fromNode = iEdge.getFromNode();
17     reconnectInEdges(fromNode, toNode.getInEdges(), result, edgeTypeToSkip,
18         retainLoops);
19     reconnectOutEdges(fromNode, toNode.getOutEdges(), result, retainLoops);
20 }

```

Code Fragment 3.7: Part of the `GraphCollapser` class that implements lifting in the SIG Graph library

3.3.2 Software Design Metric

An essential part of a software risk assesment is the calculation of design metrics which can help to evaluate the quality and maintainability of the software.

A typical example of an object oriented design metric is the package instability metric⁶. This metric is a measure of how hard it is to change a package without impacting other packages within an application. The higher the value of this metric, the harder it is to make changes.

The metric is defined as $c_e / (c_a + c_e)$ where c_a stands for afferent coupling (or incoming dependencies), which is the number of classes outside the package that use classes inside the package, and c_e stands for efferent coupling (or outgoing dependencies), which is the number of classes inside the package that use classes outside the package.

In Figure 3.2 the efferent coupling (EC) relation is visualised. Afferent coupling can be visualised analogously to efferent coupling: the EC and U (the use relation) arrows in the figure will both change direction.

This can be expressed using pointfree binary relational calculus as follows:

$$EC = \llbracket \text{package} = p \rrbracket \ll (PO \circ U \circ PO^{-1}) \gg \llbracket \text{package} \neq p \rrbracket$$

We make use of predicates (denoted with $\llbracket \dots \rrbracket$) to restrict the domain (\ll) of the package of (PO) relation. Note that the above expression results in a relation between packages which for every use of a class in package p_2 by a class in package p_1 (where $p_2 \neq p_1$) contains only one tuple (p_1, p_2) . However, in this case we want to obtain the *number of classes* that use a class outside the package, which. To resolve this, we either have

⁶<http://semmlle.com/content/view/200/180/>

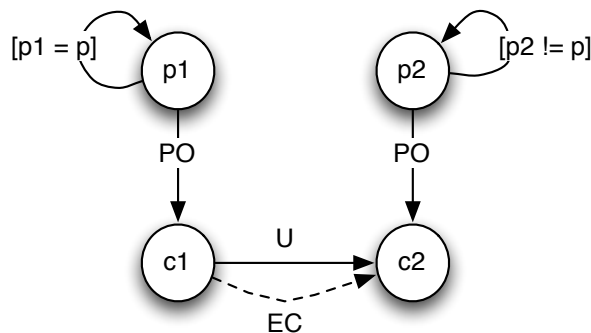


Figure 3.2: Efferent coupling

to use a multi-relation that allows multiple identical tuples, or we have to rewrite the expression to return a relation between classes as result:

$$EC = PO[[package = p], -] \ll U \gg PO[[package \neq p], -]$$

The above expression will result in a relation between classes which removes the need for bags as relations. Basically the use (U) relation is restricted in its domain by the set of classes that are in package p (obtained by taking the right image ($R[x, -]$, where x can be a set or a predicate) of the PO relation), and is restricted in its range (\gg) by the set of classes that are *not* in package p .

For the instability metric we are only interested in the cardinalities of the efferent and afferent coupling relations, these can be obtained by applying the cardinality operation $\#$. This allows us to define the computation of the actual instability metric:

$$packageinstability = \frac{\#EC}{(\#AC + \#EC)}$$

Crocopat

Code Fragment 3.8 is taken from [8].

```

1 U(x,y) := Call(x,y) | Contain(x,y) | Inherit(x,y);
2 FOR p IN Package(x)
3 {
4   CAClass(x) := !PO(p,x) & EX(y, Use(x,y) & PO(p,y));
5   CA := #(CAClass(x));
6   CeClass(x) := PO(p,x) & EX(y, U(x,y) & !PO(p,y));
7   CE := #(CeClass(x));
8   PRINT p, " ", CE / (CA + CE), ENDL;
9 }

```

Code Fragment 3.8: Instability metric in Crocopat

A FOR-loop is used to loop over all packages. Existentials and the cardinality operation are used for the computation of the instability of each package.

Rscript

This time we use another feature of the comprehensions: predicates. $P \neq \text{Pack}$ is a boolean expression that should hold for all elements of the comprehension result. The results of the comprehensions are sets and the $\#$ operation is used to obtain their cardinality.

```

1 int stability(package Pack, rel[package, package] U, rel[package, class] PO) =
2 #CE / (#CE + #CA)
3 where
4   set CE[class] = {C1 | <C1, C2> : U, <Pack, C1> : PO, <P, C2> : PO, P != Pack}
5
6   set CA[class] = {C1 | <C1, C2> : U, <P, C1> : PO, <Pack, C2> : PO, P != Pack}
7 endwhile

```

Code Fragment 3.9: Instability metric in Rscript using comprehensions

Just like in the previous queries, Rscript also allows this query to be rewritten to a style that comes close to binary relational calculus. Although Rscript does not support predicates, we can still simulate this particular problem by restriction and exclusion using sets. In particular because this query uses equality predicates that apply to one element (the package for which the metric is being computed): domain restriction by the $\llbracket package = p \rrbracket$ predicate can be expressed as domain restriction (domainR) using a singleton set {Pack}. Domain restriction by the $\llbracket package \neq p \rrbracket$ predicate can be expressed as domain exclusion with the same singleton set.

```

1 int stability(package Pack, rel[package, package] U, rel[package, class] PO) =
2 #CE / (#CE + #CA)
3 where
4   CE = domainR(PO, {Pack}) o U o inv(rangeX(PO, {Pack}))
5   CA = domainX(PO, {Pack}) o U o inv(rangeR(PO, {Pack}))

```

Code Fragment 3.10: Instability metric in Rscript using binary relational calculus

Both Rscript queries differ from the implementations in SemmlCode and Crocopat in that the latter compute the instability metric for every package. The Rscript example is only applicable for a single package.

JRelCal

Using JRelCal we can almost directly translate the relational calculus expression to Java and expose it as a packageStability method. To compute the instability metric for all packages this method simply has to be called for all packages.

```

1 public static <S,T> double packageStability(Relation<S, T> PO, Relation<T, T> U, S
2     packageName) {
3     Predicate<S> equalP = new EqualPredicate<S>(packageName);
4     Predicate<S> notEqualP = new NotPredicate<S>(new EqualPredicate<S>(packageName));
5
6     Relation<S, S> EC = PO.domainRestriction(equalP).compose(U).compose(
7         PO.inverse().rangeRestriction(notEqualP));
8
9     Relation<S, S> AC = PO.domainRestriction(notEqualP).compose(U).compose(
10        PO.inverse().rangeRestriction(equalP));
11
12    return EC.cardinality() / (AC.cardinality() + EC.cardinality());

```

Code Fragment 3.11: Instability metric in JRelCal

This example also shows the use of predicates to restrict the domain and the range of a relation. One might argue that the approach as taken in the Rscript relational calculus query (Code Fragment 3.10) is in this case

more appropriate. However, in this case using predicates is useful for demonstrating the feature, and comes more closer to the original relational calculus formulation at the beginning of this section.

SemmlCode

In SemmlCode, the package instability metric is included in a library with predefined functions that is shipped with the product. The code shown below is taken from this library where it is defined as part of the MetricPackage class. The SemmlCode implementation is more general than the standard definition of the metric: c_a is defined for any reference type (which includes classes, methods, constructors and interfaces) that depends on any reference type inside the package (as opposed to any class). This dependency is defined in the `depends()` method, and corresponds to the *use* relation in the other examples.

```

1 class MetricPackage extends Package, MetricElement, Commentable {
2   ... // This class contains more methods, but these are left out in this fragment.
3
4   int getAfferentCoupling() {
5       result = count(RefType t | t.getPackage() != this and
6                   exists(RefType s | s.getPackage()=this and depends(t,s)))
7   }
8
9   int getEfferentCoupling() {...} // This method is defined analogously to
   getAfferentCoupling()
10
11   float getInstability() {
12       exists(int ecoupling, int sumcoupling |
13           ecoupling = this.getEfferentCoupling() and
14           sumcoupling = ecoupling + this.getAfferentCoupling() and
15           sumcoupling > 0 and
16           result = ecoupling / sumcoupling)
17   }
18 }
```

Code Fragment 3.12: Instability metric in .QL

In the example we see that SemmlCode supports aggregation functions like `count` (sum, avg, etc.). Furthermore, it shows that .QL has an operation `exists` for existential quantification.

The above code fragment showed how to compute the metric for a single package. To compute the instability metric for all the packages in the source, the following query can be used:

```

1 from MetricPackage p
2 where p.fromSource()
3 select p, p.getInstability()
```

Code Fragment 3.13: Instability for all packages in the project's source in .QL

GReQL 2

GReQL 2 also supports aggregation constructs which can be mixed with the arrow syntax that was also shown in the previous section.

```

1 from p1:V{package}
2 report p1, EC / (AC+EC) where
3   AC := count({thisVertex <> p1}& -->{Use} <--{PackageOf} p1)
4   EC := count({thisVertex <> p1}& <-->{Use} <--{PackageOf} p1)
5 end
```

Code Fragment 3.14: Instability metric in GReQL 2

SIG Graph library

In the SIG graph library the instability metric has to be implemented as a subclass of the `AbstractObservationVisitor`. This visitor visits all vertices in the graph that are connected via edges specified by `EdgePredicate`. For this case, the predicate specifies the dependency edges like call edges, contain/defines, and inherit edges. For each vertex the fan-in and fan-out can be computed by counting the number of incoming and outgoing edges. The results of the metric computations are stored at the vertices themselves. Code Fragment 3.15 shows a fragment of the class that partly implements the metric.

```

1  public class EfferentAfferentCouplingCalculator extends
      AbstractHierarchicalFanInOutVisitor {
2  ...
3  static final List<String> NODES_TO_POPULATE = new ArrayList<String>();
4      static {
5          NODES_TO_POPULATE.add(Nodes.CLASS);
6          NODES_TO_POPULATE.add(Nodes.INTERFACE);
7          NODES_TO_POPULATE.add(Nodes.NAMESPACE);
8      }
9
10     public List<String> getNodesToPopulate() {
11         return NODES_TO_POPULATE;
12     }
13
14     static final List<String> DEPENDENCY_EDGES = new ArrayList<String>();
15     static {
16         DEPENDENCY_EDGES.add(Edges.CALL);
17         DEPENDENCY_EDGES.add(Edges.EXTENDS);
18     }
19 }

```

Code Fragment 3.15: Instability metric in SIG Graph library

3.3.3 Graph Pattern detection

In [8] Beyer and Noack list several applications of graph pattern detection, these include:

- the detection of implementation patterns, object-oriented design patterns and architectural styles,
- the detection of potential design problems (anti-patterns),
- the identification of code clones.

An example of an anti-pattern is the degenerate inheritance pattern [10]: when a class c inherits from another class a directly and indirectly via a class b , the direct inheritance is probably redundant or even misleading. This is illustrated in figure 3.3, where I denotes the inheritance relation. Since Java does not support multiple inheritance this anti-pattern can not be realised in Java with classes. However, with interfaces it can be realised. In that case it is still considered to be an anti-pattern.

This can be expressed in pointfree binary relational calculus as follows:

$$DI = I \circ I^+ \cap I$$

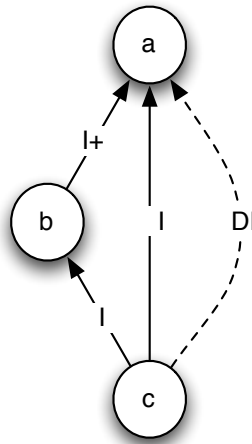


Figure 3.3: Degenerate inheritance

This will result in a relation DI containing all pairs (a, c) that participate in a degenerate inheritance pattern and represent the probably redundant direct inheritance. In the figure this relation is depicted as a dashed arrow.

The difference between the binary relational calculus expression and the implementations of this query in the tools is that they support n -ary relations that allow us to return triples that also include b .

The support for n -ary relations can also be useful in other instances of graph pattern detection such as cycle detection.

Croccopat

```
1 DI(a, b, c) := I(c, b) & I(c, a) & TC(I(b, a));
```

Code Fragment 3.16: Detecting degenerate inheritance in RML

TC is the transitive closure operation. The logical connective $\&$ corresponds to intersection in set theory. The ternary result relation $DegInh$ contains tuples that represent all three classes that are involved in the pattern.

Rscript

In Rscript you can implement this pattern matching query using a comprehension and the support for n -ary relations:

```
1 {<A, B, C> | <A, B> : I, <C, B> : I, <B, C> : tc(I)}
```

Code Fragment 3.17: Detecting degenerate inheritance in Rscript using relations

Alternatively, Rscript also allows you to express this pattern in a binary, pointfree style:

```
1 I o I+ inter I
```

Code Fragment 3.18: Detecting degenerate inheritance in Rscript using relational calculus

In this expression the resulting relation only contains binary tuples (a, c) .

JRelCal

For this example we have not defined the JRelCal expression in a method. Again, the JRelCal query is a direct mapping from the relational calculus expression to a JRelCal method chain.

```
1 i.compose(transitiveClosure(i)).intersection(i);
```

Code Fragment 3.19: Detecting degenerate inheritance in JRelCal

SemmlCode

SQL supports n-ary relations, and so does .QL: the `select` clause allows the user to influence the arity and type of the result. Transitive closure is expressed with the `+` operation appended to the method call.

```
1 from Class a, Class b, Class c
2 where c.hasSupertype(b) and c.hasSupertype(a) and b.hasSupertype+(a)
3 select a, b, c
```

Code Fragment 3.20: Detecting degenerate inheritance in .QL

GReQL 2

In GReQL 2 we can implement this pattern detection in two ways. Both queries are intended to find all triples of classes which match to the degenerated inheritance anti-pattern. The first one is similar to the SemmlCode query, while the second one looks directly at the elements (or edges for a graph) of the inheritance relation itself. Instead of reporting just triples of classes, it directly reports the inheritance edges that are candidates to be removed together with the two classes affected by the degenerated inheritance.

```
1 from a, b, c : V{Class}
2 with c -->{I} a && c -->{I} b -->{I}+ a
3 report a, b, c
4 end
```

Code Fragment 3.21: Detecting degenerate inheritance in GReQL 2 variant one

```
1 from e : E{I}
2 with startVertex(e) -->{I} -->{I}+ endVertex(e)
3 report e, startVertex(e), endVertex(e)
4 end
```

Code Fragment 3.22: Detecting degenerate inheritance in GReQL 2 variant two

SIG Graph library

Currently, there is no implementation of this analysis in the SIG graph library. However, an implementation in the SIG graph library is likely to take the following approach: for every class vertex in the graph paths to its superclasses will be traversed, if we detect two paths to the same superclass we can report this as degenerate inheritance.

3.3.4 Slicing

With forward slicing we mean the selection of an exact subgraph consisting of the vertices and edges encountered when traversing a graph starting at a set of starting vertices (slicing criteria). A more specific notion of slicing is program slicing as defined in [74]. Depending on the direction of the traversal two kinds of slicing are distinguished: *backward slicing* which can be used for dependency analysis, and *forward slicing* (Figure 3.4) which can be used for impact analysis.

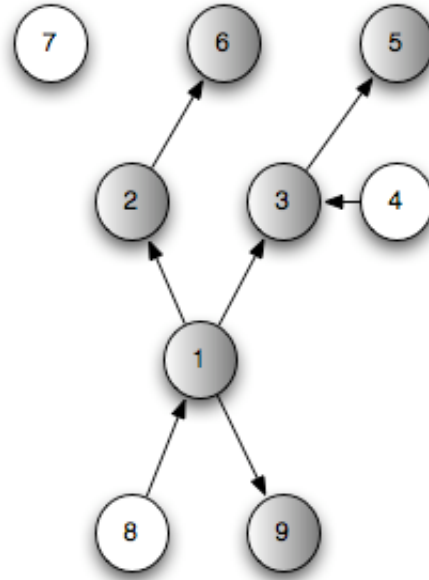


Figure 3.4: Forward slicing, starting at vertex 1

G is the source graph, V is the set of vertices in the forward slice, and C are the slicing criteria. In the following expression the set of vertices V that are part of the slice is computed

$$V = \text{carrier} (C \ll G^+)$$

This expression can be read as follows:

- First we compute the transitive closure ($^+$) of source graph G . All vertices that were directly or indirectly reachable from the slicing criteria are now directly reachable from the slicing criteria.
- The domain of the resulting graph is restricted with the slicing criteria C , which leaves us with only the vertices that are (directly or indirectly) reachable from the slicing criteria.
- We obtain the set of vertices V that are part of the slice by computing the carrier of this relation.

Now that we know the vertices that are part of the slice we can compute the actual subgraph by restricting the domain of the original graph by the vertices that are part of the slice:

$$V \ll G$$

Merging these two expressions into a single expression, results in this:

$$\text{carrier} (C \ll G^+) \ll G$$

In almost exact the same way a backward slice can be obtained: merely inverse the innermost source graph G before computing a forward slice.

Croccopat

```

1 Calls(x,y) := CALL(x,y);
2 CallsTC(x,y) := TC(Calls(x,y));
3 Slice(x,y) := EX(vertex, Criteria(vertex) & CallsTC(vertex, x) & Calls(x,y))

```

Code Fragment 3.23: Forward slicing in Croccopat

In this Croccopat query the starting criteria are in the set `Criteria`. The actual slicing is specified by a existential quantification.

Rscript

Rscript allows us to almost literally translate the relational calculus expression and expose it as a `fwdSlice` method:

```

1 rel[&T, &T] fwdSlice(set[&T] Criteria, rel[&T, &T] Graph) =
2   domainR(Graph, carrier(domainR(Graph, Criteria+)))

```

Code Fragment 3.24: Forward slicing in Rscript

JRelCal

In Java we can implement this expression using JRelCal, and expose it as a reusable, generic, forward slicing method as follows:

```

1 public static <T> Relation<T, T> fwdSlice(Relation<T,T> g, Set<T> slicingCriteria) {
2   Set<T> vertices = carrier(reflexiveTransitiveClosure(g).domainRestriction(
3     slicingCriteria));
4   return g.domainRestriction(vertices);
5 }

```

Code Fragment 3.25: Forward slicing in JRelCal

SemmlCode

```

1 from Method m1, Method m2, Method start
2 where start.getName() = "criteriaName" and start.calls+(m1) and m1.calls(m2)
3 select m1, m2

```

Code Fragment 3.26: Forward slicing in SemmlCode

The .QL version of the query uses a similar strategy as the binary relational calculus approach that is introduced in the beginning of this section. First, it filters out the method that has a name equal to the name of the desired starting criterion. Every method that is reachable from that method, is selected together with its direct callee (which also has to be reachable from the starting vertex).

Note that in SemmlCode it is not possible to implement a single slicing query that supports a set of slicing criteria. However, one could decide to lists all slicing criteria in the where clause. The disadvantage of this approach is that for large sets this quickly becomes unmanageable. Also, it is not possible to define the query generically because SemmlCode does not support parametric polymorphism.

GReQL 2

```

1 from start:V{Method}
2 with start.name = "criteriaName"
3 report start -->{Calls}+
4 end

```

Code Fragment 3.27: Forward slicing in GReQL 2

A distinct feature of GReQL 2 is that it supports *regular path expressions* this makes it possible to compute more complex slices. This is demonstrated in the following code fragment:

```

1 from start:V{Method}
2 with start.name = "criteriaName"
3 report nodes(completePathSystem(start, -->{Calls} -->{ClassOf} <-->{Inherit}))
4 end

```

Code Fragment 3.28: Forward slicing using regular path expressions in GReQL 2

This slice returns the subgraph that results from following paths that contain Calls, ClassOf, or Inherit edges.

SIG Graph library

```

1 // Slice forward from the seedNode(s) getting all nodes that are linked with edgeType
2 public IGraph sliceForward(IGraph graph, Collection<? extends INode> seedNodes,
3     Collection<IEdge> edgeTypes) {
4     IGraph resultGraph = new Graph();
5     for (INode node : seedNodes) {
6         INode graphNode = graph.lookupNode(node.getIdentity(), node.getType());
7         if (graphNode == null)
8             continue;
9         INode n = resultGraph.lookupNode(node.getIdentity(), node.getType());
10        if (n == null) {
11            INode newNode = resultGraph.createNode(node.getIdentity(), node.getType());
12            newNode.setDataMap(graphNode.getDataMap());
13            sliceForward(graph, graphNode, newNode, resultGraph, edgeTypes);
14        }
15    }
16    return resultGraph;
17 }
18 public static IGraph getInducedSubgraph(Collection nodes) {
19     IGraph subgraph = new Graph();
20     for (Iterator iterator = nodes.iterator(); iterator.hasNext();) {
21         INode originalNode = (Node)iterator.next();
22         INode newNode = subgraph.lookupOrCreateNode(originalNode);
23         addOutEdgesToSubGraph(nodes, subgraph, originalNode, newNode);
24         addInEdgesToSubGraph(nodes, subgraph, originalNode, newNode);
25     }
26     return subgraph;
27 }

```

Code Fragment 3.29: Forward slicing in SIG Graph library

3.4 Summary of results

In this section we summarise and discuss the results of both the language and the tool comparison.

3.4.1 Language comparison results

Evaluating languages is a non-trivial task that often depends on personal taste. However, the benchmark queries and criteria do give us means to compare the tools objectively. Due to the subtle differences in the various constructs and facilities the languages offer, there is no language that is clearly the best. Every language has its strong and its weak points. Table 3.2 summarizes the results of the language comparison.

Table 3.2: Summary of language comparison results

	Paradigm & characteristics	Type system					Abstraction	Extendability
		<i>String</i>	<i>Integer</i>	<i>Real</i>	<i>Boolean</i>	<i>Other</i>		
Rscript	Relational & Comprehensions	x	x	-	x	Composite	yes	no
JRelCal	Relational & API	x	x	x	x	Java	yes	yes
SemmlCode	SQL-like & OO	x	x	x	x	Object	no	no
Crocopat	Imperative & FO logic	x	x	x	-	-	no	no
GReQL 2	SQL-like & Path expressions	x	x	x	x	Object	no	no

Per tool, we will discuss in more detail its language and the result of the language comparison.

Crocopat offers an imperative language based on first order logic that required us to specify queries in a point-wise style using existential quantifiers. *Crocopat* does not offer abstraction facilities. Its type system is limited to the primitive types string, integer, and real.

Rscript allows us to specify a query either using point-free binary relational calculus, or in a point-wise style using comprehensions. The language has scalar types (boolean, string, integer, and location) and composite types (sets and relations). Expressions can be constructed from comprehensions, function invocations and operators. Both forms of abstraction are available in *Rscript*. Extending the language is expectedly a non-trivial task since this would require adapting the compiler.

JRelCal queries are binary relational calculus expressions expressed using Java method call chains. *JRelCal* offers a lot of functionality that is also available in *Rscript*, but without any syntactic support. Language constructs that heavily depend on syntax constructs for their power, e.g., comprehensions, are therefore not available in *JRelCal*. Part of *JRelCal*'s features are provided by its host language Java, e.g., its abstraction facilities. Parametric polymorphism is realised by using Java Generics. This makes *JRelCal* queries applicable to relations containing Java objects of any type. Parametrization is possible by defining *JRelCal* queries in Java methods. For users with knowledge of Java or a similar programming language extending *JRelCal* is relatively easy. Extensions involve adding methods to its API or changing the implementation of existing methods.

SemmlCode's .QL language required us to specify a query in a point-wise, SQL-like style. The similarity to SQL makes .QL familiar and therefore easy to learn language for a wide range of users. A convenient feature of *SemmlCode* is its object-orientation. Functionality for relations can be grouped as methods of a class representing that relation. Combined with the auto-completion in the editor this eases the writing of queries significantly, because functionality that is available for a class can now easily be found. The only form of abstraction offered is ad-hoc polymorphism; using the object orientation of .QL it is possible to override

existing code. Subclasses of existing classes are given a "replacement method" for methods in the superclass. Parametrization of queries is not possible. Note that one could argue that methods taking parameters is a form parametrization. Extending .QL is not possible; SemmleCode is proprietary and closed-source which makes it impossible to make changes to the language.

On the query level, *GReQL 2*'s syntax is very similar to SQL. However, on the individual clause level it differs from SQL considerably. An example is the arrow syntax in the where class: this is an intuitive way to express graph queries, however, due to some particularities in the syntax, it may be difficult for many users to use it to its full potential. Another distinctive feature of GReQL 2 is regular path expressions. These expressions allow you to express advanced path queries conveniently and concisely. Whether RPE's are useful in the context of high level software analysis remains a question to be answered. Although the source code is available, we expect that extending GReQL 2 will be a difficult task involving changes to the GReQL compiler.

3.4.2 Tool comparison results

The comparison of the tool criteria consisted of checking for the presence or absence of features. The results of this comparison are summarized in Table 3.3. We will limit the discussion of the tool comparison results to

Table 3.3: Summary of tool comparison results.

	User Interface	API	Interchange	Extractor	Licensing
RScript	CLI & GUI	-	Rstore	-	BSD
JRelCal	API	x	RSF	-	BSD
SemmleCode	Eclipse plug-in	x	-	Java & XML	Proprietary
Croccopat	CLI	x	RSF	-	GNU LGPL
GReQL 2	CLI	x	TGraph	-	BSD

the most interesting issues:

Although the summary suggests that most solutions offer an API, notice that most of these are call-level interface API's. Only JRelCal offers an API in the form of a Java library. Rscript does not offer the convenience of an API, however, the command-line interface (CLI) allows a (limited) form of interoperability. SemmleCode is shipped as an Eclipse plugin, offering the most advanced graphical user interface of the compared tools. It includes a query editor with user friendly features such as syntax highlighting and auto syntax completion. The results of a query can be presented as a table, graph or chart. SemmleCode can only be invoked interactively via its user interface; there is no CLI and API available. In combination with its commercial license, the integration of SemmleCode in an existent tool becomes virtually impossible. It comes with integrated extractors for Java and XML, but does not allow the use of third party extractors. Since it comes with its own extractors it does not support interchange formats. It does have a text export function available in the GUI.

3.5 Conclusions

We have subdivided the conclusion into two conclusion: a generic conclusion and a specialized conclusion. The former draws some general conclusion, and is useful for every interested reader or potential user. The latter is the conclusion specific for this thesis, it discusses which tool is the best for use within the SIG.

3.5.1 General conclusion

We have compared five code querying tools with respect to ten criteria. Some of the criteria – paradigm & characteristics, type system, abstraction facilities, and extendability –, were meant to compare the code querying language provided by the tools, the others – user interface, API support, output formats, interchange

formats, extraction support, and licensing – were meant to compare the tools themselves. The criteria were motivated by two usage scenarios: one in which a user wants to interact directly with the tool, and one in which the tool needs to be used indirectly from other tools. The comparison of the language oriented criteria was performed by implementing four queries in detail in each of the languages under consideration.

Based on the findings summarized in the previous section we conclude that although each offers a rich set of features, none of the tools rate well on all criteria. In particular we found that the combination of good abstraction facilities and extendability is not available in most languages, and that an API in the form of a library is rarely provided. We pose the challenge to create a solution that offers extendability as well as abstraction on the language level, and on the tool level provides an API for optimal integration possibilities.

In addition to the challenge posed above we would like to suggest some other avenues of future work that can be explored. Currently, code querying technologies are applied to obtain information from a large body of code. We would be interested to see how code querying can be combined with *code transformation*. Another extension is the ability to query not just a static code repository, but to use code querying to compare different versions of the same system, e.g., by providing capabilities and abstractions to effectively query svn repositories. A final application for code querying is to automatically support *architecture checking*: from a (formal) description of the system architecture, a number of code queries is automatically generated that verify whether the implementation satisfies the constraints set by the architecture.

3.5.2 Conclusion for the SIG

Recall that in the context of this thesis we are looking for the best code query technology for SIG. Ultimately to be used as a replacement for the SIG graph library, the tool will need to be fully integrated with SIG's SAT. Using a code query technology in this situation corresponds to the tool integration scenario as defined in Section 3.2.

As we can see in Table 3.1, the most important criteria for the tool integration scenario are: "paradigm & characteristics", API support, abstraction facilities, extendability, and licensing. Below, we discuss why these criteria are important for the SIG:

- *Paradigm & characteristics*: For SIG, the query language should offer a more declarative and concise way to specify queries in comparison to the SIG graph library. During our comparison we have observed that, in this sense, all tools offer a "better" language. When we leave ease of use out of consideration, all compared tools meet SIG's requirements for this criterion. If we do take ease of use into consideration then one could argue that SemmleCode's .QL is the most easy to use language: it offers a SQL-like language which is familiar to a lot of users, and the object-orientation in combination with auto completion in the editor makes it easy to find available functionality.
- *API support*: For SIG the ideal level of API support would be a tool that offered an API written in the language of the SAT: Java. In this way full integration with the SAT can be achieved with no effort at all.
- *Abstraction facilities*: For SIG it is important that the query language offers good abstraction facilities, this makes it possible to write a query only once and reuse it as much as possible across different projects of the SIG. Both Rscript and JRelCal offer parametrization and parametric polymorphism
- *Extendability*: Extending a language is a complex task requiring knowledge of compiler technology. In an industrial setting such as the SIG such knowledge is not always at hand. This leaves only the Java library JRelCal as suitable candidate: Java knowledge is sufficiently available at the SIG.
- *Licensing*: If SIG wants to integrate the tool into its own tooling and products, this will have to be allowed by the license. Most tools come with a permissive license that allows integration in SIG's products. Only SemmleCode comes with a proprietary license that prohibits inclusion in (commercial) third party products.

Based on these requirements and the results of the comparison we conclude that only JRelCal sufficiently meets the requirements of the SIG. Still, there is room for improvement: useful features we have seen in the other tools are not all present in JRelCal, and some aspects of the implementation can be improved upon. Therefore we will create a new implementation of JRelCal that removes these limitations. In the next chapter we introduce JRelCal and discuss most of our improvements.

Chapter 4

Improving JRelCal: from version 0.1 to 1.0

JRelCal is a prototype Java library that is based on Tarski's binary relational calculus (BRC). It was developed by Tijs van der Storm as a side project of his work on his doctoral thesis at the CWI. JRelCal is inspired by Rscript and was meant to be an efficient implementation of this relational query tool. To move JRelCal past the prototype stage we have created a new and mature version of JRelCal.

The new JRelCal includes the following improvements: support for predicates, optimisation of reachability queries, efficient representations of relations, a mapping from SIG graphs to JRelCal relations, and additional unit tests.

Currently, our work on JRelCal is done in a branch of the JRelCal repository hosted at the CWI. On our request, this branch is merged into the trunk of the JRelCal project. A daily build of this trunk is publicly available at the build farm of the CWI¹.

In this chapter we introduce JRelCal in more detail and discuss the previously mentioned improvements.

4.1 Introduction to JRelCal

JRelCal is based on BRC; all BRC operations are implemented as a method. In addition, JRelCal includes several other operations that are useful for source code querying that are not part of Tarski's BRC (e.g., domain restriction).

BRC is compositional, which is reflected in JRelCal in the fact that every JRelCal method takes one or two relations as argument, and returns a relation as result. This makes it possible to compose a complex query from basic operations. An example of this is the following BRC expression where $(A, B \subseteq \text{int} \times \text{int})$:

$$([\geq 2] \ll A) \circ B \circ A$$

which directly translates to the following JRelCal query:

```
1 Relation<Integer, Integer> A, B;  
2 Predicate<Integer> greaterEqualTwo;  
3 ... //Instantiation of predicate and relation variables.  
4  
5 A.domainRestriction(greaterEqualTwo).compose(B).compose(A);
```

Code Fragment 4.1: Example of a JRelCal query

The original JRelCal included two implementations of relations: an implementation based on an adjacency list structure, and a bag-based (as opposed to set-based) implementation that allows duplicate tuples in a relation. The bag-based implementation can be useful for expressing queries that for their answer rely on the number of occurrences of a single tuple. An example of such a query is the query for the computation of the package instability metric, which we discussed in Section 3.3.2.

¹<http://sisyphus.meta-environment.org>

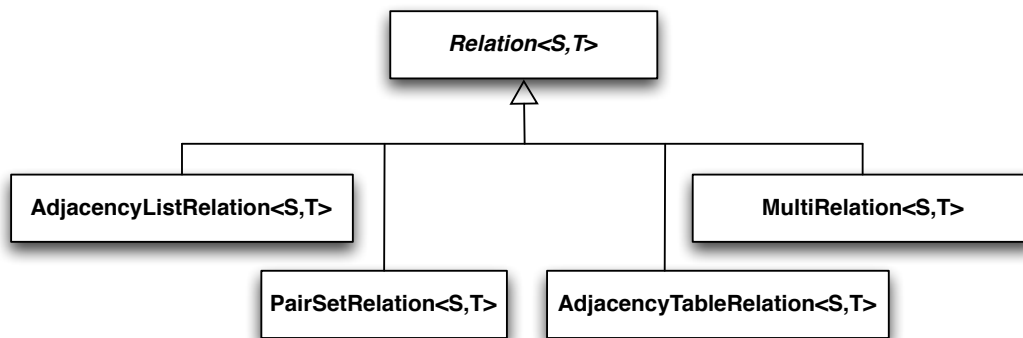


Figure 4.1: Design of JRelCal on class level

Next, we will discuss two interesting design decisions found in JRelCal: defining operations on relations as "side-effect free" methods, and using Java Generics to type these methods.

4.1.1 Side-effect free methods

JRelCal operations modify neither the relation they are executed upon, nor their argument relations. Instead, an operation creates a new relation to store and eventually return its results. In that sense, a JRelCal method can be considered to be side-effect free. Removing side-effects from the equation allows expressions to be evaluated in any order. Note that if a JRelCal method would modify its arguments, the expression in Code Fragment 4.1 would be incorrectly evaluated: A would have been modified by the domain restriction, which influences the result of the composition with A at the end of the expression.

Another advantage of this design decision is that it becomes possible to use method chaining. On his wiki² on DSL's Martin Fowler describes *fluent interfaces*: an API style primarily designed to be readable and to flow. The fluent interface style is often used to realise an EDSL in an object oriented language. One way of achieving "fluency" is by using method chaining. Martin Fowler describes this as follows: "*Internal DSL's are all about providing a flowing API, which often involves a sequence of calls on a single object. Method chaining is an idiom that achieves this through a sequence of modifier calls where each call returns the host object for further modification*"³. The method chaining found in JRelCal differs from Fowler's definition in that JRelCal methods do not modify their host object (a relation). Despite this difference, the advantages of method chaining, readability and "flow", still hold for JRelCal.

4.1.2 Generics

The other design decision we adopt from the original JRelCal is using Java Generics to type a relation and its operations. This makes it possible to statically check the type correctness of a JRelCal expression. Without the use of Java Generics the type of the domain and the range of the relation would not have been available at compile time, making it impossible to check the type correctness of an expression.

Unary operations that require homogeneous relations as argument (e.g., transitive closure and carrier) form a special case. To consistently make use of the typing imposed by generics, these methods are defined as static methods that take the subject relation as argument. This allows us to impose that the argument relations is homogeneous. This would not have been possible if we defined such an operation as an instance method without parameter.

This is best illustrated by using the `transitiveClosure` method as an example. The transitive closure operation is a graph theoretic operation and therefore requires the domain and the range of a relation to be of the same type.

The method can be defined statically as:

²<http://martinfowler.com/dslwip/index.html>

³<http://martinfowler.com/dslwip/MethodChaining.html>

```
public static <T> transitiveClosure<T,T>(Relation<T,T> rel) {...}
```

or as an instance method:

```
public Relation<T,T> transitiveClosure() {...}
```

The static definition is used as `Relation.transitiveClosure(rel)`. In this case, the typing of the method imposes that `rel` is homogeneous. The definition as instance method is used as `rel.transitiveClosure()`. Since the `Relation` class is typed as `Relation<S, T>` it is not statically imposed that the relation instance `rel` is a homogeneous relation. This may lead to type errors at runtime.

One can take full advantage of both design decisions by using an IDE with auto-completion: because of the method chaining and the typing with Generics the auto-completion function is able to show only the methods that are applicable in the current context. An example of this is shown in Figure 4.2. The suggestions offered by the auto-completion are based on the fact that relation `A` is a homogeneous relation of the type `Relation<Integer, Integer>`: since the range restriction on `A` will return a relation of the same type, composition is only possible with a relation of type `Relation<Integer, U>`.

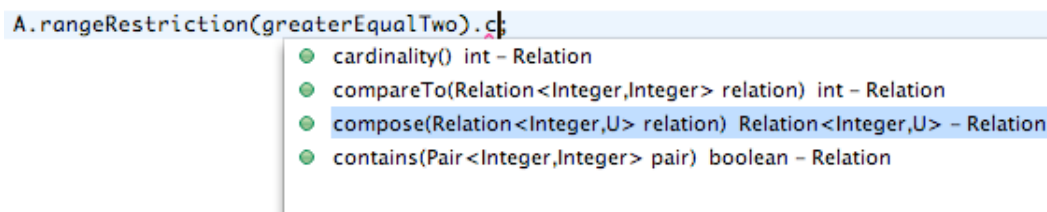


Figure 4.2: Auto completion in Eclipse while creating a JRelCal query

4.1.3 The new JRelCal

Since the original implementation of JRelCal was a prototype, it had several shortcomings: it lacked functionality, showed poor performance for some operations, and contained several bugs. To remove these shortcomings we have created a new version of JRelCal. Although we have created a completely new implementation of JRelCal, we do adopt the two previously discussed design decisions from the original JRelCal.

In our new version of JRelCal we have defined the relation abstract data type as an abstract class (Code Fragment 4.2) that serves as a super class for all relation implementations (Figure 4.1(b)). The result is a separation of the specification of a relation from its implementation, which eases the addition of new implementations. In the new JRelCal we have concentrated on removing the shortcomings of the original implementation. We will discuss these improvements in the remaining sections of this chapter.

```
1 public abstract class Relation<S, T> implements
2     Iterable<Pair<S, T>>,
3     Comparable<Relation<S, T>> {
4
5     public abstract boolean contains(Pair<S,T> pair);
6     public abstract Set<Pair<S, T>> asPairs();
7
8     public abstract Relation<T, S> inverse();
9     public abstract Set<S> domain();
10    public abstract Set<T> range();
11    public abstract Relation<S, T> union(Relation<S, T> relation);
12    public abstract Relation<S, T> intersection(Relation<S, T> relation);
13    public abstract Relation<S, T> difference(Relation<S, T> relation);
```

```

14 public abstract <U> Relation<S, U> compose(Relation<T, U> relation);
15 public abstract int cardinality();
16
17 // Static definition of transtive closure
18 public static <T> Relation<T, T> transitiveClosure(Relation<T, T> rel){...}
19
20 public abstract Relation<S, T> domainRestriction(Set<S> set);
21 public abstract Relation<S, T> domainExclusion(Set<S> set);
22 public abstract Set<T> rightSection(Set<S> set);
23 public abstract Relation<S, T> rangeRestriction(Set<T> set);
24 public abstract Relation<S, T> rangeExclusion(Set<T> set);
25 public abstract Set<S> leftSection(Set<T> set);
26
27 // Predicate versions
28 public abstract Relation<S, T> domainRestriction(Predicate<S> p);
29 // ...
30
31 public abstract Iterator<Pair<S, T>> iterator();
32 // Default implementations of equals, compareTo, based on iterator
33 public int compareTo(Relation<S, T> relation) {...}
34 public boolean equals(Object o) {...}
35 // ...

```

Code Fragment 4.2: The abstract class Relation in JRelCal

4.2 Predicates

Restriction and exclusion of relations are common operations that can be used to filter a relation. Domain restriction using a set S as parameter 'restricts' a relation to only contain tuples whose first element occur in S . In the same way, domain exclusion excludes all tuples whose first element occurs in S . Application of restriction and exclusion is not limited to the domain of a relation and can be applied to any arbitrary set, e.g. range and carrier sets.

In the original version of JRelCal, restrictions and exclusions of sets was only possible using sets as parameter. Sets are extensional definitions that sum up every element that falls under its definition. There are particular situations where extensional definitions are not feasible. For example, using an extensional definition, restricting the domain to only contain elements that are greater than 1 would require us to sum up all integers that are greater than 1. In these situations, an intensional definition is more appropriate. An intensional definition gives the meaning of a term by specifying all the necessary and sufficient conditions for belonging to the set being defined. To add support for intensional definitions we extended the JRelCal restriction and exclusion operations with support for predicates. Code Fragment 4.3 shows the implementation of domain restriction with a set and with a predicate. Note that the predicate-based variant generalises the set-based variant: the set-based variant can be implemented by calling the predicate-based variant with a predicate that tests whether an element is in the set parameter of the set-based variant.

```

1 public Relation<S, T> domainRestriction(Set<S> set) {
2     Relation<S, T> result = new PairSetRelation<S, T>();
3     for (Pair<S, T> pair : this)
4         if (set.contains(pair.getFirst()))
5             result.add(pair);
6     return result;
7 }
8
9 public Relation<S, T> domainRestriction(Predicate<S> p) {
10    Relation<S, T> result = new PairSetRelation<S, T>();
11    for (Pair<S, T> pair : this) {
12        if (p.evaluate(pair.getFirst()))

```

```

13         result.add(pair);
14     }
15     return result;
16 }

```

Code Fragment 4.3: Implementation of domain restriction using predicate and set in PairSetRelation

For the implementation of predicates we reuse the Apache Commons-Collections⁴ framework that offers a predicate interface and implementation. The Apache Commons is a project of the Apache Software Foundation⁵. The purpose of the Commons project is to provide reusable, open source Java software. Commons-Collections seeks to build upon the Java Collections Framework by providing new interfaces, implementations, and utilities.

The `Predicate` interface offered by Commons-Collections defines an interface for classes that perform a predicate test on an object. The test has to be defined in the `evaluate(...)` method. This is a boolean method that takes the input object as argument. Standard implementations of common predicates are provided by `PredicateUtils`. These include `true`, `false`, `instanceof`, and `equals`. Additionally, there are predicate offered that implement logical operations like `and`, `or`, and `not`. These predicates can be used to combine simple predicates to form more complex ones.

A disadvantage is that the Apache Commons-Collections framework does not support Java 5 Generics. Without the support for Generics it is not possible to take full advantage of the generically defined JRelCal interface. For example, a version of Commons-Collections that supports Generics would allow us to define a domain restriction such that it only accepts predicates of the correct type (e.g. `public Relation<S, T> domainRestriction(Predicate<S> p)`).

However, there exists an open-source project Commons-Collections with Generics⁶ that is a Java 5 generics-enabled version of the Commons-Collections project. All appropriate classes from Commons-Collections 3.1 have been refactored to support Java generics.

Code Fragment 4.4 shows an example of the creation of a predicate and its use to restrict the domain (of type `Integer`) of a relation to contain only the integers that are greater than 1.

```

1 Predicate<Integer> greaterThanOne = new Predicate<Integer>() {
2     public boolean evaluate(Integer i) {
3         return i > 1;
4     }
5 };
6
7 Relation<Integer, String> aRelation = new PairSetRelation<Integer, String>();
8 aRelation.domainRestriction(greaterThanOne);

```

Code Fragment 4.4: Definition and use of a simple predicate using Commons-Collections with Generics

4.3 Optimisation of reachability queries

In this section we describe the work we have done to improve the performance of JRelCal for queries involving reachability issues. We start with a description of the problem, followed by a discussion of our solution and its implementation. We conclude with the validation of our work by discussing the results of the performance tests we have carried out. Note that the definitions of the graph theoretical concepts in this chapter are adapted from [18].

⁴<http://commons.apache.org/collections/>

⁵<http://www.apache.org/>

⁶<http://sourceforge.net/projects/collections>

4.3.1 Problem definition

Reachability deals with determining where we can get to in a directed graph, or, in the case of JRelCal, a homogeneous relation viewed as a digraph. A relation can be viewed as a graph as follows: given a homogeneous relation $R \subseteq V \times V$ and $v_1 R v_2$, then $v_1 \rightarrow v_2$ can be viewed as a directed edge in a directed graph (digraph) $G(V, R)$. Queries involving the notion of reachability are common in the context of code querying, e.g., "return all classes that are a direct or indirect subtype of a particular superclass", or "find all methods that are directly or indirectly called from a particular class".

Definition A directed graph G is a pair (V, E) where V is a set of elements called *vertices* and $E \subseteq V \times V$ is a set of pairs called *edges*. The cardinality of V and E is denoted by n and e respectively.

To express reachability, all the evaluated code query technologies in Chapter 3 offer a transitive closure operation. Assuming there are no optimisations performed, these operations compute the full transitive closure for a homogeneous relation R . In a full transitive closure the full successor list for each vertex $v \in R$ is computed. After the full transitive closure of a relation is constructed, all reachability questions can be answered in logarithmic time (Assuming the underlying data structure supports logarithmic time lookup of adjacent vertices). The drawback of full transitive closure, however, is that it can be a very computation-intensive operation. Implementing an efficient transitive closure operation that scales to large relations (as found in source code querying) is a non-trivial task.

Definition The *full transitive closure* of graph $G(V, E)$ is a graph $G^+(V, E^+)$ such that E^+ contains an edge (v, w) iff G contains a non-null path $v \xrightarrow{+} w$. The *successor set* of a vertex v is the set $Succ(v) = \{w \mid (v, w) \in E^+\}$, i.e., the set of all vertices that can be reached from vertex v via non-null paths.

During experimentation we discovered that the transitive closure implementation of the original JRelCal was incorrect. Therefore, our challenge is to create a new efficient and scalable implementation of transitive closure. A first naive attempt at a new implementation of transitive closure is the "least fixed-point union-compose" algorithm, as shown in Code Fragment 4.5. This is an iterative algorithm that composes the input relation with itself until it reaches a least fixed point. In essence, the algorithm is the same as the well-known Warshall [35] algorithm. The running time for this algorithm is $O(n^3)$. This time behaviour makes it infeasible to use this algorithm for the computation of transitive closure in a code query tool, since typically code queries are run on large systems. In the following sections we discuss our solution to this problem.

```

1 public static <T> Relation<T, T> transitiveClosure(
2     PairSetRelation<T, T> relation) {
3     Relation<T, T> result = new PairSetRelation<T, T>(relation.asPairs());
4     int prevResultSize = 0;
5     while (!(prevResultSize == result.cardinality())) {
6         prevResultSize = result.cardinality();
7         result = result.union(result.compose(relation));
8     }
9     return result;
10 }

```

Code Fragment 4.5: JRelCal PairSet implementation of Transitive Closure

4.3.2 Solution

Often, in the context of source code querying, the *full transitive closure* of a relation is computed while the result of the query only consists of a small subset of the transitive closure. We argue that in the context of code querying, computing a full transitive closure is rarely useful. In a query, the transitive closure operation is generally used to declaratively express a reachability issue. Often, these reachability issues involve a small

number of source nodes. For example, a software engineer may use a code query technology to support his refactoring activities. To better understand the impact of changes to a particular method, he may be interested in the methods that call this method directly or indirectly. To answer this kind of queries, only the successors of one particular vertex need to be computed instead of the successors of each vertex in the relation.

An example of this is shown in Code Fragment 4.6: to find the methods that are called from a method `foo()` the full transitive closure of the `CALL` relation is computed. The domain restriction performed on the transitive closure results in a relation between `foo()` and its successors. This demonstrates that in some cases many computations of successor lists are performed that are not required for the result.

```
transitiveClosure(CALL).domainRestriction("foo()");
```

Code Fragment 4.6: Example of a reachability issue expressed with transitive closure

Based on these observations, we approach the optimisation of reachability queries in the following way: instead of *optimising* the computation of the transitive closure we will focus on *reducing* the amount of work that is performed for this computation.

In JRelCal, the transitive closure operation is the only operation available to express reachability issues. Therefore, with the current set of JRelCal operations, there is no way for the query programmer to prevent the redundant computations from being performed. Our solution to this is to add a "Reach" operation that computes a *partial transitive closure*. A partial transitive closure operation only computes the successor list for each vertex s in a set S of source nodes. We expect that using a partial transitive closure operation will have a larger impact on the performance than using an efficient algorithm for full transitive closure. Mainly because, depending on the number of source vertices, the number of computations can be significantly reduced.

There are two types of partial transitive closure problems: single-source transitive closure and multi-source transitive closure [61, 21]. In the *single-source transitive closure problem* (Figure 4.3(a)), we are given a homogeneous relation $R \subseteq V \times V$ which we view as a digraph $G(V, R)$, and a single vertex $v \in V$ whose successors we should compute. The successors of a vertex v areThis problem can be solved by a simple graph search algorithm such as depth-first search (DFS) or breadth-first search (BFS). We start the search at vertex v and collect each vertex in G that is encountered into the result set $Succ(v)$.

In the *multi-source transitive closure problem*, we are given the graph $G(V, R)$ and a subset $S \subseteq V$. We should compute the successors of the vertices in S . This problem can be further divided into *strong multi-source transitive closure problems* (Figure 4.3(b)) and *weak multi-source transitive closure problems*. In the strong version, we should compute its own successor set for each vertex of S . In the weak version, we should compute the union of the successor sets of the vertices in S . Like the single-source problem, the weak multi-source problem can be solved by a graph search algorithm. Both the strong and the weak multi-source problem can be solved by first computing the full transitive closure. However, computing the multi-source transitive closure directly by using a graph search algorithm is likely to be more efficient.

We will make two variants of a partial transitive closure operation available in the JRelCal API. These two methods can be implemented efficiently using graph traversal algorithms. The operations will return a relation between the sources and their successors. This means that the actual paths between the sources and their successors are discarded, e.g., in Figure 4.3(a) we can see that $(D, I) \in result$ but the actual path $\{(D, F)(F, I)\}$ can not be deduced from *result*. We discuss the implementations of both methods in more detail in the next section.

4.3.3 Implementation

The single-source transitive closure problem is implemented by the algorithm "SingleSourceReach", shown in Algorithm 1. The algorithm uses DFS to find the successor vertices. The result is accumulated in the result parameter as a relation between the source vertex and its successors. If the algorithm finds a cycle back to the source, a self-arc is added to the result.

Algorithm 1 SingleSourceReach(Relation *rel*, Vertex *v*, Vertex *source*, Relation *result*)

```

1: Mark v as visited
2: for (v, w) in rel do
3:   if w has not been visited then
4:     result  $\leftarrow$  result  $\cup$  {(source, w)}
5:     Recursively call SingleSourceReach(rel, w, source, result)
6:   else if w = source then
7:     result  $\leftarrow$  result  $\cup$  {(source, source)}
8:   end if
9: end for

```

If n_s vertices and e_s edges are reachable from vertex s , this algorithm taking s as source has a running time of $O(n_s + e_s)$. This is provided that the relation is represented with a data structure that supports constant-time vertex and edge methods [32].

The SingleSourceReach algorithm is defined for a single source vertex. However, defining the strong multi-source variant MultiSourceReach for a set of vertices S in a relation R is trivial, because we can reuse the definition of Reach:

$$\text{MultiSourceReach}(S, R) = \bigcup (\forall s : s \in S : \text{Reach}(s, R))$$

The result of the MultiSourceReach operation is a relation Q between the vertices from S and their successors in R . Each tuple in Q relates a vertex $s \in S$ to one of its successor vertices $v \in V$. The answer to the weak multi-source problem can be easily be obtained by taking the range of Q .

Implementing TC using Reach

The goal of introducing the reachability operations was to replace the the transitive closure operations in reachability queries. However, The advantage of this solution is that for the implementation of the full transitive closure operation we can reuse the implementation of MultiSourceReach. The compositional definition of Reach allows us to define the transitive closure operation in terms of Reach. In this way we are reusing functionality, and removing the need for a separate implementation of a transitive closure algorithm. For a relation viewed as digraph $G(V, R)$ we can define transitive closure as follows:

$$\text{TransitiveClosure } R = \text{MultiSourceReach } V$$

Transitive closure implemented as a repeated DFS has a running time of $O(n(n+e))$. Nevertheless, when used for a graph that is *dense*, that is, if it has close to n^2 edges, this approach runs in $O(n^3)$ time. When the graph is *sparse*, repeated DFS performs better than running the Warshall algorithm once. During our experiments we found that all relations extracted from source code are sparse. Still, more empirical evidence is required to support this.

Informally, a dense graph is a graph in which the number of edges is close to the maximal number of edges. The opposite, a graph with only a few edges, is a sparse graph. The distinction between sparse and dense graphs is rather vague. One possibility is to choose a number k with $1 < k < 2$ and to define sparse graph to be a graph with $e = O(n^k)$ [65].

An optimisation for MultiSourceReach

The MultiSourceReach repeatedly runs the SingleSourceReach operation on the same relation, which can lead to redundant traversals of the same path. This is an opportunity for optimisation: vertices that have been used as source for a call of SingleSourceReach can be marked. When, in a subsequent call of SingleSourceReach a vertex is encountered that already has been marked, the traversal can be stopped and the results of the earlier call are looked up in the current result to complete the SingleSourceReach algorithm for the current source.

This optimisation can be implemented for the MultiSourceReach operation, and in this way the optimisation is propagated to the transitive closure operation.

We can illustrate this optimisation with the help of Figure 4.3(b). When we perform an optimised MultiSourceReach on relation R , it may start a SingleSourceReachMark from vertex G and will therefore mark G as having been used for a SingleSourceReach. When the SingleSourceReachMark for G returns, it may start a single-source ReachMark from vertex D . When the ReachMark from D arrives at vertex G it will detect that it has been used as a source for a previous call of ReachMark. To complete the run for D , the successors of G will be looked up from the current result.

The implementation of this optimisation only requires a small adaption of the original SingleSourceReach algorithm which we showed in Algorithm 1. Algorithm 2 shows the modified version of SingleSourceReach called SingleSourceReachMark. The modification (Algorithm 1, line 5-10) consists of an additional check to see whether the vertex that is being visited has been used as source in a previous call of the algorithm. If so, the successors of the vertex are retrieved from the current result, added to the successors of the current source, and marked as visited. Note that the MultiSourceReach operation has to be adapted such that it marks the vertices that have been used as source for a SingleSourceReachMark.

Algorithm 2 SingleSourceReachMark(Relation rel , Vertex v , Vertex $source$, Relation $result$)

```

1: Mark  $v$  as visited
2: for  $(v, w)$  in  $rel$  do
3:   if  $w$  has not been visited then
4:      $result \leftarrow result \cup \{(source, w)\}$ 
5:     if  $w$  has been used as source for SingleSourceReachMark then
6:       Mark  $w$  as visited
7:       for  $(w, q)$  in  $result$  do
8:          $result \leftarrow result \cup \{(source, q)\}$ 
9:         Mark  $q$  as visited
10:      end for
11:     else
12:       Recursively call SingleSourceReachMark( $rel, w, source, result$ )
13:     end if
14:   else if  $w = source$  then
15:      $result \leftarrow result \cup \{(source, source)\}$ 
16:   end if
17: end for

```

The effectiveness of the marking optimisation depends on the order in which the sources for the MultiSourceReach are used as source for a single call of SingleSourceReachMark. Ideally, the sources are used in a reverse topological order. This ensures optimal use of the previously computed successor sets, thereby preventing redundant traversals of paths.

4.3.4 Run-time optimisation of reachability queries

We have presented the Reach operations as cheaper alternatives for computing the full transitive closure, and added them as methods to the JRelCal API. This means, however, that the programmer is responsible for identifying optimisation opportunities.

Instead of relying on the programmer for identifying optimisation opportunities we can do this automatically at run-time. We will explain our optimisation technique using an example in which we rewrite JRelCal expressions involving the composition of a transitive closure relation with another relation ($A \circ B^+$ and $B^+ \circ A$) to equivalent expressions that use MultiSourceReach.

The optimisation is based on the use of a "wrapper class": when the transitive closure operation is called, instead of computing the full transitive closure, it returns a wrapper class `TC` containing the rela-

tion the transitive closure was called upon. This can be observed in Figure 4.4 in the code snippet for the `transitiveClosure` method.

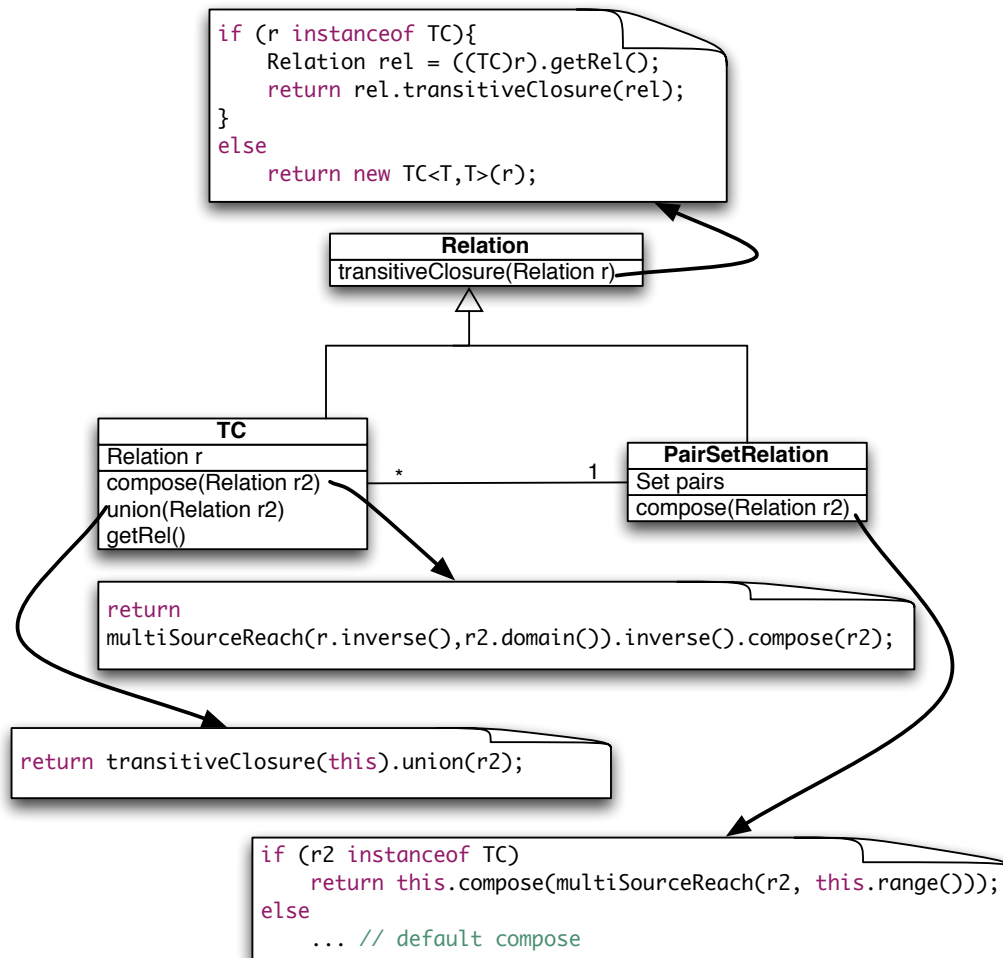


Figure 4.4: Run-time optimisation of reachability queries

The actual optimisation occurs when the TC is composed with another relation. In the TC wrapper class the `compose` method is implemented such that instead of computing the full transitive closure, it will compute a partial transitive closure using `MultiSourceReach`. This is shown in the code snippet of the `compose` method: the expression at the `return` statement is obtained by applying the following rules:

$$\begin{aligned}
 A \circ B^+ & \\
 & \equiv \{\text{compose-restrict}\} \\
 A \circ (\text{rng } A \ll B^+) & \\
 & \equiv \{\text{reach}\} \\
 A \circ \text{MultiSourceReach}(B, \text{rng } A) &
 \end{aligned}$$

When a "none-TC" relation is composed with a TC relation (i.e. $A \circ B^+$), the none-TC relation is responsible for performing the optimisation. An example of this can be found in the code snippet for the `compose` method of the `PairSetRelation`. The `compose` method of this `PairSetRelation` checks whether the relation is an instance of the `TC` class. If so, an equivalent expression will be evaluated, based on the rules below. This is one of drawbacks of this optimisation technique: the implementation of the optimisation is not entirely local to the wrapper class. However, the only consequence of not adding the optimisation to a relation implementation, is that the optimisation will not work for that particular implementation. Moreover, for any relation that is

used as argument for the compose of TC, the optimisation will work, even when the relation itself does not implement the optimisation.

$$\begin{aligned}
& B^+ \circ A \\
& \equiv \{\text{compose-inverse}\} \\
& (A^{-1} \circ B^{+-1})^{-1} \\
& \equiv \{\text{compose-restrict}\} \\
& (A^{-1} \circ (\text{rng } A^{-1} \ll B^{+-1}))^{-1} \\
& \equiv \{\text{reach}\} \\
& (A^{-1} \circ \text{MultiSourceReach}(B^{-1}, \text{rng } A^{-1}))^{-1} \\
& \equiv \{\text{inverse domain range}\} \\
& (A^{-1} \circ \text{MultiSourceReach}(B^{-1}, \text{dom } A))^{-1} \\
& \equiv \{\text{compose-inverse}\} \\
& (\text{MultiSourceReach}(B^{-1}, \text{dom } A))^{-1} \circ A
\end{aligned}$$

For operations for which the Reach optimisation is not applicable (e.g. $A^+ \cup B$), the full transitive closure will be computed. This is done by calling the transitive closure method again on the current TC wrapper class, which can be observed in Figure 4.4 in the code snippet for `union` method of the TC class. In the code snippet of the `transitiveClosure` method we can see that when its called on an instance of TC, it will "unpack" the wrapped relation and compute its full transitive closure.

4.3.5 Representation

To efficiently implement relational operations, we have to select a representation for relations that has good time and space characteristics. Two important operations are the *rightSection* and *leftSection* operations that respectively return the vertices that are adjacent from and to a vertex. DFS, which we use to implement the SingleSourceReach operations, depends mostly on the *rightSection* to obtain the neighbours of a vertex. Therefore, an efficient implementation of the SingleSourceReach operations requires a relation representation that supports efficient lookup of vertices adjacent *from* a vertex. Relational composition requires efficient lookup of vertices adjacent *to* a vertex.

Two common representations for graphs (and binary relations) are the *adjacency matrix* and the *adjacency list*. The *adjacency matrix* of a graph $G(V, E)$ with $|V| = n$ and $|E| = e$ is an $n \times n$ boolean matrix A such that $A[i, j]$ is *true* if and only if G has an edge (v_i, v_j) . Enumerating the vertices adjacent to and from a vertex takes $\Theta(n)$ time, regardless of the in-degree or out-degree of v . The main disadvantage of the adjacency matrix is that it takes $\Theta(n^2)$ space, even when the graph is sparse. An example of an adjacency matrix representation of the graph from Figure 4.3 is shown in Figure 4.5.

	A	B	C	D	E	F	G	H	I	J
A	0	0	0	1	0	0	0	0	0	0
B	0	0	0	1	0	0	0	0	0	0
C	0	0	0	0	1	0	0	0	0	0
D	0	0	0	0	0	1	1	0	0	0
E	0	0	0	0	0	1	0	0	0	0
F	0	0	0	0	0	0	0	0	1	0
G	0	0	0	0	0	0	0	1	0	1
H	0	0	0	0	0	0	0	0	0	1
I	0	0	0	0	0	0	0	0	0	0
J	0	0	0	0	0	0	1	0	0	0

Figure 4.5: Adjacency matrix representation of example graph

An *adjacency list* is in essence a list of lists: the first list contains lists of vertices and is indexed by each of the connected vertices in the graph. Each vertex v in this list points another list $AdjFrom(v)$ that contains the vertices adjacent from v . An adjacency list takes $O(n + e)$ space, which makes it an economical representation for sparse graphs. Enumerating the vertices adjacent from a vertex v takes $O(Outdeg(v))$ time. However, enumerating the vertices adjacent to a vertex v takes $O(n + e)$ time, since we must check the presence of v in each $AdjFrom(v)$ list. If the vertices adjacent *to* (as opposed to *from*) a vertex are often needed, one can consider to add another list $AdjTo(v)$ for storing these vertices, which will result in a doubling of the space taken by the adjacency list. However, the time for enumerating the vertices adjacent to v will improve to $O(Indeg(v))$. An example of an adjacency matrix representation of the graph from Figure 4.3 is shown in Figure 4.5.

$$\begin{aligned}
 AdjFrom(A) &= \{D\} \\
 AdjFrom(B) &= \{D\} \\
 AdjFrom(C) &= \{E\} \\
 AdjFrom(D) &= \{F, G\} \\
 AdjFrom(E) &= \{G\} \\
 AdjFrom(F) &= \{I\} \\
 AdjFrom(G) &= \{H, J\} \\
 AdjFrom(H) &= \{J\} \\
 AdjFrom(I) &= \emptyset \\
 AdjFrom(J) &= \emptyset
 \end{aligned}$$

Figure 4.6: Adjacency list representation of example graph

Since the relations involved in source code querying are typically sparse, and obtaining the vertices adjacent to a vertex is an important operation for implementing Reach algorithms, we have chosen to use the adjacency list data structure to represent relations in JRelCal.

In Java, an adjacency list can be implemented with two collection interfaces from the Java Collections Framework (JCF): the `Map` and the `Set`. A `Map` is a structure that associates one object with another: it is a collection of entries, where each entry is a key-value pair. To implement an adjacency list we use a `Map` to associate a vertex with a list of its adjacent vertices. Since every tuple in a relations is unique, we use a `Set` for storing the adjacent vertices.

4.3.6 Benchmarks

The aim of the optimisation presented in this section is to improve the performance of JRelCal for queries involving reachability issues. Naturally, we can expect queries that use the new Reach operations (which compute a partial transitive closure) to perform better than queries that rely on the transitive closure operation (which computes a full transitive closure), depending on the number of source nodes. The effect of our marking optimisation is more difficult to predict. Therefore, we have experimentally compared the performance aspect time of our new implementation to previous implementations. This allows us to verify the theoretical performance gain in practice.

As subjects in our performance measurements we use call relations extracted from open-source Java projects of different sizes, since these represent relations that are typically query subjects in source code querying. Table 4.1 shows the systems from which we statically extracted a call graph. Most systems were parsed using SemmlCode, and a .QL query was used to obtain the call relation from the SemmlCode fact database. To be able to load the relation into JRelCal, it was written to a RSF file. The call relations for JDK 1.4.2 and for Eclipse 2.1.2 were obtained from the Crocopat distribution. This distribution includes the RSF files of the relations that were used for the performance benchmark in Beyer’s paper [10].

System name	Author/Owner	LOC	# Methods	# Calls
JPacMan 3.0.4	Arie van Deursen	2,499	335	757
JHotDraw 7.1	IFA Informatik & Erich Gamma	25,451	5,204	23,277
JDK 1.4.2	Sun Microsystems	784,244	2,785	17,533
JFreeChart 1.0.9	JFree	127,672	8,437	65,252
Eclipse 2.1.2	Free software community	1,440,346	6,066	48,135
Analyses 1.3.9	SIG	267,542	20,171	135,718

Table 4.1: Characterisation of the systems used for performance tests

We defined two queries involving reachability: the computation of the full transitive closure and the reachability relation between the top and bottom vertices. We implemented these queries as minimal and optimal as possible in order to get precise timing results.

Note that the shown results depend on the properties of the hardware, the operating system, and installed software of the machine the experiments were run on. The machine used for the experiments was a Intel Xeon dual core 2 Ghz with 8 Gb RAM running Linux 2.6.16.

Finally, it is important to stress that this benchmark focusses on the performance of reachability queries. It gives **no** indication of the overall performance of the code query tools whatsoever. An evaluation of the overall performance would require a balanced approach that includes a wider range of queries, each highlighting the strong points of each evaluated technology.

Reachability from top

Reachability from top is a query that returns a relation that relates each top vertex to its successors. Figure 4.7 shows the input and the produced output of this example. A top vertex is a vertex v such that $AdjTo(v) = \emptyset$.

Although this query is not directly useful for software re-engineering and comprehension tasks, it is useful as a test for measuring the performance of reachability algorithms. Most graphs have a considerable amount of top vertices that in this benchmark serve as sources for the partial transitive closure algorithms, and from the definition of a top vertex it follows that it always has vertices that are adjacent to it, unless the source is an isolated vertex.

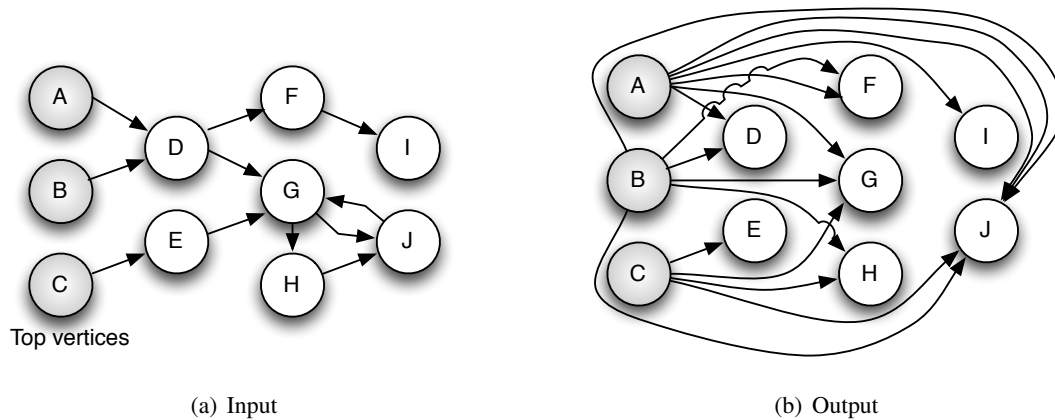


Figure 4.7: Example input and output of the top reachability analysis.

The code fragments below show the implementation of the query in JRelCal and Crocopat. The first JRelCal query uses the `MultiSourceReach` operation with the relation `CALL` and the `top` vertices as argument. The second JRelCal query computes the full transitive closure for the `CALL` relation, and then uses the `top` vertices to obtain the final result by restrict the domain of the transitive closure of the `CALL` relation. As we can observe from the query, Crocopat, in essence, uses the same strategy of computing the full transitive closure followed by a domain restriction.

```
1 Set<String> top = CALL.domain().difference(CALL.range());
2 Relation.reach(CALL, top);
```

Code Fragment 4.7: Reachability from top in JRelCal using `MultiSourceReach`

```
1 Set<String> top = CALL.domain().difference(CALL.range());
2 Relation.transitiveClosure(CALL).domainRestriction(top);
```

Code Fragment 4.8: Reachability from top in JRelCal using transitive closure

```
1 Top(x) := !CALL(_, x);
2 Reach(x, y) := TC(CALL(x, y)) & Top(x);
```

Code Fragment 4.9: Reachability from top in Crocopat

Figure 4.8 shows the results of this performance benchmark.

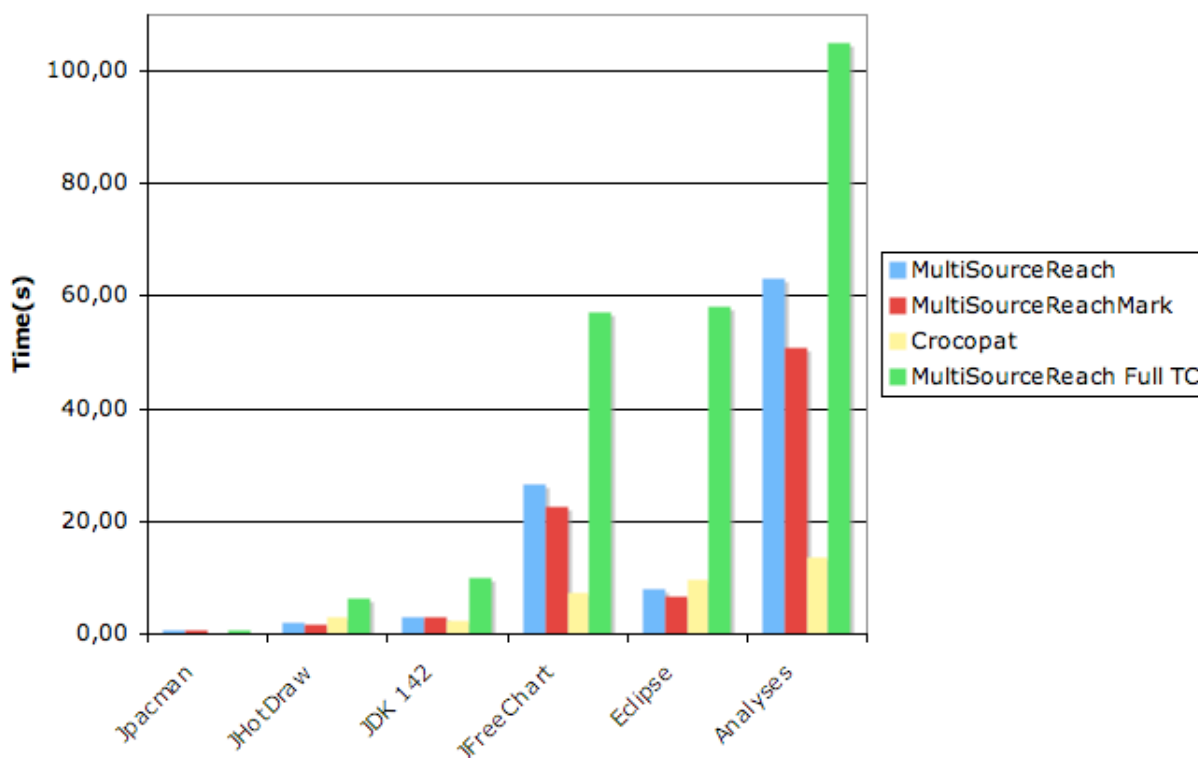


Figure 4.8: Runtime reachability from top

From these results we can observe that, as was expected, the queries that use a full transitive closure operation are generally slower than those that use a partial transitive closure operation. Since the full transitive closure is based on the Reach operations used for the partial transitive closure, it is clear that the selectivity of the Reach algorithms is the dominant factor affecting their performance.

Notice that the MultiSourceReach with the marking optimisation shows almost exact the same performance results as the one without marking. This can be explained by the fact that top nodes are used as sources: for every call of Reach for every vertex $v \in Top$, v will be marked as a source vertex used in a call of Reach. However, because of the definition of a top vertex, it will never be encountered by another call of Reach. The performance of the Reach algorithms heavily depends on the properties of the graph and the number of sources that is used. An example of this can be found in the peak for the JFreeChart call relation.

Table 4.2 gives a characterisation of the results of this query, which gives insight into the characteristics of the resulting relations.

	# Top vertices	# Result
JPacMan 3.0.4	88	4441
JHotDraw 7.1	686	131871
JDK 1.4.2	603	315621
JFreeChart 1.0.9	2421	1663442
Eclipse 2.1.2	514	810529
Analyses 1.3.9	7035	4881438

Table 4.2: Characterisation of the result relations of the reachability from top query

Transitive closure

In [10] Beyer carried out a performance comparison of the computation of full transitive closure in six different technologies (RelView, Grok, Quintus Prolog, MySQL, and Crocopat itself). To compare the performance of

the transitive closure operation, the tools were run on six call graphs extracted from open-source systems of different sizes. In this experiment Crocopat proved to be the most efficient tool for computing the transitive closure of large call relations.

Our optimisation approach improves the performance of reachability queries by computing partial transitive closures instead of full transitive closure. However, our approach to optimising reachability queries allows us to implement a transitive closure operation without much effort. We carried out an experiment similar to Beyer’s experiment in order to compare the efficiency of the different JRelCal transitive closure implementations with Crocopat.

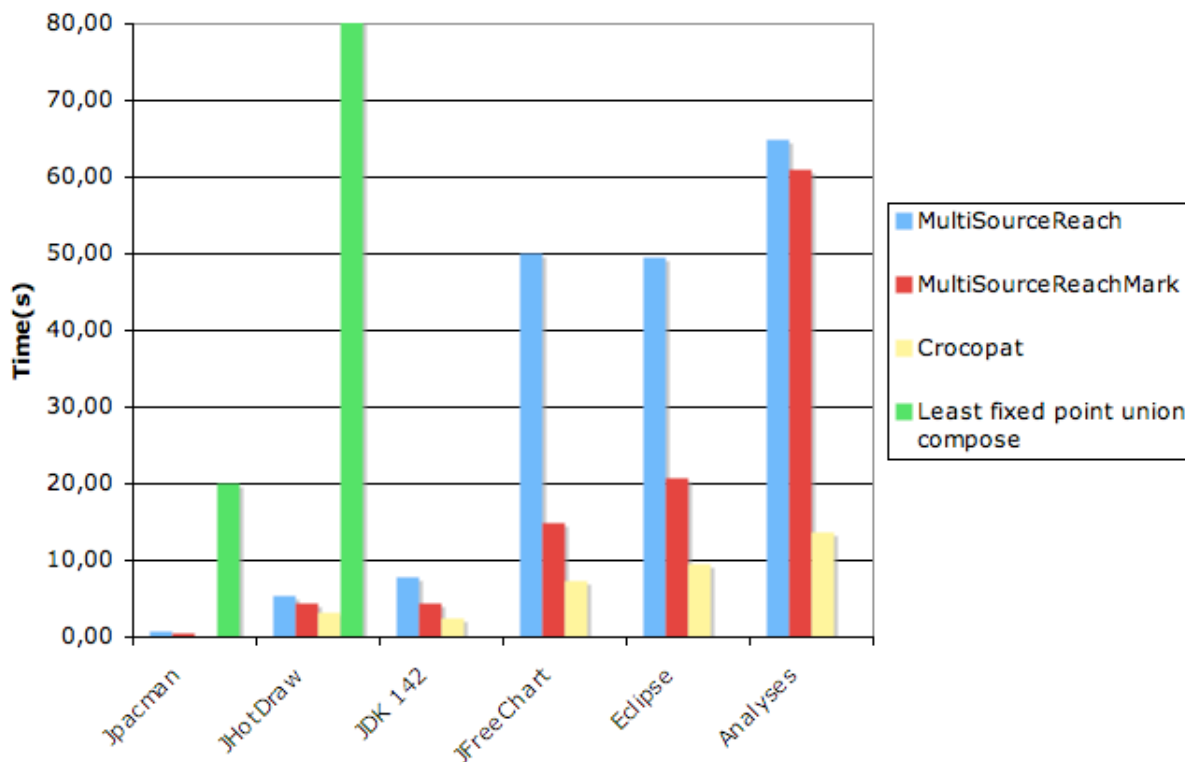


Figure 4.9: Runtime transitive closure

The results of the experiment in Figure 4.9 show that our marking optimisation for multi-source partial transitive closure is an effective optimisation. We can also observe the poor performance of the naive implementation of the original JRelCal implementation: for the largest inputs it ran out of memory. As was already shown in Beyer’s experiments, Crocopat proves to be very efficient in computing transitive closure. Nevertheless, our optimised MultiSourceReach shows comparable performance characteristics. However, for the largest input graph we see a sudden increase in runtime. This may be explained by the fact that the effectiveness of the marking optimisation depends on the order in which the sources for the MultiSourceReach are used as source for a single call of SingleSourceReachMark. Ideally, the sources are used in a reverse topological order. This ensures optimal use of the previously computed successor sets, thereby preventing redundant traversals of paths.

4.3.7 Discussion

As is shown by the benchmarks in this section, using Reach operations for reachability queries improves the performance compared to using a transitive closure operation to compute the full transitive closure. The transitive closure operation based on MultiSourceReach in combination with the marking optimisation has proved to be considerably more efficient than the original transitive closure operation of JRelCal. The two main reasons for this performance improvement are a better representation for relations and the new algo-

rithm. Our relatively simple optimisation technique of marking vertices that have been used a start for a single source Reach, proved to be an effective optimisation technique. Another advantage of our approach is that it is compositional: the single source Reach operations can be reused to implement multi-source partial transitive closure, and thus to implement a full transitive closure. This reduces the amount of code, which in turn reduces the effort required to maintain the code. We have shown how the Reach optimisation can be automatically applied at runtime. This relieves the programmer from manually having to identify opportunities for optimisation. A trade-off is that it adds complexity to the JRelCal project.

4.4 Integration with SAT

Integration of JRelCal with SAT makes the power of JRelCal available in the SAT, and vice versa: the various source code fact extractors of the SAT become available to use in combination with JRelCal.

There are two alternatives for integrating JRelCal with the SAT. Either we take a SIG graph and convert it to a set of JRelCal relations, or we implement JRelCal using the SIG graph library. Since a lot of effort has gone into the creation of a mature and efficient implementation of JRelCal, it is sensible to choose the former. An additional advantage of the conversion approach is that it is relatively simple to implement.

We have created a class with facilities to convert a SIG graph to JRelCal relations. The conversion method takes a SIG graph and an `EdgePredicate` as arguments, and converts it to a JRelCal relation. We use the unique identities of the graph's vertices as elements of the relation. Code Fragment 4.10 shows the implementation of the conversion method.

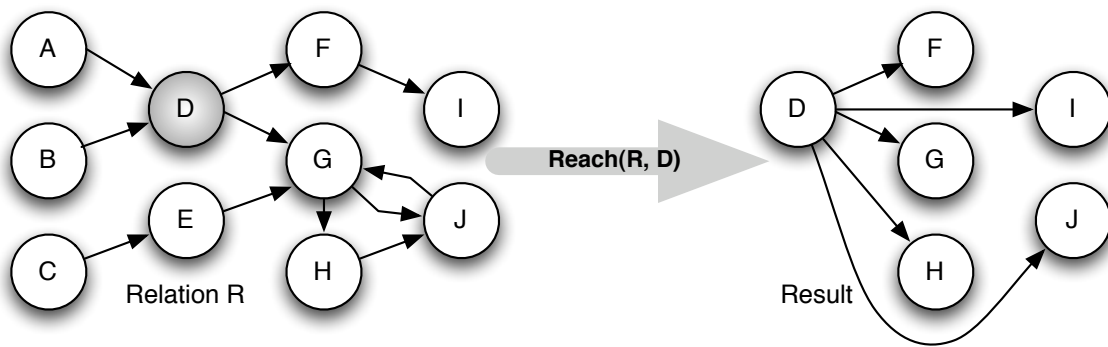
Instead of implementing the converter in a separate utility class, another possibility would be to add a constructor to the `Relation` class that takes a SIG graph as an argument. The disadvantage of this approach, however, is that a distribution of JRelCal would have to include the SIG graph classes.

```

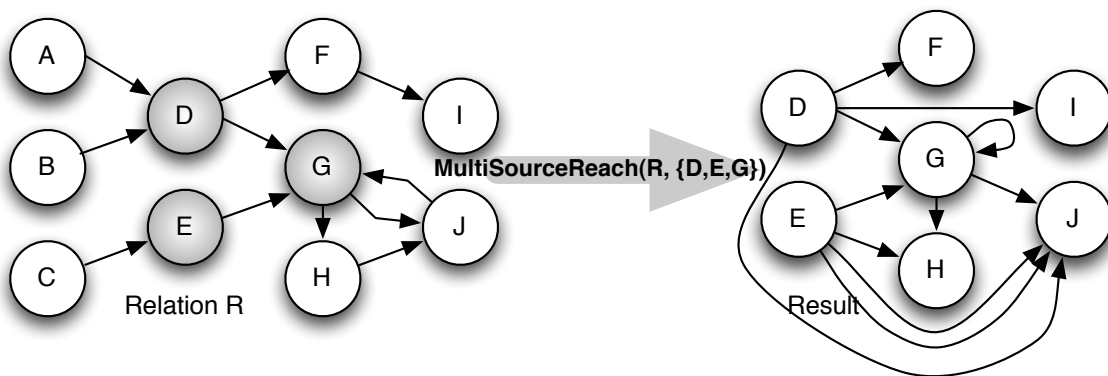
1     public Relation<String, String> getRelation(EdgeTypePredicate edgeTypePredicate) {
2         Relation<String, String> result = new PairSetRelation<String, String>();
3         for (IEdge e : graph.getEdges(edgeTypePredicate)) {
4             result.add(new Pair<String, String>(e.getFromNode().getIdentity().
5                 toNodeString(),
6                 e.getToNode().getIdentity().toNodeString()));
7         }
8         return result;
9     }
10    SIGGraph2Relation converter = new SIGGraph2Relation(f);
11    assertEquals(rel, converter.getRelation(new EdgeTypePredicate("_")));

```

Code Fragment 4.10: Method for converting a SIG graph to a JRelCal relation and its use



(a) Single-source transitive closure



(b) Strong multi-source transitive closure

Figure 4.3: Example inputs and outputs of single- and multi-source transitive closure

Chapter 5

Case study: Static estimation of test coverage

To verify the feasibility of the new JRelCal for use in industry (in particular for use at SIG) we have conducted a case study in which we re-implement queries used in the research work on the static estimation of test coverage of Alves et al. [4]. First, we will discuss the work of Alves et al., followed by an explanation of how we have re-implemented their work using JRelCal. We validate our re-implementation by repeating some of the original experiments and comparing our results with the results obtained by Alves et al.

5.1 Introduction

Test coverage is a measure for indicating how much of the production code is reached (or covered) by test code. Test coverage is a good complement to unit testing: unit tests indicate whether a unit of code performs as expected, and code coverage indicates what remains to be tested.

There are various tools available such as Clover¹ and EMMA² that compute code coverage dynamically by instrumenting the source code with logging functionality to keep track of the executed parts of the software system. Due to the dynamic nature of these solutions they require the availability of a running installation of the analysed software.

In the context of standard software development projects, a running installation is inherently available. However, in the context of third party evaluations of software, such as SIG's Software Risk Assessments, this may not be the case for various reasons: the software may require hardware that is not available to the SIG, proprietary libraries may be required that are under a non-transferable license, the build and deployment procedure may not be reproducible, or instrumentation may not be feasible, e.g., due to space or time limitations in case of embedded software.

Alves et al. show that it is possible to estimate test coverage by static analysis of the source code, which removes the need for having a running installation of the software. The static estimation is implemented both as relational queries in SemmlCode's .QL and as a graph algorithm in the SIG Graph library. For validation, the results are compared with those of the dynamic analysis tool Clover. The experimental results show a high correlation between static estimation of coverage and true coverage at all levels for a significant number of projects. However, outliers and high dispersion rates at package and class level lead to the conclusion that static analysis is a good estimator of test coverage at the system level only.

Static estimation of test coverage is a suitable candidate for our case study since it is a typical example of an analysis that SIG would like to perform on the source code of its clients. Furthermore, it is a scenario that entails multiple aspects of the use of a source code query tool.

¹<http://www.atlassian.com/software/clover>

²<http://emma.sourceforge.net>

5.2 Approach

We follow the same approach as Alves et al. This approach involves reachability analysis on a graph structure that is based on relations derived from the source code by static analysis. Although the derived structural information is exact, the static nature of the derivation process implies that the call information is an approximation of the actual dynamic execution. This implies that the precision of the resulting estimation depends on the precision of the call graph extraction. In their paper, Alves et al. discuss various sources of imprecision including dynamic dispatching and control flow. Please refer to the paper for a more detailed discussion [4].

The approach can roughly be divided into the following steps:

1. Static analysis is performed to extract relations from the source code (both test and production code). These relations represent structural and call information.
2. The test classes are identified and collected as a set.
3. To determine the production methods covered by the test class methods, a reachability analysis is performed on a graph view of the extracted relations. The reachability analysis is done by performing the MultiSourceReach operation using the test methods as sources.
4. To take into account call edges originating from class initialisers, the *defines method* relation is used to determine the classes that define the covered methods. The class initialisers of these classes are used as sources for the MultiSourceReach operation to determine the production methods covered by class initialisers.
5. For each production class, the coverage is computed by taking the ratio of the number of methods it defines and the number of covered methods.
6. The coverage estimations at package and system level are computed as ratio of the coverage counts at class level.

In the traditional, dynamic, computation of test coverage three levels of code coverage are commonly distinguished: statement, branch, and path coverage. Statement coverage indicates which statements in a method or class have been executed, branch coverage which decision outcomes have been executed, and path coverage which of the possible execution paths have been executed. Alves et al. compute coverage by statically determining the methods that have been called by the test code. We call this method coverage. The statically determined method coverage is less precise than the statement coverage level for dynamic test coverage, e.g., 100% method coverage does not guarantee 100% statement coverage.

5.3 Implementation

The re-implementation of the static coverage query in JRelCal is guided by the steps listed in the previous section. For this case study, the three most interesting steps of the approach are 2, 3, and 4: distinguishing test code from production code, and identifying the covered methods using MultiSourceReach. The JRelCal implementation of both steps is discussed in more detail below. The extraction of relations from the source code in the first step is done using SemmleCode. This way, we ensure that we use the same relations as Alves et al. The last two steps, the computation of the coverage ratios and the counting of the methods, are based on the information derived at step 2, 3, and 4. The computations in these steps involve basic arithmetical operations and are therefore, for the purpose of validating a code query technology, not interesting.

5.3.1 Distinguishing test code from production code

Often, production and test code is stored in different file system paths. Using this knowledge, a simple heuristic for identifying test classes is to check whether a class is stored in a subdirectory of a known test directory. For systems where this convention does not apply, other heuristics can be used, e.g. checking

whether the class subclasses the JUnit `TestCase` class, or checking for a naming convention such as class names ending with `Test`.

```

1 Predicate<String> containsTest = new Predicate<String>() {
2     public boolean evaluate(String s) {
3         return s.contains("/test/");
4     }
5 };
6
7 Set<String> testClasses = LOC.leftSection(containsTest);
8 Set<String> testMethods = DM.rightSection(testClasses);

```

Code Fragment 5.1: Distinguishing test code from production code in JRelCal

In JRelCal, this heuristic can be expressed as a predicate. Code Fragment 5.1 shows the creation of the predicate and the identification of the test methods. To improve readability the names of relation variables are in uppercase. `DM` is the *defines method* relation representing that a class or interface defines a method, and `LOC` relates classes to the file system path of the file they are contained in. Both relations are extracted by `SemmlCode`, written to disk in a RSF file, and read into memory by `JRelCal`. After creation of the predicate, we take the left section (or image) of the `LOC` relation using the predicate `containsTest` to arrive at the test classes. We take the right section of the `DM` relation using the test classes to arrive at the test methods. This example shows that the extension of JRelCal with support for predicates allows us to elegantly express filtering of relations.

5.3.2 Identification of covered methods

In order to detect all covered methods we have to perform the `MultiSourceReach` operation twice: the first time using the test methods identified in the previous section as sources, the second time using class initialisers as sources. Figure 5.1 depicts the traversals of both instances of the `MultiSourceReach` operation.

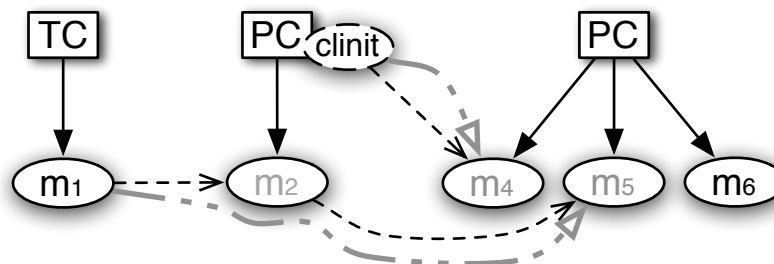


Figure 5.1: `MultiSourceReach` using both methods defined in test classes (TC) and class initialisers (`clinit`) as sources. The grey arrows depict the `MultiSourceReach` traversals, the solid black arrows depict method definition edges, and the dashed black arrows depict methods calls. The methods indicated in grey font and defined in production classes (PC) are identified as covered methods.

As discussed in Section 4.3, we have extended JRelCal with operations that allow us to express reachability concepts in a query more efficiently. This can be observed in the first line of Code Fragment 5.2: the `multiSourceReach` method computes a partial transitive closure and hides the actual traversing of the graph.

To take into account calls originating from class initialisers we identify the classes that are initialised by using the results from the first call to `multiSourceReach` to take the left section of the `DM` relation. This results in all the classes that define a covered method, and therefore are initialised. From these classes we collect the class initialisers using the predicate `isClInit` and use these as sources for the second call to `multiSourceReach` (Code Fragment 5.2, line 5) to obtain the methods that are covered indirectly via class

initialisers. Finally, we merge the results of both calls to `multiSourceReach`. The result is a set containing all covered production methods.

```

1 Set<String> directlyTestedMethods = multiSourceReach(CALL, testMethods);
2
3 Set<String> initializedClasses = DM.leftSection(directlyTestedMethods);
4 Set<String> clInits = DM.rangeRestriction(isClInit).rightSection(initializedClasses);
5 Set<String> indirectlyTestedMethods = multiSourceReach(CALL, clInits);
6
7 Set<String> allTestedMethods = directlyTestedMethods.union(indirectlyTestedMethods);

```

Code Fragment 5.2: Identification of covered methods using `MultiSourceReach`

5.4 Discussion

In this section we discuss the validation of our `JRelCal` implementation and compare the query to the `.QL` and `SIG` graph library variants by Alves. Since we have already compared the query languages in detail in Chapter 3 we will limit the discussions in this section to the most significant parts of the queries: the identification of test methods, and the identification of covered methods.

5.4.1 Validation

To validate our `JRelCal` re-implementation we have rerun the query for three of the 12 systems that were used in the experiment of Alves. These systems are listed in Table 5.1. For all three systems the `JRelCal` queries returned the exact same results as the original `.QL` query, making it safe to conclude that the re-implementation is correct with respect to the original query.

System name	Version	Author / Owner	Description	LOC	#Packages	#Classes	#Methods
JPacMan	3.04	Arie van Deursen	Game used for OOP educational purposes	2,499	3	46	335
Utils	1.61	SIG	Library of tools for static code analysis	37,738	37	506	4,533
Analysis	1.39	SIG	Tools for static code analysis	267,542	284	3,199	22,315

Table 5.1: Characterisation of the systems used in the experiment order by LOC.

5.4.2 `JRelCal` compared to `SIG` Graph library

The implementation of the algorithm in the `SIG` Graph library is less declarative and less concise compared to the `JRelCal` implementation. In this section we discuss the most significant differences to the `JRelCal` implementation.

Code Fragment 5.3 demonstrates how the identification of test methods is implemented in the `SIG` graph library: first, all nodes of the graph that are in the "test" java source context are collected using a predicate. From these nodes the class and interface nodes are selected, again using a predicate. For every class and interface, the test methods are collected by selecting the out nodes of `METHOD_DEFINE` edges using a `EdgeTypePredicate`. When we compare this implementation to how the `JRelCal` implementation we observe that in `JRelCal` the for-loops for iterating through the nodes is hidden by the `leftSection` and `rightSection` operations, which results in a more concise and declarative query.

```

1 private Collection<INode> collectTestMethods(FileGraph graph) {
2     Collection<INode> result = new ArrayList<INode>();
3     Collection<INode> sourceNodes = graph.getNodes(new JavaSourceContextPredicate("test"))
4     ;
5     for (INode sourceNode : sourceNodes) {

```

```

5     Collection<INode> testClassesAndInterfaces = sourceNode.getOutNodes(new
        SourceNodeDefinedTypesPredicate());
6     for (INode classOrInterface : testClassesAndInterfaces) {
7         Collection<INode> testMethods = classOrInterface.getOutNodes(new EdgeTypePredicate
            (Edges.METHOD_DEFINE));
8         result.addAll(testMethods);
9     }
10    }
11    return result;
12 }

```

Code Fragment 5.3: Identification of test methods using the SIG graph library

The slice method is similar to the `multiSourceReach` method in that it implements a graph traversal to collect the successors of a set of source nodes. The main differences can be found in that for a SIG graph we have to indicate which edge types are used for the traversal (the `AllCallsEdgePredicate`), and that slice is defined less generically than its counterpart `multiSourceReach`, since its definition only considers call edges. To make it more general the predicate for selecting the edges should be exposed as parameter of the method.

```

1 private Collection<INode> slice(Collection<INode> criterium, FileGraph graph) {
2     Collection<INode> visited = new ArrayList<INode>();
3     Collection<INode> toVisit = new ArrayList<INode>();
4     toVisit.addAll(criterium);
5     while (!toVisit.isEmpty()) {
6         INode methodNode = toVisit.iterator().next();
7         toVisit.remove(methodNode);
8         visited.add(methodNode);
9         Collection<INode> calledMethods = methodNode.getOutNodes(new AllCallsEdgePredicate()
            );
10        calledMethods.addAll(getInitializerCalledMethods(methodNode));
11        for (INode calledMethod : calledMethods) {
12            if ((!visited.contains(calledMethod)) && (!toVisit.contains(calledMethod)))
13                toVisit.add(calledMethod);
14        }
15    }
16    return visited;
17 }

```

Code Fragment 5.4: Slicing to collect covered methods using the SIG graph library

5.4.3 JRelCal compared to .QL

Code Fragment 5.5 and 5.6 show the two most essential aspects of the .OL implementation of the detection of production methods that are covered by test methods.

```

1 predicate invoke(Callable m1, Callable m2) {
2     myPolyCall(m1,m2)
3     or
4     exists(Class c, Callable mi, Callable mj |
5         myPolyCall(m1,mi) and
6         c.contains(mi) and
7         c.contains(mj) and
8         mj.getName() = "<clinit>" and
9         myPolyCall(mj,m2)
10    )
11 }

```

Code Fragment 5.5: .QL fragment showing the adapted notion of method invocation to include calls from class initialisers

Fragment 5.5 shows Alves' variation on the default `invoke` definition of `SemmlCode`. The difference with the original implementation is that the new implementation takes into account methods that are called from class initialisers. The predicate expresses that an invocation is either a `polyCall` from method `m1` to method `m2` (defined in the standard .QL library to include virtual calls as well as standard method calls), or a call of method `m1` to a method that is defined in a class whose class initialisers calls method `m2`.

```
1 int numberOfTestMethodCallers() {
2   result = count(TestClass tc, Callable tm | tc.contains(tm) and invoke+(tm,this))
3 }
```

Code Fragment 5.6: .QL fragment showing the use of `invoke` in the .QL class that represents production code

Code Fragment 5.6 shows how the reachability issue is expressed by appending the `+` operation to the `invoke` method. While in `JRelCal` we use the `multiSourceReach` method to express partial transitive closure, in `SemmlCode` it depends on the optimisations performed by the query compiler whether the full transitive closure for the `invoke` relation will be computed. The `numberOfTestMethodCallers` method is defined as a member of a .QL class that represents a production class, this class defines predicates that define a production class, analogously to a class that defines a test class. In this particular implementation production and test classes are distinguished by their location on disk. An example of such a predicate is shown in Code Fragment 5.7

```
1 predicate isInTestPath(CompilationUnit cu) {
2   cu.getPath().matches("%/test/%")
3 }
```

Code Fragment 5.7: .QL fragment showing the predicate used to identify test classes

5.5 Conclusion

In this case-study `JRelCal` has shown to be a good alternative for the `SIG` graph library. `JRelCal` allows us to express queries in a more concise and declarative way than is possible with the `SIG` graph library.

The main goal of this thesis project was to find a successor for the SIG graph library that allows SIG to concisely, declaratively, and efficiently formulate queries on relations extracted from the source code by their SAT.

Our first step was to compare existing source code querying tools. To do this, we have formulated a set of criteria to compare tool features and suggested four benchmark queries to compare language features. From the comparison we concluded that none of the tools entirely met the requirements of the SIG. Only JRelCal offered both abstraction facilities and extendability in combination with an API for easy integration with the SAT.

Since the original JRelCal was a prototype that had shortcomings in both design and implementation, we have created a new and mature version that includes new features and a new implementation. First, we have created an abstract relation class which facilitates adding new implementations. In the new version we represent relations as adjacency lists which allows efficient implementation of many relational operations. Furthermore, we have extended JRelCal with support for predicates. Predicates make it possible to conveniently express restrictions and exclusions on sets and relations. To integrate JRelCal with SIG's SAT we have created a converter that converts a SIG graph to a JRelCal relation. In this way, a seamless integration with the SAT is obtained.

Transitive closure is one of the most computation intensive operations of JRelCal. Our approach in optimising queries involving transitive closure is to add a partial transitive closure operation. In many queries the full transitive operation can be replaced by the cheaper partial transitive closure operation, which prevents unnecessary computation of successor sets. The optimisation can either be applied manually or by using a runtime optimisation technique. For the validation of our transitive closure optimisation we have carried out several experiments to measure the performance on different types of graphs. We compared the performance with Crocopat and previous JRelCal implementations. The results show that it is an effective optimisation.

We have validated the new JRelCal in a case study in which we re-implement the work of Alves et al. The case study shows that JRelCal allows us to re-implement the queries of Alves both concisely, declaratively, and efficiently.

6.1 Research question

In Section 1.3 we formulated our research question:

Can we find a successor of the SIG graph library that allows the SIG to formulate source code queries concisely, declaratively and efficiently?

Based on the results described in this thesis we can positively answer this question: JRelCal allows SIG to formulate their source code queries concisely, declaratively and efficiently.

6.2 Contributions

The contributions of our work can be summarised as follows:

- The formulation of a set of benchmark queries to compare and characterise the languages of five source code querying tools.
- The formulation of a set of ten criteria: six criteria to compare tool features, four to compare language features.
- A comparison of the languages and tool features of five different source code querying tools with respect to the benchmark queries and the criteria.
- A declarative, concise, and efficient way to express source code queries.
- A new source code query tool library with the following unique combination of features:
 - Predicates to conveniently express restrictions and exclusions on sets and relations.
 - An efficient representation of relations based on adjacency lists.
 - Efficient evaluation of reachability queries.
 - The possibility to add new implementations of relations.
 - Conversion mechanisms allowing the creation of relations from SIG graphs and RSF files.
 - A large collection of unit tests.

6.3 Future work

As a result of this thesis project, we now essentially have two projects: the comparison of code query technologies, and JRelCal. Both can be further improved and extended. First, we explore some ideas for further work on our tool comparison, followed by a discussion of several possibilities for future work on JRelCal.

6.3.1 Tool comparison

As we explained in Section 3.1, some code querying tools we have encountered have not been considered in this paper. This is certainly something that can be addressed in the future. Besides the ones we have seen thus far, there may be tools that we have missed, and in the future new tools will probably emerge.

A different dimension to extend the tool comparison in is to add more criteria. The most important of these is performance. Although we have actually measured the performance for most of the tools, the results are still inconclusive and we need to spend more time to be able to compare the tools fairly and correctly. For this reason, we have omitted the performance criterion from the tool comparison.

6.3.2 Optimisation by algebraic transformation

Numerous algebraic transformations can be used to rewrite an expression into an equivalent expression. A trivial example of an algebraic simplification is the following:

$$R^{-1^{-1}} = R$$

To avoid unnecessary computations, the double inverse can be eliminated.

Another transformation is called "reduction in strength" [2]. This is the replacement of a statement by a cheaper, but equivalent statement. An example of such an transformation is the optimisation of transitive closure operations by replacing them with a Reach operation when applicable. Besides this example, there may be more opportunities for applying the "reduction in strength" pattern.

Note that domain-specific optimisations and transformations may be hard to realise in a general-purpose language such as Java without resorting to using program transformation tools or adapting the compiler. However, in Section 4.3.4 we have shown how the "Reach optimisation" can be implemented within Java as a runtime optimisation.

6.3.3 Optimisation by improving implementation

Another approach to optimising JRelCal is to improve its implementation. We distinguish two types of implementation optimisations: representation optimisation and optimisation of operations.

Depending on the properties of a relation there may be an optimal representation. For example, to represent sparse graphs, we currently use an adjacency list representation. However, when a graph is dense, an adjacency matrix is a better representation. Other examples of properties of relations are the size of a relation, or whether a relation is injective, bijective, etc. The performance of operations partially depends on the representation of a relation. As we have mentioned in Section 4.3.5, a depth first search algorithm relies heavily on obtaining the neighbours of a vertex. It mainly depends on the representation whether this operation can be implemented efficiently.

Relational operations such as computing the transitive closure or the inverse of a relation can also be optimised by improving the algorithm or replacing it by a new algorithm. For example, it would be interesting to implement Tarjan's [67] algorithm for computing transitive closure and see how it compares to our current implementation.

6.3.4 Supporting queries involving n -ary relations

In examining the expressiveness of his Calculus of Binary Relations [68], Tarski found that the following can not be expressed in his binary relational calculus:

$$\forall x \forall y \forall z \exists u (xRu \wedge yRu \wedge zRu)$$

If we would try to express this first order logic expression in binary relational calculus the first step is to use relational composition to 'simulate' the existential quantification $\exists u$. To do this, we first rewrite the previous expression to the following, equivalent expression:

$$\forall x \forall y \forall z \exists u (xRu \wedge uR^{-1}y \wedge uR^{-1}z)$$

This makes it possible to express the first part of the expression like this:

$$R \circ R^{-1}$$

However, we cannot simulate the last part of the expression using relational composition. Veloso [72] explains that the reason for this is that BRC has no variables over individuals and the relational composition $R \circ R^{-1}$ "consumes", so to speak, the variable u .

We can overcome this difficulty by extending BRC with a new operation: the fork operation. In [72], Veloso shows that the extension of binary relational calculus with a fork operation will make it possible to express this query. Moreover, Veloso proves that by extending the BRC with the fork operation and an additional concatenation operation and two new constants, BRC will have the expressive power of first-order logic. The fork operation is defined as follows:

$$R \nabla S = \{(x, (y, z)) \mid xRy \wedge xSz\}$$

The extension of BRC with the fork operation allows us to express Tarski's problem as follows:

$$R \circ (R^{-1} \nabla R^{-1})$$

In JRelCal we can provide the fork operation with the following method signature:

```
public Relation<S, Pair<T,T>> fork(Relation<S,T> relation){...}
```

It would be interesting to see how we can properly add the fork operation to JRelCal and see whether it is a useful operation in the domain of source code analysis.

6.3.5 Other language constructs

During the tool comparison we found that some tools had useful language constructs that are not available in JRelCal. It would be interesting to take a closer look at these constructs and see whether they can be added to JRelCal. Since most of these language constructs rely on syntactic constructs, we expect that it will be hard to elegantly implement these in Java.

- **List Comprehensions** A powerful construct we found in Rscript is list comprehension. The Rscript implementations of the benchmark queries in Section 3.3 show that list comprehensions allow the concise expression of many code queries.
- **Existentials** Crocopat offers an existential quantifier operation. An existential quantifier can be useful to express queries that only need one condition to hold to be able to conclude something, e.g., lifting from class to package level only requires the discovery of one relation on class level to have a relation on package level.
- **Aggregates** SemmlCode offers an aggregate constructs that allows for concise expression of counting, summing, minning, maxing, or averaging of values. A concrete example is `select sum(Class c | c.getPackage().getName().matches("org.jhotdraw%") | c.getNrOfMethods())` in which the sum of the lines of code in all packages that match the regular expression `org.jhotdraw%` is computed. The general format for an aggregation is: `aggfunc(declarations | condition | value)` Here `aggfunc` is one of `count`, `sum`, `max`, `min`, or `avg`(for average).
- **Regular path expressions** GReQL 2 supports regular path expressions, which make it possible to search for paths that satisfy particular properties. They can be useful to express reachability queries that require the path to the reachable vertices to satisfy particular properties.

6.3.6 Creation of a DSL on top of JRelCal

By creating a DSL on top of JRelCal we reuse its implementation, and a DSL can make it easier to implement some of our previous future work suggestions: implementing algebraic optimisations becomes easier because these can be performed at compile time in the DSL compiler, and since we can create or own syntax, adding new syntactic constructs becomes possible.

6.4 Acknowledgments

In this section I would like to thank the people that helped and supported me during the time I worked on my thesis project.

First of all, thanks to my supervisors Joost Visser (SIG) and Jurriaan Hage (Utrecht University) for their diligent guidance and their time invested in reviewing the various drafts of my thesis.

Together with Tiago Alves I have written the position paper. During that time he helped me in sharpening the tool comparison in my thesis, thanks! Thanks to Tijs van der Storm for helping me to get started with the JRelCal project.

The authors of some of the query tools were very helpful in providing feedback on our position paper and helping us with example queries, which was much appreciated: Dirk Beyer, Ric Holt, Oege de Moor, Mathieu Verbaere, Rudolf Berghammer, and Daniel Bildhauer.

And last but not least my colleagues at the SIG. The students that shared the office with me: Bart, Christiaan, Tim, Zeeger, and the system administration team: Gerard and Leo.

Bibliography

- [1] Hilde Abold-Thalmann, Rudolf Berghammer, and Gunther Schmidt. Manipulation of concrete relations: The relview-system. Technical Report 8905, Universität der Bundeswehr München, 1989.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools*. Pearson Education Singapore, 1986.
- [3] Tiago M. Alves and Peter Rademaker. Evaluation of code query technologies for industrial use. In *Proceedings of the Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, 2008.
- [4] Tiago M. Alves and Joost Visser. Static estimation of test coverage.
- [5] Rudolf Berghammer and Gunther Schmidt. The relview-system. In *STACS 91: Proceedings of the 8th annual symposium on Theoretical aspects of computer science*, pages 535–536, New York, NY, USA, 1991. Springer-Verlag New York, Inc.
- [6] Dirk Beyer. Relational programming with crocopat. In *International conference on Software engineering (Tool Demo)*, pages 807–810, Shanghai, China, 2006. IEEE.
- [7] Dirk Beyer and Claus Lewerentz. CrocoPat: A tool for efficient pattern recognition in large object-oriented programs. Technical Report I-04/2003, Institute of Computer Science, Brandenburgische Technische Universität Cottbus, January 2003.
- [8] Dirk Beyer and Andreas Noack. Crocopat 2.1 introduction and reference manual. Technical Report UCB/CSD-04-1338, EECS Department, University of California, Berkeley, Jul 2004.
- [9] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the Tenth IEEE Working Conference on Reverse Engineering (WCRE 2003, Victoria, BC, November 13-16)*, pages 216–225. IEEE Computer Society Press, Los Alamitos (CA), 2003.
- [10] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31(2):137–149, 2005.
- [11] Daniel Bildhauer and Jürgen Ebert. Querying software abstraction graphs. In *Proceedings of the Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, 2008.
- [12] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [13] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *Knowledge and Data Engineering, IEEE Transactions on*, 1(1):146–166, 1989.
- [14] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The c information abstraction system. *IEEE Trans. Softw. Eng.*, 16(3):325–334, 1990.

-
- [15] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
- [16] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *CASCON '91: Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, pages 17–35. IBM Press, 1991.
- [17] Thomas A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.
- [18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
- [19] Roger F. Crew. Astlog: a language for examining abstract syntax trees. In *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.
- [20] Peter Dahm. Architektur des GReQL–Auswerters. Projektbericht 11/97, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1997.
- [21] Shaul Dar and Raghu Ramakrishnan. A performance study of transitive closure algorithms. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 454–465. ACM Press, 1994.
- [22] Oege de Moor, Elnar Hajiyev, and Mathieu Verbaere. Object-oriented queries over software systems: (abstract of invited talk). In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 91–91, New York, NY, USA, 2007. ACM.
- [23] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .ql for source code analysis. In *SCAM '07: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 3–16, Washington, DC, USA, 2007. IEEE Computer Society.
- [24] Jan Van den Bussche. Applications of alfred tarski's ideas in database theory. *Lecture Notes in Computer Science*, 2142:20–??, 2001.
- [25] A. van Deursen and T. Kuipers. Building documentation generators. In *Proc. Int. Conf. on Software Maintenance*, pages 40–49. IEEE Computer Society, 1999.
- [26] Michael Eichberg, Michael Haupt, Mira Mezini, and Thorsten Schafer. Comprehensive software understanding with sextant. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 315–324, Washington, DC, USA, 2005. IEEE Computer Society.
- [27] Institute O. Electrical and Electronics E. (ieee). *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology*. 1990.
- [28] Solomon Feferman. Tarski's influence on computer science. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, page 342, Washington, DC, USA, 2005. IEEE Computer Society.
- [29] L. Feijs, R. Krikhaar, and R. Van Ommering. A relational approach to support software architecture analysis. *Softw. Pract. Exper.*, 28(4):371–400, 1998.
- [30] Norman E. Fenton and Shari L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach, Revised*. Course Technology, February 1998.

- [31] Alexander Fronk and Rudolf Berghammer. Considering design problems in oo-software engineering with relations and relation-based tools. *Journal on Relational Methods in Computer Science (JoRMiCS)*, (1):73–92, December 2004.
- [32] Michael T. Goodrich and Roberto Tamassia. *Data structures and algorithms in Java*. Second edition, 2001.
- [33] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris De Volder. Codequest: querying source code with datalog. In Ralph Johnson and Richard P. Gabriel, editors, *OOPSLA Companion*, pages 102–103. ACM, 2005.
- [34] C. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
- [35] Jr. Henry S. Warren. A modification of warshall’s algorithm for the transitive closure of binary relations. *Commun. ACM*, 18(4):218–220, 1975.
- [36] R. Holt, A. Winter, and J. Wu. Towards a common query language for reverse engineering, 2002.
- [37] R. C. Holt. Binary relational algebra applied to software architecture. CSRI Tech Report 345, University of Toronto, March 1996.
- [38] Richard C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *WCRE ’98: Proceedings of the Working Conference on Reverse Engineering (WCRE’98)*, page 210, Washington, DC, USA, 1998. IEEE Computer Society.
- [39] Richard C. Holt. Wcre 1998 most influential paper: Grokking software architecture. In *WCRE*, pages 5–14. IEEE, 2008.
- [40] Richard C. Holt and James R. Cordy. The turing programming language. *Commun. ACM*, 31(12):1410–1423, 1988.
- [41] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a Standard Exchange Format. Fachberichte Informatik 1–2000, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2000.
- [42] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD ’03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA, 2003. ACM.
- [43] Stan Jarzabek. Design of flexible static program analyzers with pql. *IEEE Transactions on Software Engineering*, 24, 1998.
- [44] Manfred Kamp. GReQL - eine Anfragesprache für das GUPRO-Repository 1.1. Projektbericht 8/96, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1 1996.
- [45] Manfred Kamp. GReQL - eine Anfragesprache für das GUPRO-Repository. In Jürgen Ebert, Rainer Gimnich, H. Stasch, and Andreas Winter, editors, *GUPRO — Generische Umgebung zum Programmverstehen*, pages 173–202. kein Verlag zugeordnet, 1998.
- [46] P. Klint. How understanding and restructuring differ from compiling—a rewriting perspective. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC03)*, pages 2–12. IEEE Computer Society, 2003.
- [47] Paul Klint. *A tutorial introduction to Rscript*. CWI, May 2005.

- [48] T. Kuipers and J. Visser. A tool-based methodology for software portfolio monitoring. In Mario Piattini and Manuel Serrano, editors, *Proc. 1st Int. Workshop on Software Audit and Metrics, (SAM 2004)*, pages 118–128. INSTICC Press, 2004.
- [49] Tobias Kuipers and Joost Visser. Object-oriented tree traversal with *jjforester*. *Electr. Notes Theor. Comput. Sci.*, 44(2), 2001.
- [50] Tobias Kuipers, Joost Visser, and Gerjon de Vries. Monitoring the quality of outsourced software. In Jos van Hillegersberg et al., editors, *Proc. Int. Workshop on Tools for Managing Globally Distributed Software Development (TOMAG 2007)*. Center for Telematics and Information Technology (CTIT), The Netherlands, 2007.
- [51] Leonid Libkin. Expressive power of SQL. *Lecture Notes in Computer Science*, 1973:1–??, 2001.
- [52] Mark A. Linton. Implementing relational views of programs. *SIGPLAN Not.*, 19(5):132–140, 1984.
- [53] B. J. MacLennan. Introduction to relational programming. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 213–220, New York, NY, USA, 1981. ACM.
- [54] B. J. MacLennan. Overview of relational programming. *SIGPLAN Not.*, 18(3):36–45, 1983.
- [55] B. J. MacLennan. Four relational programs. *SIGPLAN Not.*, 23(1):109–119, 1988.
- [56] Bruce J. MacLennan. A simple software environment based on objects and relations. In *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, pages 199–207, New York, NY, USA, 1985. ACM.
- [57] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
- [58] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, 1976.
- [59] H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 80–86, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.
- [60] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of moose: an agile reengineering environment. *SIGSOFT Softw. Eng. Notes*, 30(5):1–10, 2005.
- [61] Esko Nuutila. Efficient transitive closure computation in large digraphs. *Acta Polytechnica Scandinavia: Math. Comput. Eng.*, 74:1–124, 1995.
- [62] J. N. Oliveira. First steps in pointfree functional dependency theory. 2005.
- [63] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.
- [64] Santanu Paul and Atul Prakash. Querying source code using an algebraic query language. In *In Proceedings of the International Conference on Software Maintenance*, pages 127–136. IEEE Computer Society Press, 1994.
- [65] Bruno R. Preiss. *Data structures and algorithms with object-oriented design patterns in C++*. John Wiley & Sons, Inc., New York, NY, USA, 1999.
- [66] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. Osborne/McGraw-Hill, Berkeley, CA, USA, 2000.

- [67] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
- [68] A. Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6:73–89, 1941.
- [69] Sander Tichelaar, StÉphane Ducasse, and Serge Demeyer. Famix and xmi. *Reverse Engineering, Working Conference on*, 0:296, 2000.
- [70] Tijs van der Storm. Jrelcal.
- [71] Arie van Deursen and Tobias Kuipers. Source-based software risk assessment. In *ICSM '03: Proc. Int. Conference on Software Maintenance*, page 385. IEEE Computer Society, 2003.
- [72] P A S Veloso and A M Haebeler. A finitary relational algebra for classical first-order logic. In *Presented at 9th international congress of Logic, Methodology and Phylosophy of Science*, pages 52–62, 1991.
- [73] Mathieu Verbaere, Elnar Hajiyev, and Oege De Moor. Improve software quality with semmlecode: an eclipse plugin for semantic code search. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 880–881, New York, NY, USA, 2007. ACM.
- [74] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.