Program Queries

A Survey

by

Kwun Kit, Lo

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2010

Abstract

Program query systems allow a user to extract program information by viewing a program as a piece of queryable data. They have long been a research interest and have been shown to be useful in supporting wide-ranging software engineering tasks.

The design space of a program query system is complex. To help organize this survey, we will structure our presentation around two major design dimensions. One dimension is the choice of language paradigm for the query language. The other dimension is the choice of storage technologies of the program data. In other words, these two dimensions are the front-end interface and the back-end of a query system respectively. They affect other major properties of a program query system such as expressive power, usability, performance and scalability.

In this essay, we will survey the applications of program queries, as well as explore the relative strengths and weaknesses of different query language paradigms and storage technologies.

Table of Contents

List of Tables

List of Figures

List of Programs

Acknowledgements

I would like to thank my supervisor Kris De Volder, for his support, patience and guidance that make this work possible.

Chapter 1

Introduction

Source code is a complicated artifact. It is not uncommon that a software project exceeds a million lines of code. Investigation in such a large code base is not an easy task. Research has shown that programmers rely heavily on navigation and search tools during development [77].

Most source code editors provide support for text searching. Some even allow a user to express their text search as a regular expression. Text search is useful in tasks such as locating the implementation of particular method or renaming a particular function. However, it provides poor support for understanding the structural relations between different code elements. For example, a textual search cannot help to find "*Java classes that override the `equals()` method without overriding the `hashCode()` method*"[1].

Program query systems go a step further. They allow a user to extract program information by viewing a program as a piece of structured, queryable data. Usually it comes with a query language that allows the user to extract program information.

This idea has long been a research interest. Early systems can be dated back to the 1970s. For example, Browne developed a program analysis system that allows a user to query FORTRAN code in 1978 [16].

Since then, program queries have been applied to aid wide ranging software engineering tasks including Architectural Visualization [57, 58, 74, 94], Design Pattern Recovery [34, 62, 79], Design Rule Verification [6, 48, 51, 72, 76] and Debugging [41, 68, 71].

The goal of this essay is to review the applications of program queries, as well as the design considerations involved in building a program query system. The design space is complex. To help organize this survey, we will structure our presentation around two major design dimensions.

One dimension is the choice of language paradigm for the query language. The other dimension is the choice of storage technologies of the program data. These two dimensions are significant as they affect other major

[1]If two objects are equal according to the `equals()` method, they must have the same `hashCode()`. See [10] for more details.

properties of a program query system such as expressive power, usability, performance and scalability.

Program query languages are the front-end interfaces of program query systems. The choice of language paradigms affects the expressive power and the usability of a query system. In this essay, we will review four popular program query language paradigms: Logical, Relational, Syntactic and Visual.

Storage technologies are the back-ends of program query systems. The choice of storage technologies determines the performance and the scalability of a query system. In this essay, we have selected some common storage technologies to review: from standard logic engines and relational databases, to less well-known technologies such as Tabled Logic Engines, Binary Decision Diagrams, Object and XML Databases and Reflection technologies.

The two design dimensions are not completely orthogonal. For example, some storage technologies may favor some particular kinds of program query paradigms, and vice versa. We will see more examples in Chapter 4 and 5.

The rest of this essay is organized as follows. In Chapter 2, we will present a few early query systems to introduce the history of program queries. Chapter 3 discusses some of the modern applications of program queries. The next two chapters provide a survey of the design space of program query system. Chapter 4 reviews query language paradigms. Chapter 5 discusses storage technologies. Finally, Chapter 6 concludes.

Chapter 2

Early Development

Research in program queries can be dated back to the 1970s. In this chapter, we will compare and discuss a few early program query systems.

2.1 FACES and FAST

In the mid-1970s, Ramamorrthy and Ho developed FACES (FORTRAN Automatic Code Evaluation System) [86] for analyzing FORTRAN program. FACES supports a few predefined queries allowing the user to perform program analysis such as variable tracing.

FACES parses the source code and stores the data in three tables: a *Symbol* table contains the variable names and subroutine names along with their types, a *Use* table stores the information of a symbol and its usage, and a *Node* table records the predecessor and the successor of each statement. So basically, FACES stores the symbol information and statement location.

FAST [16] is built on top of FACES's table generator. Instead of creating its own data store, FAST maps the tables generated by FACES into a commercial hierarchical database, System 2000. This can enhance the scalability of the system and reduce the cost of development. Although FAST and FACES store the same kind of data, FAST provides a query language such that the user can have more control during analysis. For example, FAST can handle compounded query likes "*find all the modules that are referenced modules A but not modules B*".

2.2 SCOPE

SCOPE is a programming analysis environment for LISP [73]. It supports simple display of program structure, cross-reference, variable usage, and side-effect queries.

In contrast to FAST, SCOPE does not use a relational database but stores program data in some indexed data structures in memory. The authors made these design choices because of several reasons: first, the data stored in SCOPE is small so it does not require a mechanism for maintaining large data; second, SCOPE uses binary relations for the internal representation of a program. Tailor-made data storage could be employed to advantage; and third, they think that the kind of retrieval request is more complex than that normally found in database systems.

SCOPE is actually part of an integrated development environment (IDE) called INTERLISP [96]. This makes it different in some of the design choices when compared with the other systems. SCOPE can interact with other modules in the INTERLISP system. For example, when a user queries about the usage of a variable, SCOPE will allow the user to open the INTERLISP editor and go to the lines where these usages are found.

Furthermore, SCOPE provides two interfaces. One is an English-like query language that a user can use to interact with SCOPE; another one is an API for development. Therefore, SCOPE can act as a building block for a program analysis system.

2.3 OMEGA

Mark Linton developed OMEGA [69] in the early 1980s. It allows a user to query structural information of Pascal-like programs.

OMEGA is an ambitious system. Compared with FAST or SCOPE, OMEGA tries to store detailed information about a program. A table is created for each kind of program construct in the OMEGA system, such as *call statement*, *ifthen statement*, *loop statement*, etc. This allows OMEGA to support complicated program queries like "*find all the functions which contain a variable with name x inside an if-then statement*", which is impossible to express in either FAST or SCOPE.

Similar to FAST, OMEGA stores the program information in an INGRES relational database. The use of a database management system allows OMEGA to store a large amount of data.

Instead of creating its own query language, the user has to use the relational query language of INGRES, QUEL, to query the database directly.

Due to technological limitations at the time and the large amount of detailed information, OMEGA is extremely slow. For example, it takes 446 seconds to answer a query which displays 1000 lines of code.

2.4 Summary

The above systems are some of the earliest research related to program queries. Although all the above systems are designed to support program development and intended to support general purpose queries, their designs were quite different. They differ in the following design aspects:

Storage Mechanism: Instead of building the storage system from scratch, FAST and OMEGA use existing database management systems. This can enhance the scalability of the system and allows the developers to focus on other technical aspects during development. SCOPE stores the program data using its own data storage system. This makes it possible to design an optimized storage mechanism to fit its internal data representation.

Query Language: The query language of the three systems differ in their expressiveness. OMEGA uses the relational database query language directly. This allows the user to express complicated queries. FAST and SCOPE invent their own query languages. They are much more restrictive than the OMEGAs one. On the one hand, creating a new language allows developer to fully control the languages expressiveness. On the other hand, it is not easy to design a good language.

Granularity: Apart from the above differences. The three systems make a different choice in data granularity. OMEGA stores the program data in a much greater detail than the other two. This, together with an expressive query language, allows complicated queries to be expressed.

Also we see that the implementations struggle with performance. A query system with an expressive query language and support for fine-grained details is obviously desirable, but this may lead to unacceptable performance as shown in the OMEGA example.

These days, program queries have been applied to various software engineering areas. There is also research in query language paradigms and storage mechanisms. In the rest of this essay, we will review these issues one by one.

Chapter 3

Applications of Program Queries

In the last chapter, we have seen a few general program query systems in the early day. In this chapter, we will look at some of the modern applications of program query technologies.

Program queries have been used in different areas, including Architectural Visualization [57, 58, 74, 94], Design Pattern Recovery [34, 62, 79], Design Rule Verification [6, 48, 51, 72, 76] and Debugging [17, 25, 41, 68, 71, 83]. In the following sections, we will review some of the research in these areas.

3.1 Architectural Visualization

Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns [92]. Having an understanding of software architecture is beneficial to tasks that require a software engineer to understand the source code, for example, software maintenance. An Architectural Visualization tool such as SHriMp [95] presents the architectural information in graphical form. A user can browse an architectural diagram by panning and zooming. However, the diagram is usually very large. The user can easily get lost during navigation.

Program queries can be applied to Architectural Visualization by adding a search function to visualization tools [57, 58, 74, 94]. This allows the user to navigate to the targeted portion of the diagram efficiently. In this section, we are going to look at some of the query systems in this area.

3.1.1 Hy+/Graphlog

Hy+ is a general graph visualization and query system. In [31, 74], Mendelzon and Sametinger demonstrated the use of the Hy+ system in Architectural Visualization. After parsing the source code into a Hy+ readable format, Hy+ can display the architectural diagram in the user interface.

Hy+ uses GraphLog [33] as its query language. GraphLog is a graphical query language which can be translated to other logical languages. Hy+ provides an interface which allows users to edit their GraphLog queries in graphical format. A user can specify the structure of interest in the query. For example, Figure 3.1 shows a GraphLog query "*find all the classes that inherited from class DomainObject and override the toString() method*". The GraphLog query is then processed by a logical engine and the result is displayed in graphical format.

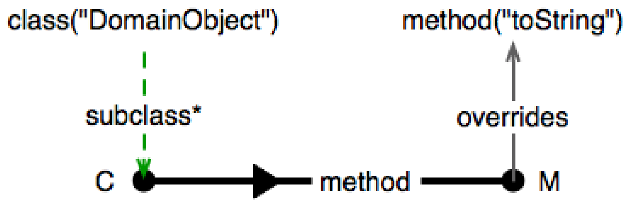


Figure 3.1: A Graphlog Query

3.1.2 IAPR

IAPR [58] is a program query system that based on Constraint Satisfaction Techniques. It provides a graphical user interface that allows a user to specify the structure of interest in graphical form. Elements in a query can be marked as optional. This approximate matching feature is useful when a user want to query a number of structures which are very similar in structure but not identical.

IAPR compares the query patterns with the architectural graph using Constraint Satisfaction Techniques. The optional structural elements are modeled as soft constraints in the constraint solver. The result is a set of full and near pattern matches in the architecture displayed in graphical format.

3.2 Design Pattern Recovery

Design patterns have been widely used in software development as recurring solutions to some common design problems. However, identifying the patterns in the source code is not easy. The implementations of design patterns are usually spread across multiple classes. There is a lot of research in design pattern discovery. A good survey in this area can be found in [93]. Some of the research involves program queries [34, 62, 79].

These tools have a lot in common with the program query tools in Architectural Visualization introduced in Section 3.1. Both involve querying design level patterns in the architectural representation. Architectural Visualization tools such as Hy+ [74] can also be applied to Design Pattern Recovery, but not all of them. Since design patterns are implemented across multiple classes, query systems in design pattern recovery are required to provide support in specifying class relationships such as inheritance, aggregation, association and call dependency. In this section, we will discuss a few design pattern recovery tools that were based on program queries.

3.2.1 Pat

Pat [62] is a system for searching for design patterns in C++ source code. An interesting feature of Pat is that the user can define the structure of a design pattern in an OMT diagram [88], which is a predecessor of the UML class diagram [12]. Figure 3.2 shows an example of OMT diagram for the Adapter Pattern.

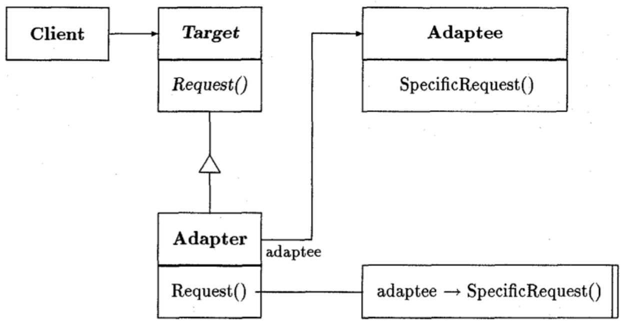


Figure 3.2: OMT Diagram for the Adapter Pattern [62]

Pat parses the C++ header files in the source code to construct an architectural diagram and then converts it into Prolog facts. The design pattern diagram specified by the user is also converted into Prolog rules. These Prolog facts and rules are processed by a Prolog engine to find the design patterns.

As the system only analyzes the header files, it is completely unaware of calling dependencies. Not surprisingly, the approach suffers from finding a lot of false positives. They reported the false positive rate is more than 50%.

3.2.2 FUJABA

A complication with design pattern recovery is that developer may implement a design pattern in ways that slightly differ from its formal definition. These implementation variants are hard to recognize by structural matching techniques. FUJABA [79] overcomes this problem by iterative querying.

Both source code and pattern queries are represented by an Abstract Syntax Graph (ASG). This allows a user to define their query structure patterns in fine-grained details. The system tries to match the query structure to the source code ASG in an iterative process. The user can edit their queries based on the intermediate results.

3.2.3 DPRE

DPRE [34] is design pattern recovery tool based on visual grammar parsing. Visual grammar [36] is a formal grammar that can be parsed using LR parsing techniques. A design pattern in DPRE is defined by step-by-step grammar reduction rules. For example, Figure 3.3 shows the reduction rules that instructs the parser to recognize the Bridge pattern.

Once the grammar is specified, DPRE creates a visual language parser to recognize the grammar patterns. The parser then parses the UML representation of the source code to recognize the design patterns. This approach discovers the design patterns efficiently in linear time.

3.3 Design Rule Verification

Software Design Rules are constraints on the behavior and structure of a program [76]. Enforcing design rules can help software quality. There are certain types of errors that can be caught by design rule verification, but are hard to discover by software testing. SQL injection vulnerability is one example. It can be relatively easily detected by checking whether every user input string is processed by the escape function [2]. Design rule checking can also enforce programming practices. For example, one may use a design rule checker to make sure whether proper naming conventions are followed.

[2]Function that escapes the characters in a string that are treated as special operators in the system.

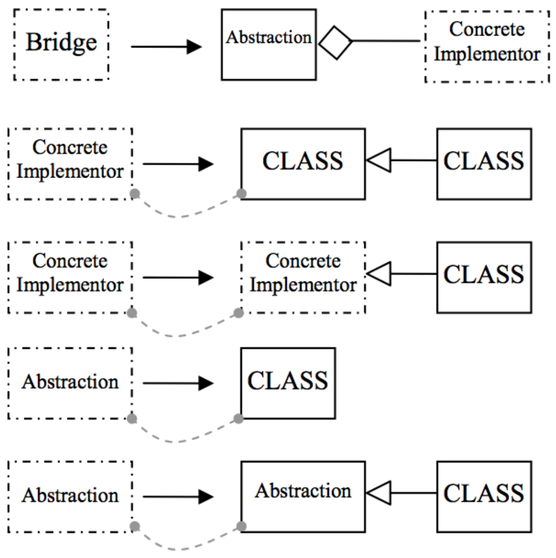


Figure 3.3: Reduction Rules for the Bridge Pattern [34]

Query systems in Design Rule Verification focus on different aspects of design rule constraints. Some focus on verifying the static structure of a program [51, 76]. Others allow the user to query runtime behavior [6, 72].

3.3.1 Metal/xgcc

Metal/xgcc [51] is an error checker based on static analysis. It provides a language to construct a state machine for matching execution sequences in the source code. When certain statements of interest are parsed, the system performs an associated action as specified in a state machine. For example, to check whether a concurrency lock is locked twice, the user can create a state machine with 2 states (*locked* and *unlocked*). When a lock statement is parsed, the state machine will transition to *locked* state. If another lock statement is parsed while the machine is still in the *locked* state, an error report action is performed. This kind of analysis is also useful in detecting resource leak by ensuring every call of a resource allocation statement is followed by a call of a resource free statement.

The state machine query is performed by parsing the Abstract Syntax Tree (AST) of the source code. As a static checker, Metal/xgcc does not track variable values or evaluate branches precisely. Loops are also analyzed conservatively. False positives are unavoidable. Metal mitigates this problem by ranking. The query results are ranked by their possibility of being false positives and the severity of the errors. So the true errors are reported first in the result list.

3.3.2 PDL

PDL [76] is a query language designed for checking structural design rules. It allows a user to query static program entities such as classes, fields and methods in the source code. By specifying the structural characteristics of certain code smell [3], the user can check whether a program conforms with a given coding style. However, behavioral rules such as constraints on API calling sequences, can more naturally be expressed in Metal/xgcc than in PDL.

PDL is designed for the .Net framework [1]. The PDL rules are compiled by a source-to-source compiler into a C# program. This allows the compiler to perform type-checking and optimization on the query code. The result is then executed to validate the design rules.

3.3.3 PQL

PQL [72] is a design rule checker for verifying the runtime behavior of a program. Since matching is done at runtime, queries in PQL are not limited to structural program entities. Runtime variable values and exact execution sequence are also accessible. This allows PQL to outperform Metal/xgcc in term of accuracy.

The user queries are parsed into matching code. These matchers are then instrumented into the target locations in the program. Analysis is performed when the user runs the program.

3.4 Debugging

A debugger is a tool that allows a developer to control the execution of a program. A user can set up breakpoints to stop a running program and examine the runtime value of the variables. Most debuggers such as Visual Studio .Net and Eclipse even support conditional breakpoints. Such a breakpoint is only activated if the condition is satisfied. However, all these kinds of breakpoint mechanisms require the user to know the exact locations (e.g. line number) where breakpoints should be set. Furthermore, if

[3] A code smell is a surface indication that usually corresponds to a deeper problem in the system [45].

a breakpoint condition spans over a number of classes, e.g. setting conditional breakpoints to all Observers in an Observer pattern, the user has to set a breakpoint in every Observer class and repeat the dynamic condition for each one. This problem can be addressed by program queries. Instead of placing breakpoints at individual lines of code, the user can specify the breakpoints by writing a query. A single query can specify both static and dynamic conditions. It is the debugger's responsibility to determine the static breakpoints locations where the breakpoint might apply, and to attach any dynamic conditions as necessary.

There are many query systems designed for debugging [17, 25, 41, 68, 71, 83]. In this section, we will review some of them.

3.4.1 Coca

Coca [41] is a query-based debugger for C programs. It is built on top of the *gcc* compiler and the *gdb* debugger. Before allowing the user to query, Coca has to precompile the source code. This is to enable the *gdb* breakpoint mechanism and generate prolog tables for the program information that is not available via *gdb*.

Coca provides a breakpoint language in Prolog. The user can specify conditional breakpoints by predicating on the entry and exit events of the function calls. Coca then computes a superset of lines that may correspond to the requested event and set them as line-based breakpoints in *gdb*. These *gdb* breakpoints are skipped by Coca if the breakpoint condition is not satisfied. Once the breakpoint condition is reached, the user can query the variable values or step to the next breakpoint using an interactive interface.

3.4.2 On-the-fly Query-based Debugger

Lencevicius et al. developed an On-the-fly query-based debugger [68] for Java programs. It offers an SQL-like language to define the event-based conditional breakpoints.

The debugger supports *on-the-fly* queries. This means that the user can issue new query whenever a breakpoint is reached. For example, suppose a conditional breakpoint is triggered when the program extracts an element from a heap data structure. The user is not quite sure what happened and therefore creates another query to monitor the inserting function. This allows the user to further examine the breakpoint without restarting the program. However, since the system does not store the execution history of the program, the user cannot query an event happened in the past.

To support *on-the-fly* queries, the debugger needs to instrument the Java byte-code at every location that may be queried in runtime. This results in significant overhead in performance.

3.4.3 Omniscient Debugger

Pothier et al. developed the Omniscient Debugger for Java programs [83]. Similar to other query-based debugger, the Omniscient Debugger allows user to specify dynamic breakpoints using program queries. The omniscient debugger works on post-mortem database of stored program history. This enabled the users to query the program state at any time and allow them to step over the execution trace back and forth. For example, when the program paused at a breakpoint, a user can trace back to the point in execution "when variable x was previously assigned value y".

Although this is very powerful, their approach requires extensive hardware resources. A large amount of disk space is needed to store the execution history of a program. To cope with this problem, the authors suggest the use of a distributed database for efficient data storage.

3.5 Summary

In this chapter, we have reviewed research in different application areas. Program querying technologies have been applied in supporting different purposes, such as querying structural patterns in Architectural Visualization and Design Pattern Recovery, specifying structural constraints in Design Rule Verification and defining runtime conditions in Debugging.

Chapter 4

Program Query Language Paradigms

Program query languages can be classified into different language paradigms. Each language paradigms has its own strengths. In this chapter, we will review some common query language paradigms used in various query systems, as well as provide some code snippets as examples.

4.1 Logical Query Languages

Logical Query Languages represent program information in terms of logical facts and rules. Facts are assertions about an application domain, such as "*A is a class*". Rules are sentences that allow us to deduce facts from other facts. For example, "*A is subclass of B if and only if B is the parent of A*". Program queries in Logical Query Languages are expressed in predicates. The answer of a query is a set of variable bindings, based on given facts and rules that satisfy the query.

Logics have been studied extensively and there are many variants that strike a different balance between expressiveness and performance. For example, Prolog is a Turing complete programming language. Sometimes this expressive power is not necessary and may even not be suitable for database system. For example, the evaluation strategy of Prolog does not guarantee termination when evaluating a query [22]. This is why Datalog [22, 46] was developed. Datalog is designed specifically for querying databases. Syntactically it is a subset of Prolog and does not provide any data structures such as list. It also poses certain stratification restrictions on the use of negation and recursion. Though less expressive, Datalog queries are more efficient to evaluate and termination is guaranteed.

Logical Query Languages have several attractive properties. First, the same language can be used to represent the program data and query the program information. The same deductive engine can be used for both data storage and query evaluation purposes. Second, deductive engines for popular logical languages like Prolog and Datalog are widely available. There is no need to reinvent a new language and a new deductive engine. Third, logical languages are expressive. For example, although Datalog has less expressive power than Prolog, it is able to express complex queries such as those involving recursive definition or transitive closure. These kinds of queries are poorly supported by Relational Query Languages, but they are important for program queries. For example, queries about inheritance or call graphs typically require recursion or transitive closure.

There are many query systems using Logical Query Languages [7, 8, 30, 50, 57, 61, 98, 102]. In this section, we will have a look at some of them.

4.1.1 CodeQuest

CodeQuest [50] is program query system for the Java language. It provides a logical query language but the program information is stored in a relational database. Although CodeQuest can store the program data in a logical representation, it is more efficient to store it in a relational database due to better support for query caching and optimization. A program query is specified using Datalog. It is then translated into SQL by CodeQuest and eventually evaluated by the database system.

In CodeQuest, queries are expressed in Datalog clauses. For example, Program 4.1 shows a CodeQuest query which finds all methods M that write a field with a particular type T. The query result is a set of method and type pairs that satisfied all the clauses in the query.

Program 4.1 A Simple CodeQuest Query

```
q(M, T) :- method(M), write(M,F), hasType(F,T).
```

CodeQuest supports recursive definition. This is hard to achieve in other query language paradigms. Program 4.2 shows an example of defining transitive subtype relationship *hasSubTypeStar* in CodeQuest. *hasSubTypeStar* is satisified if and only if T and S are the same type or S is a subtype of T.

Program 4.2 Recursive Definition in CodeQuest

```
hasSubTypeStar(T, T) :- type(T).
hasSubTypeStar(T, S) :- hasSubType(T, S);
                        hasSubType(T, MID), hasSubTypeStar(MID, S).
```

4.1.2 JTL

JTL [30] is a query language for Java code. Compared to other logical query languages, JTL does not use a Prolog-like syntax. Instead, the JTL syntax closely mimics the Java syntax. This allows JTL query to look syntactically intuitive to a Java programmer without scarifying the expressiveness of a logical language. For example, Program 4.3 shows a JTL query to search for abstract classes in which there is a field of type `long` or `int` and that has no abstract methods.

Program 4.3 JTL Syntax

```
abstract class{
        [long | int] field;
        no abstract method;
}
```

Although JTL looks like a Syntactic Query Language (see Section 4.3), we classify it under the logical language paradigm. A JTL query consists of a set of predicate definitions. The underling language model is based on logical relations rather than the Abstract Syntax Tree (AST) of the program. For example, a user can define a predicate in a syntax that resembles Datalog. Program 4.4 shows a user-defined predicate $p(S)$ which is satisfied if and only if *an abstract class S extended another abstract class X*.

Program 4.4 Defining Predicate in JTL

```
p(S) := abstract, extends X, X abstract;
```

Queries in JTL can be translated to Datalog. For example, Program 4.5 shows an equivalent Datalog code of the query shown in Program 4.4. The resulting Datalog query can then be evaluated using a Datalog engine.

Program 4.5 Equvalent Datalog Code for the JTL Predicate in Program 4.4

```
p(S) :- abstract(S), extends(S,X), abstract(X).
```

4.2 Relational Query Languages

Relational Query Languages are based on the theoretical foundation of relational algebra or relational calculus [28]. The program data is viewed as relations, which are defined as sets of tuples [27]. For example, *fields* in an Object-oriented language can be represented using the relation defined in Table 4.1.

Table 4.1: A Relational Heading for Field

Field	field_id	type_id	name	inClass_id

Using Relational Query Languages has several advantages and limitations. On the one hand, commercial strength relational database management systems are widely available. Caching, query optimization and scalability techniques have long been investigated. A query system developer can also choose to use existing database query languages directly without worrying about inventing a new language. On the other hand, relational algebra cannot handle transitive closure. Queries for inheritance or call dependency analysis may be impossible to express. Some Relational Query Languages address this by providing additional language constructs. For example, PQL [56] provided a Kleene Star operator "$*$". A transitive call relation between p and q is expressed as $Call * (p, q)$.

Despite these complications, Relational Query Languages are popular in program query systems [24, 42, 44, 48, 56, 60, 67, 69, 82]. We will discuss some of them in this section.

4.2.1 OMEGA

As introduced in Chapter 2.3, OMEGA [69, 70, 84] is a program analysis system for Pascal code. OMEGA stores the program data in a relational database and uses the database query language, QUEL [52], as the query language.

QUEL syntax is similar to SQL. A QUEL query consists of a set of **Range** statements and **Retrieve** statements. The **Range** statement indicates the relation(s) from which data is to be retrieved and the **Retrieve** statement is a selection statement. For example, Program 4.6 shows a simple QUEL query for *retrieving all variable names in the variables relation table.*

Program 4.6 A Simple QUEL Query

```
range of v is variables
retrieve(v.name)
```

OMEGA represents the program data in a large number of relational tables. Querying useful program information may require a large number of relational joins. For example, Program 4.7 shows a query for "*finding all statements that assign a value to a variable with name 'a'* ". This simple query requires 4 relational joins.

Program 4.7 Relational Joins in QUEL

```
range of v is variables
range of s is statements
range of a is asgstmts
range of n is names
retrieve (s.all) where s.stmt-id == a.id and
                       a.lhs-id == v.id and
                       v.name == n.id and
                       n.identifier == "a"
```

4.2.2 PTQL

PTQL is a Relational Query Language for querying execution traces of a running program [48]. The system keeps track of every method invocation and object allocation during execution. This is an example of how a Relational Query Language can be applied to querying runtime data.

The syntax of PTQL is similar to SQL. PTQL provides two relations, *MethodInvoc* and *ObjectAlloc*, for retrieving runtime data. These data, including the starting and ending time of the event, runtime values of the variable and the id of the corresponding running thread. For example, Program 4.8 shows a query for retrieving the starting time and the ending time of method *bar* in class *Foo*.

Program 4.8 PTQL Query for Querying the Starting Time and Ending Time of a Given Method

```
SELECT Y.startTime, Y.endTime
FROM MethodInvoc('Foo.bar') Y
```

PTQL provides relational join operations for querying the timing relationships among different events. Program 4.9 shows a query for validating the consistency of the `equal()` and `hashCode()` method. It reports if there is any instance when the two Java objects, `x` and `y`, `x.equals(y)` but `x.hashCode() != y.hashCode()`.

Program 4.9 PTQL Query for Checking Consistency of `equal()` with `hashCode()`

```
SELECT xhc.implClass, yhc.implClass, eq.implClass
FROM MethodInvoc('Object.equals') eq
        JOIN MethodInvoc('Object.hashCode') xhc
                ON eq.receiver = xhc.receiver
        JOIN MethodInvoc('Object.hashCode') yhc
                ON eq.receiver = yhc.receiver
                AND xhc.result != yhc.result
WHERE eq.result = true
```

4.3 Syntactic Query Languages

In Syntactic Query Languages, queries are not specified by predicates or relations. Instead, program queries are specified by syntactic patterns, for example, lexical patterns on source code or structural patterns on Abstract Syntax Trees (AST). Regular expression matching is one of the simplest examples. Program 4.10 shows a Unix `grep` command for finding all functions defined in file *code.c*.

Program 4.10 Simple Syntactic Query with Regular Expression

```
grep -Eho "^s*function w+" code.c
```

Syntactic Query Languages can be further divided into 2 types [4], Concrete Syntax and Abstract Syntax. Their difference is illustrated in the following example. Suppose a user wants to match the looping statements in the source code. In Concrete Syntactic Query Languages, the user may need to declare different patterns for matching different kinds of loops (`for`, `while` and `doWhile`). In Abstract Syntactic Query Languages, an abstract `LOOP` elements may be provided and the user only needs to specify one pattern to match all kinds of looping constructs.

As the pattern queries follow the actual structure in the source code.

Syntactic Query Languages are more intuitive than Logical or Relational Query Languages in general. However, Syntactic Query Languages are usually less expressive. For example, a query like "*finding all functions that called by function A but not function B*" may be hard or impossible to express in some Syntactic Query Languages.

There are a number of query systems that adopted Syntactic Query Languages [4, 37, 39, 64, 78, 80, 91]. In this section, we will discuss some of them.

4.3.1 LSME

LSME is a light weight tool for quickly extracting program information [78]. The source code is not actually parsed but simply separated into tokens. Therefore, analysis can be performed even when the source code is not entirely correct.

Queries in LSME are based on lexical matching using regular expression. Therefore, matching is based on concrete syntax. For example, Program 4.11 shows a LSME pattern for matching C function declaration. The names within angle brackets are the variables that will be matched to tokens. Special characters are escaped by the backslash character. Optional elements are indicated by square brackets.

Program 4.11 LSME Pattern for Matching C Function Declaration

```
[ <type> ] <functionName> \( [ <argumentList> ] \) \{
```

The power of Regular expressions is sometimes too limited. So in LSME, we can augment with action code written in the ICON [49] language. The action will be checked when the pattern is matched. In our example, suppose we want to exclude the functions *foo, bar, baz* in our analysis. We can put some action code to check the function name matched (Program 4.12).

Program 4.12 Action Code in LSME

```
[ <type> ] <functionName> \( [ <argumentList> ] \) \{
        @ if functionName == ("foo" | "bar" | "baz") then fail @
```

4.3.2 GENOA

GENOA [38–40] is a Syntactic Query Language. Unlike LSME which treats the program source code as plain text, GENOA queries are matched to the Abstract Syntax Tree (AST) of the program. Since the matching is based on abstract syntax, queries in GENOA are less intuitive than LSME. However, query patterns can be more precisely specified.

In GENOA, a query pattern is specified according to the AST structure. Program 4.13 shows an example for matching a C style `switch` statement in GENOA. Since the query code is based on AST traversal, the syntax is quite verbose. The `Root CFile` declaration (line 2) specified that our pattern is to be matched to the whole file. The matching starts by looking for `Switch` node in the AST (line 4). If it is found, we move to the child `switchbody` under Switch (line 5) and search for `Block` node (line 6). If it is found, we move to its child node `blockbody` (line 7). And then, for each of the `Statement` node (line 8) under `blockbody`, we search for the `Case` and `Default` nodes (line 9 and 10). Actions such as "*printing the corresponding line number*" can also be put in there.

Program 4.13 GENOA Query for Matching C Switch Statement

```
 1 PROC cswitch
 2 ROOT CFile
 3 {
 4 (?Switch
 5     <switchbody
 6         (?Block
 7             <blockbody
 8                 {Statement
 9                     (?Case /* Do something */ )
10                     (?Default /* Do something else*/ )}>)>)
11 }
```

4.4 Visual Query Languages

In Visual Query Languages, queries are depicted by visual representations of domains of interest. For example, using box-and-arrow diagrams or graphical icons to formulate the query [21].

Visual Query Languages have some advantages over Textual Query Languages. For example, human perception is more efficient to recognize relations in visual form then textual form. Visual Queries are more intuitive to represent relations between different query elements. In addition, graphical notations such as UML diagrams are adopted in software design processes, so it is more natural to use a Visual Query Language if the system is designed for querying design level program data. However, complicated queries are hard to express in Visual Query Languages. Visualization of large amount of data is also challenging [53].

There are a number of query systems that adopted Visual Query Languages [32, 35, 58, 62, 74, 79, 85]. In this section, we are going to review some of them.

4.4.1 Hy+/Graphlog

As introduced in Section 3.1.1, Hy+ is a software visualization and query tool. The user can specify structural patterns using a Visual Query Language called Graphlog [33]. A simple Graphlog query has been shown in Figure 3.1. A node represents element constant or variable while an edge denotes a relation.

Queries in Hy+ are divided into two types: *define* and *filter*. Both of them using the same syntax but different in semantic meaning. The *define* queries specify the structure of interest in the program while the *filter* queries control *which structures of interest* and *how* the data is presented. Figure 4.1 shows an example. A *define* queries is enclosed by a *defineGraphLog* box and a *filter* query is enclosed by a *showGraphLog* box.

Graphlog queries support transitive closure, which is useful for expressing inheritance or call dependencies. Figure 4.2 shows an example of transitive closure. The *subclass** edge denotes that *C1* can be any direct or indirect subclass of class *CompositVObject*.

4.4.2 FUJABA

Niere et al. developed a design pattern recovery tool using FUJABA [79], which is a model transformation tool that can convert UML diagrams into Java code. In their system, query patterns are specified in terms of Abstract Syntax Graph (ASG). For example, Figure 4.3 shows the a FUJABA query representing a class association relationship. The structural elements are denoted by nodes while their relations are denoted by edges.

The oval-shaped nodes represent the sub-patterns. In FUJABA, the users can build up a complex query pattern from simple sub-patterns. For example, creating the sub-patterns to specify some simple class relationships, such as generalization and association, and combine them to specify some

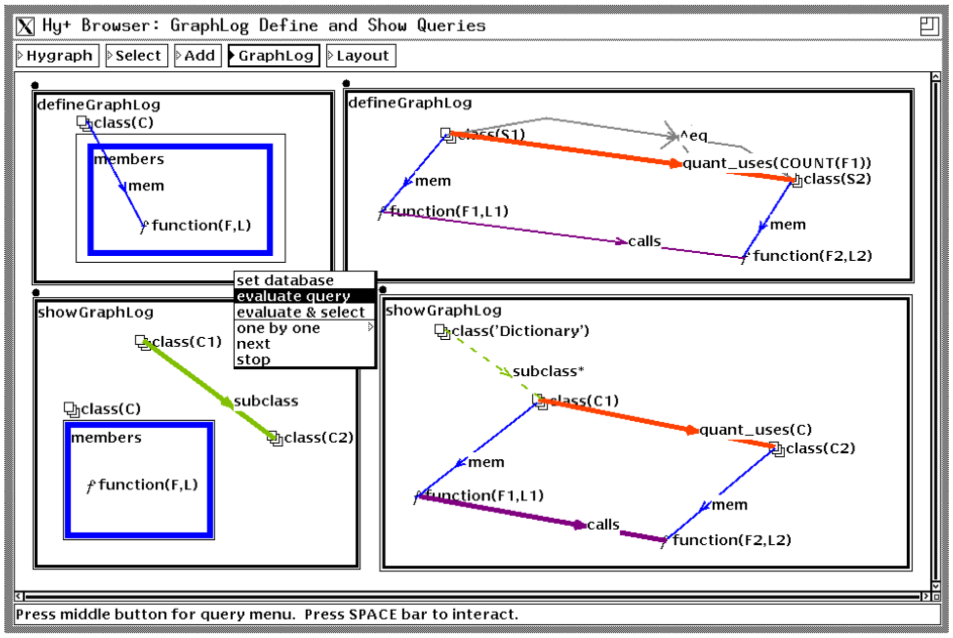


Figure 4.1: Define and Filter Queries in Hy+ [74]

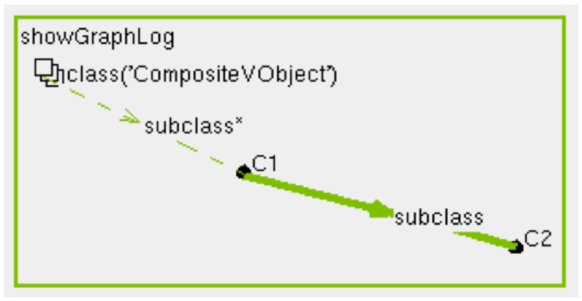


Figure 4.2: Transitive Closure in Hy+ [74]

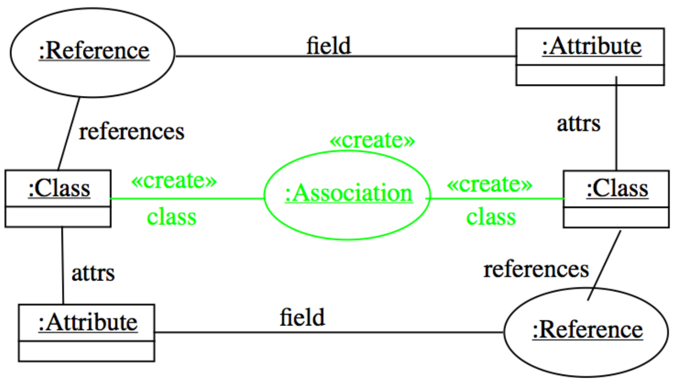


Figure 4.3: Association Relation in FUJABA [79]

more complicated design patterns. For example, Figure 4.4 shows a query pattern for the Composite Pattern. There are three sub-patterns in this query: generalization, delegation and association.

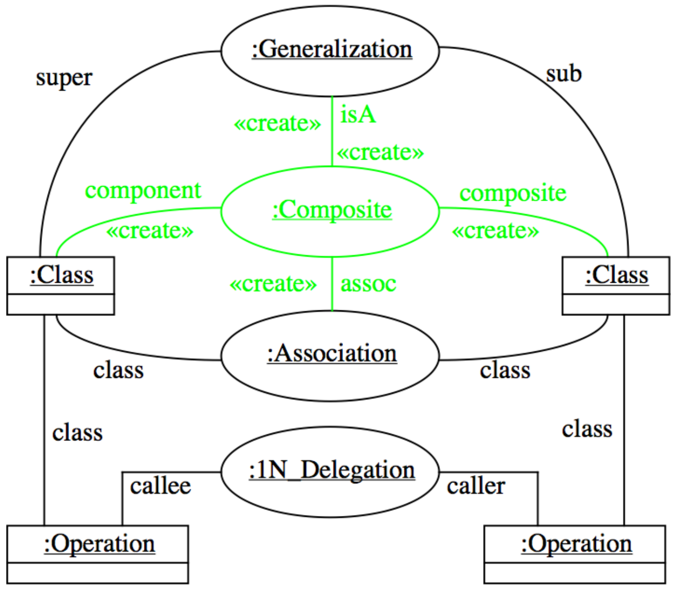


Figure 4.4: Sub-patterns in FUJABA [79]

4.5 Summary

In this chapter, we have reviewed four common query language paradigms: logical, relational, syntactic and visual. Each of these language paradigm has its own strengths. When integration with industrial strength storage is important, Logical and Relational query languages are better choices. Among the two, Logical query languages are more expressive while Relational query languages are better understood in query optimization. Syntactic and Visual query languages are more intuitive to the user. Among them, Visual query languages are even more intuitive, but they have less scalability and expressiveness. Query System Developers should select a suitable language paradigm which conform with the design goals.

Chapter 5

Program Data Storage

In the last chapter, we have seen different kinds of query language paradigms. In this chapter, we will review some techniques that have been employed to store the data about programs.

Many program query systems implement their own storage mechanism [4, 6, 13, 23, 40, 73, 78, 80]. However, these storage mechanisms are usually too tailor-made to the supporting query systems and are usually not well illustrated in the literature. We are not going to discuss them in this chapter.

Some more general storage technologies have also been applied in program query systems. For example, relational databases and standard logical engines are standard and popular technologies in program queries [16, 24, 30, 50, 57, 63, 69, 74, 75, 91]. In addition, some less well-known technologies, such as Tabled Logical Engines, Binary Decision Diagrams, XML Databases, Object Databases and Reflection, have also been used in program query systems [8, 20, 43, 50, 54, 59, 65, 67, 72, 81, 98, 102]. In the following sections, we will have a look at them.

5.1 Standard Storage Technologies

There are numerous ways to store program data. The most standard ways are storing the data in relational databases and standard logical engines.

5.1.1 Relational Databases

Using relation databases is a popular choice in program query systems [16, 24, 50, 63, 69, 75, 91]. The reasons for this are straightforward. First, relational databases are natural fit for query systems that adopted relational query languages. Second, the technology is mature and industrial strength relation databases are available. This performance advantage is attractive. For example, CodeQuest [50] uses a relational database for storage even though it adopted a logical query language.

However, as mentioned in Section 4.2, relational model does not support transitive closure. Although some database vendors do provide support for recursive queries, not all relational database systems provide such features.

5.1.2 Standard Logic Engines

Some program query systems [30, 57, 62, 74] store their data in Standard Logic Engines, for example, Prolog and Datalog engines. They are attractive for the query systems that adopted logical query languages because a query system can serve as both a query engine and a place for data storage. In addition, unlike relational databases, recursive queries are well supported in logic engines.

Although logic engines are widely available, some standard Prolog engines suffer from efficiency and scalability problems [37]. However, there are some more sophisticated logic engines that have been developed to address these issues. We will review them in Section 5.2.1.

5.2 Other Storage Technologies

Besides the standard storage technologies mentioned above, some more specific storage technologies have also been applied to program queries. For example, Tabled Logical Engines and Binary Decision Diagrams are optimized for performance. Object and XML databases provide a better support in storing complex structures such as Abstract Syntax Trees (AST). Reflection is a programming language feature, but it can act as a program data repository that comes for free. In this section, we will review these technologies one by one.

5.2.1 Table-based Deductive Engine

As mentioned earlier in Section 4.1, one of the advantages of using Logical Query Languages is the abailability of well optimized deductive engines. XSB [3, 89] is a logical deductive engine that supports both Prolog and Datalog. In contrast with a standard Prolog engine, XSB employs a *Tabled Resolution* strategy. The queries are evaluated bottom-up and redundant computations are avoided using memoization. This does not only improve evaluation efficiency, but also allows queries to terminate correctly in many cases where Prolog does not [3, 87].

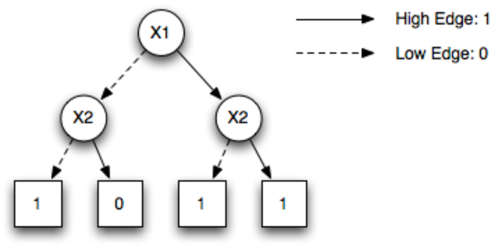
There are several logical query systems that use table-based deductive engines. For example, CodeQuest has been has been evaluated using XSB [50, 61]. DIMPLE [8] and JQuery [98] are not using XSB. However, the deductive engines they used (YAP [97] and TyRuBa [2]) also provide a similar table-based evaluation feature.

The major drawback of using these deductive databases is high memory consumption. Both XSB and JQuery have been reported as memory hungry and lacking in scalability [50, 61].

5.2.2 Binary Decision Diagram (BDD)

A Binary Decision Diagram is a data structure for efficient representation and manipulation for a large number of Boolean functions [18]. It has been widely used in hardware verification and model checking which require efficient storage of a large number of states [99].

Before introducing BDD, we first introduce Binary Decision Trees. A binary decision tree is consists of decision nodes and two types of terminal nodes (0 and 1). Each decision node represents a Boolean variable and has 2 children, which correspond to the cases where the variable is assigned to value 0 and 1 respectively. For example, Figure 5.1 shows a truth table and its binary decision tree representation.



X1	X2	f
0	0	1
0	1	0
1	0	1
1	1	1

Figure 5.1: A Binary Decision Tree Representation of a Truth Table

A BDD is a compressed representation of a binary decision tree. It can be derived from a binary decision tree by merging isomorphic sub-trees and eliminating nodes with two identical children. For example, Figure 5.2 is a possible BDD representation for the binary decision tree shown in Figure 5.1. Although the use of BDDs does not improve the worst case space complexity, Burch el al. have reported that BDDs were capable to handle extremely large number of states in practice [19].

Query systems that use BDD for storage include CrocoPat [9] and PQL [65, 72]. As a BDD can only handle binary functions, a query system has to encode the data before storing them into BDD. For example, suppose we have a binary relation *Call* over the functions $\{A, B, C\}$, where $Call =$

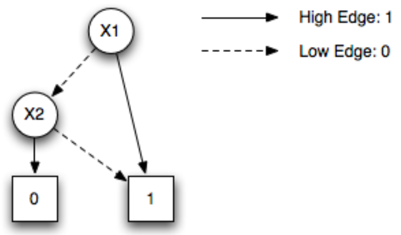


Figure 5.2: A Corresponding Binary Decision Diagram for Figure 5.1

$\{(A, B), (A, C), (B, A), (C, A)\}$. We can encode A by $(0, 0)$, B by $(0, 1)$ and C by $(1, 0)$ and result in $Call' = \{(0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0)\}$. The corresponding BDD of $Call'$ is shown in Figure 5.3.

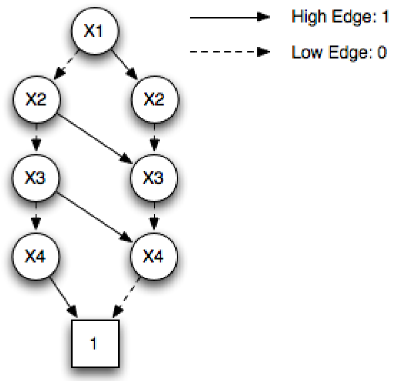


Figure 5.3: A BDD Reprentation of *Call* Relationships

In PQL, Lam et al. use a BBD to aid context-sensitive point-to analysis[4] [101]. This kind of analysis is challenging because there are over 10^{14} contexts in a typical large program. However, with the use of BDD, only several hundreds megabyte of memory is enough to perform analysis in some large Java projects [101].

While BDD is powerful, Lam et al. have reported that BDDs were not easy to use. They reported that direct implementation of BDDs caused a memory exhaustion problem and they had spent one man-year of effort in optimization [65]. However, this may be due to the fact that they imple-

[4]Point-to analysis computes which objects each variable may refer to.

mented their BDD database [100] from scratch[5] and the implementation of efficient BDD requires advanced optimization techniques [14].

5.2.3 Object-Oriented Database

An Object-Oriented Database (OODB) is a database management system with an underlying Object-Oriented model [5]. In contrast with Relational databases, OODBs are more suitable for storing complex, hierarchical data. Query System such as [20, 59, 81] make use of OODB to store the Abstract Syntax Tree (AST) of a program. The hierarchical structure of an AST makes it hard to store in relational database. For example, the OMEGA query system has used 57 relation tables to represent an AST [69].

However, the performance complications of using an OODB are unclear. On the one hand, an OODB is more efficient than a relational database in handling complex data[6]. This is because data retrieval in an OODB can be done by navigation through the pointers instead of table joins [29, 66]. On the other hand, query optimization and caching in OODB are far less well understood than those in relational databases.

It is also unclear whether OODB can help improve the performance of program queries. Very little research has been done in evaluating the use of OODB in querying program data.

5.2.4 XML Database

An XML [15] document is a text document with a hierarchical tree structure. This makes it a natural fit for storing an Abstract Syntax Tree (AST) of a program. In addition, XML is widely adopted as an industrial standard for electronic data interexchange. A standardized query language, XQuery [11], is also available for querying XML documents. Due to its popularity, industrial strength XML databases are also available. They allow import and export of XML data, and support XQuery as a query language.

Sextant [43, 90] is a program query system that uses an XML database. It exploits the generic nature of XML. Different kinds of software artifacts, including documents and source code, are all converted into XML representation and stored in an XML database. This allows a user to create cross-artifacts queries using XQuery.

[5]Some BDD packages are publicly available. Janssen has written a survey comparing different existing BDD packages [55].

[6]However, it is not true for Object-Relational (O/R) Mapping technologies, which convert objects into relational database tables.

Performance of using an XML Database in program query systems is not well evaluated. However, Schäfer et al. have tested their Sextant system using a Java project that consists of 200K lines of code [90]. Although no scientific evaluation result is reported, they claimed that the speed is satisfactory, although a large amount of memory is required.

5.2.5 Reflection

Reflection refers to the ability of a programming language to observe or modify its structure or behavior. Unlike other storage mechanisms mentioned in this chapter, query systems based on reflection mechanisms do not need to store the program data in external repositories. Instead, program data can be accessed using some special language constructs provided by the programming language.

There are several query systems that are based on reflection mechanisms [54, 67, 102].For example, SOUL provides a logical query language for analyzing Smalltalk program [102, 103]. Since SOUL is also implemented in Smalltalk and Smalltalk provides a rather strong support for reflection [47], SOUL can access the program data using reflection in Smalltalk.

While using reflection for building query systems can help reducing development effort, the powers of the query systems are highly depend on the reflection models supported by the underlying program languages. Some programming languages such as C or C++ even do not support reflection at all. However, this problem can be addressed by using reflection libraries. For example, Hobatr and Mallory use a reflection library, OpenC++ [26], in their C++ query-based debugger [54].

5.3 Summary

In this chapter, we have reviewed different technologies for storing data about programs. These technologies are selected because they are general and are not confined to any particular program query system. Relational databases and standard logic engines are standard and popular technologies. There are industrial strength implementations available in the market. Several less well-known technologies have also been applied to program queries. Tabled logical engines are logic engines that optimized for data storage. They are better in performance and scalability when compared to standard logic engines. Binary Decision Diagrams are special data structures that optimized for storing huge amount of data. However, they are complicated to use. XML databases and Object databases are more suitable for storing complex structures, such as the Abstract Syntax Tree (AST) of a program. However, their performance in program queries is not well studied. Reflection is a programming language feature, but it can also be used as a built-in data repository for a program query system. However, the powers and efficiency of the query systems are entirely dependent on the support of the underlying reflection models.

Chapter 6

Summary

Program query systems allow a user to extract program information by viewing a program as a piece of queryable data. They have long been a research interest and have been shown to be useful in many software engineering tasks. In Architectural Visualization and Design Pattern Recovery, program queries allow the users to specify the structures of interest. In Design Rule Verification, program queries allow the users to specify structural or behavioral constraint rules. In Debugging, program queries can be used to specify the breakpoint conditions.

The design space of a program query system is complex. To help organize this survey, we will structure our presentation around two major design dimensions. One dimension is the choice of language paradigm for the query language. The other dimension is the choice of storage technologies of the program data.

A Program query language is the end-user interface of a program query system. The choice of query language paradigms is important because it affects the expressive power and the usability of a query system. In this essay, we have reviewed four common query language paradigms: Logical, Relational, Syntactic and Visual. Each of each paradigm has its own strengths. When integration with industrial strength storage is important, Logical and Relational query languages are better choices. Among the two, Logical query languages are more expressive while Relational query languages are better understood in query optimization. Syntactic and Visual query languages are more intuitive to the user. Among them, Visual query languages are even more intuitive, but they have less scalability and expressiveness

Storage technologies are another major design dimension. The choice of storage technologies affects the scalability and the performance of a query system. In this essay, we have reviewed two standard storage technologies and five less well-known technologies.

Standard logic engines and relational databases are standard and popular storage technologies. There are many industrial strength implementations in the market. Tabled logical engines are logic engines that optimized for data storage. They are better in performance and scalability when compared to standard logic engines. Binary Decision Diagrams are special data structures that optimized for storing huge amount of data. However, they are complicated to use. XML databases and Object databases are more suitable for storing complex structures, such as the Abstract Syntax Tree (AST) of a program. However, their performance in program queries is not well studied. Reflection is a programming language feature, but it can also be used as a built-in data repository for a program query system. However, the powers and efficiency of the query systems are entirely dependent on the support of the underlying reflection models.

In addition, the two design dimensions are not completely orthogonal. For example, logic engines and relational databases are natural fit for the logical and relational language paradigms respectively, providing a unified representation in both language and storage level. However, a program query system designer can always choose any possible combination to fit the design goal. For example, we have seen that CodeQuest [50] uses a relational database for storage even though it adopted a logical query language.

Bibliography

[1] Microsoft .net framework developer center. `http://msdn.microsoft.com/netframework`.

[2] Tyruba. `http://tyruba.sourceforge.net`.

[3] Xsb project summary. `http://xsb.sourceforge.net/about.html`.

[4] Darren C. Atkinson and William G. Griswold. Effective pattern matching of source code using abstract syntax patterns. *Softw. Pract. Exper.*, 36(4):413–447, 2006.

[5] Malcolm Atkinson, David Dewitt, David Maier, Francois Bancilhon, Klaus Dittrich, and Stanley Zdonik. The object-oriented database system manifesto. In *Readings in database systems (2nd ed.)*, pages 946–954. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[6] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[7] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The pan language-based editing system for integrated development. *SIGSOFT Softw. Eng. Notes*, 15(6):77–93, 1990.

[8] William C. Benton and Charles N. Fischer. Interactive, scalable, declarative program analysis: from prototype to implementation. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 13–24, New York, NY, USA, 2007. ACM.

[9] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. Software Eng.*, 31(2):137–149, 2005.

[10] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[11] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. Xquery 1.0: An xml query language. `http://www.w3.org/TR/xquery`.

[12] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[13] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *SDE 3: Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 14–24, New York, NY, USA, 1988. ACM.

[14] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *DAC '90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, New York, NY, USA, 1990. ACM.

[15] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0 (fifth edition). `http://www.w3.org/TR/REC-xml`.

[16] J. C. Browne and David B. Johnson. Fast: A second generation program analysis system. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 142–148, Piscataway, NJ, USA, 1978. IEEE Press.

[17] Bernd Bruegge and Peter Hibbard. Generalized path expressions: a high level debugging mechanism. *SIGSOFT Softw. Eng. Notes*, 8(4):34–44, 1983.

[18] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[19] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[20] S. Burson, G. B. Kotik, and L. Z. Markosian. A program transformation approach to automating softwarere-engineering. In *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International*, pages 314–322, Chicago, IL, USA, October/November 1990.

[21] Tiziana Catarci, Maria F. Costabile, Stefano Levialdi, and Carlo Batini. Visual query systems for databases: A survey. *Journal of Visual Languages and Computing*, 8(2):215–260, 1997.

[22] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, 1989.

[23] Y.-F. R. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach. Ciao: a graphical navigator for software and document repositories. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 66, Washington, DC, USA, 1995. IEEE Computer Society.

[24] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The c information abstraction system. *IEEE Trans. Softw. Eng.*, 16(3):325–334, 1990.

[25] Rick Chern and Kris De Volder. Debugging with control-flow breakpoints. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 96–106, New York, NY, USA, 2007. ACM.

[26] Shigeru Chiba. Openc++ reference manual. `http://opencxx.sourceforge.net/opencxx/html/index.html`.

[27] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[28] 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.

[29] Vincent Coetzee and Robert Walker. Experiences using an odbms for a high-volume internet banking system. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 334–338, New York, NY, USA, 2003. ACM.

[30] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. Jtl: the java tools language. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 89–108, New York, NY, USA, 2006. ACM.

[31] Mariano Consens and Alberto Mendelzon. Hy+: a hygraph-based query and visualization system. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 511–516, New York, NY, USA, 1993. ACM.

[32] 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.

[33] Mariano P. Consens and Alberto O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *PODS '90: Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 404–416, New York, NY, USA, 1990. ACM.

[34] Gennaro Costagliola, Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Design pattern recovery by visual language parsing. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 102–111, Washington, DC, USA, 2005. IEEE Computer Society.

[35] Gennaro Costagliola, Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Case studies of visual language based design patterns recovery. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 165–174, Washington, DC, USA, 2006. IEEE Computer Society.

[36] Gennaro Costagliola and Giuseppe Polese. Extended positional grammars. In *VL '00: Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, page 103, Washington, DC, USA, 2000. IEEE Computer Society.

[37] 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.

[38] Prem Devanbu and Laura Eaves. Gen++ - an analyzer generator for c++ programs. Technical report, AT & T Bell Lab, New Jersey, 1994.

[39] Premkumar T. Devanbu. Genoa: a customizable language- and front-end independent code analyzer. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 307–317, New York, NY, USA, 1992. ACM.

[40] Premkumar T. Devanbu. Genoa: a customizable language- and front-end independent code analyzer. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 307–317, New York, NY, USA, 1992. ACM.

[41] Mireille Ducassé. Coca: an automated debugger for c. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 504–513, New York, NY, USA, 1999. ACM.

[42] J"urgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO — generic understanding of programs, an overview. Fachberichte Informatik 7–2002, Universit"at Koblenz-Landau, 2002.

[43] 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.

[44] L. Feijs, R. Krikhaar, and R. Van Ommering. A relational approach to support software architecture analysis. *Softw. Pract. Exper.*, 28(4):371–400, 1998.

[45] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[46] Herve Gallaire and Jack Minker, editors. *Logic and Data Bases.* Perseus Publishing, 1978.

[47] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[48] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 385–402, New York, NY, USA, 2005. ACM.

[49] Ralph E. Griswold and Madge T. Griswold. *The ICON Programming Language*. Annabooks, 1996.

[50] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In Dave Thomas, editor, *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.

[51] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 69–82, New York, NY, USA, 2002. ACM.

[52] G. D. Held, M. R. Stonebraker, and E. Wong. Ingres: a relational data base system. In *AFIPS '75: Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 409–416, New York, NY, USA, 1975. ACM.

[53] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[54] Chanika Hobatr and Brian A. Malloy. The design of an ocl query-based debugger for c++. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 658–662, New York, NY, USA, 2001. ACM.

[55] Geert Janssen. A consumer report on bdd packages. In *SBCCI '03: Proceedings of the 16th symposium on Integrated circuits and systems design*, page 217, Washington, DC, USA, 2003. IEEE Computer Society.

[56] Stan Jarzabek. Design of flexible static program analyzers with pql. *IEEE Trans. Softw. Eng.*, 24(3):197–215, 1998.

[57] Shahram Javey, Kin'ichi Mitsui, Hiroaki Nakamura, Tsuyoshi Ohira, Kazu Yasuda, Kazushi Kuse, Tsutomu Kamimura, and Richard Helm. Architecture of the xl c++ browser. In *CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, pages 369–379. IBM Press, 1992.

[58] R. Kazman and M. Burth. Assessing architectural complexity. In *CSMR '98: Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, page 104, Washington, DC, USA, 1998. IEEE Computer Society.

[59] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 226–235, New York, NY, USA, 1999. ACM.

[60] Paul Klint. *A Tutorial Introduction to RSCRIPT.* Centrum voor Wiskunde en Informatica, draft edition, MAY 2005.

[61] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *LATE '07: Proceedings of the 3rd workshop on Linking aspect technology and evolution*, page 6, New York, NY, USA, 2007. ACM.

[62] Christian Kramer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. *Reverse Engineering, Working Conference on*, 0:208, 1996.

[63] B. Kullbach and A. Winter. Querying as an enabling technology in software reengineering. In *Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on*, pages 42–50, Amsterdam, Netherlands, 1999.

[64] David A. Ladd and J. Christopher Ramming. A*: A language for implementing language processors. *IEEE Trans. Softw. Eng.*, 21(11):894–901, 1995.

[65] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM.

[66] Neal Leavitt. Whatever happened to object-oriented databases? *Computer*, 33(8):16–19, 2000.

[67] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 304–317, New York, NY, USA, 1997. ACM.

[68] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Dynamic query-based debugging of object-oriented programs. *Automated Software Engg.*, 10(1):39–74, 2003.

[69] Mark A. Linton. Implementing relational views of programs. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 132–140, New York, NY, USA, 1984. ACM.

[70] Mark Andrew Linton. *Queries and views of programs using a relational database system*. PhD thesis, University of California, Berkeley, 1983.

[71] Guillaume Marceau, Gregory H. Cooper, Jonathan P. Spiro, Shriram Krishnamurthi, and Steven P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engg.*, 14(1):59–86, 2007.

[72] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. *SIGPLAN Not.*, 40(10):365–383, 2005.

[73] Larry M. Masinter. *Global program analysis in an interactive environment*. PhD thesis, Stanford University, Stanford, CA, USA, 1980.

[74] Alberto Mendelzon, Alberto Mendelzon, and Johannes Sametinger. Reverse engineering by visualizing and querying. *SOFTWARE — CONCEPTS AND TOOLS*, 16:170–182, 1995.

[75] 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*, pages 3–16, Washington, DC, USA, 2007. IEEE Computer Society.

[76] Clint Morgan, Kris De Volder, and Eric Wohlstadter. A static aspect language for checking design rules. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 63–72, New York, NY, USA, 2007. ACM.

[77] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? *IEEE Softw.*, 23(4):76–83, 2006.

[78] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, 1996.

[79] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *ICSE*, pages 338–348, 2002.

[80] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Trans. Softw. Eng.*, 20(6):463–475, 1994.

[81] Santanu Paul and Atul Prakash. Querying source code using an algebraic query language. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 127–136, Washington, DC, USA, 1994. IEEE Computer Society.

[82] Santanu Paul and Atul Prakash. Supporting queries on source code: A formal framework. *International Journal of Software Engineering and Knowledge Engineering*, 4, 1994.

[83] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *SIGPLAN Not.*, 42(10):535–552, 2007.

[84] Michael L. Powell and Mark A. Linton. Database support for programming environments. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1983.

[85] Ansgar Radermacher. Support for design patterns through graph transformation tools. In *AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 111–126, London, UK, 2000. Springer-Verlag.

[86] C. V. Ramamoorthy and S. F. Ho. Testing large software with automated software evaluation systems. In *Proceedings of the international*

conference on Reliable software, pages 382–394, New York, NY, USA, 1975. ACM Press.

[87] Prasad Rao, Konstantinos Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A system for efficiently computing well-founded semantics. In *Logic Programming And Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Computer Science*, pages 430–440. Springer Berlin / Heidelberg, 1997.

[88] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[89] Konstantinos Sagonas, Terrance Swift, and David S. Warren. Xsb as an efficient deductive database engine. *SIGMOD Rec.*, 23(2):442–453, 1994.

[90] Thorsten Schafer, Michael Eichberg, Michael Haupt, and Mira Mezini. The sextant software exploration tool. *IEEE Trans. Softw. Eng.*, 32(9):753–768, 2006.

[91] Mirko Seifert and Roland Samlaus. Static source code analysis using ocl. In *the 8th International Workshop on OCL Concepts and Tools (OCL 2008) at MoDELS 2008*, volume 15, 2008.

[92] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[93] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In *Proceedings - 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006*, pages 123 – 132, Piscataway, NJ 08855-1331, United States, 2006. Pattern detection tools;System behavior;Code structure;.

[94] Susan Elliott Sim, Charles L. A. Clarke, Richard C. Holt, and Anthony M. Cox. Browsing and searching software architectures. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 381, Washington, DC, USA, 1999. IEEE Computer Society.

[95] Margaret-Anne Storey, Casey Best, Jeff Michaud, Derek Rayside, Marin Litoiu, and Mark Musen. Shrimp views: an interactive environment for information visualization and navigation. In *CHI '02:*

CHI '02 extended abstracts on Human factors in computing systems, pages 520–521, New York, NY, USA, 2002. ACM.

[96] W. Teitelman and L. Masinter. The interlisp programming environment. *Computer*, 14(4):25–33, 1981.

[97] Version No Value, Vtor Santos Costa, Lus Damas, Rogrio Reis, Rben Azevedo, and V. Santos Costa. Yap user's manual.

[98] Kris De Volder and Kris De Volder. Jquery: A generic code browser with a declarative configuration language. *IN PRACTICAL ASPECTS OF DECLARATIVE LANGUAGES, 8TH INTERNATIONAL SYMPOSIUM, PADL 2006*, 3819:88–102, 2006.

[99] Ingo Wegener. *Branching programs and binary decision diagrams: theory and applications.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[100] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer Berlin / Heidelberg, 2005.

[101] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not.*, 39(6):131–144, 2004.

[102] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 112, Washington, DC, USA, 1998. IEEE Computer Society.

[103] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, Pleinlaan 2,B-1050 Brussel,Belgium, 2001.