A Comparative Study of Code Query Technologies

Tiago L. Alves

Jurriaan Hage

Peter Rademaker

Technical Report UU-CS-2011-009
April 2011

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

A Comparative Study of Code Query Technologies

Tiago L. Alves
Software Improvement Group, Netherlands
University of Minho, Portugal
t.alves@sig.eu

Jurriaan Hage
University of Utrecht, Netherlands
jur@cs.uu.nl

Peter Rademaker
University of Utrecht, Netherlands
peter.rademaker@gmail.com

Abstract

When analyzing software systems we are faced with the challenge of how to implement a particular analysis for different programming languages. A solution for this problem is to write a single analysis using a code query language abstracting from the specificities of languages being analyzed. Over the past ten years many code query technologies have been developed, based on different formalisms. Each technology comes with its own query language and set of features.

To determine the state of the art of code querying we compare the languages and tools for seven code query technologies: Grok, Rscript, JRelCal, SemmleCode, JGraLab, CrocoPat and JTransformer. The specification of a package stability metric is used as a running example to compare the languages. The comparison involves twelve criteria, some of which are concerned with properties of the query language (paradigm, types, parametrization, polymorphism, modularity, and libraries), and some of which are concerned with the tool itself (output formats, interactive interface, API support, interchange formats, extraction support, and licensing). We contextualize the criteria in two usage scenarios: interactive and tool integration. We conclude that there is no particularly weak or dominant tool. As important improvement points, we identify the lack of library mechanisms, interchange formats, and possibilities for integration with source code extraction components.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—code query languages

General Terms code query, software analysis, comparative study, tools

Keywords Code query, Grok, Rscript, JRelCal, SemmleCode, JGraLab, CrocoPat, JTransformer.

1. Introduction

Code query technologies play an important role in software analysis. Applications of these technologies can be found in software architecture analysis [13], reverse engineering [13, 16], applying consistency checks [36], enforcing coding conventions [36], and finding crosscutting concerns [26].

The extensive use of code query technologies is due to the possibility to analyze different software artifacts. This is achieved based on the use of the extract-abstract-present paradigm [13, 34]. The extract-abstract-present paradigm defines three steps:

- *Extract*: take the source code and map it to some intermediate structure such as a graph;
- *Abstract*: apply operations and queries to this abstract intermediate structure to obtain results;
- *Present*: graphically display the results.

Language-dependent extractors are responsible for mapping program sources to an intermediate, usually relational structure. Queries are specified in a domain specific language offering specific constructs (e.g. representation of files and source code locations) and operators (e.g. recursion) that match the problem domain of the source code analysis. Queries are then executed on the intermediate relation structure that abstract the specific details of the programming languages, allowing immediate reuse of queries across programming languages.

To effectively query source code repositories it is essential that the query language support some form of recursion to help deal with the recursive structure of modern programming languages. This is why using the regular expressions of, for example, `grep` quickly becomes unmanageable. Partly for the same reason, and partly due to performance problems, the first attempt at using code query technologies in 1984 was unsuccessful [25].

Many code query technologies have surfaced through the years differing in many essential ways. Some only provide means for querying code, but leave extraction and presentation to 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. Therefore a comparison of these technologies is in order. Indeed, the direct motivation for our work was to enhance code querying productivity at the Software Improvement Group (SIG), by replacing an existing imperative implementation for code querying with a more declarative alternative. Within SIG, there is much need for effectively computing large-grained information from code-bases, and this is something that code query technologies tend to be most useful for. We believe the outcome of our comparison to be useful to others as well.

In this paper, we describe the results of an extensive comparison of seven existing code querying technologies: Grok [16], Rscript [20], JRelCal [31], SemmleCode [36], JGraLab [12], CrocoPat [4] and JTransformer [21]. The criteria for tool selection was that the tools be available and actively maintained.

The comparison is based on twelve criteria. These criteria are derived from two typical usage scenarios: interactive use and tool integration. The criteria are divided into two categories: those re-

lated to the query language features, and those related to the tool features. The comparison of the code query languages is based on the computation of the package instability metric [28]. This query is justified since it requires both architecture analysis and metrics computation, two of the main tasks in software analysis.

We stress that this paper is a comparison and not a competition between tools. The ultimate goal is to provide enough information for anyone to make an informed decision on which code query technology is best for a given context. As such, we do not make assumptions about the relative *importance* of the criteria we used, nor do we provide a qualitative assessment.

This paper provides a significant contribution over an earlier paper [1]. The previous version only presented a general overview of three code query technologies while here we present an in-depth comparison of seven. In the previous version a trivial example was used and there was no discussion about the comparison. In contrast, the current version presents a more elaborated example capable to show differences between the tools and a comprehensive discussion of the comparison results.

This paper is organized as follows. Background information about formalisms underlying code query languages is presented in Section 2. The tools compared in this paper as well as the criteria for selection are presented in Section 3. Section 4 introduces the usage scenarios and defines the criteria to compare the query languages and the tools. We compare the implementation of an example query in Section 5. The language comparison is done in Section 6 and the tool comparison in Section 7. We summarize our findings in Section 8. An overview of related work is provided in Section 9 and the paper is concluded in Section 10.

2. Background

In this section we discuss the formalisms that underlie the tools we have considered. We also list some tools that we have not considered in our comparison, but that can be considered to belong to the field.

The tools we consider are based on logics of varying expressivity. The advantage of logical languages is that they tend to be more *declarative* than their procedural and functional counterparts: one specifies the properties of the answer, leaving it implicit how the answer should be computed.

With regards to expressivity, Prolog is a language that is Turing complete: anything expressible in any programming language we know can be expressed in it. However, expressivity comes at a price: it is much harder to guarantee bounds on the execution time of queries, which might become a real problem if the size of the input is very large. Moreover, many other properties of the queries, e.g., termination, are (theoretically) undecidable. This does not mean, however that Prolog implementations are always slower than those based on less expressive logics. It does mean that fewer guarantees can be given.

Database query languages are often based on Tarski's relational algebra (RA) or Codd's relational calculus (RC) both based on first order logic, a much less expressive, but also more tractable form of logic. The application and formulation of these languages for database querying is mainly due to Codd [7, 8], although Kuhns [23] considered RC somewhat earlier. However, the basis of these languages goes back much further (see Tarski [33]). It is well-known that RA and RC are computationally equivalent; the main difference between the two is that RC is somewhat more declarative: in RA an expression also specifies how the result should be computed. Many database languages are based on a combination of both logics, although as a rule they tend towards RC; SQL is the most well known example. SQL also adds, e.g., grouping and aggregate operators that neither RC and RA contain.

DATALOG [6] lies inbetween SQL and Prolog, inhabiting what many consider to be a sweet spot for querying: every DATALOG query can be evaluated in polynomial time and, contrary to SQL, it does provide a least fixed-point operator vital for querying recursive structures (such as program sources).

In the case of JGraLab, the basis of the language is a particular language of graphs called *grUML*, a subset of the UML class diagrams. These structures are queried with a language that is much like relational algebra, but tailored to graphs, e.g., by providing so called regular path expressions [5].

We are not the first to make a comparison of code querying tools. Holt, Winter and Wu did a similar comparison in 2002 [18], although with a different goal. They wanted to determine the requirements (or "wishes") for a common code querying language, by comparing two widely different tools: Grok and GReQL. They also mention some other tools: Progres is a language based on graph transformations [32] and does not seem to be particularly suited for code querying. ESCAPE [30] is a tool based on many-sorted algebras extended with subtyping polymorphism, of which the authors contend that it is more powerful than relational algebra. We did not find any evidence that it has ever passed beyond the prototype stage. Finally, RPA was based on relational algebra extended with a "lifting" operator [13]. As far as we have been able to tell it does not exist anymore.

3. Code Query Technologies

Table 1 provides an overview of each of the code query technologies used in this comparison. For each one we report the code query language formalism, original author, release version, implementation language, and implementation date.

These tools were selected based on the following criteria:

- only tools built specifically for code querying were selected. Although other (more generic) querying technologies, such as SQL, can be used for querying source code our focus is to investigate features that explicitly benefit source code querying.
- only the most active tools based on Tarski's relational calculus, Codd's relational algebra and predicate logic were selected. Although quite a few other tools could be included, we decided to limit our study to active tools (both in terms of publications and tool support). Our comparison can then serve as a basis for a further comparison to lesser known and active technologies.

For each tool, we provide a brief historical overview below. All except SemmleCode were developed in university environments for both research and teaching. SemmleCode, on the other hand, is the only tool that was developed in an industrial environment, and is commercially available.

Grok Based on relational calculus, Grok was implemented in 1995 by Ric Holt at the University of Toronto, Canada. Grok was implemented in the Turing Programming Language [17] also developed by Holt. The first use of Grok was to manipulate graph models to aid the reverse engineering of the software architecture implicit in the source code. Grok was first referred to in a 1996 technical report [15] and the first publication dates from 1998 [16].

Sharing the same concepts and language of Grok, JGrok [37] started in 2000 in the context of Jingwei Wu's PhD work, under supervision of Holt. JGrok, implemented in Java, was developed and maintained independently from Grok, implementing a richer set of operations. Since only Grok is being maintained, JGrok will not be included in our comparison. For the comparison we used Grok version 89, as available in the SWAG Kit version 3.03.

Rscript Also based on relational calculus, Rscript was developed in 2002 by Paul Klint at the Centrum voor Wiskunde & Infor-

Table 1. Overview of the analyzed tools (sorted by formalism and date).

Tool	Formalism	Author	Version	Implementation	Date
Grok	Tarski	Ric Holt	89 (SWAG Kit 3.03)	Turing	1995
Rscript	Tarski	Paul Klint	1.1, rev 26884	ASF+SDF	2002
JRelCal	Tarski	Tijs van der Storm	0.7	Java	2007
SemmleCode	Codd	Oege de Moor	pre-rel. 1.0, 2009-01-12	Java	2006
JGraLab	Codd	Steffen Kahle	1095/64	Java	2006
CrocoPat	Predicate Logic	Dirk Beyer	2.1.4	C	2002
JTransformer	Predicate Logic	Günter Kniesel	2.6.1	Prolog	2002

matica (National Research Institute for Mathematics and Computer Science), in Amsterdam, The Netherlands. Implemented in ASF+SDF [35], Rscript was developed especially for program analysis and querying source code. The first publication mentioning Rscript dates from 2003 [20]. For the comparison we used Rscript version 1.1, rev. 26884.

JRelCal Based on Rscript, the JRelCal project started as an efficient Java implementation of Rscript datatypes and its operations. JRelCal started in 2007 in the context of the PhD thesis of Tijs van der Storm, under supervision of Klint. The purpose of JRelCal is not a Java re-implementation of the Rscript tool, but an API with the same functionality. Hence, JRelCal can be used as an embedded DSL and can be easily integrated with Java projects. The authors have no publications about JRelCal, but Rademaker discusses it in his Master's thesis [31]. Although JRelCal is similar to Rscript, we decided to included because it contrasts with all other code query technologies for being just an API. For the comparison we used JRelCal version 0.7.

SemmleCode Based on relational algebra, but with an objected oriented *style* of programming, SemmleCode started in 2006. Although it has an academic background, at the University of Oxford, UK, it is commercialized by a spin-off company, Semmle Ltd., led by the mentor of the project, Oege de Moor. SemmleCode was developed with the purpose of offering a competitive tool (as an Eclipse plug-in) for code analysis. The ideas were initially presented in January 2007 [9] and the first publications date from the same year [10, 36]. For the comparison we used SemmleCode a pre-release version 1.0 dated from 2009-01-12.

JGraLab Based on relational algebra and graph theory, JGraLab started in 2006 in the context of the diploma thesis of Steffen Kahle. Since then, JGraLab has been developed and maintained by several people, among them Volker Riediger and Daniel Bildhauer, the current maintainers of the tool. JGraLab replaced GraLab initially developed in 1985 by Jürgen Ebert. Both GraLab and JGraLab were developed at the University of Koblenz and Landau, Germany, in the research group of Jürgen Ebert. The first publication of GraLab dates from 1987 [11] and JGraLab was first referred to in 2007 [12]. For the comparison we used JGraLab release 1094/64.

CrocoPat Based on first-order predicate logic, CrocoPat was started in May 2002 by Dirk Beyer as a first project as PostDoc at the University of Cottbus, Germany. CrocoPat, implemented in C, was meant to replace the SQL front-end of the Crocodile reengineering tool developed at the same university. The ideas of CrocoPat were first described in a technical report from 2003 [3]. CrocoPat was described in more detail in 2005 [4] and in 2006 [2]. For the comparison we used CrocoPat version 2.1.4.

JTransformer Based on first-order predicate logic, JTransformer was initially implemented by Tobias Rho and Uwe Bardey, under supervision of Günter Kniesel. JTransformer development started in 2002 as a program transformation tool to prove the effectiveness of logic-based conditional transformations. In 2003, the first Eclipse plug-in was developed and since then JTransformer has been used for program analysis in both research and teaching. Although much research was done using JTransformer, the first publication describing the tool is from as late as 2006 [21]. JTransformer was described in more detail in 2007 [22]. For the comparison we used JTransformer version 2.6.1.

It is not surprising that the tools reveal different strong points as advertised by their authors and maintainers. For instance, Grok advocates its language conciseness, Rscript its static type checking and the expressiveness of set comprehensions, JRelCal its seamless integration with Java, and SemmleCode the accessibility of the language to O.O. programmers due to their borrowing syntax from well-known O.O. languages and integration with the Eclipse environment. Furthermore, JGraLab advocates its ease of manipulating graph based structures, CrocoPat its efficiency in computing transitive closures, and JTransformer the use of Prolog as basis for its code query language, combined with the Eclipse environment integration.

Notwithstanding these differences, we do not expect the tools to differ very much in their suitability for code querying, certainly not to the extent that one tool is inferior to one other tool in all respects and thereby can be considered to be superfluous.

4. Criteria

In this section, we define the criteria for comparing language and tool features of the analyzed code query technologies. Table 2 summarizes the criteria used and the scenarios for which these criteria are of importance.

The following notation is used to describe the importance of the criteria: "$\triangle$" is used when the criterion is important for a particular scenario; "o" is used when the criterion is relevant but not important; and "$\triangledown$" is used when the criterion is not important.

Although we expect every potential user of code query technologies to have his own set of requirements, we indicate the relative importance of the criteria for two typical usage scenarios: interactive use and tool integration. In the interactive use scenario, the code query technology is used directly by a software analyst. The analyst is responsible for the specification and execution of the queries, and the extraction of results for further processing. This scenario imposes strong requirements on the language, but also on support provided by the tool. However, the presence of an API is not an issue, and since the tool is used directly, the particular license plays only a minor role. In the tool integration scenario the code query technology is used as a component by a developer with the purpose of being used indirectly from a tool. The developer is required to interface with the code query language, and therefore API support, Interchange format and Output format criteria are crucial. Style/Paradigm is important when interfacing is done through the language instead of the API. The remaining language criteria are

relevant but are not as important as in the interactive use scenario, since shortcomings in these areas can typically be compensated for by the constructs in the language of the host tool. Clearly, an interactive user interface is not relevant in this scenario. Additionally, extraction support is relevant but not important since it is likely that extractors are already available in the host tool.

Language criteria To compare language features we consider the following criteria:

- *Paradigm* specifies the paradigm the code query language is based upon which typically has implications for the conciseness of the code queries.
- *Types* make explicit which data types are supported by the query languages.
- *Parametrization* indicates that the behavior of queries may depend on parameter values;
- *Polymorphism* specifies whether queries that abstract away from the types from which the relations are built are supported. We consider both parametric and subtyping polymorphism.
- *Modularity* determine the extent it is possible to reuse queries for constructing other queries; and
- *Libraries* determine the possibility to use and/or define libraries of queries.

Tool criteria To compare tool features we consider the following criteria:

- *Output formats* specifies the output formats provided by the tools, e.g., charts, plain text, and XML.
- *Interactive interface* lists the types of interfaces available to access the code query technology functionalities, e.g., command-line interface (CLI), graphical user interface (GUI) or Eclipse plug-in.
- *API support* indicates the availability of an API, allowing the functionality to be used from a host program.
- *Interchange format* specifies the file formats supported to read and write extracted facts and query results. Users of code querying technologies benefit from being able to interchange extracted facts and query results. For example, if a user needs to analyze a language L, but tool Y does not support it, then if tool Z can extract facts for L and both Y and Z support a particular format, then the problem is solved. Or, consider that Y is optimized for a particular class of queries, and Z for another. In an application for which both kinds of queries are necessary, it is desirable to be able to easily switch tools.** **RSF** **
- *Extraction support* indicates the existence of fact extractors for known programming languages, e.g., C or Java.
- *Licensing* specifies the license under which the code query technology was released. This can be essential, because the license may restrict the user in how the tool may be employed. A technology is proprietary if the company has reserved some measure of control over the software. Otherwise, the technology is free (open-source). The drawback of proprietary software is that the source code is typically not made available to its users and changing the software is not allowed or simply impossible. Within the open-source community, various licenses exist. BSD 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. Software under the less permissive GNU GPL (GNU General Public License) may not be used in proprietary software at all.

Table 2. Language and tool criteria vs. usage scenarios. We use △ for important, o for relevant, and ▽ for not important.

			Scenario	
			Interactive use	Tool integration
Criteria	Language	Paradigm	△	△
		Types	△	o
		Parametrization	△	o
		Polymorphism	△	o
		Modularity	△	o
		Libraries	△	o
	Tool	Output formats	△	△
		Interactive interface	△	▽
		API support	▽	△
		Interchange format	△	△
		Extraction support	△	o
		Licensing	o	△

These criteria equally apply to assess the tools in both industrial and academic environments.

5. Examples Comparison

To compare the languages of the code query technologies, we implemented a query to compute the package instability metric [28] in each one. Despite that the original motivation of this metric was to be applied to OO languages, when considering the notion of package in a broader term, e.g. component, this metric can be applied to any programming language. The usage of this metric is motivated by the fact that it combines two of the main tasks in software analysis: architectural analysis and metrics computation. By coding this analysis in different code query technologies we expect to stress three main aspects: the syntactical differences between the code query languages, how the respective models can be enriched with new facts, and how results can be extracted.

The package instability metric is derived from two other metrics: afferent coupling and efferent coupling of a package.

Afferent coupling Ca = number of classes outside the package that depend upon classes within the package.

Efferent coupling Ce = number of classes inside the package that depend upon classes outside the package.

Instability $I = Ce/(Ca + Ce)$. Instability $I = 0$ indicates a maximally stable package, and $I = 1$ indicates a maximally unstable package.

In this section we show excerpts of the package instability metric implemented in each of the tools. We first formulate what we aim to compute, and define the input relations for the queries. Then, to make it a fair and easy comparison, we describe some implementation guidelines that should be followed. Finally, we show excerpts for all the implementations and comment on key issues.

5.1 Implementation Guidelines

To be fair when comparing examples it is necessary to define implementation guidelines. Hence, we will define the input relations and the implementation steps of the example.

The defined input relations are:

$PackageOf : Package \times Class$: records for each package the classes that belong to that package or, more precisely, if $(P, C) \in PackageOf$ then class C is defined in package P;

$ClassOf$: $Class \times Method$: records for each class the methods that belong to that class or, more precisely, if $(C, M) \in ClassOf$ then method M is defined in class C; and

$MethodCall$: $Method \times Method$: records methods invocations or, more precisely, if $(M, N) \in MethodCall$ then the method M calls the method N.

For SemmleCode and JTransformer similar relations were used, albeit with different names, because the naming of these relations are under the control of the extractors.

For the examples, the following implementation steps were defined:

$ClassDep$: $Class \times Class$ where $(A, B) \in ClassDep$ records that a class A depends on a class B: A has a method that calls a method from class B.

$ClassDepInterPackg$: $Class \times Class$ records that $(A, B) \in ClassDep$ and A and B belong to different packages.

$AffCoupling$: $Package \times Class$ records for each package P all classes outside P that depend on classes within P.

$EffCoupling$: $Package \times Class$ records for each package P all classes inside P that depend on classes outside P.

$AfferentCoupling$: $Package \times \mathbf{N}$ records for each package the number of classes outside the package that depend upon classes inside the package.

$EfferentCoupling$: $Package \times \mathbf{N}$ records for each package the number of classes inside the package that depend upon classes outside the package.

$PackageInstability$: $Package \times \mathbf{R}$ records for each package the instability value I computed as indicated above.

The $ClassDep$ and $ClassDepsInterPackages$ are intermediate relationships that allow us to compute the afferent and efferent coupling metrics.

5.2 Language examples

For each code query technology, excerpts of the implementation of the package instability metric will be presented. Due to to space constraints, only the implementation of the $ClassDepInterPackg$, $AffCoupling$, $AfferentCoupling$ and $PackageInstability$ relations will be shown. Instead of describing each example in detail, we focus on the language constructs we employed.

Grok

```
PackageDep := PackageOf o ClassDep o (inv PackageOf)

PackgDepInterPackg := PackageDep - (id dom PackageOf)

ClassFowardRel
   := (inv PackageOf) o PackgDepInterPackg o PackageOf

ClassDepInterPackg := ClassForwardRel ^ ClassDep

AffCoupling := PackageOf o (inv ClassDepInterPackg)

AfferentCoupling
   := (dom AffCoupling) outdegree AffCoupling
```

Listing 1. Grok fragment of expressible package instability metric.

Grok provides a very concise language requiring few operators or symbols. However, this requires construction of queries by small pieces leading to some fragmentation at the expense of being hard to understand, e.g. three auxiliary relations were required to implement the $ClassDepInterPackg$ relation. Since the language is untyped and there is no checking, debugging a wrongly composed relations is hard. Finally the $PackageInstability$ relation is missing since it is not possible to implement the computation of the instability. Grok does not offer combinators to combine two relations by their domain applying a function to the range of the relations, operation necessary to finalize the example. Grok is not Turing complete.

Rscript

```
rel[str,str] PackageOf
rel[str,str] ClassOf
rel[str,str] MethodCall

rel[&T1, int] outdegree(rel[&T1,&T2] R)
   = { <D, #R[D]> | <&T1 D, &T2 U> : R }

rel[str,str] ClassDepInterPackg
   = { <C1,C2> | <str C1, str C2> : ClassDep
               , PackageOf[-,C1] != PackageOf[-,C2] }

rel[str,str] AffCoupling
   = PackageOf o inv(ClassDepInterPackg)

rel[str,int] AfferentCoupling = outdegree(AffCoupling)

rel [str,int] PackageInstability
    = { <P1, (100*N1)/(N1+N2)>
        | <str P1, int N1> : EfferentCoupling
        , <str P2, int N2> : AfferentCoupling
        , P1 == P2 }
```

Listing 2. Rscript package instability metric.

Rscript requires the declaration of the input relations in the beginning of the program (first three lines) enabling type-checking of the queries. In addition to relations, user-defined functions are allowed (see $outdegree$ definition, which computes the cardinality of the range of a relation). This function makes use of parametric polymorphism. Besides relational calculus, set comprehensions are also supported (see $outdegree$ and $ClassDepInterPackg$ definitions). Finally, since Rscript does not support reals, it is necessary to multiply the numerator by 100.

JRelCal

```
Relation<String, String> packageDep
   = packageOf.compose(classDep.compose(
                          packageOf.inverse()));

Relation<String, String> packgDepInterPackg
    = packageDep.difference(packageOf.domain().id());

Relation<String,String> classForwardRel
    = (packageOf.inverse())
      .compose(packgDepInterPackg)
      .compose(packageOf);

Relation<String,String> classDepInterPackg
    = classForwardRel.intersection(classDep);

Relation<String,String> affCoupling
    = packageOf.compose(classDepInterPackg.inverse());

Relation<String,Int> afferentCoupling
    = affCoupling.outdegree();
```

Listing 3. JRelCal package instability metric.

JRelCal queries are a mix of Java code and relational calculus. Except minor syntactic sugar, due to the use of a fluent-interface API [14], JRelCal queries are identical to those for Rscript. JRelCal supports Java generics which enables the use of any Java type

and allows for static checking support. The implementation of the *packageInstability* relation was omitted, because its code is straightforward Java.

SemmleCode

```
predicate classDepInterPackg( Class c1, Class c2) {
    c1.getPackage() != c2.getPackage() and
    classDep(c1,c2)
}
class MyPackage extends Package {
    MyPackage() { this.fromSource() }

    predicate affCoupling(Class c) {
        exists(Class c1 | this.contains(c1) and
                          classDepInterPackg(c1, c))
    }
    int afferentCoupling() {
        result = count(Class c | this.affCoupling(c))
    }
    float packageInstability() {
        result = (1.0 * this.efferentCoupling()) /
            (this.afferentCoupling() +
             this.afferentCoupling())
    }
}
```

Listing 4. SemmleCode package instability metric.

SemmleCode offers a language based relatation algebra with a syntax with a distinctive object-oriented flavour. The excerpt above clearly shows the OO influence. Relations can be implemented as a *classless* predicates, e.g. *classDepInterPackg* definition, and/or by using extension mechanisms. An example of model extension is shown in the *MyPackage* definition which extends the class *Package* predefined in the library (subtyping polymorphism). Since there is no control over the extraction, it was required to specify functionality not related with our problem (see the use of the *fromSource*() predicate in the class constructor to filter out library classes). SemmleCode is statically typed.

JGraLab

```
from p : V {JavaPackage}
reportMap p,
   from outerClass : V {JavaClass}
   with
     (not p --> {PackageOf} outerClass) and
     (p --> {PackageOf} <-- {ClassDep} outerClass)
   report outerClass end
end store as AffCoupling

using AffCoupling:
from p : V {JavaPackage}
reportMap p, count(get(AffCoupling,p)) end
store as AfferentCoupling

using AfferentCoupling, EfferentCoupling:
from p : V {JavaPackage}
reportMap p, get(EfferentCoupling,p) /
         ( get(EfferentCoupling,p) +
           get(AfferentCoupling,p))
end store as PackageInstability
```

Listing 5. JGraLab package instability metric.

JGraLab is based on relational algebra and path expressions. JGraLab uses an inverted SQL notation, which can be identified in the beginning of the query. Path expressions, which can be observed by the use of arrows, specify how to identify objects by specifying how to navigate a graph structure. Modularity is achieved by saving the query results into variables, e.g., in the first query we specify that the result is a map (*reportMap*) and that the result should be stored in the *AffCoupling* variable.

CrocoPat

```
ClassDepInterPackg(c1,c2)
    := EX( p1, PackageOf(p1, c1)
         & EX( p2, PackageOf(p2, c2) & !=(p1,p2)
             & ClassDep( c1, c2)));

AffCoupling(p,c) := EX( c1, PackageOf( p, c1) &
                        ClassDepInterPackg( c, c1));

Package(x) := PackageOf(x,_);
FOR p IN Package(x) {
    ca := #(AffCoupling(p,c));
    PRINT "AfferentCoupling ", p, " ", ca, ENDL;

    ce := #(EffCoupling(p,c));
    PRINT "EfferentCoupling ", p, " ", ce, ENDL;

    i := ce / (ca + ce);
    PRINT "Instability ", p, " ", i, ENDL;
}
```

Listing 6. CrocoPat package instability metric.

CrocoPat queries are specified in a mix of predicate logic and imperative code. Except for the use of quantifiers and minor syntactic sugar the implementation of the *AffCoupling* and *ClassDepInterPackg* relations are identical to Rscript. However, the implementation of afferent and efferent coupling and package instability metrics is completely different. CrocoPat does not allow these metrics to be specified as relations. As a workaround, we have to write a loop printing these relations and import them later for further processing.

JTransformer

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

affCoupling(P, C) :-
   packageOf(P, C1), classDepInterPackg(C, C1).

afferentCoupling(P,N) :-
   setof(C, affCoupling(P, C), AffClasses),
   length(AffClasses, N).

packageInstability(P, I) :-
   efferentCoupling(P, Ec), afferentCoupling(P, Ac),
   I is Ec/(Ec + Ac).
```

Listing 7. JTransformer package instability metric.

JTransformer is based on SWI-Prolog. Hence, relations are defined as logic predicates. Comparing JTransformer implementation with others, e.g. Rscript, we observe that it is not much different. The *classDepInterPackg* and *affCoupling* implementations are nearly identical, and the implementation of *afferentCoupling* requires only a minor additional effort.

6. Language comparison

Table 3 presents a summary of the comparison of the language features for each code query technology. Tarski relational calculus is supported by three tools (Grok, Rscript and JRelCal), while the others support Codd relational algebra (SemmleCode and JGraLab) or logic (CrocoPat and JTransformer). However, some of the tools offer constructors from other paradigms. For example, Rscript supports set comprehensions which can elegantly express a wide variety of properties. SemmleCode is the only tool that borrows inspiration from the Object-Oriented programming community, in an effort to be more easily adopted by this large community of

Table 3. Summary of the language comparison results.

Criteria vs. tools		Grok	Rscript	JRelCal	SemmleCode	CrocoPat	JGraLab	JTransformer
Paradigm		Relational	Relational and Comprehensions	API Relational	OO & SQL-like	FO-logic Imperative	SQL-like & Path Expr.	FO-logic
Types	String	x	x	x	x	x	x	x
	Int	x	x	x	x	x	x	x
	Real	x	-	x	x	x	x	x
	Bool	-	x	x	x	x	x	x
	Other	-	Composite and Location	Java	Object	-	Edge and Node	Logic terms
Parametrization		-	x	x	-	x	x	x
Polymorphism		-	x	x	x	-	x	x
Modularity		x	x	x	x	x	-	x
Libraries		-	-	x	x	-	-	x

programmers. CrocoPat, on the other hand, supports constructors taken from imperative programming and, finally, JGraLab features path-expressions. The example is fully expressable in all languages except Grok which lacks the necessary operators. Most common types are supported by all tools. The only exception is that Rscript does not provide any reals. JRelCal types are inherited directly from Java. Parametrization is not present only in Grok and in SemmleCode. Polymorphism is not present at all in Grok and CrocoPat. However different types of polymorphism are supported: SemmleCode supports subtype polymorphism, while the others support parametric polymorphism. Modularity is only lacking in JGraLab. Finally, libraries are supported by JRelCal (through the use of Java), SemmleCode and JTransformer (through the use of Prolog). All others lack the support of libraries.

7. Tool comparison

Table 4 presents a summary of the tool features of each code query technology.

Text is the dominant output format, supported by five out of seven tools. However, in both Grok and JGraLab the output text cannot be customized. Rscript only outputs Rstore, a textual format that is also used for interchange. With JRelCal, since it is a library, output can only be done through Java. Finally, SemmleCode supports a rich set of output formats (charts, maps, graphs).

Interactive use is mostly provided via the command-line. Exceptions are Rscript which provides a GUI in addition to the command line interface, and SemmleCode and JTransformer which offer an Eclipse plug-in. JRelCal is meant to be used as a library, so no interactive interface is available.

API support is absent only in Grok and Rscript, although interfacing is possible through interchange support.

Interchange format support is diverse. RSF is supported by only three tools: Grok, JRelCal and CrocoPat. Rscript and JGraLab use their own formalisms, and JTransformer uses Prolog fact bases. SemmleCode does not support interchange format and relies on its integrated fact extractor only.

Java extraction support is available for three tools: SemmleCode, JGraLab and JTransformer. SemmleCode supports additional XML extraction support. In addition to Java, JGraLab also supports C extraction, although neither extractor is publicly available. Grok uses the C++ extractor that is provided by the SWAG kit. Rscript, CrocoPat and JRelCal lack extraction support. The lack of extraction support can be overcome by using either Grok, JTransformer or SemmleCode to extract and then convert to a suitable interchange format. The file formats RSF and Rstore can be automatically generated by the tools that support them. Also automatic conversion between RSF, Rstore and Prolog is possible with a small amount of work. In all three formats, nodes and edges are defined as untyped constructs, so conversion can be achieved by simple syntactical transformations. In the case of JGraLab's TGraph, however, both nodes and edges are typed and declared in the header of the format. Converting RSF files to TGraph is a semi-automatic process. The header file describing the used types must be manually defined. After this, the nodes and edges can be automatically converted.

Finally, most of the tools are available under an open-source license: Rscript is available under a BSD license, JRelCal and CrocoPat under LGPL, and JTransformer under EPL. Exceptions are Grok, SemmleCode and JGraLab. For Grok no license could be found. SemmleCode is only available under a commercial license. JGraLab is offered in a dual license: for non-commercial projects GNU GPL 2 is available otherwise a commercial license is applicable.

8. Summary

In summary, Grok offers a very concise and simple language which is its strongest point. However, note that in Grok it was not possible to fully specify the package instability metric. Parametrization, polymorphism and libraries would be desirable to enable query reuse. Tool support is limited to the essential only, lacking API support and other output options.

Rscript, like Grok, has as its strongest point the language. Typechecking is a valuable addition at the expense of more verbose queries. The lack of support for reals is relevant when computing metrics. Support for libraries is missing. Tool support is rudimentary, missing support for other interchange formats and support for text output functionality.

JRelCal was specifically designed to be used as a library to be embedded in Java programs. As such, it is only recommended for this particular scenario that it was built for.

SemmleCode offers a simple Java-styled language based on SQL. The language is simple to learn since these two paradigms are commonly known by programmers. However SemmleCode's strongest point is the seamless integration with Eclipse and the supported output formats. The lack of parametrization is relevant but not essential. More important is the lack of interchange formats restricting the use of other extractors.

CrocoPat strongest point is the conciseness of the language although with some limitations in expressivity. Note that in order to compute package instability, it is necessary to print out the results and read them back. Also, it lacks support for polymorphism and libraries.

JGraLab strongest point is the use of path expressions. On the downside, the language is difficult to learn and understand mainly

Table 4. Summary of the tool comparison results.

Criteria vs. tools	Grok	Rscript	JRelCal	SemmleCode	CrocoPat	JGraLab	JTransformer
Output formats	Text	Rstore	Sets and relations	Text, charts, maps, graphs	Text, RSF	Text, HTML	Text
Interactive interface	CLI	CLI, GUI	-	CLI, Eclipse	CLI	CLI	Eclipse
API support	-	-	x	x	x	x	x
Interchange format	RSF, TA	Rstore	RSF	-	RSF	TGraph	Prolog
Extraction support	C++	-	-	Java, XML	-	Java, C	Java
Licensing	-	BSD	LGPL	Proprietary	LGPL	GPL 2 Proprietary	EPL

due to the lack of documentation. Support for modularity and libraries is lacking. Tool support is rudimentary.

Finally, JTransformer's strongest point is the conciseness. However due to the use of tuples it is sometimes difficult to memorize the order and what each component of the tuple represents. Eclipse integration is a bonus, but the ease of interfacing could be improved. Also, more output formats would be desirable.

In conclusion, the variation of the tools is not as large as may seem at first glance. Most code query technologies can be used in both the interactive and tool integration scenarios. In the interactive scenario, only JRelCal is less suitable, since only an API is available. In the tool integration scenario, only Grok and Rscript are less suitable since they do not support an API (however this is compensated with interchange format(s) support). Also, when considering the various implementations of the example, ignoring syntactic sugar, the differences are not significant (although, admittedly, we could not fully implement the query in Grok). Indeed, some tools provide a handsome interface, other provide a useful API, but when looking at the comparison results there is not a particularly weak or dominant tool among them. However, there is space for improvement and we hope the authors use our comparison as basis for improvement by adopting other tools strongest points.

9. Related work

Alves and Rademaker presented a similar comparison between three code query technlogies: CrocoPat, SemmleCode and Rscript [1]. Features were compared using language and tool criteria, and languages were compared through an example. However, this work presents just a brief overview of the technologies. The current paper is an extension of that overview, presenting more and more clearly defined criteria, highlighting differences between tools. Also the example used in this paper is far less trivial, showing that for some code query technologies its implementation is not possible or obvious.

Holt et al. [18] present a comparison between Grok and JGraLab with the goal of deriving requirements for a new code query language. Their comparison is based only on code examples in a tool manual style. In contrast, we provide a comparison between seven tools, present a well-defined criteria and example challenge. Finally, we provide a discussion of each tool covering their strongest and weakest points.

Lange et al. [24] present a comparison between GUPRO, which implements the JGraLab language, and relational databases for reverse engineering of a real world software system. This comparison highlights the advantages of code query languages over traditional relational databases. Although examples are used, this work compares two tools that are based on two (different) paradigms. Our work, however, is meant to show differences between different alternatives of the same technologies. Moreover, we do so by comparing language and tool features, and provide example implementations.

Beyer et al. [4], in their paper about CrocoPat, present a performance comparison between CrocoPat, Prolog and the relational database MySQL. However this comparison deals with the issue of performance. In contrast, we focus on focus on language or tool features and the expressiveness and conciseness of the languages.

10. Conclusions and Future work

We have compared seven code querying tools on a total of twelve criteria. To compare language features – paradigm, types, parametrization, polymorphism, modularity, and libraries– were used, while properties of the tools were address through the comparison of others – output formats, interactive interface, API support, interchange formats, extraction support, and licensing. The criteria were motivated by two usage scenarios: one in which a user interacts directly with the tool, and one in which the tool is used indirectly from other tools. The comparison of the languages was performed by implementing a query in each of the languages under consideration. In Section 8, we have already summarized our findings.

The main goal of our paper is to allow potential users of code querying technologies to make an informed decision which tool to select to address their specific needs by having an initial understanding about each tool. This paper can be used as a guideline for comparing code query technologies. The selected criteria form a substantial basis that can, of course, be further extended.

Moreover, the comparison can be used by the tool developers to decide on the directions in which to further extend their tools. In particular, we hope that more attention will be paid to the support of libraries, interchange formats and integration with code extractors.

A criterion that was left out of the comparison is performance. Measuring performance would provide additional information about if the tools can be used in practice. While experimenting with the queries we observed that some tools were faster than other for specific analysis, yet we did not observe major issues with any of them.

However, our focus was in showing how things can be done rather then how fast they can be done. In the absence of an accepted benchmark for code query technologies, a fair comparison of performance should include the development of such a benchmark. Also, these queries should be executed under identical conditions, which is hard considering that some tools are executed via the command line while others are executed from a GUI. For those reasons we believe that a fair performance comparison deserves a study by itself and is beyond the scope of this paper.

Future Work Our work can be extended in various directions. There are quite a few more tools that operate within the chosen formalisms. Examples include JQuery [19], SOUL [29], and PQL [27]. We can also extend along a different dimension by contrasting the technologies with technologies based on other formalisms, e.g., generic tools for analysis and transformation of code, or including more generic querying technologies. In this paper there was room for only one example query, but our work can

be strengthened by adding more example queries. We already discussed the possibility of addressing performance as an additional criterion for the comparison, a second additional criterion would be a description, for each tool, of the type of queries for which the tool is most eminently suited. We certainly welcome outside participation to broaden our study.

Challenges The first challenge to be met by code querying tools is to adopt each other's strong points. Also, better support for libraries, interchange formats and extraction is required. As previously stated, tool integration can be achieved through an API or via the combination of CLI and interchange format. However, an API exposing the same functionality is preferable since this allows a cleaner integration. All tools should be both programmable via an API and be useable in an interactive way, preferably as plugin to an IDE.

Currently, code querying technologies are applied to obtain information from a large body of code, typically a single single version of some software system. It would be interesting to provide capabilities and abstractions to be able to use code querying to analyze and compare several revisions of the same code.

A final application for code querying is to automatically support *architecture checking*: automatically generate code queries to verify whether the implementation satisfies the constraints specified set by an architecture specification.

Acknowledgements We are grateful to the various authors/maintainers of all the considered tools for their help in getting the historical details right, answering questions about the tools and providing helpful comments: Ric Holt for Grok, Bas Basten and Paul Klint for Rscript, Tijs van der Storm for JRelCal, Oege de Moor and Mathieu Verbaere for SemmleCode, Daniel Bildhauer for JGraLab, Dirk Beyer for CrocoPat and Günter Kniesel for JTransformer. Additionally, we are grateful to Bart Luijten, Xander Schrijen, Eric Bouwers and Joost Visser of Software Improvement Group for providing valuable comments on an earlier version of the paper. The first author is supported by the *Fundação para a Ciência e a Tecnologia* (FCT), grant SFRH/BD/30215/2006 and the SSaaPP project, FCT contract no. PTDC/EIA-CCO/108613/2008.

References

[1] T. L. Alves and P. Rademaker. Evaluation of code query technologies for industrial use. In *QTAPC'08*, 2008.

[2] D. Beyer. Relational programming with CrocoPat. In *ICSE'06*, pages 807–810, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: http://doi.acm.org/10.1145/1134285.1134420.

[3] D. Beyer and C. 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.

[4] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31 (2):137–149, Feb. 2005. ISSN 0098-5589. doi: 10.1109/TSE.2005.23.

[5] D. Bildhauer and J. Ebert. Querying software abstraction graphs. In *Proceedings of the Working Session on Query Technologies and Applications for Program Comprehension (QTAPC'08)*, 2008.

[6] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 01(1):146–166, 1989. ISSN 1041-4347. doi: http://doi.ieeecomputersociety.org/10.1109/69.43410.

[7] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362384.362685.

[8] E. F. Codd. Relational completeness of data base sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[9] O. de Moor, E. Hajiyev, and M. Verbaere. Object-oriented queries over software systems: (abstract of invited talk). In *PEPM'07*, pages 91–91, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-620-2. doi: http://doi.acm.org/10.1145/1244381.1244396.

[10] O. de Moor, M. Verbaere, and E. Hajiyev. Keynote address: .QL for source code analysis. *SCAM*, pages 3–16, 2007. doi: 10.1109/SCAM.2007.31.

[11] J. Ebert. A Versatile Data Structure For Edge-Oriented Graph Algorithms. *Communications ACM*, 30(6):513–519, 6 1987. URL `http://www.uni-koblenz.de/~ist/documents/Ebert1987AVD.pdf`.

[12] J. Ebert, D. Bildhauer, H. Schwarz, and V. Riediger. Using difference information to reuse software cases. *Softwaretechnik-Trends*, 27(2), May 2007. URL `http://pi.informatik.uni-siegen.de/stt/27_2/SE2007/Ebert_Bildhauer_Schwarz_Riediger_2007_stt.pdf`.

[13] L. Feijs, R. Krikhaar, and R. van Ommering. A relational approach to support software architecture analysis. *Softw. Pract. Exper.*, 28 (4):371–400, 1998. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/(SICI)1097-024X(19980410)28:4⟨371::AID-SPE154⟩3.0.CO;2-1.

[14] M. Fowler. MF Bliki: FluentInterface. `http://martinfowler.com/bliki/FluentInterface.html`.

[15] R. C. Holt. Binary relational algebra applied to software architecture. CSRI Technical Report 345, Computer Systems Research Institute, University of Toronto, March 1996.

[16] R. C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *WCRE'98*, page 210. IEEE Computer Society, 1998. ISBN 0-8186-8967-6.

[17] R. C. Holt and J. R. Cordy. The Turing programming language. *Commun. ACM*, 31(12):1410–1423, 1988. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/53580.53581.

[18] R. C. Holt, A. Winter, and J. Wu. Towards a common query language for reverse engineering. Technical Report 8/2002, Fachbereich Informatik, Universität Koblenz Landau, June 2002.

[19] D. Janzen and K. 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. ISBN 1-58113-660-9. doi: http://doi.acm.org/10.1145/643603.643622.

[20] P. Klint. How understanding and restructuring differ from compiling—a rewriting perspective. In *IWPC'03*, pages 2–12. IEEE, 2003.

[21] G. Kniesel and U. Bardey. An analysis of the correctness and completeness of aspect weaving. In *WCRE'06*, pages 324–333. IEEE, 2006. ISBN 0-7695-2719-1. doi: http://dx.doi.org/10.1109/WCRE.2006.10.

[22] G. Kniesel, J. Hannemann, and T. Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *LATE'07*. ACM, 2007. ISBN 1-59593-655-4. doi: http://doi.acm.org/10.1145/1275672.1275678. URL `http://www.cs.uni-bonn.de/~gk/papers/knieselHannemannRho-late07-preprint.pdf`.

[23] J. L. Kuhns. Answering questions by computer: a logical study. Technical Report RM-5428-PR, The RAND Corporation, December 1967.

[24] C. Lange, H. M. Sneed, and A. Winter. Applying the graph-oriented GUPRO-approach in comparison to a relational database based approach. In *IWPC'01*, 2001.

[25] M. A. Linton. Implementing relational views of programs. In *SESPSDE'84*, pages 132–140, New York, NY, USA, 1984. ACM. ISBN 0-89791-131-8. doi: http://doi.acm.org/10.1145/800020.808258.

[26] M. Marin, L. Moonen, and A. van Deursen. Soquet: Query-based documentation of crosscutting concerns. In *ICSE'07*, pages 758–761. IEEE, 2007. URL `http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2007-005.pdf`.

[27] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. *SIGPLAN Not.*,

40(10):365–383, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1103845.1094840.

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

[29] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Journal on Expert Systems with Applications*, pages 236–243, 2001.

[30] S. Paul and A. Prakash. Querying source code using an algebraic query language. In H. A. Müller and M. Georges, editors, *ICSM*, pages 127 – 136. IEEE Computer Society, 1994.

[31] P. Rademaker. Binary relational querying for structural source code analysis. `http://www.cs.uu.nl/wiki/Hage/MasterStudents`.

[32] A. Schürr, A. J. Winter, and A. Zündorf. *The PROGRES approach: language and environment*, pages 487–550. World Scientific Publishing Co., Inc., 1999. ISBN 981-02-4020-1.

[33] A. Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3):73–89, 1941. ISSN 00224812. URL `http://www.jstor.org/stable/2268577`.

[34] S. R. Tilley, S. Paul, and D. B. Smith. Towards a framework for program understanding. In *IWPC'96*, pages 19 – 28. IEEE Computer Society, 1996.

[35] M. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, H. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *CC'01*, LNCS. Springer, 2001.

[36] M. Verbaere, E. Hajiyev, and O. de Moor. Improve software quality with SemmleCode: an Eclipse plugin for semantic code search. In *OOPSLA'07*, pages 880–881, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-865-7. doi: http://doi.acm.org/10.1145/1297846.1297936.

[37] J. Wu. *Open source software evolution and its dynamics*. PhD thesis, University of Waterloo, Canada, 2006.