# *Evaluating optimization and reverse engineering techniques on data-intensive systems*

Csaba Nagy

Department of Software Engineering
University of Szeged

Supervisor: Dr. Tibor Gyimóthy

A thesis submitted for the degree of Doctor of Philosophy
of the University of Szeged


University of Szeged
Ph.D. School in Computer Science

December 2013
Szeged, Hungary

*"No man runs the race until he sees the dream."*

Fred C. White

# Preface

Life is full of challenges. Finishing a PhD thesis is one of those challenges, and a huge one at that. I came to realize that it is probably one of the greatest challenges in my life - at least, up to today. In some ways I found it similar to long distance hiking, my main hobby. You have a dream that you want to accomplish something in your life. You prepare for it by training on minor trails, for example. On some of those trails, you might succeed, and on some you might fail. Either way, you learn something from each and perhaps the most from your failures. And when the time comes, you rely on all the knowledge and all the experience that you have gathered in your life, and you need to do your best to reach the finishing line. So let us suppose that my PhD is the hardest hiking trail in my life, and the research problems I encountered are those minor trails that I successfully completed or just tried to finish earlier. Now, I am making my dream come true. It is a dream that I had over six years ago.

It was my aim, so I had to work on it; but of course, I could not even get close to the finishing line without the support of so many great colleagues and friends. First of all, I would like to express my gratitude to my supervisor, Dr. Tibor Gyimóthy, who gave me the opportunity to carry out this research study. Secondly, I would like to thank my article co-author and mentor, Dr. Rudolf Ferenc, for offering me interesting and challenging problems while guiding my studies and helping me. Also, I wish to thank David P. Curley for reviewing and correcting my work from a linguistic point of view. My thanks also go to my colleagues and article co-authors, namely Spiros Mancoridis, Gábor Lóki, Árpád Beszédes, Tamás Gergely, László Vidács, Tibor Bakota, János Pántos, Gabriella Kakuja-Tóth, Ferenc Fischer, Péter Hegedűs, Judit Jász, Zoltán Sógor, Lajos Jenő Fülöp, István Siket, Péter Siket, Ákos Kiss, Ferenc Havasi, Dániel Fritsi, Gábor Novák and Richárd Dévai. Many thanks also to all the members of the department over the years.

An important part of the thesis deals with the Magic language. Most of this research work would not have been possible without the cooperation of SZEGED Software Inc, our industrial partner. Special thanks to all the colleagues at the company, particularly István Kovács, Ferenc Kocsis and Ferenc Smohai.

Last, but not least, none of this would have been possible without the invaluable support of my family. I would like to thank my parents, my sister and my brother for always being there and for supporting me.

*Csaba Nagy, December 2013.*

# Contents

# List of Figures

# List of Tables

To my family

*"Any fool can write code that a computer can understand;*
*good programmers write code that humans can understand."*

Martin Fowler

# 1 Introduction

## 1.1 Data-intensive systems

At the beginning of the 21st century, information systems are not simple computer applications that we sometimes use at work, but large systems with complex architectures and fulfilling important roles in our daily life. The purpose of such systems is to get the right information to the right people at the right time in the right amount and in the right format [101].

In 1981, Pawlak published a paper reporting some of the activities of the Information Systems Group in Warsaw [99]. According to this study the basic component of an information system is a finite set of objects, like human beings and books. Pawlak also added that "*with every information system a query language is associated and its syntax and semantics is formally defined.*" Their information system was implemented for an agriculture library that had some 50,000 documents.

Since then, information systems have evolved as data volumes have steadily increased. There are reports on systems, like those in radio astronomy, where systems need to handle 138 PB (peta bytes) of data per day [102]. Another example from the high-energy physics community, the Large Hadron Collider (LHC) machine, generates 2 PB of data per second when in operation [75]. Imagine how well organized this data must be and the system handling it, to perform its operations correctly! These systems are usually referred to as data-intensive systems [27, 84–86].

The big data which data-intensive systems work with is stored in a database, typically managed by a database management system (DBMS), where it is structured according to a schema.

In relational DBMSs (RDBMS), this schema consists of data tables with columns where the tables usually represent the current state of the population of a business object and columns are its properties.

In order to support the maintenance tasks of these systems, several techniques have been developed to analyze the source code of applications or to analyze the underlying databases for the purpose of reverse engineering tasks like quality assurance and program comprehension. However, only a few techniques take into account the special features of data-intensive systems (e.g. dependencies arising via database accesses). As Cleve et al. remarked in a study on data-intensive system evolution [30]: "*... both the software and database engineering research communities have addressed the problems of system evolution. Surprisingly, however, they've conducted very little research into the intersection of these two fields, where software meets data.*"

Growing complexity and the evolution of large-scale software systems is an evergreen topic of software maintenance [16, 17]. However, besides software maintenance tasks, many further research challenges have been identified in the area of data-intensive systems such as architectural challenges (e.g. data volumes, data dissemination; data curation; use of open source software; search; data processing and analysis and information modeling) [86] and the data reengineering of legacy systems [90].

## 1.2 Goals of the Thesis

In this thesis, we describe studies carried out to analyze data-intensive applications via different reverse engineering methods based on static analysis. These are methods for recovering the architecture of data-intensive systems, a quality assurance methodology for applications developed in Magic, identifying input data related coding issues and optimizing systems via local refactoring. With the proposed techniques we were able to analyze large scale industrial projects like banking systems with over 4 million lines of code, and we successfully retrieved architecture maps and identified quality issues of these systems.

We seek answers to the following research questions:

**RQ1:** *Can automated program analysis techniques recover implicit knowledge from data accesses to support the architecture recovery of data-intensive applications?*

**RQ2:** *Can we adapt automatic analysis techniques that were implemented for 3rd generation languages to a 4th generation language like Magic? If so, can static analysis support the migration of Magic applications with automated techniques?*

**RQ3:** *How can we utilize control flow and data flow analysis so as to be able to identify security issues based on user-related input data?*

**RQ4:** *Can we use local refactoring algorithms in compilers to optimize the code size of generated binaries?*

## 1.3 Outline of the Thesis

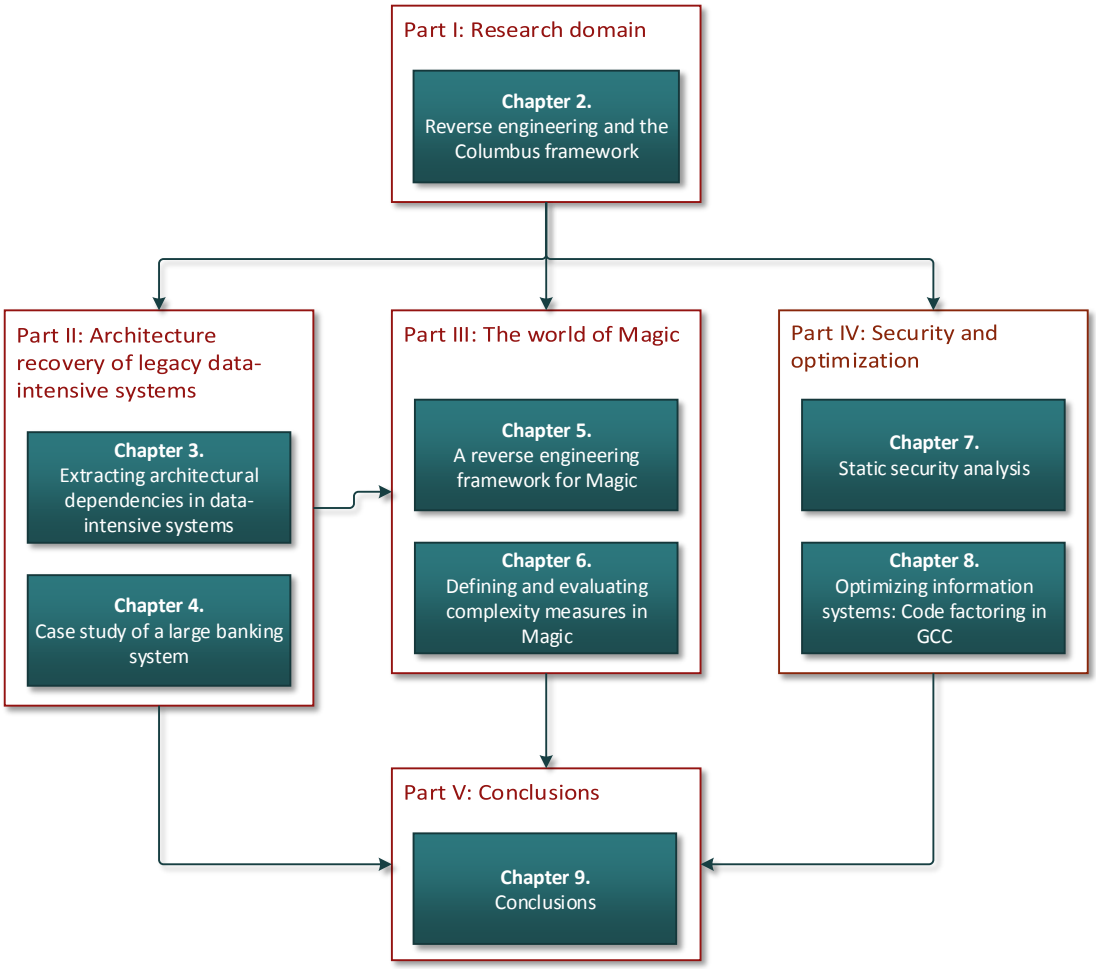



**Figure 1.3.1:** General structure of the thesis.

The thesis contains nine chapters (including the introduction and conclusion chapters) in three main parts and the additional research domain and conclusions parts, as depicted in Figure 1.3.1.

**Part I** groups introductory chapters to introduce the reader to the definitions, terms and technologies we use in later chapters.

- **Chapter 2** gives an introduction to reverse engineering legacy data-intensive systems and to the Columbus methodology.

**Part II** describes the analysis techniques we developed for data-intensive systems with a relational database management system in the centre of the architecture.

- **Chapter 3** presents a new approach for extracting dependencies in data-intensive systems based on data accesses via embedded SQL queries (CRUD dependencies). The main idea

was to analyze the program logic and the database components together; hence we were able to recover relations among source elements and data components. We analyzed a financial system written in ForrásSQL with embedded Transact SQL or MS SQL queries and compared the results with Static Execute After/Before relations. The results show that CRUD relations recovered new connections among architectural components that would have remained hidden using other approaches.

- **Chapter 4** describes a case study in which we use the above introduced methods as bottom-up reverse engineering combined with top-down techniques to construct the architecture map of large legacy data-intensive systems. The target system is a banking system written in Oracle PL/SQL. As a result, we were able to validate the technique in an industrial environment and found it useful in the architecture recovery of large PL/SQL systems. Top-down and bottom-up techniques completed each other.

**Part III** explains how we applied the Columbus methodology in a fourth generation language environment. We developed a framework for analyzing quality attributes of Magic applications and extracting architectural dependencies introduced in the previous chapters.

- **Chapter 5** describes our technique for analyzing an application written in a 4th generation language called Magic. For the analysis, we develop a reverse engineering framework based on the Columbus methodology which can be used for software quality assurance purposes and re-engineering tasks. As a result we successfully adapt the Columbus methodology in a 4GL environment and show that automatic analysis techniques for 3GLs (e.g. for quality assurance or supporting migration) support the development in this environment too.

- **Chapter 6** introduces a new complexity metric for the Magic language because during previous research studies we realized that current 3GL complexity metrics did not fulfill the needs of experienced Magic developers. With an experiment we compare Magic programs based on McCabe, Halstead and our complexity measure and evaluate the results with experienced Magic developers.

**Part IV** discusses analysis techniques for security and optimization purposes. These techniques are more general than the previous ones in the sense that they do not depend on a data-centric architecture.

- **Chapter 7** presents static analysis techniques for applications working with user input. A specific technique (working with data flow and control flow analysis) targets buffer overflow errors in C applications. The resulting algorithm is able to handle medium-size applications and identify buffer overflow errors with high precision.

- **Chapter 8** discusses the performance of local code factoring algorithms for optimizing the size of applications written in C or C++. We implemented local hoisting and sinking algorithms in GCC and we analyze the results we obtained with these algorithms here. The algorithms were implemented on different abstraction levels of GCC and successfully reduced the code size of the generated binaries.

**Part V** outlines the general conclusions of this thesis.

- **Chapter 9** summarizes the contributions of the thesis with respect to the above research questions.

Figure 1.3.1 depicts the general structure of the thesis and identifies the main dependencies among the successive parts and chapters.

## 1.4 Publications

Most of the research results presented in this thesis were published in proceedings of international conferences and workshops or journals. Section 9.1 provides a list of selected peer-reviewed publications. Table 1.4.1 is a summary of which publications cover which results of the thesis.

| Chapter | Contribution - short title | Publications |
|---|---|---|
| 3. | Extracting architectural dependencies in data-intensive systems | [128] |
| 4. | Case study of a large banking system | [125, 130] |
| 5. | A reverse engineering framework for Magic | [123, 129, 132] |
| 6. | Defining and evaluating complexity measures in Magic | [131] |
| 7. | Static security analysis | [126] |
| 8. | Optimizing information systems: Code factoring in GCC | [124, 127] |

**Table 1.4.1:** Relation between the thesis topics and the corresponding publications.

Here, the author adds that although the results presented in this thesis are his major contribution, from this point on, the term 'we' will be used instead of 'I' for self reference to acknowledge the contribution of the co-authors of the papers that this thesis is based on.

# Part I

# Research domain

*"There is a single light of science, and to brighten it anywhere is to brighten it everywhere."*

Isaac Asimov

# 2 Reverse engineering

This thesis deals with reverse engineering data-intensive systems and its applications. Reverse engineering is a hot research topic at the moment in software engineering. As large companies tend to develop large and complex software systems, reverse engineering techniques have become more important to help deal with such systems (e.g. in software maintenance). Here, we present the state-of-the-art in reverse engineering data-intensive systems. We begin by defining the terms used, then discuss reverse engineering and give an overview of the current approaches available. After, we briefly introduce the Columbus framework and the way we used it and extended it in our thesis.

## 2.1 Legacy systems

Legacy systems are typical targets of reverse engineering techniques. As Sommerville says in [109]: *"Legacy systems are old systems that are still useful and are sometimes critical to business operation. They may be implemented using outdated languages and technology or may use other systems that are expensive to maintain. Often their structure has been degraded by change and documentation is missing or out of date. Nevertheless, it may not be cost effective to replace these systems. They may only be used at certain times of the year or it may be too risky to replace them because the specification has been lost."*

From a more pragmatic point of view, Bennett says that "*legacy software is software which is vital to our organization, but we don't know what to do with it*".

In the context of data-intensive systems, it is important to note that most data-intensive systems are legacy systems. There are large industrial systems developed in AS400, COBOL, Fortran, PL/I, etc. technologies which were common solutions for data-intensive systems. Today, these systems are more than 10-20 years old and they are still evolving; hence their developers often face challenging maintenance issues [106, 107].

## 2.2 Reverse engineering

The term 'reverse engineering' covers many methods and tools for understanding and modifying software systems. It has its origin in the analysis of hardware since it is common practice in the military and in industry to analyze a competitor's product or to decipher its design.

In the context of software engineering, the most widely used and accepted definition for the term *reverse engineering* was given by Chikofsky and Cross II in [25]. The definition is based on the assumption that an ordinary life-cycle model exists for the software-development process. This cycle may be represented as the traditional waterfall or spiral, or generally as a directed graph. Within the stages of the life-cycle model, it is possible to define forward and backward activities.

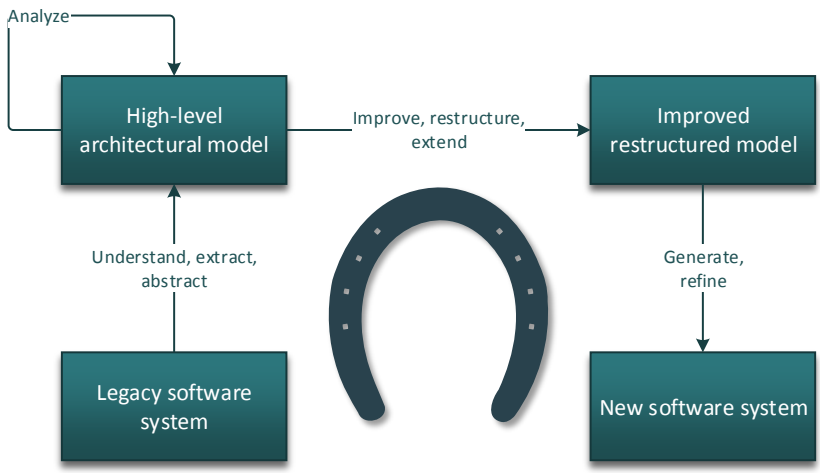



**Figure 2.2.1:** The horseshoe model for reengineering.

Chikofsky and Cross II define both forward engineering and reverse engineering based on this life-cycle model. Before giving the definition, they identify three life-cycle stages for describing key terms. These are:

- *Requirements* (specification of the problem being solved, including objectives, constraints, and business rules);

- *Design* (specification of the solution); and
- *Implementation* (coding, testing, and delivery of the operational system).

**Forward engineering** is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

**Reverse engineering** is the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction.

The re-engineering process is typically visualized by the so-called horseshoe model, a simplified version of which can be seen in Figure 2.2.1 [72, 89].

There are also some important things we should mention about the definition. For instance, reverse engineering often involves an existing functional system as its subject, but this is not a key requirement. Actually it is possible to perform reverse engineering starting from any stage of the life-cycle. Another issue is that the reverse engineering process itself does not involve changing the subject system or creating a new system based on the reverse-engineered subject system. It is simply a process of examination and not a process of change or replication.

Reverse engineering has many sub-areas. Two sub-areas that are widely referred to are redocumentation and design recovery. *Redocumentation* [25] is the creation or revision of a semantically equivalent representation within the same relative abstraction level. *Design recovery* [13] recreates design abstraction from a combination of code, existing design documentation (where available), personal experience and general knowledge of problem and application domains.

## 2.3 Reverse engineering data-intensive systems

**Data-intensive systems** are large-scale distributed software systems which integrate information taken from heterogeneous data sources [84]. Such systems are usually composed of a software system and a data system that co-evolve during the lifecycle of the system [30]. The *software system* is responsible for implementing the business logic of the application and the *data system* is responsible for storing and handling the data. An example of a typical data-intensive system can be seen in Figure 2.3.1.

Reverse engineering techniques may target the software system or the data system of a data-intensive application, while some approaches may target both of them together.

### 2.3.1 Software reverse engineering

Standard software reverse engineering techniques focus on the software system and not on the data. It still is a hot topic on research conferences where researchers investigate topics such as the

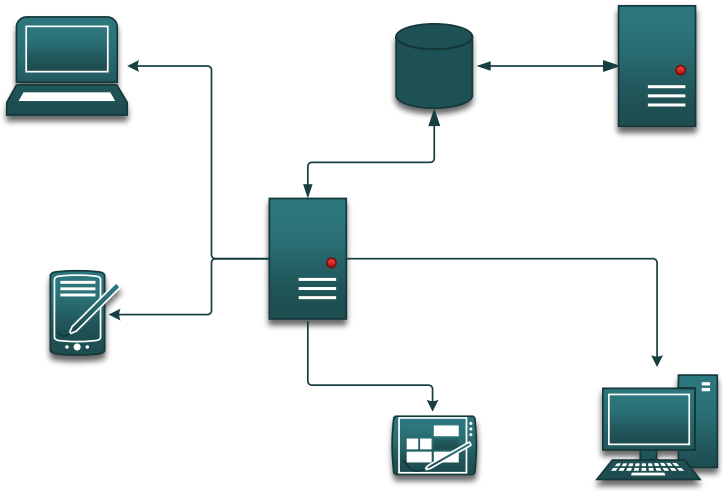


**Figure 2.3.1:** Illustration of a typical data-intensive system.

automation of techniques that answer questions like: "*What does this program do ... ?*" "*Why does this program do ... ?*" or "*How does this program perform ... ?*" [2].

The main objective is to extract information about the static structure and the dynamic behaviour of the code and then give it to some abstract representation [46]. The information extraction is performed by a *parser* or *semantic analyzer*, which is a basic component of most reverse engineering tools. This component performs a syntactic and semantic analysis and constructs an *abstract syntactic tree* (AST) or an *abstract semantic graph* (ASG) representation of the system being analyzed. ASG is the input of many further analyses or transformations. For instance, it is the basic structure for compilers too as an intermediate representation which is then compiled to the target language.

One common analysis technique is called *control flow analysis*. Its purpose is to determine control dependencies among statements (usually nodes of the ASG); that is, to determine on which other statements depends the execution of a certain one. The outcome of the control flow analysis is the *control flow graph* (CFG) or the *control dependence graph* (CFG).

Similar to control flow analysis, *data flow analysis* determines data dependencies among statements. The goal here is to determine which statements depend on a certain variable and its assignments, or vice versa. The outcome of a data flow analysis is the *data dependence graph* (DDG).

An ASG extended with control and data flow dependencies is also known as a *system dependence graph* (SDG), which is the structure used by program slicing techniques, for instance [112]. An SDG is the interprocedural extension of the *program dependence graph* (PDG). It consists of interconnected PDGs (one per procedure in the program) and extends the control and data dependencies with interprocedural dependencies.

Reverse engineering techniques come in two possible types: static or dynamic. *Static analysis*

methods are applicable to systems without executing them. Thus, one can observe the effect on all the potential executions of a system with one such analysis, while with *dynamic analysis* one can observe only specific executions of the system.

### 2.3.2 Database reverse engineering

When the focus of a reverse engineering effort is on the data system or organizational data, it is called *data reverse engineering* (DRE). DRE is defined as "*the use of structured techniques to reconstitute the data assets of an existing system*" [1, 2]. A special kind of DRE is *database reverse engineering* (DBRE) which seeks to recover the conceptual schema that expresses the semantics of the source data structure. It is defined as "*the process through which the logical and conceptual schemas of a legacy database, or of a set of files, are recovered from various information sources such as DDL code, data dictionary contents, database contents, or the source code of application programs that use the database*" [57, 89]. A historical survey on data reverse engineering was published by Davis and Alken in [36].

There are a number of reasons for performing database reverse engineering [57]. These include: knowledge acquisition in system development, system maintenance, system reengineering, system extension, system migration, system integration, quality assessment, data extraction/-conversion, data administration and component reuse.

### 2.3.3 Database component of data-intensive systems

A *database* is a collection of related data [43] that is typically organized according to a *data model*.

Computerized databases are managed by so-called *database management systems* (DBMS). By definition, the "*DBMS is a general-purpose software system that facilitates the processes of defining, constructing, manipulating, and sharing databases among various users and applications*" [43].

The database administrator defines the structure or the *schema* of the database in the DBMS using the *data-definition language* (DDL). Once the schema is defined, data can be stored, queried and manipulated in the DMBS. DBMSs usually provide a *data-manipulation language* (DML) for this purpose. A DML command does not affect the schema of the database but may affect its content. The purpose of these commands is usually only to query data, in which case such a command is simply called as a *query*.

Today the most widespread database management systems are *relational database management systems* (RDBMS), where data items are organized into tables called *relations*. Although the storage structure of the data may be different and more complex, the aim is to present the data to the user in a simple way. The user can express queries in a very high-level language called *structured query language* (SQL). The idea was first introduced by Codd in 1970 [31]. Typical examples of RDBMSs include *MySQL*, *PostgreSQL*, *Oracle Database* and the *Microsoft SQL Server*.

Later, in the mid-80s, *object-oriented database management systems* (OODMS), or simply *object databases* were introduced [121]. Examples of such systems include *ObjectDB* and *Versant Object Database*.

Today, *post-relational databases* (so-called *NoSQL databases*) are widely used by developers. These include fast key-value stores, document-oriented databases and XML databases. Some examples are *MongoDB*, *CouchDB* and *Oracle NoSQL Database*.

## 2.4 State-of-the-art methods in reverse engineering data-intensive systems

In this thesis we deal with architecture recovery and optimization techniques, but there are several other uses of reverse engineering data-intensive systems as well. Some of these relate to our studies, so to have a better view on the main topic of the thesis, now we give an overview of the state-of-the-art in reverse engineering data-intensive systems.

### 2.4.1 Research topics

*Database reverse engineering.* Reverse engineering databases (e.g. modeling data with OMT models [100]) can be regarded as a different field, but it has some things in common with data-intensive systems. Hainaut published a number of papers, books and book chapters on this topic [54–57] and Henrard wrote his PhD thesis on it [63]. Both sum up previous work in this area.

*Program analysis and transformation.* Cleve wrote his PhD thesis [27] on program analysis and transformation for data-intensive system evolution mostly in the area of dependency analysis and migration to support data-intensive system evolution [30]. Cleve et al. used a program dependence graph [29] and dynamic analysis of SQL queries [28, 56] for database reverse engineering purposes.

*Testing database applications.* There have been a number of papers published in different subareas of testing database applications such as test input generation [44, 82, 97], test case generation [22], test coverage measurement [110], regression testing [53] and test suite reduction [71].

*Source Quality.* Static analysis was applied earlier by Pantos et al. for the source code-based quality assessment of ForrásSQL [98]. Brink et al. used the Usage Matrix to calculate metrics for applications with an embedded SQL [20]. Wassermann et al., Gould et al., Brass and Goldberg published papers on the static code checking of embedded SQL queries [19, 52, 117]. Saha et al. analyzed ABAP code for fault localization in [104].

*Program comprehension.* In the area of program comprehension, Van Deursen and Kuipers published a paper on the rapid system understanding of two large COBOL systems [114]. Di Lucca et al. recovered class diagrams from large COBOL applications in [41].

*Impact analysis.* Example topics of impact analysis in database-intensive systems include the analysis of schema changes [51, 87] and supporting test case selection (e.g. in regression testing) [96].

### 2.4.2 Tool support

There are a number of database tools that have database reverse engineering features. Most of them are able to reverse engineer and visualize E/R (Entity-Relationship) diagrams (e.g. MySQL Workbench, JDeveloper, MyEclipse, SchemaSpy). As database reverse engineering is not the main topic of this thesis, we will not deal with these tools, but we note that there is a list of available tools on Wikipedia[1] that compares their key features.

*TOAD* (Tool for Oracle Application Developers) is a software application from Dell (previously from Quest Software) for database administration and development supporting various databases, although it was first developed for Oracle. TOAD has a static analysis tool called *Code Xpert* which is able to identify software defects via rule checking and to calculate software metrics such as the *Halstead Complexity Measure*, *McCabe's Cyclomatic* Complexity and the *Maintainability Index*.

*SonarSource* is a continuous code quality management tool provided by SonarSource S.A. SonarSource has several front-end supporting different languages. It has a front-end for PL/SQL, ABAP and COBOL too, which are typical programming languages for data-intensive systems. Similar to TOAD, SonarSource is able to calculate software product metrics and identify problematic code segments.

## 2.5 Magic applications as data-intensive systems

*Magic* was introduced by Magic Software Enterprises (MSE) in the early 80's. It was an innovative technology to move from code generation to the use of an underlying meta model within an application generator. The resulting application was run on popular operating systems including DOS and UNIX. Since then newer versions of Magic have been released called *eDeveloper*, *uniPaaS* and *Magic xpa*. Recent versions support novel technologies including RIA (Rich Internet Applications), SOA (Service Oriented Architecture) and mobile development.

### 2.5.1 Generations of programming languages

Programming languages are usually grouped into five levels or 'generations' [69]. With just binary numbers, the machine languages are the *first generation languages* (1GLs). Lower level programming languages (e.g. assembly) are the *second generation languages* (2GLs) and currently popular

[1] http://en.wikipedia.org/wiki/Comparison_of_database_tools

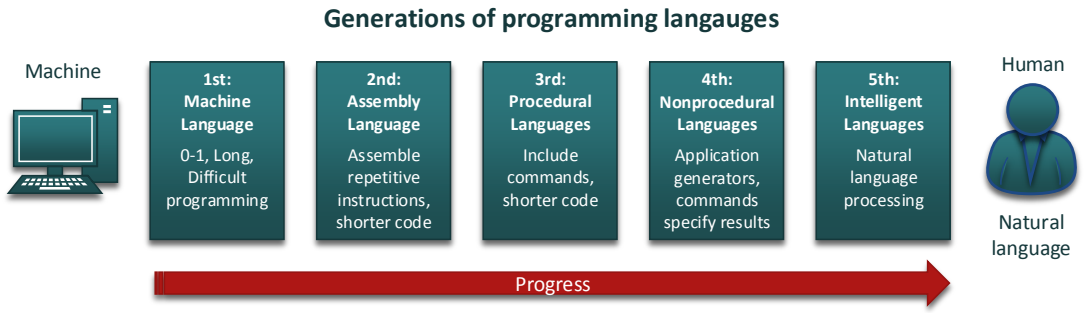



**Figure 2.5.1:** Generations of programming languages.

procedural and object-oriented languages are the *third generation languages* (3GLs). The higher level languages are all closer to human thinking and spoken languages. Using *fourth generation languages* (4GLs) a programmer does not need to write source code, but can program an application at a higher level of abstraction, usually with the help of an application development environment. Lastly, *fifth generation languages* (5GLs) would involve a computer which responds directly to spoken or written instructions, like English language commands. An illustration of the various generations of programming languages can be seen in Figure 2.5.1.

4GLs are also referred to as *very high level languages* (VLLs) [83]. A developer who develops an application in such a language does not need to write 'source code', but he can program his application at a higher level of abstraction and higher statement level, usually with the help of an application development environment. These languages were introduced and widely used in the mid-1980s. At that time many 4GLs were available (such as Oracle, FOCUS, RAMIS II and DBASE IV), but today most of the information systems are developed in third generation languages. However, large systems developed earlier in a 4GL are still evolving and there is still a continuous need for *rapid application development and deployment* (RADD) tools, which are usually based on these higher level languages.

### 2.5.2 Magic as a 4th generation language

The heart of a Magic application is the *Magic Runtime Engine* (MRE), which allows one to run the same application on different operating systems. When one develops an application in Magic, one actually programs the MRE using the unique meta model language of Magic, which is – at a higher level of abstraction – closer to business logic. This meta model is what makes the development in Magic unique and what really makes Magic a RADD tool.

The architecture of Magic applications follows the design principles of typical data-intensive applications. That is, Magic was invented to develop business applications for data manipulating and reporting, so it comes with many GUI screens and report editors. Hence the most important elements of its meta model language are the various entity types of business logic, namely

the *data tables*. A table has its columns and a number of *programs* (consisting of subtasks) that manipulate it. The programs or *tasks* are linked to *forms, menus, help screens* and they may also implement business logic using logic statements e.g. for selecting variables (virtual variables or table columns), updating variables, conditional statements and expressions.

The meta model of a Magic application serves as a 'source code' that can be analyzed for reverse engineering purposes. With this model we can describe the main characteristics of an application and we can locate potential coding problems or structures which may contain bugs or bad design within them.

## 2.6 The Columbus framework

The Columbus reverse engineering framework was developed by FrontEndART Ltd., the University of Szeged and the Software Technology Laboratory of Nokia Research Center. It was developed so as to be able to analyze large C/C++ projects and extract facts from them; e.g. calculate product metrics for them. The framework was designed in such a way as to combine a number of reverse engineering tasks and provide a common interface for them (see Figure2.6.1).

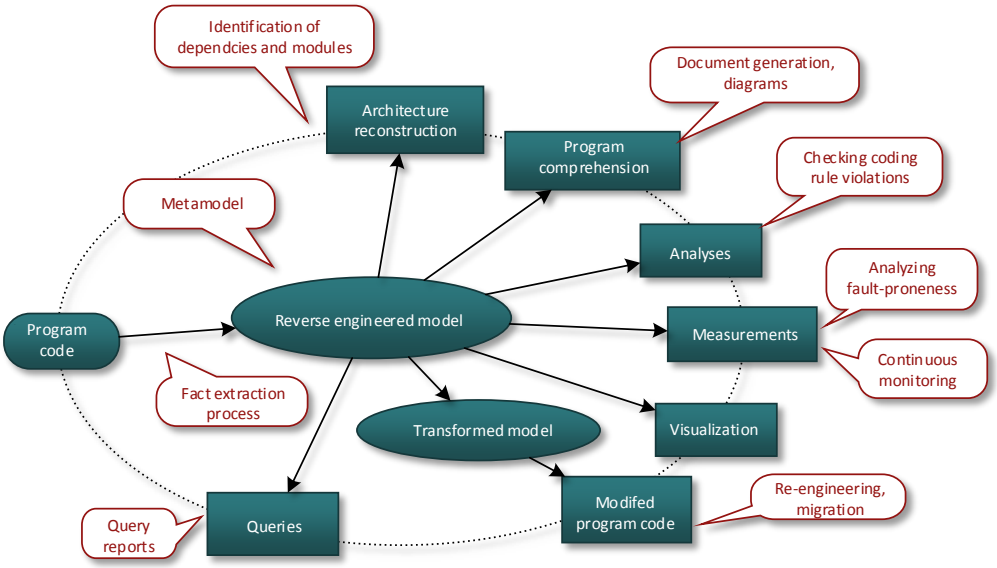



**Figure 2.6.1:** An overview of the Columbus methodology.

Since its introduction [47], it has been extended to support additional languages such as Java, C#, Python, ForrásSQL, PL/SQL and Transact SQL.

### 2.6.1 The Columbus framework adapted to data-intensive systems

Figure 2.6.2 shows the SQL package of the Columbus framework. *SQLAn* denotes the SQL Analyzer which takes SQL files as its input and parses the DDL and DML commands in those

files. SQLAn constructs the ASG which will then be the input of the different extractors (tools for calculating metrics, finding code clones, etc.). *SQL2Metrics* calculates software metrics such as size and complexity metrics. *SQLCheckGEN* looks for coding rule violations. *SQLDuplicate-CodeFinder* identifies code clones. *SQL2Arch* identifies main components in the architecture of the system being analyzed, and dependencies between them.

The SQL front-end of Columbus is able to analyze Oracle SQL, Oracle PL/SQL, MS SQL and Transact SQL dialects of SQL. Through its API it can be used to analyze embedded SQL statements as well. To support this, it was designed to be able to parse SQL statements with missing code fragments too. Most of this front-end was developed during this study. Chapter 3 discusses how we implemented this method and used it to investigate safe dependency relations by analyzing embedded SQL queries in large legacy data-intensive systems developed in ForrásSQL. Chapter 4 uses the same technique to recover the architecture of a large Oracle PL/SQL system.

In Chapter 5, we adapt the Columbus technology to a specific data-intensive environment called the Magic 4th generation programming language. Chapter 6 relies on this adaptation where we utilize the adapted technology for quality assurance purposes and for supporting the migration of Magic applications.

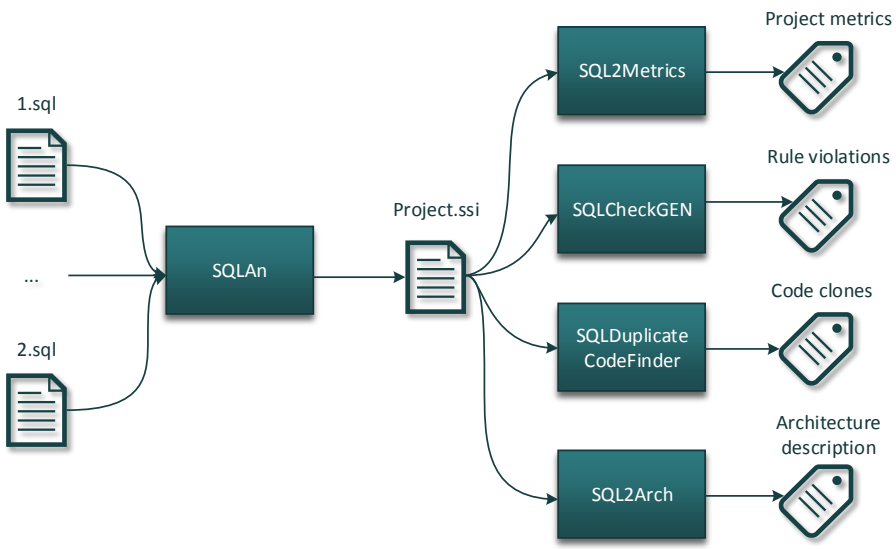



**Figure 2.6.2:** SQL analyzers of the Columbus framework.

Lastly, in chapters 7 and 8, we will use the GrammaTech CodeSurfer[2] technology and GNU GCC[3] as the main technologies, and we will also utilize the C++ front-end of Columbus for validation tasks.

[2]http://www.grammatech.com/research/technologies/codesurfer

[3]http://gcc.gnu.org/

# Part II

# Architecture recovery of legacy data-intensive systems

*"It is good to have an end to journey toward; but it is the journey that matters, in the end."*

Ernest Hemingway

# 3

# Extracting architectural dependencies in data-intensive systems

Extracting dependencies between source elements is a key step towards recovering the design of a legacy system. Here, we introduce a method for computing dependencies (CRUD dependencies) among source elements of data-intensive systems. These are the so-called CRUD dependencies between procedures on the software side and data tables on the database side. We will compare these relations with the previously published SEA/SEB [70] relations.

## 3.1 Overview

Analyzing program dependencies is helpful in many different phases of the software development process, such as in architecture reconstruction, reverse engineering, regression test case selection, and change impact analysis. A certain class of automated methods for computing program dependencies uses static analysis, usually by employing call, control flow, and data flow information. The granularity of these methods ranges from the basic block level through procedure[1] and class

[1] Here the term *procedure* means any general procedure. A stored procedure of a database system will be called a *stored procedure*.

levels to module and system levels, each granularity level having potential application areas.

Static analysis has its own pitfalls, though: in many situations, we have to decide if want a safe result (meaning that no important dependency is missed) or a concise one (focusing on certain dependencies only). On the one hand, a high-level analysis can be made safe by including all the possible dependencies that might arise during the executions of the code being examined, but this might result in many fake dependencies, so what we get is not very useful. On the other hand, a detailed low-level static analysis might find just the relevant dependencies, but have high computation costs and make it unfeasible on real-size systems. It is also possible that safety is sacrificed to make a method faster, and heuristics are used instead of detailed computations that might miss some rare, but important, dependencies.

In addition, large information systems are usually heterogeneous in the sense that more than one programming language and technology are used in them. The most common situation is that relational databases are used behind procedural programs. In this case dependencies can arise through the database, which are usually not detected by using static analysis tools. It is possible to extract some SQL instructions that access the databases, but a static analysis is usually not enough to recover all of them exactly (consider, for example, SQL query-strings assembled at execution time).

Here we propose two methods that compute dependencies at the procedure level, are applicable on real-size systems, and when properly applied, can provide safe results. The first method is based on Static Execute After/Before relations (*SEA*/*SEB* [70]), which uses the static call- and control flow graph and a lightweight interprocedural analysis of the system. The second method analyses the embedded SQL statements of the code and the database schemas to discover dependencies via database access. It computes *CRUD*-based dependencies [20, 114] between the SQL statements, and propagates them to the procedure level.

We applied these methods on program code obtained from one of our industrial partners. Here, architecture reconstruction and test coverage evaluation were performed. The main contributions of the chapter are:

- The application of a *CRUD*-based Usage Matrix for dependency analysis among program elements, which we think is a novelty (Usage Matrix is a common way of recovering dependencies between client applications and database tables);

- Adapting *SEA* relations to recover dependencies in data-intensive systems;

- Applying a combination of these two methods and empirically evaluating them on real-life data.

## 3.2 Related work

The System Dependence Graph (SDG), which describes traditional software dependencies (e. g. control and data dependencies) among different source elements of a system, is a common tool for software dependency analysis [66]. The construction of the SDG has been a real challenge for the slicing community [14, 112] for a long time, because – depending on the source code being analyzed – one has to tackle a range of problems like context sensitivity, pointers, and threads. This is why many studies have focused on software dependencies. However, only a few of them concentrated on dependencies arising via database access like Nyary and Sneed in [95].

Another goal of identifying these kind of dependencies is the impact analysis of schema changes. Maule et al. published a technique [87] that is based on a k-CFA algorithm for extracting SQL queries. They use the SDG to identify the impact of relational database schema changes upon object-oriented applications. Gardikiotis and Malevris use a Database Application specific version of the PDG (Program Dependence Graph) to perform a static program slicing analysis [51]. They extended the PDG with special pseudo nodes for SQL statements and their relations to the database schema.

Another potential application of the identified dependencies is test case selection, and testing database applications in general. Haraty et al. introduced a regression test selection method for database applications [60, 61]. Besides using traditional control and data dependencies, they introduced the notion of a dataflow analysis method that uses database interactions and it is based on identifying the usage of table columns. This idea is similar to our *CRUD* relation idea. Their method works on SQL's PSM extension (Persistent Stored Modules, e.g. Pl/SQL). Thus they do not need to deal with the problems of dynamically concatenated SQL queries, which is important if one would like to generalize this technique to other procedural or object-oriented languages.

A *CRUD* or Usage Matrix is also useful for system understanding and quality assessment purposes. Van Deursen and Kuipers used it in their study [114] to identify conspicuous table usage in COBOL applications. For instance, tables employed just for retrieving data and top used tables. Brink et al. utilized the Usage Matrix to calculate metrics for applications with embedded SQL [20].

The dependencies computed from the source code can be applied to support data reverse engineering too. Henrard and Hainaut published papers in this area [29, 64] and evaluated different dependency analysis techniques (variable dependency graph, program slicing) via database access positions applied to dependency elicitation.

Most of the above-mentioned methods rely heavily on the extraction of SQL statements extracted from the source code. Finding solutions for this problem has also been a big challenge for researchers and many approaches have been proposed for static and dynamic analysis techniques as well. Most of the static methods apply string analysis techniques [26, 52], as we do too, but

ours does not implement control or data flow analysis in order to keep the extraction of query strings fast and scalable for large systems. Hainaut and Cleve not long ago published two papers [28, 56] that describe different techniques and also some applications in this area. Cordy published a number of papers [33, 37] describing the TXL language and its applications including embedded SQL parsing. Their TXL language and agile parsing technology could also be used to extract embedded SQL statements and/or to parse incomplete SQL statements. However, they admit that applying their agile technology would require more resources than 'normal' parsers.

## 3.3 Methods

### 3.3.1 Overview

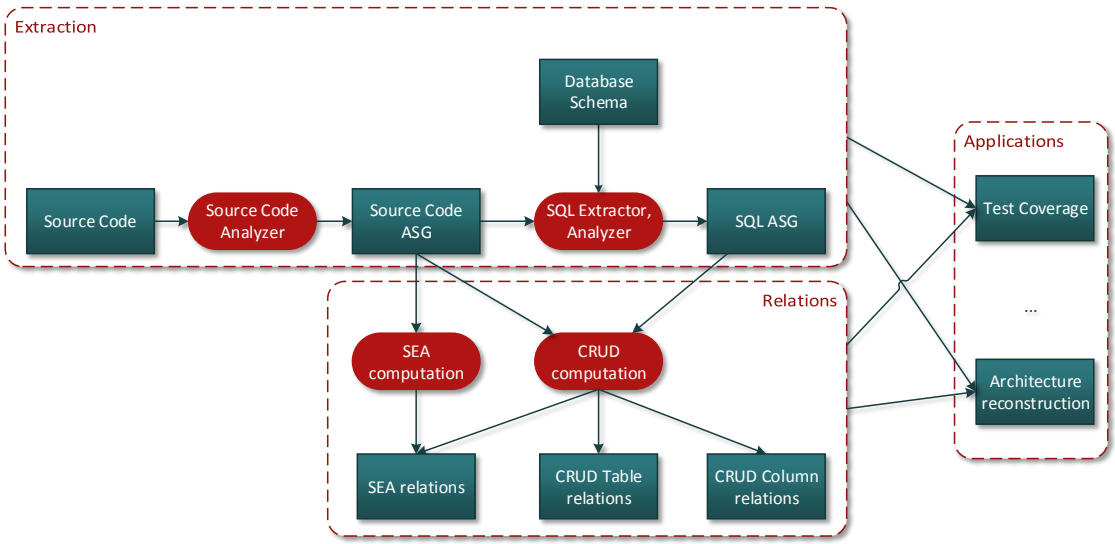



**Figure 3.3.1: Methods overview.** The main steps for extracting source information (analyzing the source code, extracting and analyzing SQL instructions) and computing *SEA* and *CRUD* relations. The information gathered can be used for a variety of purposes.

#### Motivation

The main idea behind our methods is that in data-intensive systems many dependencies among the program elements arise via database accesses, which are usually not recovered by traditional dependency algorithms.

For instance, suppose that one procedure (called $f$) inserts data into a table $T$ and later during execution another procedure (called $g$) reads data from table $T$ to execute a complex algorithm which takes the same data as its input parameter. Obviously, when we modify the first procedure and perform a change impact analysis, we must examine the second procedure too. This is the only

way to ensure that our algorithm implemented in $g$ still works as expected. Sometimes traditional algorithms are able to determine whether $g$ depends on $f$ ($f \rightarrow g$), but there are many situations where $f \rightarrow g$ will not be detected. For example, a simple call or a control flow analysis will be able to recover the $f \rightarrow g$ relation if there is a chain of call or control flow dependencies from $f$ to $g$. However, suppose that our application is multi-threaded and $f$ and $g$ run in different threads; furthermore suppose $f$ never calls $g$ directly or transitively. In this case, the relation between the two procedures will appear neither in the call graph nor the control flow graph of the system.

In another common situation, let us suppose that the above-mentioned $f$ and $g$ are called in a procedure body one after the other; but $g$ is never called directly or indirectly from $f$. The traditional call graph-based methods will not recover their dependency relation either. Therefore, it would not be safe to perform a change impact analysis in this situation.

#### Methods

Here, we propose two new methods for recovering dependencies in data-intensive systems that complement traditional software dependencies. One is a previously published algorithm that determines Static Execute After or Static Execute Before (*SEA*/*SEB*) [70] relations and the other is based on the Usage Matrix (*CRUD* for short) [20, 114].

Dependencies computed by both *SEA*/*SEB* and *CRUD* describe special relations which are not recognizable by traditional software dependencies, and when properly applied they, can provide safe results at a reasonable cost, even for large data-intensive systems.

An overview of our analysis system is given in Figure 3.3.1. Both algorithms take their input from the Abstract Semantic Graph (ASG) extracted from the source code, but in the case of the Usage Matrix we also need to extract the SQL instructions embedded in the source code and analyze them separately. The outputs of the algorithms are a list of computed relation pairs: procedure-to-procedure, procedure-to-table, table-to-table, and column-to-column pairs. These relations can be used in application areas like architecture reconstruction and test coverage measurement.

### 3.3.2 *CRUD* relations

After a successful extraction and analysis of embedded SQL statements, it is possible to determine the relations between the program statements and between the accessed tables or columns. To achieve this, an analysis of the database schema is required along with an analysis of the extracted SQL statements. Should the schema source code not be available, it can be obtained from the database management system. After retrieving this additional piece of information, the computed relations can be used to construct the Usage Matrix of the system. This matrix describes the usage relations among the procedures and among the tables of the system. In our case, the relations are the basic *CRUD* (*Create*, *Retrieve*, *Update*, *Delete*) operations. Namely,

- `INSERT` statements *create* data in their target table;
- A typical way of *retrieving* data from a table is a `SELECT` statement, but any other statements (even an `INSERT` or `DELETE` statement) can retrieve data from tables;
- `UPDATE` statements *update* data in their target table;
- `DELETE` statements *delete* data from their target table.

A typical *CRUD* matrix can be seen in Figure 3.3.1.

| | `Customers` | `Rentals` | `Cars` |
|---|---|---|---|
| `NewCustomer` | C | | |
| `CarRental` | R | C | R |
| `AddressModification` | RU | | |
| `CarCrash` | | | D |

**Table 3.3.1: A typical *CRUD* matrix.** `NewCustomer` inserts data into the `Customers` table. `CarRental` reads data from the Customers and Cars tables, and inserts data into the Rentals table. `AddressModification` retrieves and updates the `Rentals`, and `CarCrash` deletes data from the `Cars` table.

The same information can also be displayed in a graph called a *Usage Graph*, which shows the different kinds of relations among the procedures and tables. A typical Usage Graph can be seen in Figure 3.3.2. Solid and dashed arrows with different directions represent different relation types.

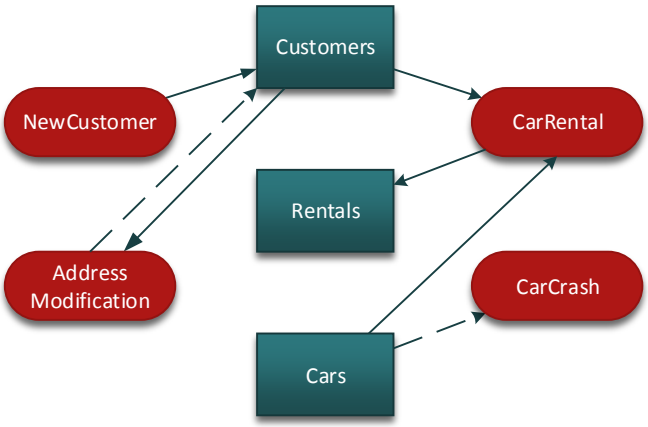



**Figure 3.3.2: A typical *CRUD* graph.** The tables are in the centre of the figure and the procedures are in the oval shapes on both sides of the figure. The arrows from the tables to the procedures represent data retrieve operations, while the arrows from the procedures to the tables represent updates. The dashed arrows from tables to procedures represent delete operations and the dashed arrows from procedures to the tables represent create operations.

### Relations between procedures via table access

The computed *CRUD* matrix can be used to determine relations at different granulation levels. We say that procedures are related by $CRUD^T_{PP}$ if they share at least one accessed table.

Formally, for $f$ and $g$ procedures $(f, g) \in CRUD^T_{PP}$ if and only if $\exists T$ table which is accessed by both $f$ and $g$.

This sort of relation has no direction, hence it is symmetrical: $(f, g) \in CRUD^T_{PP} \Leftrightarrow (g, f) \in CRUD^T_{PP}$. It is not necessarily transitive because it may happen that $(f, g) \in CRUD^T_{PP}$ as they access only $T_i$ and $(g, h) \in CRUD^T_{PP}$ as they access only $T_j$, but $(f, h) \notin CRUD^T_{PP}$ as they do not share an accessed table; that is, $i \neq j$.

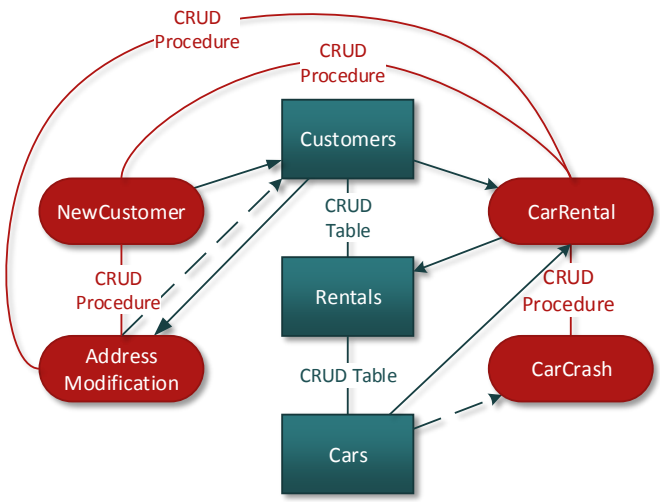



**Figure 3.3.3:** *CRUD* relations between procedures and between tables.

A visual representation of these relations can be seen in Figure 3.3.3.

This relation can be computed in a safe way even though the SQL extraction may not recover the exact SQL statements for each embedded SQL string in the source code. When an extracted SQL instruction contains an unrecognized fragment in the place of a table identifier, a conservative approach inserts the procedure into the Usage Matrix as it would be related to all the tables of the database. For example, in the case of a procedure containing an SQL command like the one in Figure 3.3.4, a conservative approach relates the procedure to each procedure that accesses any table in the system.

Here, notice that the most common reason for unparsable SQL strings is that they are sometimes constructed with code fragments in a position where it makes the full statement syntactically incorrect. (An example can be seen in Figure 3.3.5.) However, there are several other reasons for it, which we will elaborate on in Section 3.5.4.

```
SELECT firstname, lastname
  FROM @@customer_table@@
  WHERE firstname
    LIKE('%@@name@@%');
```

**Figure 3.3.4:** Sample code of an extracted SQL command where the table name is determined by a variable.

```
SELECT firstname, lastname
  FROM @@customer_table@@
  WHERE firstname
    LIKE('%@@name@@%') @@condition@@;
```

**Figure 3.3.5: Example of an extracted SQL command which is syntactically incorrect.** The `@@condition@@` fragment may be an additional condition of the executed SQL, which may become a syntactically correct condition during execution.

#### Relations between procedures via column access

*CRUD* operations can be lowered to the database column level by considering exact column access instead of a table access. The idea behind this low-level consideration is that even if a procedure modifies a table and another one reads data from the same table, there is no data dependency between them unless they modify and read the same record(s) of the table. However, determining the accessed record(s) of the table is not possible via a static analysis. Nevertheless, it is still possible to find the accessed columns of the table, and the scope of the dependency relation can be narrowed. Haraty et al. suggest this level of granularity in [60].

We shall define *CRUD* operations for the relations between procedures via column access like so:

- `INSERT` statements *create* data in all the columns of their target table;
- `SELECT` and those statements which do not modify data, *retrieve* data from columns which are accessed only for reading. The asterisk in a `SELECT` means a data retrieval for all the columns of the corresponding tables;
- `UPDATE` statements *update* specified columns of their target table;
- `DELETE` statements *delete* data from all the columns of their targeted table.

Formally, for $f$ and $g$ procedures $(f, g) \in CRUD^{C}_{PP}$ if and only if $\exists C$ column which is accessed by $f$ and $g$.

$CRUD^{C}_{PP}$ can be also computed via a conservative approach. However, in procedural languages where SQL commands are constructed in a dynamic way, the noise of such a conservative method would result in too many false positive relations.

#### Relations between tables or between columns

The Usage Matrix can be used to determine relations among tables ($CRUD_{TT}$) or among columns ($CRUD_{CC}$) of the system too. This approach is based on database and program reverse engineering [64]. The idea behind it is that there are many kinds of dependencies among table columns which are not recognizable by the traditional database reverse engineering techniques that only analyze the database of a system. Certain kinds of dependencies require taking into account the embedded queries in the source code as well. Columns or tables accessed by the same procedures are related to each other and these relations must be considered when carrying out data reverse engineering (i.e. modularization).

Similar to $CRUD^T_{PP}$, the $CRUD_{TT}$ and $CRUD_{CC}$ relations can be defined as follows: for $t$ and $q$ tables (columns) $(t, q) \in CRUD_{TT}$ ($CRUD_{CC}$) if and only if $\exists P$ procedure which accesses both $t$ and $q$.

These kinds of dependencies can be also recovered by utilizing the Usage Matrix. Note that a conservative implementation should be applied with care as an unrecognized code fragment will mean that all the tables or all of the columns in the system will always be related to each other.

### 3.3.3 *SEA*/*SEB* relations

#### *SEA*/*SEB* in general

The Static Execute After/Before dependencies and an algorithm for their computation were previously published in [70]. According to their definition $(f, g) \in SEA$ if and only if it is possible that any part of $g$ is executed after any part of $f$ in some execution of the program. Similarly, $(f, g) \in SEB$ if and only if it is possible that any part of $g$ is executed before any part of $f$. The two relations are inverses of each other, so $(f, g) \in SEA \Leftrightarrow (g, f) \in SEB$.

*SEA*/*SEB* relations involving $(f, g)$ can be formally defined as follows:

$$SEA = CALL \cup RET \cup SEQ[\cup ID],$$

where $(f, g) \in CALL \Leftrightarrow f$ (transitively) calls $g$, $(f, g) \in RET \Leftrightarrow f$ (transitively) returns into $g$, $(f, g) \in SEQ \Leftrightarrow \exists h : h$ (transitively) calls $f$ followed by $g$, and the second call-site is flow-reachable from the first one. Lastly, $(f, g) \in ID \Leftrightarrow f = g$. *SEB* can be formally defined as the inverse of *SEA*.

#### *SEA*/*SEB* for procedures

The reason why *SEA* and *SEB* describe safer relations among procedures compared to simple call relations is due to *SEQ* relations. Thanks to this set, *SEA*/*SEB* will discover those $(f, g)$ relations among procedures where $f$ is called followed by $g$, but $g$ is not called (not even transitively) by $f$.

In order to compute *SEA*/*SEB*, the traditional call graph is not sufficient since the order of call-sites within a procedure body is required to determine the above-mentioned *SEQ* set of relations. To compute *SEA*/*SEB*, the control flow graph (CFG for short) of the system is required. Once we have the CFG, we can compute the dependencies with the help of a language independent algorithm.

An extended example of the above *CRUD* example (Figure 3.3.3) can be seen in Figure 3.3.6 with additional *SEA*/*SEB* dependencies.

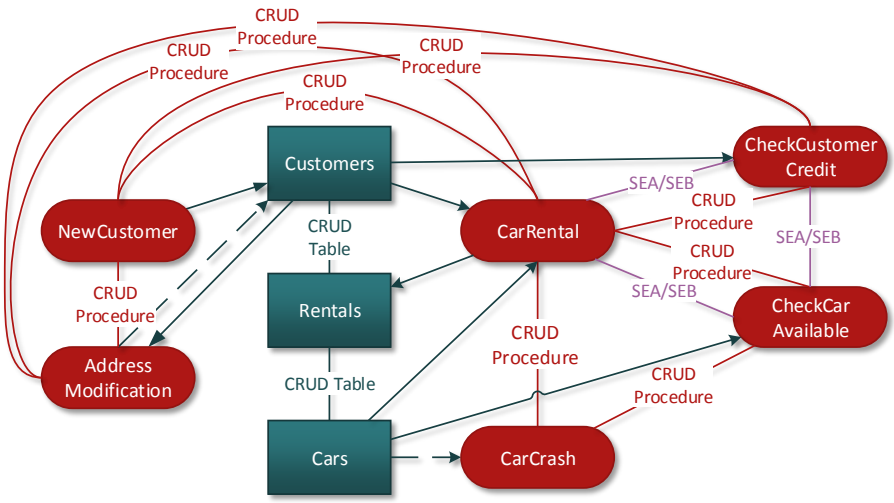



**Figure 3.3.6:** *CRUD* and *SEA*/*SEB* relations between procedures and between tables.

### Directed *SEA*-*CRUD* relations

We mentioned previously that *CRUD* relations are not directed. The reason for this is that it makes no sense to distinguish between two procedures reading from the same table. It is the same for procedures when updating, inserting, or deleting data from the same table, but it is slightly different in the case of two procedures where one of them modifies the table and the other reads data from the same table. Simply by using *CRUD* relations for procedures over tables, we cannot determine the execution order of the procedures; so it is not possible to determine whether the procedure reading data from the table reads it after or before the other procedure modifying the same table.

By combining *SEA* and *CRUD*, it becomes possible to compute which procedure accessed the same table before the other one. We will combine *SEA* and *CRUD* relations in a simple way; by computing different set operations on the two relations. Doing this, we can use the union as a combined conservative approach, and the intersection as a way to see the stronger relations, say. This information can be used to evaluate special relations among procedures not otherwise identifiable.

## 3.4 SQL EXTRACTION

Although the extraction of SQL commands is not part of the dependency analysis algorithm itself, it has a great influence on the effectiveness of the two methods proposed here. In general, if we can achieve a better extraction of SQL commands, the algorithms will be more precise and more effective.

In many programming languages the SQL queries are sent to the database server by specific procedures which take the SQL commands as a string parameter. These procedures are internal procedures or library procedures accessible via an API. For these languages it is common programming practice to prepare SQL commands by simple string concatenations and to execute the prepared SQL string by giving it as a parameter to one of the internal or API procedures. It was previously shown that for these languages the mere examination of the source code should provide enough information to determine the most significant fragments of the SQL queries. These fragments are sufficient to parse embedded SQL statements and determine the relations using them via a careful static analysis [20].

```
name = readString();
sql = "SELECT firstname, lastname " +
   "FROM customers " +
   "WHERE firstname " +
   "LIKE('%" + name + "%')";
executeQuery(sql);
```

**Figure 3.4.1: Example of an embedded SQL query.** The query string is concatenated on the second line using a variable in the `WHERE` clause of the SQL query.

However, the code fragments of the SQL query may be defined at a certain distance from the place they are used in the source code. In special cases this may result in a situation where determining the exact syntax of SQL commands is unfeasible via static analysis. For instance, when the SQL statement which will be executed is a concatenation of strings where one or more strings are read from the standard input (Figure 3.4.1). In this case the executed SQL instruction could only be captured via dynamic analysis techniques, but it would produce results for only a certain set of executions of the application. Despite this, the SQL command coming from the user input will probably not be the same as that for the different executions of the application. In order to capture all the possible query strings, one execution is not enough and one must execute the application as many times as the user input may vary. This is usually unfeasible for a large and complex system.

We implemented a static approach for extracting and analyzing embedded SQL commands from the source code. We should mention here that the system on which we evaluated the proposed methods was written in a special procedural language called ForrásSQL. The programming

style of this language makes the whole system strongly database dependent and it makes the use of SQL queries very common in the language. The SQL statements to be executed are embedded as strings sent to specific library procedures and their results will be stored in given variables. This method is actually the same as that for procedural languages where embedded queries are sent to the database via libraries like JDBC or ODBC. This makes our method quite general and applicable for other languages too.

The approach we implemented is based on the simple idea of substituting the unrecognized query fragments in a string concatenation with special substrings. For instance, it is possible to simply replace the `name` variable with a string "@@name@@". If the SQL parser is able to handle this string as an identifier, then the received query string will be a syntactically correct SQL command (see Figure 3.4.2). Of course, for a parser with an SQL grammar it will not mean the same semantically, but the main characteristics of the SQL command (e.g the kind of statement, the tables and rows used in it) will remain the same.

```
SELECT firstname, lastname
  FROM customers
  WHERE firstname
    LIKE('%@@name@@%');
```

**Figure 3.4.2: Example of an extracted SQL command constructed from a string concatenation with a variable.** The embedded statement can be seen in Figure 3.4.1. The variable `name` is replaced by the value "`@@name@@`".

With this simple idea we only need to locate the library procedures sending SQL commands to the database in order to perform the string concatenation, and the above-mentioned substitution of variable, procedure name and other source elements. Whenever the constructed string is syntactically correct, it will have the same key characteristics as the executed SQL command.

Developers usually like to prepare statements as close to their execution place as possible and they prefer to keep SQL keywords in separate string literals. In most cases, it is possible to substitute the variables with their last defined values within the same control block. In other cases the variable can be replaced with the variable name.

With this technique the percentage of syntactically correct SQL statements found for the system analyzed was as high as 85% and this enhancement still did not require data flow analysis of the whole source code.

## 3.5 Evaluation

We performed our measurements on the code supplied by one of our industrial partners. The IT architecture of this company is heterogeneous and it is made up of many different technologies,

with a central role of a proprietary technology provided by another local software company. Most of the core systems are built upon this technology, which is an integrated administrative and management system made up of modules (subsystems) using Windows-based user interfaces and MS SQL databases. The modules contain programs, and the programs are aggregates of procedures. The language is procedural, and its syntax is similar to the Pascal programming language. SQL statements are embedded as strings sent to specific library procedures.

In previous projects, we implemented a source code analyzer for this language (including an analyzer for the embedded SQL language), and many different supporting tools (some of which are language independent). We implemented our methods in this environment and applied them on the working module of a core system.

| Metric name | Value |
|---|---:|
| (Logical) Lines | 315, 078 |
| Programs | 776 |
| Procedures | 2, 936 |
| Triggered procedures | 41, 479 |
| Embedded SQL statements | 7, 434 |
| Tables | 317 |
| Temporary tables | 641 |

**Table 3.5.1:** Metrics representing the key characteristics of the system.

In Table 3.5.1, metrics are presented to highlight some of the characteristics of the system being analyzed. We identified 7, 434 embedded SQL strings (based on the specific SQL library procedure calls) and we successfully analyzed 6, 499 SQL statements, which is 87% of all the embedded SQL strings. Here, we differentiate between 'normal' and *triggered* procedures. Triggered procedures are assigned to database schemas, tables, and columns. They are never called directly; instead an automated mechanism calls them at runtime whenever a table or column of the schema is used (e. g. for writing or reading). Note that most of the 41, 497 triggered procedures are empty and including them in the measurements adds only 10% more call edges to the call graph (when these triggered procedures call 'normal' procedures). Hence, in the following evaluation of the proposed methods, we focused on 'normal' procedures.

### 3.5.1 Quantitative analysis

In Table 3.5.2 basic statistical indicators of the identified relations are presented. The relations are (number of) call graph edges and *SEA*/*SEB* relations, while $CRUD^{T}_{PP}$ represents the conservative implementation of *CRUD* relations among procedures, and $CRUD^{*}$ relations representing the variants where only certain dependencies are considered (dependencies which arise only because of recognized code fragments). The first column shows the total number of computed

dependencies for each relation type. The second column shows the maximum number of dependencies of a procedure, and the last two columns are the average and the deviation of dependencies per procedure.

The results in Table 3.5.2 make it clear that there are many relations among procedures via table access which cannot be found using a call graph only. Furthermore, the differences between ${CRUD^*}^T_{PP}$ and $CRUD^T_{PP}$ tell us that the price of a conservative analysis can be quite high.

The average values reveal that when taking into account the *SEA*/*SEB* relations, a procedure may depend on about 8% of the whole module on average. In a similar way, with $CRUD^T_{PP}$ relations, a procedure might be related to about 6% of the other procedures via database access.

In addition to the different relations among procedures, we also examined relations among procedures and tables. The procedures of the system accessed 1.81% of tables on average and 25 was the highest number of accessed tables by the same procedure. This measurement was performed by taking into account only those relations that were not influenced by unrecognized code fragments of SQL instructions.

| | sum | max. | avg. | dev. |
|---|---|---|---|---|
| *CGedges* | 18,595 | 764 | 6.33 | 23.75 |
| *SEA*/*SEB* | 727,303 | 2,347 | 247.72 | 361.86 |
| $CRUD^T_{PP}$ | 576,095 | 1,066 | 192.22 | 338.87 |
| ${CRUD^*}^T_{PP}$ | 156,527 | 615 | 53.31 | 120.61 |
| ${CRUD^*}_{CC}$ | 1,024,180 | 2,358 | 99.69 | 203.19 |
| ${CRUD^*}_{TT}$ | 11,817 | 330 | 12.23 | 24.60 |

**Table 3.5.2:** Basic statistics for different relations.

The *SEA*/*SEB* and *CRUD* relations have a different basis. Thus, they are comparable as different sets, and one can check whether any of them contains the other, or they are distinct. In Table 3.5.3, the difference, the intersection, and the union of the $CRUD^T_{PP}$ and *SEA*/*SEB* relations are given. The columns represent the same statistical indicators that were used in Table 3.5.2. It tells us that *CRUD* and *SEA*/*SEB* are different kinds of relations as they have only a few dependencies in common. This means that in the program the two kinds of dataflow (via the normal control flow and via databases) are well separated. Thus, neither of these two relations seems to be better than the other; they simply complement each other. However, the intersection of these two types of relations is also interesting. It will determine those dependencies that arise via database access but are potentially used by the same execution (the same operative task). We think that this combined dependency is stronger than any of its components alone, and it can be used to prioritize procedures (e. g., for testing applications) as it will mark only a small fraction of the original relations.

| Relations | sum | max. | avg | dev. |
|---|---|---|---|---|
| $CRUD^T_{PP} \setminus SEA/SEB$ | 542, 078 | 1, 065 | 184.63 | 97.75 |
| $SEA/SEB \setminus CRUD^T_{PP}$ | 691, 782 | 2, 346 | 235.62 | 107.54 |
| $CRUD^T_{PP} \cap SEA/SEB$ | 36, 037 | 369 | 12.27 | 10.00 |
| $CRUD^T_{PP} \cup SEA/SEB$ | 1, 267, 361 | 2, 347 | 431.66 | 162.41 |
| $CRUDRW^T_{PP} \cap SEA$ | 29, 532 | 342 | 10.06 | 32.08 |

**Table 3.5.3:** Difference, intersection, and union of *SEA*/*SEB* and *CRUD* relations between procedures.

In Table 3.5.3, we present data for $CRUDRW^T_{PP}$ relations. This is a special type of the combined relations discussed in Section 3.3.3, where a read operation in a procedure is followed by a write in another, or a write followed by a read. $CRUDRW^T_{PP} \cap SEA$ approximates the database-based dataflow relation of the program. We measured this kind of relation without taking into account those relations that were influenced by unrecognized code fragments of SQL instructions. We found that the rougher relations ($CRUD^T_{PP}$, $SEA/SEB$), and their combination contained 20% to 70% more edges than the finer ones (with $CRUDRW^T_{PP}$, $SEA$).

### 3.5.2 Qualitative analysis

As a qualitative analysis, we manually inspected the computed relations by selecting random samples. We focused on special types of dependencies (e. g. $CRUD^T_{PP} \cap SEA$, which describes dataflow between two procedures) and we inspected the source code to see whether the chosen dependency actually described a real dependency between the two items.

Most of the evaluated $CRUD^T_{PP} \cap SEA$ relations were real dependencies among procedures. In some cases, we found that the developers used temporary tables to pass data from one procedure to another. It is common practice in table manipulation to select data from one table, place it into a temporary table, and later insert the retrieved data from a temporary into a different, persistent one. These relations can be easily found when the procedures working with the same temporary tables are inside the same program.[2] However, it may also happen that procedures in different programs and in different source files of the system have these types of dependencies. An example of a $CRUD^T_{PP} \cap SEA$ dependency among procedures via a temporary table can be seen in Figure 3.5.1.

Another example concerns the implementation of *menus* in this framework, which are intensively used so as to let the user access different features. One menu entry executes one procedure of the system. During manual inspection, we found procedures which were in $CRUD^T_{PP}$ relations, but they implemented functionalities of different menu entries. This means that $f$ and $g$ proce-

[2]In this framework, *program* is a higher level compilation unit. It contains procedures and it is usually in a separate source file.

dures are in $CRUD^T_{PP}$ relations, and $f$ is transitively called from the $M_1$ menu entry while $g$ is transitively called from the $M_2$ menu entry, but $f$ is not called (transitively) from the $M_2$ entry, and $g$ is not called (transitively) from the $M_1$ entry. Finding the relations among these procedures is especially important in this large system where there are around 200 menus that use 2, 936 procedures.

```
procedure procA:
  sql = "SELECT DISTINCT * " +
     "INTO #temptable " +
     "FROM table WHERE condition";
  executeQuery(sql);

procedure procB:
  sql = "INSERT INTO table2" +
     "SELECT * FROM #temptable";
  executeQuery(sql);

procedure procC:
  procA();
  procB();
```

**Figure 3.5.1: Example use of a temporary table to pass data from one procedure to another one.** There is close connection between `procA` and `procB`, as they are in the $CRUD^T_{PP} \cap SEA$ relation.

### 3.5.3 Potential applications

Based on the information extracted from the source, we provided support to the company in different areas including software architecture management and software testing. Here, we overview our experiences related to these two potential applications of the dependencies computed with our methods.

#### Architecture reconstruction

In architecture reconstruction, static analysis is used to automatically detect the various relations among software components. With our industrial partner, we previously performed an analysis like this on programs using a call graph. We then extended this architecture graph using information obtained from *CRUD* and *SEA*/*SEB*. The two relations added 49, 754 additional edges to the 2, 459 original ones among the 584 programs included in the architecture graph. We were also able to include 785 tables and 1, 921 relations among tables and programs in the graph.

#### Regression test selection

Regression tests are carried out to ensure that modifications in the code do not introduce new bugs. However, regression test suites are usually very large, and executing all the test cases for small modifications – despite being safer – can be very expensive resource-wise. Thus, regression test case selection is important. However, executing only those test cases that directly cover the modifications is not always sufficient, because the change might have an impact on other areas of the system. Impact analysis can be performed based on many kinds of dependencies including the call graph, *SEA/SEB* and *CRUD*.

We computed code coverage (for the change and the corresponding test selection result) using the test execution data of a real testing project. Figure 3.5.2 shows how the average coverage value varies with the different impact sets. As can be seen, inspecting more procedures results in a lower coverage. The relation between these two values seems to be linear in our case, but the call graph-based firewall impact results in a smaller coverage value, which was surprising to us. As for *SEA/SEB* vs. *CRUD*, it can be seen that using *SEA/SEB*, larger impact sets are obtained and this naturally results in a lower coverage.

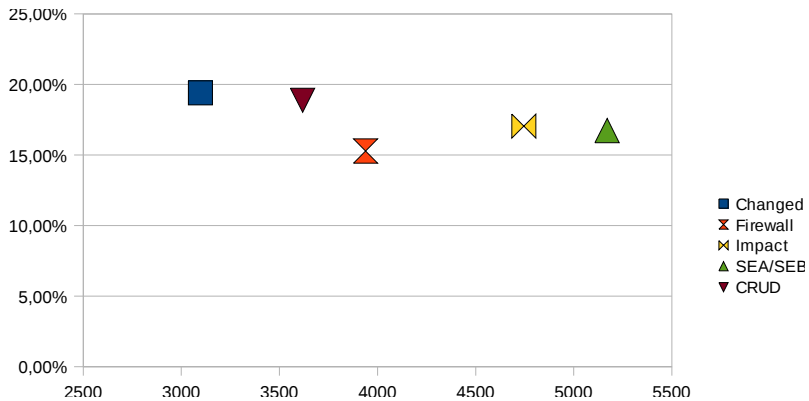



**Figure 3.5.2: Coverage of different procedure sets.** On the X axis the size of the procedure set (corresponding to the different impact analyses) is shown, with the Y axis denoting the coverage values. *Changed* denotes the procedures containing the modifications only; and the other four sets denote procedures that are accessible from Changed procedures via some relations. *Firewall* and *Impact* use the call relation, with only directly accessible procedures, and all procedures that can be reached through a series of call edges, respectively. *SEA/SEB* and *CRUD* denote those procedures that can be reached by traversing the edges of *SEA/SEB* and $CRUD^{T}_{PP}$ relations.

### 3.5.4 Limitations

Here we overview some possible limitations that may be encountered when implementing the proposed methods. We encountered these issues as well, so they may serve as 'threats to validity' of the results presented above, especially regarding the safety of the analysis.

### Imprecise ASG

The first main step, which may be unsafe in a complex analysis system, is the source code analysis itself. In this step we extract the AST (Abstract Syntax Tree) from the source code and compute the ASG (Abstract Semantic Graph). In some cases, it may not be possible to build a proper ASG via a static analysis. For example, in the languages where dynamic procedure calls are allowed, it is not hard to construct source fragments where the called procedure cannot be determined. As the input of our methods, we assume that the input ASG is precise and safe.

### Unrecognized code fragments in SQL queries

Our SQL extraction method reconstructs the embedded SQL queries with a string substitution rule. Here, we will assume that the reconstructed and syntactically correct SQL commands have the same key characteristics as the SQL commands that will be executed by the application. Whenever an SQL query is not parsable, it is handled conservatively and we suppose that it accesses all the tables of the system. However, there are some cases where the SQL query is syntactically correct, but it is not possible to tell which tables it accesses. In Figure 3.3.4 we provide an example of this case. It can be handled by recovering the unknown table, but other problems may arise as well. It may happen that the unrecognized code fragment is in a place of an identifier which is recognized as a column, but it is actually a subquery that accesses several other tables of the database (Figure 3.5.3).

```
SELECT firstname, lastname
  FROM customers
  WHERE firstname IN (@@subquery@@);
SELECT @@subquery@@, lastname
  FROM customers;
```

**Figure 3.5.3: Example of a constructed SQL command where with an unrecognized subquery.** The unrecognized code fragment is in the place of a value, but it is actually a subquery.

The potential error-prone places of the unrecognized code fragment can be determined by a simple rule which states that whenever an unrecognized code fragment is located at a place where it may refer to a subquery or table, it is assumed that it accesses all the tables of the database.

In later steps, we will assume that the SQL analyzer and the algorithm which constructs the Usage Matrix are both able to recognize all these error-prone cases.

#### Database modifications during code execution

It may happen that while the application is running, the database gets modified. If the executed SQL command that produces the change in the database is embedded in the source code, it can be located; but it is hard to tell its influence on the other commands. It may still happen that the database is modified outside the scope of a source code analysis. Our system recognizes the database modification SQL statements, but it does not evaluate them individually. Therefore, they are handled like every other kind of table access.

#### Dependencies via stored procedures

In data-intensive systems, it is normal to use stored procedures. Stored procedures are declared and they run on the database side of the application, but it is possible to create and execute them from the application by embedding specific SQL statements (e.g. CREATE PROCEDURE, EXEC). It should be mentioned that a stored procedure can access database tables like any other SQL command; hence if a procedure of the application executes a stored procedure its Usage Matrix should be properly updated with the accessed tables.

#### Dependencies via internal database dependencies (e.g. triggers, foreign keys)

Some dependencies may arise via internal database structures like triggers or foreign keys. These dependencies may lead to a situation where the database manager updates a table due to some modification made in another table. These dependencies can be handled by using an accurate database scheme analysis.

#### Dependencies through temporary tables and views

It is also common in relational database systems to use views for queries or temporary tables to store temporary data. Both of them are sources of hidden dependencies similar to internal database dependencies. Views – like structures selecting data from other tables – can be handled like any other table of the database, but their columns must point to the columns of the original table columns. In the case of temporary tables it is important to bear in mind that it is very hard to follow the lifecycle of a temporary table via a static analysis. Our system currently handles temporary tables like any other table of the system. If one temporary table is created only once inside a compilation unit, all of its references will be properly identified. However if there are other temporary tables created with the same name, it is impossible to determine statically which one is used in a query string.

## 3.6 Conclusions

Determining program dependencies in the right way via code analysis is a difficult task. Although many kinds of dependencies and the corresponding methods of analysis have been presented in the literature, they are usually not safe, precise, and efficient at the same time.

We presented two methods for recovering program dependencies in data-intensive applications, whose combination is safe in certain situations. One is based on the *SEA*/*SEB* relations, and the other uses *CRUD*-based Usage Matrices. We think that the use of these two methods for recovering program dependencies is a novelty here. We performed measurements with experimental implementations of the methods in an industrial context, and presented preliminary results that contain a quantitative and qualitative comparison of the methods, and also some potential applications.

The results show that the disjoint parts of the relation sets of the two methods are similar in size, and that their intersection is considerably smaller (it is about 3% of the union). So, based on this empirical evaluation, our main conclusion is that neither of the relations is definitely better (safer and more precise) than the other; they are simply different. Thus they should be applied together in situations where a safe result is sought. However, as the corresponding dependency sets are usually different, their intersection could also be interesting in some other situations, such as when a prioritization of the dependencies is necessary; in which case the intersection can act as a higher priority dependency set.

*"It is good to have an end to journey toward; but it is the journey that matters, in the end."*

Ernest Hemingway

# 4

# Case study of reverse engineering the architecture of a large banking system

Here, we present a case study of recovering the architecture of a large, legacy banking system. The system in question was written in PL/SQL, which is basically the procedural extension of Oracle SQL. In the case of the target system, the business logic was written in PL/SQL and was deployed to the same database which stored the data. Hence, the software system was very tightly coupled with the database. This architecture makes the system a good target for studying data-intensive systems and utilizing methods introduced in earlier sections.

## 4.1 Overview

In a banking system a simple rounding error may have a catastrophic effect on the reputation of the financial company, so there is a great pressure on developers to be precise and test their code as much as possible. However, business departments often urge the company to react to changes and implement new features rapidly. Developers emphasize reusing existing and already tested solutions with fast, minor modifications, instead of designing solutions that focus on the quality and the maintainability of their code. This usually results in a rapid evolution and growth of the system based on uncertain and ad-hoc decisions, which may lead to a loss of control over the codebase in the long term.

In this chapter we present a case study of analyzing a large banking system that was mainly written in Oracle PL/SQL. The system in question was being developed by one of our industrial partners from the financial sector whose name cannot be published owing to our signing a confidentiality agreement. After many years of development, they realized that their system's development was going in the wrong direction and they asked for help to take the necessary steps to counter the serious software maintainability problems they were experiencing. Here, we present both an analysis of their problems and our assessment.

The main contributions of this chapter are:

- A case study – performed in a real industrial environment – of analyzing quality attributes and reconstructing the architecture of a large PL/SQL banking system;
- Working solutions for emerging maintainability problems during the development of a large PL/SQL system.

## 4.2 Background story

The story began when our partner bought a boxed financial software package from India. The programming language of the software was Oracle PL/SQL and it was designed to be readily extendable with additional functionalities. The only drawback of the package was that some parts of the core system contained wrapped stored procedures and packages, hence it was not possible to modify the core functionality. However, at that time this did not seem to be necessary.

Later the company started to extend their system with new features. The system evolved rapidly, and soon it became overly large, so the small development team of the company could no longer maintain it by themselves. Instead of hiring new programmers, they decided to outsource the development of certain modules to professional companies. The companies had to take responsibility for their own code so this decision seemed reasonable. However, the vendors started to work hard and the system again started to grow rapidly. Its architecture soon became very complex. The company realized that maintaining the system and implementing new features would became increasingly expensive, so they had to stop the development process and take steps to handle the situation. An illustration of this situation can be seen in Figure 4.2.1.

Some of their most pressing problems were the following:

- The system was too complex,
- Only a few experienced developers were aware of the full architecture,
- Modifications were extremely expensive,
- Nobody was able to estimate the cost of a modification,

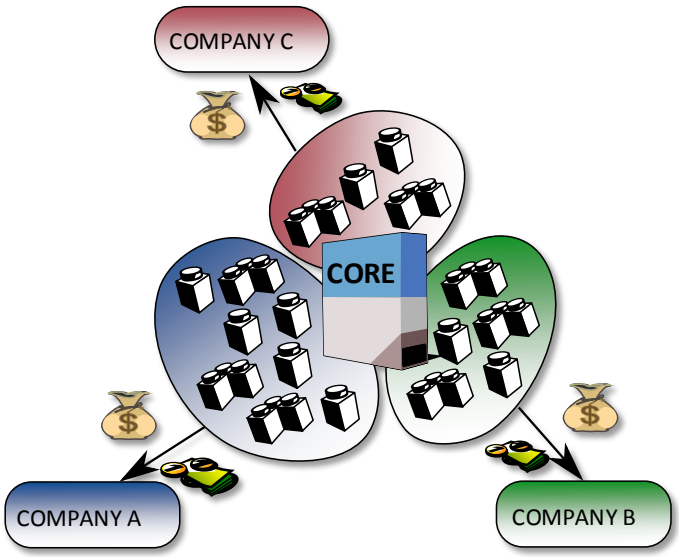



**Figure 4.2.1:** Outsourcing of the development of larger components.

- The code quality was poor,
- Maintenance was very expensive,
- Testing was expensive.

This was how things were before we were invited to participate.

## 4.3 Analysis and architecture reconstruction

A preliminary inspection of the system showed us that it was written mainly in Oracle PL/SQL with some additional subsystems (e.g. Web clients) in Java. Our main point of focus was the PL/SQL code itself because the full business logic was implemented there together with the data-management tier that was laid in the PL/SQL codebase.

Our first assessment was to create an architecture map of the system in order to learn how the system worked and visualize the interrelations among higher level components. Such a map is a useful tool for estimating the impact of a change in one component on the others. We created a map from two main sources of information. First, we performed a detailed low-level static analysis of the PL/SQL codebase; then we conducted interviews with the developers.

### 4.3.1 Low-level static analysis

A static analysis was performed on PL/SQL dumps using Columbus [47] that we extended for this study with an Oracle front-end. Our purpose was to identify low-level database objects (tables, views, triggers, packages, standalone routines) and relations among them (call relations, CRUD relations, etc.). The system turned out to be larger than we first expected. We analyzed

4.1M lines of PL/SQL and SQL code (full dump w/o data) that had 8,225 data objects, out of which 2,544 objects were packages. The total number of stored routines was over 30,000 and the system had 1.8 MLOC (million lines of code) in total (see Table 4.3.1).

| Property | Value |
|---|---|
| Total size of the full dump (w/o data) | 4.1 MLOC |
| Total size of stored procedures | 1.8 MLOC |
| Number of PL/SQL objects (tables, views, triggers, packages, routines) | 8,225 |
| Number of packages | 2,544 |
| Number of stored routines (including routines in packages) | >30,000 |

**Table 4.3.1:** Overview of the system.

### 4.3.2 Interviews

We interviewed the developers to identify higher level logical components of the system. PL/SQL was not designed to support higher level modularization, so it was necessary to get this information from the developers instead of the codebase itself. We identified 26 logical components (Accounting, Security, etc.). The developers also told us that they kept strict naming conventions, hence the corresponding component of a data object could be readily identified from its name (e.g. `PKAC_*` is a package of the Accounting component). Unfortunately, we found later that their naming convention was not that strict, so many objects remained uncategorized.

### 4.3.3 The architecture map

Based on the naming conventions, we grouped low-level data objects into components and we lifted up relations among them to the higher, component level. The final result was a dependency graph where the nodes were the components (identified via interviews) and the directed edges were the dependencies among them (identified via a static analysis). The graph had over 200 dependency edges among the 26 components, which meant that every object depended on almost every other (see Figure 4.3.1).

The results of the architecture reconstruction task made it clear that the system design was very complex. Even a simple change in a component might have an impact on almost every other component.

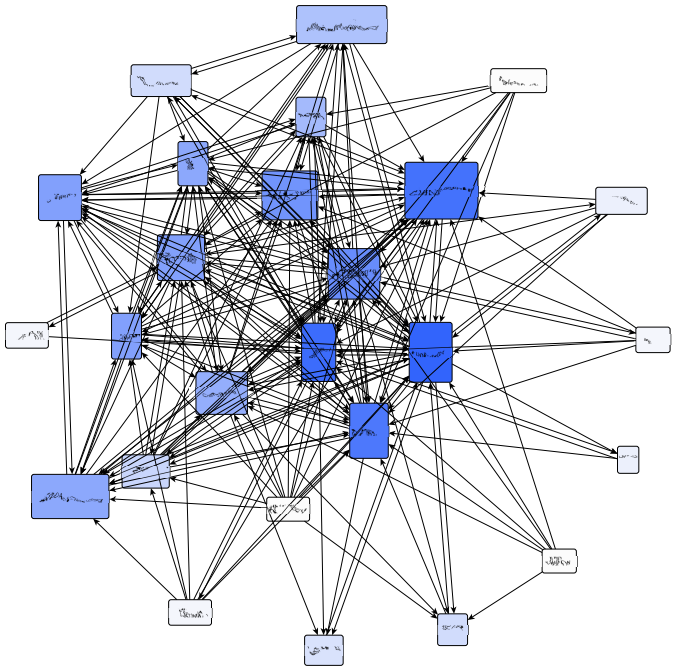


**Figure 4.3.1:** Relations among components of a large data-intensive system that evolved in an ad-hoc manner: almost all of the 26 components are related to practically all other component (names distorted).

### 4.3.4 Code quality

We investigated the quality of the source code as well. We identified many extremely large (over 3,000 LOC) and complex (McCabe's complexity larger than 1,000) stored routines in the system. In addition, we measured over 20% clone coverage (copy&paste source code fragments) and we found 5 almost identical copies of a package with over 5,000 LOC.

Apart from the most critical outlier objects of the system, the overall source code quality had a marked detrimental effect on the maintainability of the system. We found thousands of coding rule violations and dangerous error-prone constructs in the codebase.

## 4.4 Elimination of unused objects

We followed the life-cycle of the system over a half-year period. It turned out that during this period the total number of database objects increased by about 25% (see Table 4.4.1). The developers told us that during this period they had added some new features to the code, which may explain the huge number of new objects, including the large number of new tables (see Table 4.4.2). Another reason was that they usually used working (temporary) tables that often remained in the database even after the end of the development.

They also told us that they had re-implemented the biggest component of their system in Java

| Date | Total number of objects |
|---|---|
| 2010.04 | 8,255 |
| 2010.09 | 9,582 |
| 2010.11 | 10,681 |

**Table 4.4.1:** Growth in the total number of database objects in the system over a half year period.

| Date | Table | View | Trigger | Routine | Package |
|---|---|---|---|---|---|
| 2010.04 | 3,943 | 1,350 | 337 | 51 | 2,544 |
| 2010.09 | 4,868 | 1,459 | 346 | 102 | 2,807 |
| 2010.11 | 5,865 | 1,462 | 355 | 143 | 2,856 |

**Table 4.4.2:** Detailed growth of the system over a half year period.

and they had functionally cut off this component from the rest of the PL/SQL codebase. Hence, a huge number of data objects remained unused in the code. Furthermore, large data tables also remained in the database, but became useless after the code reorganization. All the unused stored procedures and packages increased the complexity of the system. Also, large and useless data tables had a negative impact on hardware maintenance costs. Note that the required table space for these data tables could be measured in TBytes.

The elimination of the obsolete components and unused data objects became especially important because of growing hardware maintenance costs.

Removing unused data objects requires careful work for such a complex system. If an unhandled reference remains in the code, its consequences may be hard to predict. Although it would be a catastrophic error, it would still be a better case when the system fails with an 'object does not exist' error, compared to miscalculating the account balance of a customer without any signs of error. Direct references to objects can be identified via the database manager or by performing a static analysis, but dynamic references may still remain hidden.

Overall, we identified a number of challenges in eliminating a single unused component from the system:

- Identifying tables/procedures of the component that had become obsolete,
- Identifying references to tables/procedures of the obsolete component,
- Validating the correct removal of the elements (e.g. make sure that no dead code remained after removing them).

### Identification of objects of obsolete components

We were able to identify data objects of the obsolete component by using our previous categorization based on the naming conventions of the company. However, this was not enough as some of the developers did not abide by the naming conventions and many objects remained uncategorized.

We defined five elimination sets and calculated them via a static impact analysis:

$SET_1$: elements of the obsolete components that match the naming conventions;

$SET_2$: uncategorized elements in (direct or transitive) relation only and only with objects from $SET_1$ or $SET_2$;

$SET_3$: uncategorized elements in (direct or transitive) relation with objects from $SET_1$ and $SET_4$;

$SET_4$: categorized elements in (direct or transitive) relation with objects from $SET_1$;

$SET_5$: elements that have no (direct or transitive) relation with $SET_1$.

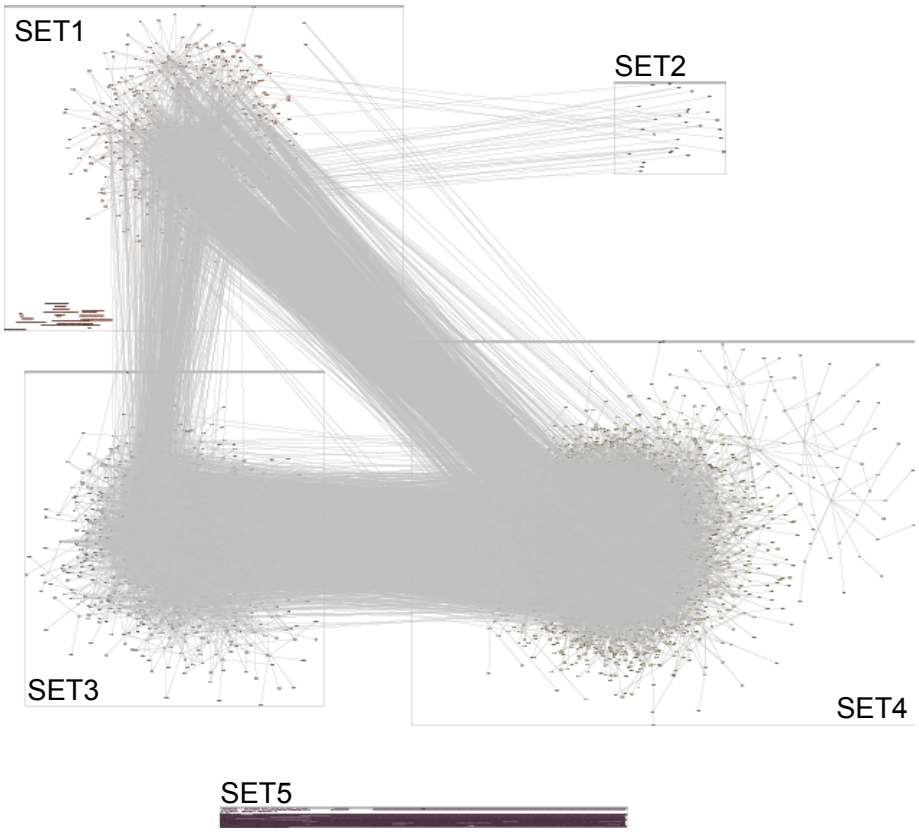



**Figure 4.4.1:** Elimination sets showing many un-cut relations between the obsolete component and others.

The relations between $SET_1$ and $SET_3$ or $SET_4$ (see Figure 4.4.1) told us that the component had not been functionally cut from other components even though the developers told us that they had done this earlier.

## 4.5 Conclusions

The case study began as our industrial partner (from the financial sector) had an information system with serious maintainability problems. We analyzed their problems, performed a thorough analysis of their system and we assisted them in eliminating unused data objects in order to simplify their system's architecture. We believe that our complex methodology for preventing software deterioration and solving maintenance issues helped them in their daily problems. Some of the above-mentioned techniques are so novel to the company that we cannot report on objective measures that compare maintenance costs or quality attributes before or after our analysis. However, it is obvious that the company had a great need for ready solutions and they were eager to try them out as soon as possible. Now, we are certain that they pay a lot more attention to the code quality and overall complexity of their system.

# Part III

# The world of Magic

*"For you to sleep well at night, the aesthetic, the quality, has to be carried all the way through."*

Steve Jobs

# 5

# A reverse engineering framework for Magic applications

Since the appearance of 4GLs, large software systems have evolved and the role supporting the maintenance of these systems with automatic techniques is of increasingly importance. Unfortunately, the main focus of current analyzer tools and techniques is on the more popular 3GL languages.

In this chapter, we design a reverse engineering framework based on the Columbus methodology in order to investigate whether the Columbus methodology is suitable for Magic as a special 4GL. In this framework we implement automatic static analyzers to identify typical coding issues, compute metrics for Magic applications and recover architectural dependencies.

## 5.1 Overview

In Magic, the whole programming process was designed for rapid application development and deployment. It includes the database component as well, which is the core element of a Magic application. The application is so much tied up with a database that everything in the programming language is related to a data table whether it is a real, persistent table or a virtual table (e.g. for variables). Magic has its own database management layer which is able to handle the most popular DBMSs, making it (conceptually) possible for us to migrate the application easily from

one DBMS to another one.

In the literature few papers are available that consider the software quality of applications written in 4GLs. When these languages became popular, many studies were published that promoted their use. These studies tried to predict the size of a 4GL project and its development effort for instance by calculating function points [115, 118] or by combining 4GL metrics with metrics for database systems [81]. Generally speaking, the goal was to demonstrate that programming in a 4GL is more efficient than programming in a 3GL. Today, some tools are available for testing purposes and for optimization purposes too, like the Magic Optimizer[1] tool.

Our motivation was to satisfy a true, industrial need for a reverse engineering framework for Magic. A company located in Szeged called SZEGED Software Inc., had developed applications in Magic for over two decades and expressed the importance of such a framework. This company is the largest Magic developer company in Hungary and their primary product is a system developed for pharmaceutical wholesalers and used by companies (e.g. TEVA Magyarország Kft., PHOENIX Pharma Inc.) not only in Hungary, but in Romania too. This system is based on the Magic technology. The company wanted to improve their quality assurance processes and they wanted to have a tool support for migrating their software package from an earlier version of Magic to the most recent one.

To successfully perform these tasks, many challenging research questions arose. Some of these were:

**RQ1:** *Can the reverse engineering method of Columbus be adapted to Magic applications where there is no source code in the traditional form, but there is a saved state of the application development environment?*

**RQ2:** *Are the quality attributes (e.g. metrics and coding rule violations) of 3GLs useful for Magic developers too?*

**RQ3:** *Can we extract architectural information from the 'source code' of a Magic application?*

## 5.2 Reverse engineering Magic applications, implementing the framework

The reverse engineering framework we implemented for Magic applications was designed by following the principles of the Columbus methodology (introduced in Chapter 2) and focusing on the previously introduced requirements.

The framework first takes the export of a Magic project that is stored in a version control environment. Currently, Magic xpa supports any third-party version control products that uses the

[1] http://www.magic-optimizer.com/

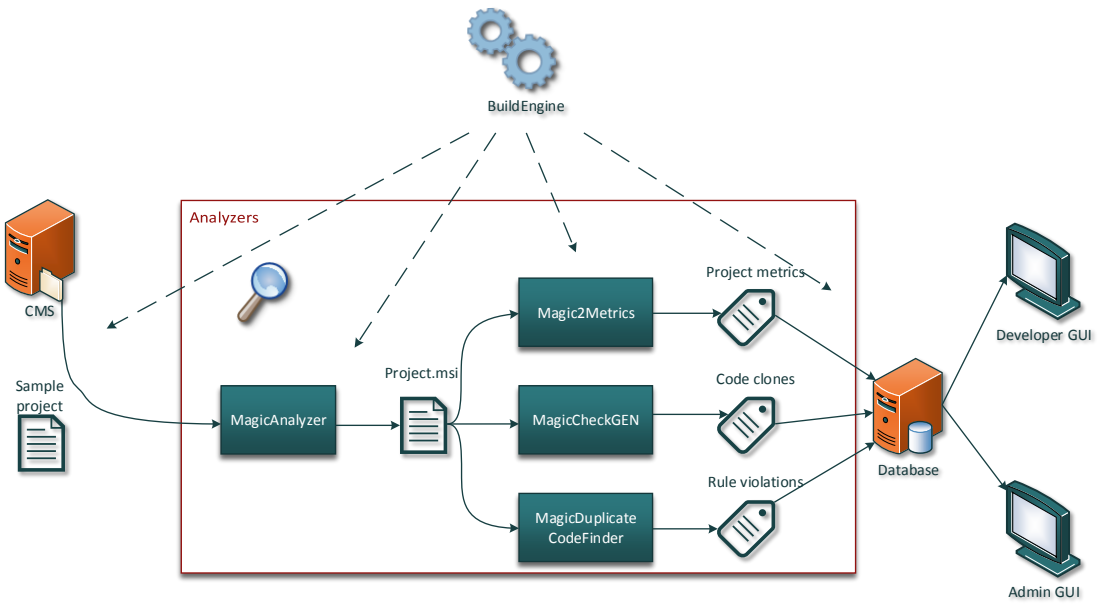



**Figure 5.2.1:** Reverse engineering framework for Magic.

SCC (Source Code Control) API, such as Visual SourceSafe and PVCS. However, some developers prefer to use the popular Subversion as version control system. During the first analysis, the ASG is constructed, which describes the internal structure of the application. This ASG is then passed to exporter tools to calculate metrics and to identify coding issues or code clones in the application. The results of these tools are then uploaded to the database of SourceInventory, a product of FrontEndART Ltd. with a GUI that was designed to allow the developers to query and analyze these results in a user-friendly way.

### 5.2.1 Constructing the ASG of a Magic application

Since Magic was invented in order to develop business applications for data manipulating and reporting, it comes with many GUI screens and report editors. All the logic that is defined by the programmer, the layout of the screens, the pull down menus, reports, on-line help, security system, reside inside tables called Repositories. The most important elements of the meta model language are the various entity types of business logic, namely the Data Tables. A Table has its Columns and a number of Programs (consisting of subtasks) that manipulate it. The Programs or Tasks are linked to Forms, Menus, Help screens and they may also implement business logic using logic statements (e.g. for selecting variables, updating variables, conditional statements).

In the Columbus methodology, the structure of a 3GL is described by a *schema*, like the *CPP Schema* [47]. The schema is given by a UML diagram where classes denote language elements and relations among classes can describe the parent relationship among nodes of the AST (aggregation) or references among them such as id nodes that refer to declaration nodes (association).

In Magic we use the *Magic Schema* to describe the internal structure of the application, hence

we use it to describe the main elements of the language and the relations among them. Figure 5.2.2 shows a small portion of the Magic Schema that focuses on some important elements of the language concerning the quality attributes of a Magic software. The full Magic Schema can be found in the Appendix B. The most important language elements are those elements that directly implement the logic of the application. A Magic *Application* consists of *Projects*, the largest entities breaking down an application into separate logical modules. A Project has *Data Tables* and *Programs* (a top-level Task is called a Program) for implementing the main functionalities. A Program can be called by a *Menu* or by other Programs during the execution of the application. When the application starts up, a special program called the *Main Program* is executed. A *Task* is the basic unit for constructing a program. A Program may consist of subtasks in a tree-structured hierarchy. The Task represents the control layer of the application and its Forms represent the view layer. It typically iterates over a Table and this iteration cycle defines so-called *Logic Units*. For instance, a Task has a Prefix and a Suffix that represent the start and the finish phases during its execution, respectively. A record of the iteration is handled by the Record Main logic unit, and before or after its invocation the Record Prefix or Suffix is executed. A Logic Unit is the smallest unit that performs lower level operations (a series of *Logic Lines*) during the execution of the application. These operations may be a simple operation like calling another Task or Program, selecting a variable, updating a variable, inputting data from a Form or outputting the data to a Form Entry.

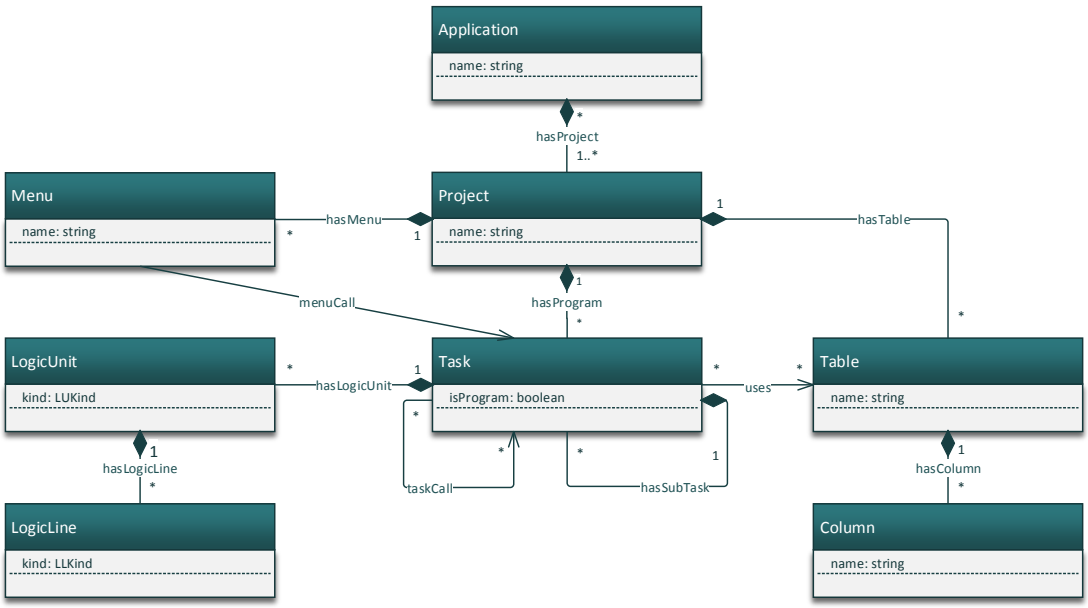



**Figure 5.2.2:** Most important Magic Schema entities. (The full Magic Schema can be found in the appendix.)

The language has evolved since its early releases, but core elements and their relations have

remained the same. Hence, the schema was designed in such a way as to represent more than one exact version of the language. That is, it contains elements of the language starting from version 5, which is an early version developed for DOS and UNIX platforms.

The internal structure of a Magic application is stored in different formats depending on the version of the Magic development environment. In Magic version 5, it is in a text file called `.ctl` which is defined by the development environment. In later versions (starting from version 9) it is stored in multiple files in XML format. An example of an export file can be found in the Appendix B.

The fact that the export format is different, but the structure of the language contains only minor changes from version to version, gives rise to the need for different parsers supporting different versions, while building the ASG according to the same Magic Schema.

We should add here that the parser of the reverse engineering framework was implemented by our industrial partner, SZEGED Software Inc., but the Magic Schema was designed mainly by the author of this dissertation.

### 5.2.2 Quality attributes of a Magic application

Once we have the ASG constructed by our static analyzer, we can use it for further computations. To investigate the quality attributes of a Magic application we first defined product metrics (like size, complexity attributes and coupling metrics). These metrics were based on the idea of similar 3GL metrics and we examined whether they were useful in the context of Magic.

We found a number of product metrics for Magic and categorized them in terms of *size*, *coupling* and *complexity*. Most of them were based on popular and well-known product metrics such as the *Lines of Code*, *Number of Classes*, *Number of Attributes*, *Coupling Between Object classes* [9].

We realized that some metrics could be readily adapted from third generation languages, but their meaning and benefits for the developers may be completely different compared to the 3GL counterparts.

In the case of size metrics, for instance, one could identify a series of 'Number of' metrics (e.g. *Number of Programs*, *Menus*, *Helps*), but they are considered less useful and interesting for developers. The reason for this is that most of these values can be easily queried through the application development environment.

The Lines of Code (*LOC*) metric can be readily adapted by taking into account the fact that the *Logical Line* language entity of Magic can be used so that it corresponds with a 'Line of Code' in a third generation language. However, the adapted metric should be used with caution because it has a different meaning compared to the original *LOC* metric. In 3GLs, *LOC* typically measures the size of the whole system and it is used to estimate the programming effort in different effort models (e.g. *COCOMO* [18]). In the case of Magic, a project is built on many repositories (Menus, Help Screens, Data Tables, etc.) and *LOC* measures just one size attribute of the sys-

tem (the Program repository). Hence, *LOC* is not the only size attribute of an application, so it cannot be used by itself to estimate the total size of the complete system.

Coupling is interesting in a 4GL as well. In object-oriented languages a typical metric for coupling is the *Coupling Between Object classes* (*CBO*) metric, which tells us the number of classes to which a given class is coupled. A class is coupled to another one if it uses its member functions and/or instance variables. 4GLs usually do not have language elements that represent objects and classes. For instance in Magic, there are no entities for encapsulating data and related functionalities, but there are separate data entities (Tables) and their related functionalities are specified in certain Tasks or Programs. Therefore it makes sense to measure the *Coupling Between Tasks and Data Tables,* just as we do for the *Coupling Between Tasks and Tasks*.

The list of the metrics we implemented for Magic can be seen in tables B.2.1, B.2.2, B.2.3 and B.2.4 of the appendix.

### 5.2.3 Identifying coding issues in a Magic application

In each programming language there are well-known best practices for developers that are sometimes written and sometimes unwritten. With the help of static analyzers, for some specific cases it is possible to automatically check whether developers follow these practices or not. PMD, CheckStyle, CodeSonar and FxCop, are some typical, common tools in the world of 3GLs. In the world of Magic, the only tool available is Magic Optimizer, which is commercially available and it is able to identify coding issues in an application.

With the help of developers of SZEGED Software Inc, we defined new coding rules for Magic and we implemented algorithms so as to recognize them on the ASG of an application.

### 5.2.4 Examining architectural dependencies in a Magic application

Another key feature of the framework is that it is able to identify architectural dependencies, hence help the developers in investigating the structure of the architecture. The motivation here was to support the migration of a Magic system from an earlier version of Magic to the most recent one.

#### Design recovery

The design recovery process includes analysis techniques that are able to present the information gathered via higher level views to the developers. These views are the following:

*Physical view.* This view presents the structure of the application. Here, we identify relevant language entities (data tables, columns, programs, tasks, logic units, etc.) and their parent-child relation that determine the main structure of the application, like packages, classes and methods that define the structure of a Java application.

*Call view.* This view presents the call-graph of the system.

*Menu view.* In Magic applications, a Menu is an entity of the language too. A Menu can fire system/user events, or can call a program within the application. The executed program can also call other programs or subtasks. Hence, in this view, we extended the Call view with the menu entities and their program call relations.

*User rights view.* Magic offers ready solutions for user and user role management. In Magic, one can define *rights* which describe the role to access menus, programs or data tables. One can also define users and user groups and these users or groups may have a number of previously defined rights. In this view we present information stating whether a user has access to a menu/program/table or not.

#### Database design recovery

Magic applications strongly depend on their databases, hence we need to identify relations among source elements and data objects to support database design recovery. These relations are the following:

*Table-Task dependencies.* We identify relations among data tables and all those tasks and programs that use the specified table. Here, we differentiate between *create*, *retrieve*, *update* and *delete* relations. Using this view, one can easily identify language elements working on the same data table. This can be a powerful tool for identifying, say, the logical components in the system.

*Data Table relations.* In older Magic versions *foreign key*s were not supported and even in new versions one can develop his application without using them. The only way to determine the relations among data tables is to analyze the application logic and identify those parts of the code where they link together two or more tables. Here, we determine whether two tables are in *one-to-one*, *one-to-many*, or *many-to-many* relation and we identify the columns that were used for linking them together.

## 5.3 Case study

Thanks to the support of our industrial partner, we were able to test our framework in a real industrial environment. We performed a static analysis, computed metrics and identified coding issues on a large-scale application using the reverse engineering framework. There are over 2,700 programs in the whole application, which is a very large number by Magic standards. The total number of non-Remark Logic Lines of this application is more than 300,000. The application itself uses more than 700 tables.

| Metric | Value |
|---|---|
| Number of Programs | 2 761 |
| Number of non-Remark Logic Lines | 305 064 |
| Total Number of Tasks | 14 501 |
| Total Number of Data Tables | 786 |

**Table 5.3.1:** Main characteristics of the system analyzed.

### 5.3.1 Usage scenarios of metrics

Once we have calculated the metrics of the system, the framework can be used to query the results. An example might be to query sample metrics of the tasks with top LLOC (Logical Lines of Code) metric values. In addition to querying metric values, it tells us how we can locate code elements with critical metric values, e.g. the longest tasks or programs of the system in question. These tasks usually play a central role in the application as they implement a relatively big part of the business logic. They usually have a relatively high complexity as well and they are coupled to many other tasks, as can be seen in Figure 5.3.1. In this figure the NOI (Number of Outgoing Invocations), TNT (Total Number of Tasks), WLUT (Weighted Logic Unit per Task) and the LLOC[2] metrics can be seen. WLUT is a special complexity metric for measuring the complexity of a task and it is similar to WMC (Weighted Methods per Class), which measures the complexity of a class in object oriented languages. Besides the sample metrics, we defined about 50 metrics and grouped them according to size, complexity or coupling.

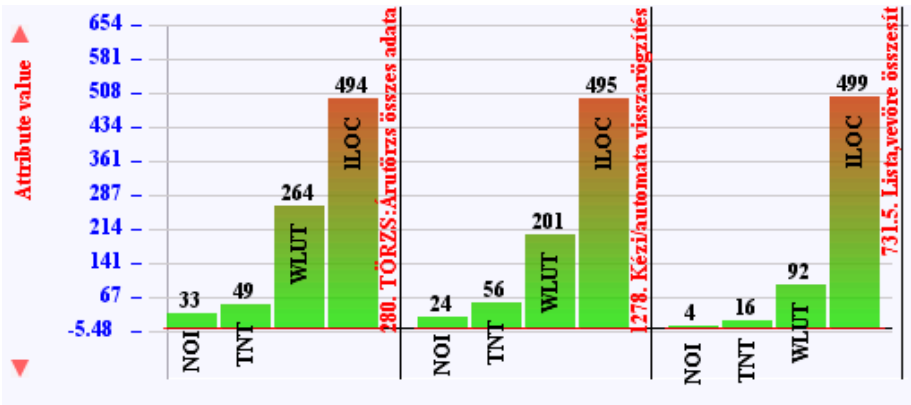



**Figure 5.3.1:** Bar chart showing the metrics of tasks with top LLOC metric values.

Figure 5.3.2 is a histogram showing the frequency distribution of LLOC metric values of tasks. Apart from the frequency distribution of the LLOC values, other information can be seen in the diagram like average LLOC value, number of items and variance. The critical tasks that implement

[2]LLOC has a special meaning in Magic too. It measures the number of non-remark (non-comment) logic lines (statements) in a task.

most statements for business logic can also be readily identified on the right hand side of the diagram.

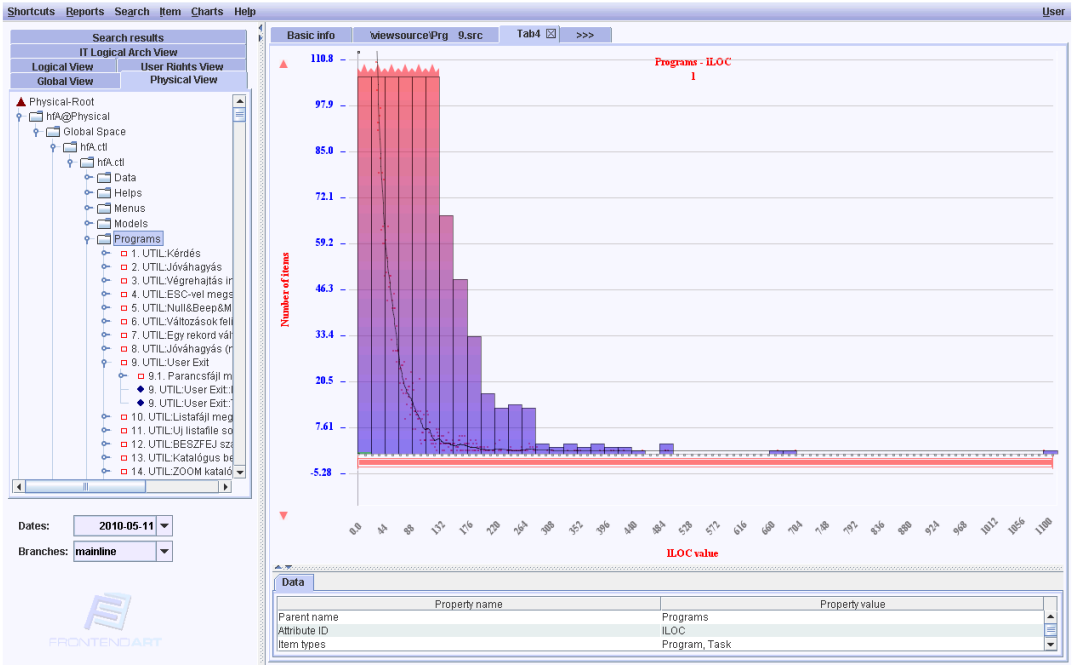



**Figure 5.3.2:** Histogram showing the frequency distribution of LLOC metric values of tasks.

### 5.3.2 Usage scenarios of architectural views

Some typical architectural views can be seen in the rest of the figures. Figure 5.3.3 shows a call view where it can be seen which program is the one which is called by most other programs of the system. Figure 5.3.4 shows a call view extended with menu accesses, so the developer can see from which menu point which programs are accessed. This query has the main advantage that the developer can readily see related code parts of a certain menu point, hence a certain feature of the system in question.

## 5.4 Conclusions

We obtained real-life experience with the reverse engineering framework thanks to our industrial partner who has over 15 years of experience with developing Magic applications and has excellent professional knowledge. Their complex logistics system designed for pharmaceutical wholesalers has attained outstanding references in Hungary. Here, we answer our initial research questions based on our experiences and the feedback got from our industrial partner.

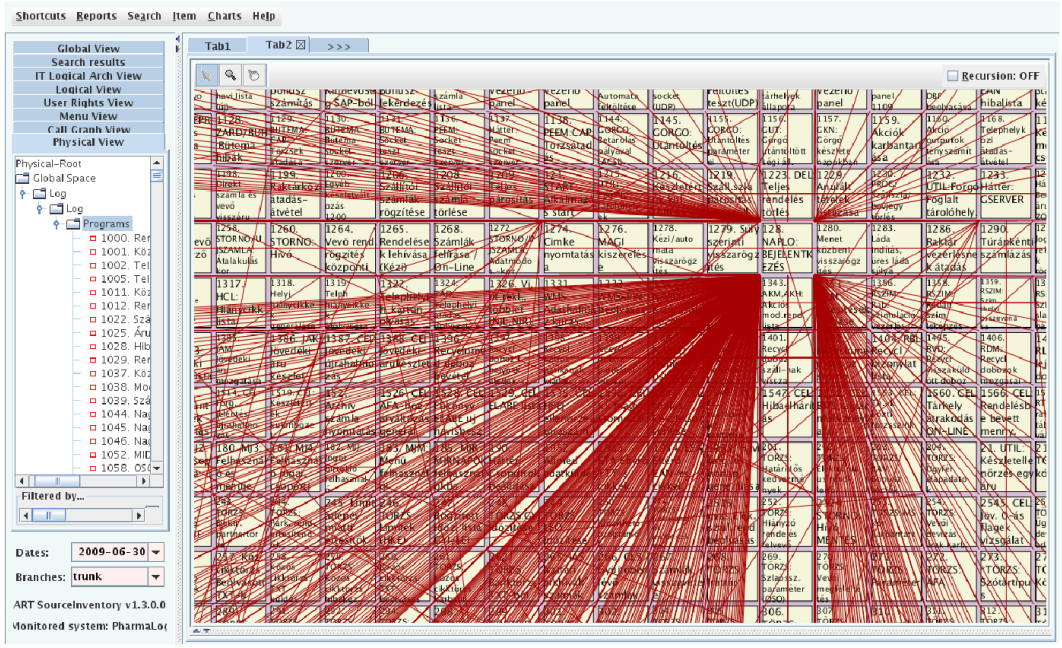



**Figure 5.3.3:** A program called from many other programs in the system.

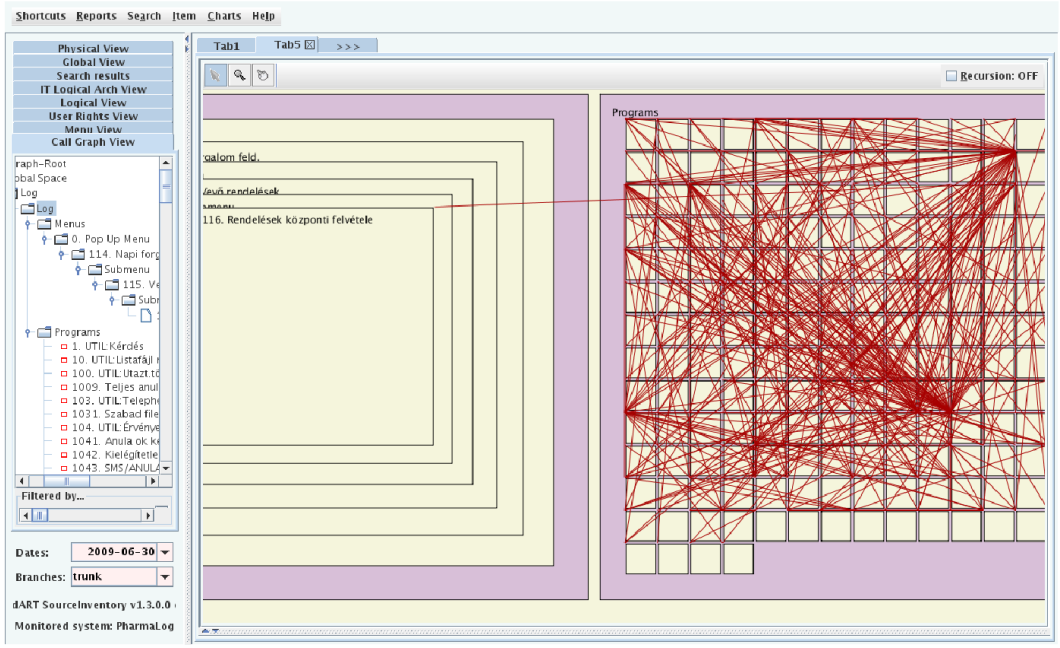



**Figure 5.3.4:** Program calls extended with menu accesses.

RQ1: Can the reverse engineering method of Columbus be adapted to Magic applications where there is no 'source code' in the traditional form, but there is a saved state of the application development environment? We designed the Magic Schema to describe the internal structure of a Magic application. Based on this schema, a directed graph could be constructed from the current save state of the application. Hence, the export file served as the source code and the constructed directed graph served an ASG in the 3GL concept.

We successfully used the constructed ASG as input for further investigations, e.g. for calculating metrics and identifying coding issues. The Magic developers were open-minded and showed special interest in using the tools. Based on our experiences, we conclude that the reverse engineering process of Columbus is definitely useful for Magic applications.

RQ2: Are the quality attributes (e.g. metrics and coding rule violations) of 3GLs useful for Magic developers too? Despite initial skepticism by the Magic developers, product metrics revealed interesting attributes of their system. There were also recommendations on how to add new Magic-specific metrics to better match developers views of Magic programs. Surprisingly, the well-known McCabe complexity metric seemed to be less useful than when we used it with object-oriented languages.

Checking rule violations was without doubt helpful to the developers as they admitted there were more violations than they had expected, and the majority of the problems found had to be corrected. They offered several suggestions for easing the handling of rule violations and for improving the quality of the checker.

RQ3: Can we extract architectural information from the 'source code' of a Magic application? We realized that the save state of a Magic application is a good source for identifying implicit information about the architecture of a system. We were able to reconstruct architectural views of an application, namely the physical view (structure of the application), call view (call-graph of the system), menu view (call view extended with the menu entities and their program call relations), user rights view (user accesses to a menu/program/table or not). In addition, we were able to identify dependencies between database elements and source elements, like table-task dependencies (*create*, *retrieve*, *update*, and *delete* relations) and data table relations (*one-to-one*, *one-to-many*, or *many-to-many* relations among data tables).

Overall, then, we conclude that 3GL concepts can be successfully adapted to a special 4GL environment like the Magic programming language. The reverse engineering framework served as a good basis for further quality assurance tasks as well. For example, in a research study we successfully automated the GUI testing of Magic applications based on UI information stored in the constructed ASG [123].

*"For you to sleep well at night, the aesthetic, the quality, has to be carried all the way through."*

Steve Jobs

# 6

# Defining and evaluating complexity measures in Magic as a special 4th generation language

During the adaptation of a 3GL reverse engineering methodology to Magic, we were confronted by the problem of the lack of software quality metrics defined for 4GLs. When we investigated the internal structure of Magic programs, we identified key points in defining new metrics and adapting some 3GL metrics to Magic. The greatest challenge we faced was the definition of complexity metrics, where experienced developers found our first suggestions inappropriate and counterintuitive. Enhancing our measures, we involved several developers in experiments to evaluate different approaches to complexity metrics. In this chapter, we will describe these experiments and the results we obtained from them.

## 6.1 Overview

Here, we present our experiences in defining complexity metrics in 4GL environment, especially in the application development environment called Magic, which was recently renamed to *Magic xpa*. Our contributions are:

- We adapted two common 3GL complexity metrics to Magic 4GL (McCabe complexity and Halstead complexities);

- We carried out experiments to evaluate our approaches (we found no significant correlation between developers' ranking and our first adapted McCabe complexity, but we found a strong correlation between a modified McCabe complexity and the developers ranking, and between Halstead's complexities and the developers ranking);
- As an outcome of the experiments, we defined new, easily understandable and applicable complexity measures for Magic developers.

Our experiment was designed to address the following research questions:

**RQ1:** *Is there a significant correlation among adapted complexity metrics of Magic programs?*

**RQ2:** *Is there a significant correlation between the complexity ranking given by developers and the ranking given by the adapted metrics?*

## 6.2 Measuring the complexity of Magic applications

We identified different quality attributes and defined a set of metrics for Magic applications. Simple size and coupling metrics indeed reflected the opinions of the developers, but this was not the case with complexity metrics. In fact, measuring the complexity of a 4GL system was our biggest challenge. Although there are many different approaches for third generation languages [21], at the source code level, well-known approaches were developed by McCabe [88] and Halstead [58] – which are widely used by software engineers, e.g. for software quality measurement purposes and for testing purposes.

> ***Complexity***: (1) The degree to which a system or component has a design or implementation that is difficult to understand and verify. (2) Pertaining to any of a set of structure based metrics that measure the attribute in (1).

**Def. 1:** Complexity according to the IEEE Standard Glossary [69].

We adapted McCabe's cyclomatic complexity and Halstead's complexity metrics to Magic, but when we showed the results to developers, their feedback was that all the programs we identified as most complex programs in their system were not really that complex, at least according to their experience. We should add here that all the programmers had been programming in Magic for more than 3 years (some of them for more than a decade) and most of them were well aware of the definition of structural complexity (see Definition 1), but none of them had heard of cyclomatic or Halstead complexity metrics before.

### 6.2.1 McCabe's cyclomatic complexity metric

First, we adapted McCabe's complexity metric [88] to Magic. McCabe used a graph-theory measure called the *cyclomatic number* to measure the complexity of the control flow of a program. It was shown that for any structured program with only one entrance and one exit point, the value of McCabe's cyclomatic complexity is equal to the number of decision points (i.e. the number of 'if' statements and conditional loops) contained in that program plus one.

McCabe's complexity is usually measured at the method or function level. For object-oriented languages it is possible to aggregate complexities of methods to the class level. The idea of *Weighted Methods per Class* (*WMC*) [24] is to give weights to the methods and sum up the weighted values. As a complexity measure, this metric is the sum of cyclomatic complexities of methods defined in a class. Therefore *WMC* represents the complexity of a class as a whole.

In the case of Magic, the basic operations are executed at the Logic Unit level. A Logic Unit has a well-defined entry and exit point as well. Likewise, a Task has predefined Logic Units. That is, a Task has a Task Prefix, Task Suffix, Record Prefix, Record Main, Record Suffix, and so on. This structure is similar to the construction of a Class where a Class has some predefined methods, like, constructors and destructors. Hence, we defined McCabe's complexity at the Logic Unit level with the same definition as defined for methods (see the definition of $McCC(LU)$ in Definition 2). So it can be simply calculated by counting the statements with preconditions (i.e., the branches in the control flow) in a Logic Unit. In a similar way, the complexity of a Task can be measured by summing up the complexity values of its Logic Units. We call this complexity measure the *Weighted Logic Units per Task* (see $WLUT(T)$ in Definition 3).

$McCC(LU) = \textit{Number of decision points in } LU + 1$
LU: a Logic Unit of a Task

**Def. 2:** The definition of McCabe's cyclomatic complexity for Logic Units.

$$WLUT(T) = \sum_{LU \in T} McCC(LU)$$
T: a Task in the Project
LU: a Logic Unit of T

**Def. 3:** The definition of Weighted Logic Units per Task (WLUT).

The $McCC(LU)$ and $WLUT(T)$ metrics were adapted using the 3GL definitions based simply on the syntactic structure of the language. When we first showed the new definitions to the developers they agreed with us and they were interested in calculating the complexity measures of their system. However, the results did not convince them. Those Tasks that we identified as the

most complex tasks of their system were regarded as not complex by the developers; and those tasks that were considered complex by the developers had lower *WLUT* values.

Developers suggested that in addition to the syntactic structure of the language, we should add the piece of semantic information that a Task is basically a loop which iterates over a table and when it calls a subtask it is rather similar to an embedded loop. After considering their suggestions we modified the McCabe complexity as follows ($McCC_2$). For a Logic Unit we simply count the number decision points, but when we find a call for a subtask it is treated as a loop and it increases the complexity of the Logic Unit by the complexity of the called subtask. In other words, the complexity of a Task is the sum of the complexity of its Logic Units. For the formal definition, see Definition 4.

$$McCC_2(LU) = \textit{Number of decision points in LU} + \sum_{TC \in LU} McCC_2(TC) + 1$$

$$McCC_2(T) = \sum_{LU \in T} McCC_2(LU)$$

T: a Task of the Project
LU: a Logic Unit of T
TC: a called Task in LU

**Def. 4:** The definition of the modified McCabe's cyclomatic complexity ($McCC_2$).

The main difference between $WLUT(T)$ and $McCC_2(T)$ is that $McCC_2(T)$ also takes into account the complexity of the called subtasks too in a recursive way. A recursive complexity measure could be similar for procedural languages when a function call might increase the complexity of the callee function by the complexity of the called function. (Loops in the call graph should be handled.)

Developers found the definition of the new metric more intuitive as it takes into account the semantics too. Later on in our experiments we found that the new metric correlates well with the complexity ranking of the developers (see Section 6.3).

### 6.2.2 Halstead's complexity metrics

Some of the developers also complained that our metrics did not reflect the complexity of the expressions in their programs. It should be added that Magic handles the expressions of a Task separately. An expression has a unique identifier and can be used many times inside different statements simply by referring to its identifier. The application development environment has an expression editor for editing and handling expressions separately. This results in a coding style where developers pay more attention to the expressions they use. They see the list of their expressions and large, complex ones can be easily spotted.

Halstead's complexity metrics [58] measure the complexity of a program based on the lexical

counts of symbols used. The basic idea is that complexity is affected by the operators used and their operands. Halstead defines four *base values* for measuring the number of distinct and total operands and operators in a program (see Definition 5). The base values are constituents of higher level metrics; namely, *Program Length* (*HPL*), *Vocabulary size* (*HV*), *Program Volume* (*HPV*), *Difficulty level* (*HD*), *Effort to implement* (*HE*). For the formal definitions, see Definition 6.

$n_1$: the number of distinct operators
$n_2$: the number of distinct operands
$N_1$: the total number of operators
$N_2$: the total number of operands

**Def. 5:** Halstead's base values for measuring the number of distinct and total operands and operators present in a program.

$$HPL = N_1 + N_2$$
$$HV = n_1 + n_2$$
$$HPV = HPL * log_2(HV)$$
$$HD = \left(\frac{n_1}{2}\right) * \left(\frac{N_2}{n_2}\right)$$
$$HE = HV * HD$$

**Def. 6:** Halstead's complexity measures.

In the case of Magic, symbols may appear inside expressions so the choice of Halstead's metrics seemed appropriate for measuring the complexity of expressions. Operands can be interpreted as symbols as in a 3GL language (e.g. variable names, task identifiers, table identifiers) and operators are the operators (plus, minus, etc.) inside expressions.

Later, in our experiments we found that Halstead's metrics correlated with the complexity ranking of the developers (see Section 6.3), but the modified McCabe's complexity was in practice closer to the opinions of the developers.

## 6.3 Experiments with complexity metrics

Although the classic complexity metrics have been successfully adapted to the Magic language, there is no empirical data available on how they relate to each other and on their applicability in software development processes. We observed that, except for the McCabe metric, complexity metrics generally do not have a justified conceptual foundation. Rather, they are defined based on experience [120]. Here, we seek to fill in the gap first by calculating and evaluating the adapted metrics on industrial size programs to see their relationship; and second, by conducting surveying with experts at a Magic developer company to learn the practical utility of the definitions. We would like to emphasize the importance of feedback given by Magic experts and include it as

input to future initiatives. There is also no extensive research literature available on the quality of Magic programs. Hence, the knowledge accumulated over many years of development is essential to justify the conceptual and practical background of our metrics.

Thus, to evaluate our metrics, metric values were computed on a large-scale Magic application, and a questionnaire was prepared for experienced Magic developers to learn their thoughts on complexity. We sought answers to the following research questions:

**RQ1:** *Is there a significant correlation among adapted complexity metrics of Magic programs?*

**RQ2:** *Is there a significant correlation between the complexity ranking given by developers and the ranking given by the adapted metrics?*

We performed a static analysis on the same large-scale Magic application that we had analyzed earlier in Chapter 5. The system had over 2,700 programs and the total number of non-Remark Logic Lines of this application was over 300,000. These numbers are regarded as enormous by Magic standards.

There were 7 volunteer developers who took part in the survey at the software developer company. The questionnaire consisted of the following parts:

1. Expertise:
   (a) Current role in development.
   (b) Developer experience in years.
2. Complexity measures for Magic:
   (a) At which level of program elements should the complexity be measured?
   (b) How important are the following properties in determining the complexity of Magic applications? (List of properties is given.)
   (c) Which additional attributes affect the complexity?
3. Complexity of concrete Magic programs developed by the company.
   (a) Rank the following 10 Magic programs (most complex ones first).

The most important part of the questionnaire is the ranking of the concrete programs. This enabled us to compare what was each the developer's mind when computing the metrics. Subject programs for ranking were selected by an expert of the application. He was asked to select a set of programs which *a)* was representative of the whole application, *b)* contained programs of various size, *c)* developers were familiar with. He was not aware of the purpose of the selection. The selected programs and their main size measures are listed in Table 6.3.1 below. The

number of programs is small as we expected a solid, established opinion of participants in a reasonable time. In the table, the Total Number of Logic Lines (containing task hierarchy) (*TNLL*), the Total Number of Tasks (*TNT*), Weighted Logic Units per Task (*WLUT*) and the cyclomatic complexity ($McCC_2$) are shown.

| Id | Name | *TNLL* | *TNT* | *WLUT* | $McCC_2$ |
|---|---|---|---|---|---|
| 69 | Engedmény számítás egy tétel | 1352 | 24 | 10 | 214 |
| 128 | TESZT:Engedmény/rabatt/formany | 701 | 16 | 14 | 63 |
| 278 | TÖRZS:Vevő karbantartó | 3701 | 129 | 47 | 338 |
| 281 | TÖRZS:Árutörzs összes adata | 3386 | 91 | 564 | 616 |
| 291 | Ügyfél zoom | 930 | 29 | 8 | 27 |
| 372 | FOK:Fökönyv | 1036 | 31 | 113 | 203 |
| 377 | Előleg bekérő levél képzése | 335 | 6 | 5 | 20 |
| 449 | HALMOZO:Havi forgalom | 900 | 22 | 3 | 117 |
| 452 | HALMOZO:Karton rend/vissz | 304 | 9 | 4 | 34 |
| 2469 | Export_New | 7867 | 380 | 382 | 761 |

**Table 6.3.1:** Selected programs with their size and complexity values.

## 6.4 Results

We will first discuss our findings on complexity measurements gathered via a static analysis of the whole application. Later, we will narrow down the set of programs given to those taking part in the questionnaire; and then we will compare them with the opinions of the developers.

### 6.4.1 RQ1: Is there a significant correlation among adapted complexity metrics of Magic programs?

Here, we investigate the correlation among the previously defined metrics. As the McCabe and Halstead metrics are basically different approaches, we will first examine them separately.

#### Halstead metrics

Within the group of Halstead metrics, a significant correlation is expected, because – by definition – they depend on the same base measures. In spite of this, different Halstead measures capture different aspects of computational complexity. We performed a Pearson correlation test to learn their relations in Magic. The correlation values we obtained are shown in Table 6.4.1. Among the high expected correlation values, the *HD* and *HE* metrics correlate slightly less well with the other metrics. We justified Halstead metrics using the *Total Number of Expressions* (*TNE*), which can be computed in a natural way as expressions are separately identified language elements. The

relatively high correlation between *TNE* and other Halstead metrics tells us that the *TNE* metric is a further candidate for a complexity metric. This also reflects suggestions given by the developers too. For the sake of simplicity, we will use the *HPV* metric to represent all five metrics of the group.

| | *HPL* | *HPV* | *HV* | *HD* | *HE* | *TNE* |
|---|---|---|---|---|---|---|
| *HPL* | 1.000 | 0.906 | 0.990 | 0.642 | 0.861 | 0.769 |
| *HPV* | 0.906 | 1.000 | 0.869 | 0.733 | 0.663 | 0.733 |
| *HV* | 0.990 | 0.869 | 1.000 | 0.561 | 0.914 | 0.773 |
| *HD* | 0.642 | 0.733 | 0.561 | 1.000 | 0.389 | 0.442 |
| *HE* | 0.861 | 0.663 | 0.914 | 0.389 | 1.000 | 0.661 |

**Table 6.4.1:** Pearson correlation coefficients ($R^2$) of Halstead metrics and the Total Number of Expressions (*TNE*) (all correlations are significant at the 0.01 level).

### Comparison of adapted complexity metrics

Table 6.4.2 contains correlation data on McCabe-based complexity (*WLUT*, $McCC_2$), *HPV* and two size metrics. The three complexity measures have a significant, but only a slight correlation, which indicates that they display different aspects of the program complexity.

Earlier, we outlined the differences between *WLUT* and $McCC_2$. The similar definitions imply a high correlation between them. Surprisingly, based on the measured 2700 programs their correlation is the weakest (0.007) compared to other metrics so they seem almost independent in practice. $McCC_2$ was measured on the subtasks too, which in fact affects the results. Our expectation was that, for this reason, $McCC_2$ should have a stronger correlation with *TNT* than *WLUT*. However, the $McCC_2$ metric only weakly correlates with *TNT*. This confirms the opinion that developers use many conditional statements inside one task, and the number of conditional branches has a higher impact on the $McCC_2$ value.

| | *WLUT* | $McCC_2$ | *HPV* | *NLL* | *TNT* |
|---|---|---|---|---|---|
| *WLUT* | 1.000 | 0.007 | 0.208 | 0.676 | 0.166 |
| $McCC_2$ | 0.007 | 1.000 | 0.065 | 0.020 | 0.028 |
| *HPV* | 0.208 | 0.065 | 1.000 | 0.393 | 0.213 |

**Table 6.4.2:** Pearson correlation coefficients ($R^2$) of various complexity metrics (all correlations are significant at the 0.01 level).

### Rank-based correlation

From this point on, we will analyze the rank-based correlation of metrics. The aim is to facilitate a comparison of results with the ranking given by the developers. The number of programs considered was now narrowed down to the 10 programs listed in Section 6.3. Ranking given by a certain metric was obtained in the following way: metric values for the 10 programs were computed, programs with higher metric values were ranked lower (e.g. the program with highest metric value has a rank no. 1).

The selection of 10 programs was justified by the fact that the previously mentioned properties (e.g. different sizes, characteristics) could be observed here as well. In Figure 6.4.1, the ranking of Halstead metrics is presented. On the $x$-axis the programs are shown (program Id), while their ranking value is shown on the $y$-axis (1-10). Each line represents a separate metric. A strong correlation can be observed as the values are close to each other. Furthermore, the *HD* and *HE* metrics can also be visually identified as a little bit outliers. (Note: Spearman's rank correlation values were also computed.)

The ranking determined by the three main complexity metrics can be seen in Figure 6.4.2. The $x$-axis is ordered by the $McCC_2$ complexity, so programs with lower $McCC_2$ rank (and higher complexity) are on the left hand side. A similar trend for the three metrics can be seen, but they behave in a variety of ways locally.

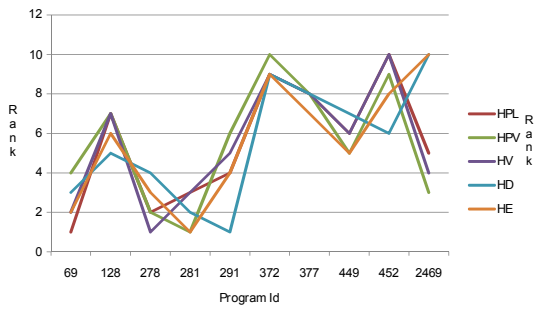



**Figure 6.4.1:** Ranking of Halstead complexity metrics (ordered by program ID).

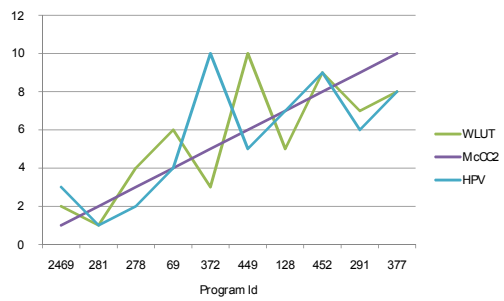



**Figure 6.4.2:** Ranking of the main complexity metrics (ordered by $McCC_2$).

Addressing our research question, we found that some of the complexity measures investigated had a strong correlation, but some of them were independent measures. We found a strong correlation among the Halstead metrics and we also found that these metrics correlate with the Total Number of Expressions. We found that our first adaptation of cyclomatic complexity (*WLUT*) had only a weak correlation with our new version ($McCC_2$), which correlates well with other measures. This also confirms the hypothesis that the new measure might be a better representative of the developers opinions on complexity.

### 6.4.2 RQ2: *Is there a significant correlation between the complexity ranking given by developers and the ranking given by the adapted metrics?*

In the third part of the questionnaire, developers were asked to rank the 10 programs in terms of their own complexity opinion. Previously, developers were given a short hint on common complexity measures, but they were asked to express their subjective opinions too. Most of the programs selected were probably familiar to the developers since the application was developed by their company. Furthermore, they were able to check the programs using the development environment during the ranking process.

Ranks given by the 7 developers are shown in Figure 6.4.3, where each line represents the opinion of one person. It can be seen that developers gave different rankings. There are diverse ranks especially in the middle of the ranking, while the top 3 complex programs were similarly selected. Not surprisingly, developers agree on the least complex program, which is 2469. Correlations of developers' ranks were also computed. A significant correlation was rare among the developers; only ranks of P4, P5 and P6 are similar (Pi denotes a programmer in Figure 6.4.3).

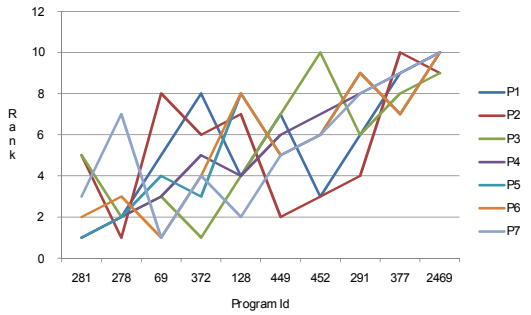



**Figure 6.4.3:** Ranks given by Magic experts.

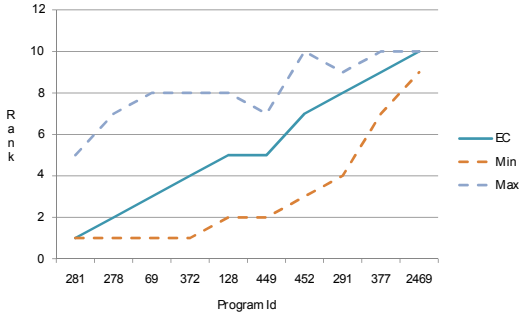



**Figure 6.4.4:** The *EC* value, *min* and *max* ranks.

We defined the *EC* value (*Experiment Complexity*) for each given program as the rank based on the average rank given by developers. In Figure 6.4.4, the *EC* value is shown together with *min* and *max* ranks of the developers. We should add that summarizing the developers' opinions on one metric may result in a loss of information as developers may have different views in their minds. We will elaborate on this later in the Threats to Validity section.

We compared the *EC* value with the previously defined complexity metrics. Table 6.4.3 contains correlation values for the main metrics. The *EC* entry displays a significant correlation with the *HE* measure.

Besides statistical information, complexity ranks are shown as well. We found that the rank-based correlation obscures an interesting relation between $McCC_2$ and the *EC* value. Ranks for each program are shown in Figure 6.4.5. The order of programs follows the $McCC_2$ metric. De-

| | *WLUT* | *McCC*$_2$ | *HPV* | *HE* | *EC* |
|---|---|---|---|---|---|
| *WLUT* | 1.000 | 0.575 | 0.218 | 0.004 | 0.133 |
| *McCC*$_2$ | 0.575 | 1.000 | 0.520 | 0.027 | 0.203 |
| *HPV* | 0.218 | 0.520 | 1.000 | 0.389 | 0.166 |
| *HE* | 0.004 | 0.027 | 0.389 | 1.000 | 0.497 |
| *EC* | 0.133 | 0.203 | 0.166 | 0.497 | 1.000 |

**Table 6.4.3:** Correlation of Magic complexity metrics and developers' view (Spearman's $\rho^2$ correlation coefficients, marked values are significant at the 0.05 level).

spite the fact that Spearman's $\rho^2$ values show no significant correlation, it can be clearly seen that the developers and *McCC*$_2$ metric give the same ranking, except for program 2469. This program was judged in a different way. The program contains many decision points, but the developers said that it was not complex since its logic was easy to understand. According to the *HE* metric, this program was also ranked as the least complex.

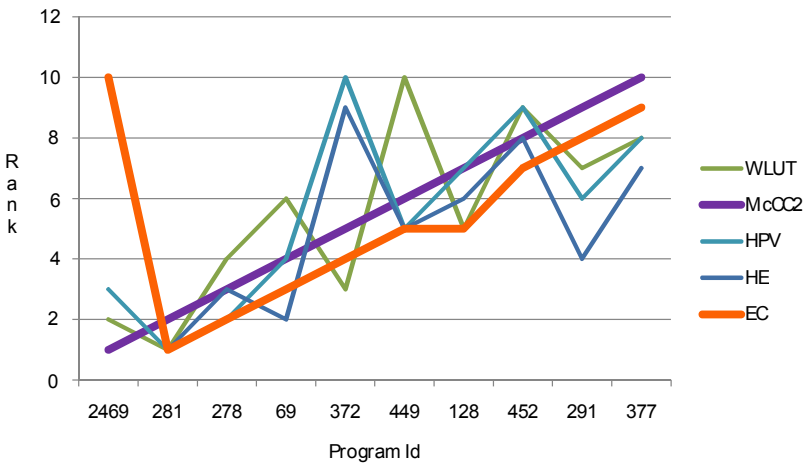



**Figure 6.4.5:** The *EC* value compared to the main complexity metrics.

Addressing our research question, we found that the rankings given by the adapted metrics have a significant and sometimes surprisingly strong relation with the rankings given by developers, except for the *WLUT* metric. Halstead's metrics have a significant correlation here, especially the *HE* metric. However, the strongest relation we discovered was that for the *McCC*$_2$ metric.

### 6.4.3 Discussion of the limitations

Although we carefully designed our experiments, there were some points which could have affected our results and observations. Complexity metrics were computed on a large-scale and data-intensive application, but the results may be affected by coding style and conventions of a single company. Measurements of Magic applications from other domains and developer com-

panies are needed. This applies to the questionnaire as well. The number of participants and given programs should be increased so as to be able to draw more reliable conclusions. Programs were selected by a person, not randomly based on a specific distribution, which could also affect our results. The evaluation of developers' view was done by means of ranking, which results in a loss of information in transforming measured values into ranks. The *EC* value is an average rank given by the developers.

## 6.5 Related work

There are many different approaches available for measuring the complexity of a system at the source code level. First, and still popular complexity measures (McCabe [88], Halstead [58], Lines of Code [3]) were surveyed by Navlakha in [91]. A recent survey which sums up today's complexity measures was published by Yu and Zhou [120]. In a 4GL environment, to our best knowledge there were no previous research study done to measure structural complexity attributes of a Magic application, even though, for other 4GLs, there were some attempts to define metrics to measure the size of a project [115], [118], [81]. There are also some industrial solutions for measuring metrics in other 4GL environments. For instance, *RainCode Roadmap*[1] for Informix 4GL provides a set of predefined metrics for code complexity (number of statements, cyclomatic complexity, nesting level), for SQLs (number of SQL statements, SQL tables, etc.), and for lines (number of blank lines, code lines, etc.). In the world of Magic, there is also a tool for optimization purposes called Magic Optimizer[2], which can be used to perform a static analysis of Magic applications. It does not measure metrics, but it is able to locate potential coding problems that also relate to software quality.

In a 3GL context there are also papers in the literature for analyzing the correlation among certain complexity metrics. For instance, Van der Meulen and Revilla analyzed about 71,917 programs from 59 fields written in C/C++ [113]. Their result showed that there was strong connection between LOC and HCM, and between LOC and CCM. Our study also found similar results, but our research study was performed in a 4GL context with newly adapted complexity metrics. We showed, in addition, that in our context traditional metrics had totally different meanings for the developers.

## 6.6 Conclusions

The main aim here was to adapt common 3GL structural complexity metrics (McCabe's cyclomatic complexity and Halstead's complexity measures) to a popular 4GL called Magic. We dis-

[1] http://www.raincode.com/fglroadmap.html

[2] http://www.magic-optimizer.com/

cussed the special features of Magic and we presented formal definitions of our metrics. After adapting the metrics, we presented a modified version of McCabe's cyclomatic complexity ($McCC_2$), which measured the complexity of a task by aggregating the complexity values of its called subtasks as well. We addressed research questions on whether our new metrics reflect developers' complexity rankings or not.

We designed and carried out an experiment to seek answers to our questions. We found that:

RQ1: Is there a significant correlation among adapted complexity metrics of Magic programs? There was a significant correlation among all the investigated metrics, and there was strong correlation among the Halstead measures, which also correlate with the Total Number of Expressions.

RQ2: Is there a significant correlation between the complexity ranking given by developers and the ranking given by the adapted metrics? The rankings given by adapted metrics had a significant and strong correlation with the rankings given by developers (especially in the case of the $McCC_2$, but not for the *WLUT* metric).

Overall, we found that our modified measure had a strong correlation with the developers' rankings.

# Part IV

# Security and optimization

*"Security is, I would say, our top priority because for all the exciting things you will be able to do with computers - organizing your lives, staying in touch with people, being creative - if we don't solve these security problems, then people will hold back."*

Bill Gates

# 7

# Static security analysis based on input-related software faults

Here, we deal with software systems from another point of view related to data. We present an approach for helping developers locate faults that are related to security by identifying parts of the source code that involve user input. The focus is on the input-related parts of the source code, since attackers commonly exploit security vulnerabilities by passing malformed input data to applications. Mishandling input data can be a source of common security faults in many languages that support pointer arithmetic such as C and C++. Examples of security faults are *buffer overflows*, *format string vulnerabilities*, and *integer overflows* [67]. The best known and, arguably, the most dangerous security faults are caused by buffer overflows, which are described in an article published in 1996 [4], and appeared in the literature as far back as 1988 [42]. This type of vulnerability is still common in software systems and is difficult to locate either automatically or by a manual code review. Another recent study has shown that code defects related to buffer overflows are still frequent in open source projects [34].

In this chapter, we introduce our analysis technique and the results we achieved with it including the metrics and algorithms underlying the analysis. We implemented the technique as a plugin to the CodeSurfer product of GrammaTech Inc. as well and validated it on open source projects. With this technique we successfully identified faults in applications including Pidgin and Cyrus Imapd.

## 7.1 Overview

Now, we provide an overview of the technique employed for a static security analysis based on input-related faults.

### 7.1.1 Technique

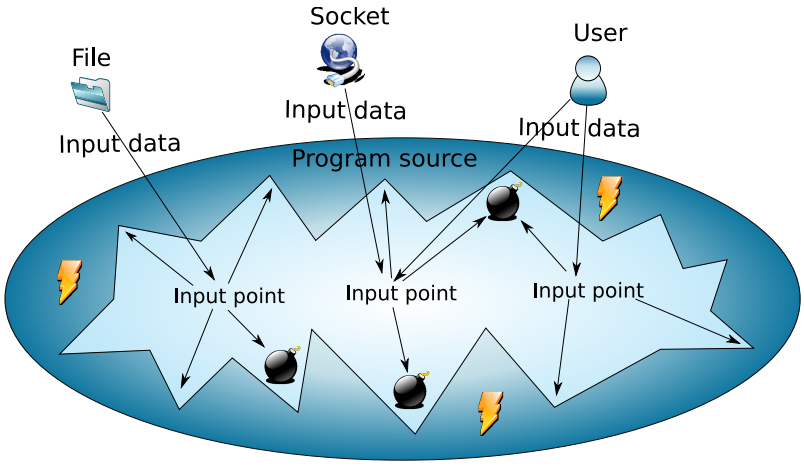



**Figure 7.1.1:** Illustration of input-related security faults. Faults related to user input are marked with 'bombs' indicating vulnerabilities.

In our approach we focus on the input-related parts of the source code, since an attacker can usually take advantage of a security vulnerability by passing malformed input data to the application. If this data is not handled correctly, it can cause unexpected behaviour while the program is running. The path which the data travels through can be tracked using *dataflow analysis* [73] to determine the parts of the source code that involve user input. Software faults can appear anywhere in the source code, but if a fault is somewhere along the path of input data it can act as a 'land mine' for a security vulnerability (see Figure 7.1.1).

The main steps of our approach (Figure 7.1.2) are the following:

1. Find locations in the source code where data is read using a system call of an I/O operation. These calls are marked as *input points*,

2. Get the set of program points involved in user input,

3. Get a list of dangerous functions using metrics,

4. Perform automatic fault detection to find vulnerabilities.

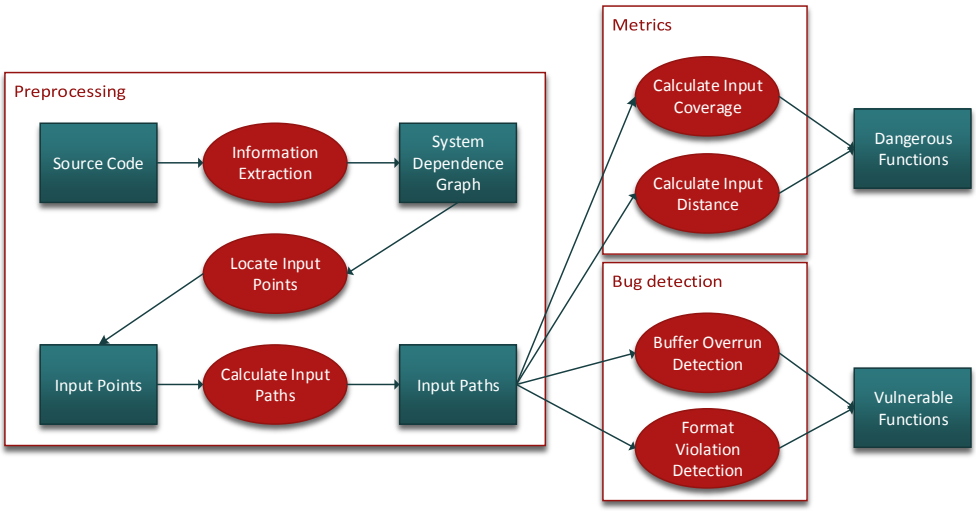



**Figure 7.1.2:** An overview of our approach.

### Locate I/O points

Input data can come from many different sources, not just from the standard input. It may come from input character devices, Internet sockets or files in the file system. In general, *input points* are statements used to read data from an external source by calling a system function to perform an I/O operation. The input data is often a string that is stored in a buffer that has been allocated on the stack or the heap.

### Extract input-related program points

After locating the input points in the source code, we can determine how the input data travels from one statement to another statement. This can be done using *dataflow analysis*, a technique for gathering information about the possible set of values calculated at various points in a program. Once we have the path for all input points, we can determine which parts of the source code involve user input by computing the union of these paths.

To perform a dataflow analysis on C/C++ code, we will use the CodeSurfer tool of GrammaTech Inc.

### Get the list of dangerous functions

We can get a list of functions that warrant an increased scrutiny by determining which parts of the source code involve user input. We will call the list of such functions *dangerous functions*.

To give developers more information about a dangerous function we measure its *coverage* as a percentage of its source code statements that are tainted by user input. We also measure the *distance* in the dataflow graph between the entry point of the function and the origin of the input

data (i.e. the statement where input occurs). These metrics are used to rank the functions in order to identify the functions that are the most tainted by user input.

#### Automatic fault detection

Automatic fault detection is performed by our technique to detect security problems in dangerous functions. These fault detections are based on algorithms that are applied to the code's corresponding data dependence graph and can point to buffer overflow or format string vulnerabilities.

### 7.1.2 CodeSurfer

Our technique is implemented as a CodeSurfer[1] plugin. CodeSurfer is a powerful static-analysis tool for C/C++ programs. This tool was chosen because it is able to create a wide range of intermediate representations [7] for a given program, including: the Abstract Syntax Tree (AST), Call Graph, Interprocedural Control-Flow Graph (CFG), Points-to Graph, set of variables used and modified for each function, the Control Dependence Graph, and the Data Dependence Graph. CodeSurfer can be extended with plugins using its internal scripting language or its C/C++ API.

The most important feature of CodeSurfer for our purposes is that, after a whole-program analysis is performed, it can build a precise *system dependence graph* [66] due to its *pointer-analysis* [6] capability.

### 7.1.3 System dependence graph

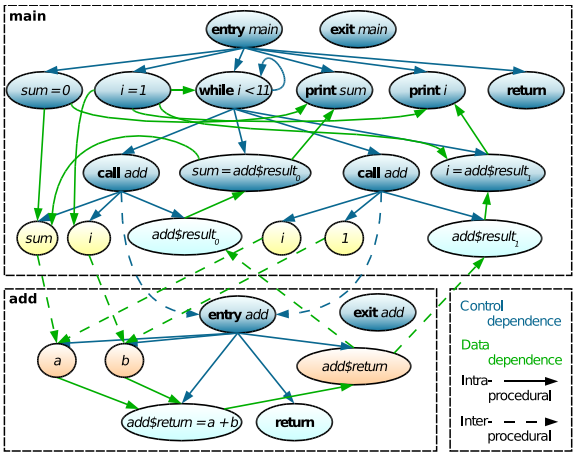
Depending on the the application there are different definitions for *program dependence graph* (PDG) [48, 66, 76]. PDG is a directed graph representation ($G_P$) of a program ($P$), where vertices represent program points (e.g., assignment statements, call-sites, variables, control predicates) that occur in $P$ and edges represent different kinds of control or data dependencies. There is a data dependence edge between two vertices if the first program point can assign a value to a variable that may be used by the second point. There is a control dependence edge between two vertices if the result of executing the first program point controls whether the second point will be executed or not.

A *system dependence graph* (SDG) [66] is the interprocedural extension of the PDG. It consists of interconnected PDGs (one per procedure in the program) and extends the control and data dependencies with interprocedural dependencies. An interprocedural control-dependence edge connects procedure call sites to the entry points of the called procedure and an interprocedural data-dependence edge represents the flow of data between actual parameters and formal parameters (and return values). Globals and other non-local variables such as file statics, and variables accessed indirectly through pointers are treated as additional parameters of procedures.

[1] http://www.grammatech.com

A system dependence graph can be used for many purposes such as code optimization [48], reverse engineering, program testing [10], program slicing [66], software quality assurance [65] and software safety analysis [108].

Here we will use an SDG, and the extracted dataflow information stored in this representation, to determine the paths on which user-related input travels from its input point.




```
int add(int a, int b) {
  return a + b;
}

void main() {
  int sum, i;
  sum = 0;
  i = 1;
  while (i<11) {
    sum = add(sum, i);
    i = add(i,1);
  }
  printf("%d\n", sum);
  printf("%d\n", i);
}
```

**Figure 7.1.3:** CodeSurfer's System Dependence Graph of an example source code [5]. The SDG represents the source code on the right side. Nodes are program points such as call-sites, assignments and return statements, while edges are inter/intra procedural data/control dependencies.

## 7.2 Technique

Now we will describe our technique and present details on how metrics and algorithms are used to locate buffer overflow and format string vulnerabilities.

### 7.2.1 Locating input points

Input points are statements in the source code that perform I/O operations to read data from standard input, network sockets, or files. To locate these statements we look for invocations of I/O functions that are declared in the header files of the *C standard library*. Examples of these function calls are: `fscanf`, `scanf`, `getc`, `gets`, and `read`. We handle 28 function calls as input functions declared in header files such as: `stdio.h`, `stdlib.h`, `unistd.h`, and `pwd.h`.

The `argc` and `argv` parameters of a program's `main` function will also be treated input points, since these parameters relate to user input.

If an input point is a function call, its call-site vertex usually does not have any forward data dependencies in the SDG, as the returned value of the function has a separate node because of interprocedural dependencies. To handle this, in our representation, each input point has a *generate set* containing the nodes that are not connected to the input call-site with data dependence edges, but are directly affected by the I/O operation. For instance, a generate set of a `scanf` call contains the parameter variables of the call site. Generate sets can be used to track the points in the SDG where the content of an input-related buffer is copied into another buffer with standard library functions such as `strcpy` or `strcat`. Since standard library functions are not defined in the user's program code, these functions require special attention for static analyzers. CodeSurfer offers a sophisticated library model to handle common used library functions, but to keep our algorithms general we will follow these operations manually.

### 7.2.2 Metrics

#### Input coverage

*Input coverage* is used to describe the percentage of statements in a function that are tainted by user input. The formal definition is the following:

$$Coverage(f_j) = \frac{|\bigcup_{i=1}^{n} L_{IO}(p_i, f_j)|}{|L(f_j)|}$$

where $p_i$ as a node of the SDG is one of the $n$ input points, $f_j$ is a function of total $m$ functions, $L_{IO}(p_i, f_j)$ is the set of statements in $f_j$ along the forward data dependence chain of $p_i$ input point and $L(f_j)$ is the set of all statements in $f_j$.

The definition can be extended to cover the full source code of a program:

$$Coverage = \frac{\sum_{j=1}^{m} \left( |\bigcup_{i=1}^{n} L_{IO}(p_i, f_j)| \right)}{\sum_{j=1}^{m} |L(f_j)|}$$

It should be mentioned that CodeSurfer's SDG contains many additional nodes (like pseudo variables because of global variables, or split statements because of AST normalization). Additional nodes may be particularly relevant in the case of global variables as described in [15], where the authors state that number of nodes per line of code may vary dramatically because of pseudo variables. Therefore, using the conventional definition of statement coverage would result in false measurements; so instead of calculating statement coverage, we will measure line coverage. The definition of line coverage is the same as that for statement coverage except that $L_{IO}(p_i, f_j)$ stands for the set of lines containing statements in $f_j$ along the path of $p_i$ input point and $L(f_j)$ stands for

the lines of $f_j$.

### Input distance

While input data travels from statement to statement in the control or data flow graphs, the data might be modified and reassigned to new variables. If input data or a variable is modified many times after reading it, developers may handle it less carefully or they may even forget the origin of the actual data. Using dataflow analysis it is possible to tell how many times the input data is modified or gets re-assigned to another variable. Thus, it is possible to compute the distance between an input point and the entry point of a function along the data dependence chains of the input in the SDG. The formal definition is:

$$Distance(p_i, f_j)\text{: number of SDG nodes on the shortest path (only for data dependence edges) from } p_i \text{ input point to the entry point of } f_j \text{ function.}$$

Input data may travel on different paths from its input point to a destination point. Selection statements and loops may cause branches along the path of the input data being investigated. Inside a loop statement the variable that stores the input data may be modified several times and static analyzers cannot determine how many times the loop body will be executed at runtime. To eliminate the effect of loops and branches, we measure the length only for the shortest path in the SDG. An illustration of this is given in Figure 7.2.1.

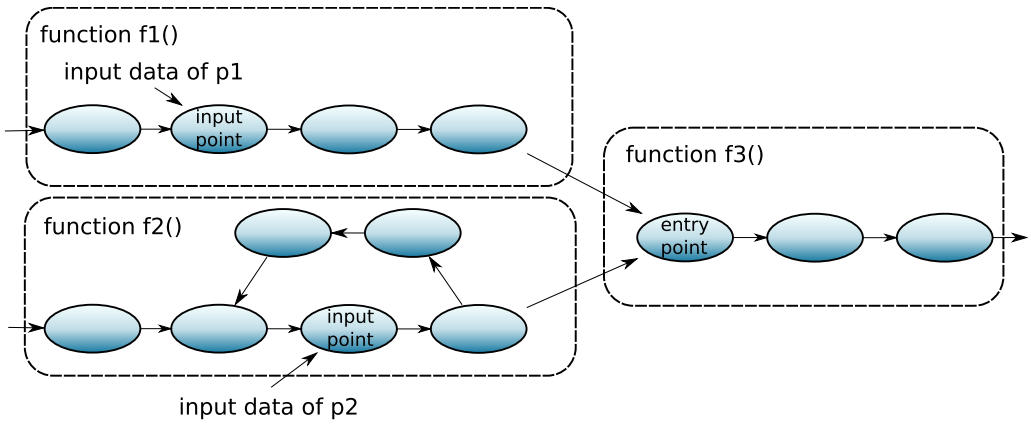



**Figure 7.2.1:** Illustration of the distance metric. The graph is a portion of an SDG showing only data dependence edges. $p_1$ is an input point in function $f_1$ and $p_2$ is an input point in $f_2$. $Distance(p_1, f_3) = 3$, $Distance(p_2, f_3) = 2$. Inside function $f_2$ there is a loop, but for calculating distance we count nodes only along the shortest path.

This metric can be used to answer two questions:

⇒ How far has a datum travelled from its original input point?

⇐ How far away does an input-related function get its input?

### 7.2.3 Fault detection

Using our metrics we can determine a software system's critical (dangerous) functions. These functions, which must be handled more carefully during code inspection and testing, are more likely to contain faults that threaten a system's reliability or security. Once these functions have been determined, we can apply automatic fault detection algorithms on them.

#### Buffer overflow detection with path sensitivity and pattern matching

Livshits and Lam published a technique for locating buffer overflow and format string vulnerabilities in C programs by tracking pointers with path and context sensitivity [79]. The novel aspect of their method was their precise and fast pointer analysis, which made it possible to analyze large projects (the largest project they analyzed was `pcre` with about 13,000 LOC) quickly. With this technique they were able to track the path of input data and warn when the data was written into a buffer with a statically declared size.

Our technique implements a similar fault detection method that uses the SDG extracted from the source code to track the path of input data and, by simple pattern matching, locates `strcpy`-kind functions along the paths that allocate buffers on the stack. These functions, including `strcpy`, `strcat` and `sprintf`, write the contents of the input data into a buffer without first performing bounds checking.

#### Format string vulnerability detection

The recognition of format string vulnerabilities may be similar to buffer overflow faults even if the two different types of vulnerabilities have different technical backgrounds. In the case of a format string fault, a mishandled buffer (related to user input) is used as a format string of a function from the `printf` family. If an attacker can insert special format characters into the format string of a vulnerable function, he may even execute malicious code during program execution. The same technique used to locate buffer overrun errors can be used to locate format string faults. In contrast to looking for function calls of the `strcpy` family, our technique looks for call-sites with system functions of the `printf` family. If the format argument of such a function is related to the user input, it is a potential format string fault, unless the content of the variable was checked earlier.

#### Buffer overflow detection with taintedness checking

After implementing our version of the buffer overflow detection technique, which was based on the algorithm described in [79] (Section 7.2.3), we realized that many of the reported faults were false positives. The most common reason for these false positives was that, before the problematic statement, there were conditions that checked the size of the buffer.

To eliminate these false positives from the set of reported warnings, we extended the algorithm with a technique usually referred to as *taint checking* [74] or *taintedness detection* [119]. The main idea of this technique is to mark the data that may cause errors as *tainted* and follow its state during execution or in the program flow.

Suppose for instance, that we mark the variables or buffers that store data from a user's input as 'tainted'. Whenever the value of the variables or buffers are assigned to a new variable or copied to a new buffer, the new variable or buffer is also marked tainted. Writing the data from a tainted buffer to another buffer without bounds checking is a dangerous operation, that generates a warning. However, if there is a selection statement (e.g. an `if` statement) which checks the size of the tainted buffer before it is copied to a new buffer, this selection statement untaints the copy operation and the new buffer. Tracking these selection statements which untaint other variables and string operations can be performed using control dependencies of the SDG along with data dependencies.

Algorithm 1 gives a formal description of our algorithm, which works as follows: We first get a set of input-related vertices from the SDG and then traverse the data dependence edges to locate `strlen` calls after the input statements. A `strlen` call is not usually used directly in a condition, but its return value is stored in a variable that is used later. To handle these cases, we get the list of variables that depend on the return value of the `strlen` calls and use this set of vertices for taintedness checking. After calculating these sets, we start a pre-order traversal to walk over the control dependencies of the input statements. When we visit a control statement (this is usually an `if` statement, but it can be any kind of selection statement) and the condition of the control statement uses a variable that depends on a `strlen` call, we skip walking over the subtree of this control statement. However, when we visit a call site of a `strcpy`-kind function, and this function was already marked as input-related, we mark this node as a place of potential buffer overflow.

## 7.3 Results

Here, we present results of applying our metrics and algorithms on open source software. We analyzed 12 security-critical products and we scanned a total 811,072 lines of C source code. The largest project we analyzed was `pidgin` with 229,825 lines of code. The full list of projects can be seen in Table 7.3.1.

The projects we analyzed were security critical in the sense that they were common targets of attackers, since most of the projects were ftp, imap or irc daemons. These daemons usually have a user management and sometimes even anonymous users can log in and send input data remotely to the daemon running on a server. If the daemon has a security vulnerability, a malicious user can readily get access to the server.

**Algorithm 1** Buffer overflow detection with taintedness checking (see Section 7.2.3).

**procedure** *DetectBufferOverflow*(*IP*, *V*)
**Input:** *IP* set of input points
**Output:** *V* set of potential buffer overflow statements

```
1:  V ← ∅
2:  InputRelated ← GetDataDeps(IP) {Get the set of input related vertices}
3:  StrLens ← StrLenCalls(IP) {Get the set of strlen calls along the SDG paths starting from points in
    IP and following forward data dependencies}
4:  StrLenVars ← GetDirectDataDeps(StrLens) {Get the variables directly depending on the return
    expressions of calls in StrLens }
5:  NextNodes ← IP {Ordered list of next nodes to visit}
6:  AlreadyVisited ← ∅ {Start a preorder traversal on SDG following forward control dependencies}
7:  while not Empty(NextNodes) do
8:     n ← First(NextNodes)
9:     Remove(NextNodes, n)
10:    if n ∈ AlreadyVisited then
11:       {this node was already visited, simply remove it from NextNodes}
12:    else
13:       if IsControlStatement(n) AND
          UsesVarIn(StrLenVars) AND
          n ∈ InputRelated then
14:          {this node dissolves tainted state of data, skip its subtree}
15:       else
16:          if IsStrcpyCall(n) AND n ∈ InputRelated then
17:             V ← V ⋃ n {This is a potential Buffer Overflow!}
18:          end if
19:          PutFirst(NextNodes, GetDirectControlDeps(n) {Get the next nodes in SDG following for-
             ward control dependencies}
20:       end if
21:       AlreadyVisited ← AlreadyVisited ⋃ n
22:    end if
23: end while
```

Our buffer overflow detection algorithm extended with taintedness checking produced 10 warnings for the systems analyzed (8 times for Cyrus Imapd, 1 for Eggdrop and 1 for Pidgin) and 6 of these warnings belonged to the same statement in a function along different input paths of Cyrus Imapd. We manually evaluated all of these warnings and we found 3 warnings representing real buffer overflow faults: 2 faults in the code of Cyrus Imapd and another one in Pidgin. These faults were not critical security threats as they were not exploitable by malicious remote users, but they could cause unexpected behaviour during program execution. In addition to the warnings of deployed algorithms, we used our metrics for further inspections, as we describe later in a case study (Section 7.3.1).

We note that because these open source projects are common targets of attacks, they are also often the subject of security research and analysis. As a result, common mistakes and errors are

| Project | Description | LOC |
|---|---|---|
| pidgin-2.4.1 | chat client | 229825 |
| cyrus-imapd-2.3.12p2 | imap daemon | 170875 |
| irssi-0.8.12 | irc client | 71253 |
| openssh-5.0p1 | ssh daemon/cl. | 62251 |
| Unreal3.2.7 | irc daemon | 59196 |
| eggdrop1.6.19 | irc robot | 58468 |
| ircd-hybrid-7.2.3 | irc daemon | 53427 |
| proftpd-1.3.2rc1 | ftp daemon | 39122 |
| wzdftp-0.8.3 | ftp daemon | 34897 |
| pure-ftpd-1.0.21 | ftp daemon | 14798 |
| vsftpd-2.0.6 | ftp daemon | 12378 |
| bftpd | ftp daemon | 4582 |

**Table 7.3.1:** List of open source projects that we analyzed.

discovered as soon as a new version is released. Widely used tools such as hybrid ircd or proftpd are also well tested. These projects are good subjects to locate vulnerabilities that are hard to find for static analyzers because the new vulnerabilities detected in this software are real faults that were probably not reported by other analysis tools.

### 7.3.1 Pidgin case study

In this case study we elaborate on the steps of our analysis (metrics, and fault detection) and demonstrate the effectiveness of our approach by applying it to Pidgin, an open source chat client.

#### Overview of Pidgin and the analyzer system

Out of the projects listed in Table 7.3.1 we chose Pidgin as the subject of a case study because of its size and popularity. Pidgin is the largest project we analyzed with 7,173 functions and 229,825 lines of C source code. CodeSurfer requires that it compile the full source code before analysis so it can build a proper ASG of the full project. We compiled Pidgin with the default configure parameters and it took CodeSurfer and GCC 31 minutes to compile and analyze the source code. In addition, our plugin required another minute to compute the metrics and perform fault detections. Compiling Pidgin simply with GCC using the same configuration options took around 8 minutes for the same system. Our analysis was conducted on an Intel Dual Xeon 2.8Ghz system with 3G memory and Ubuntu 8.04 (Hardy Heron) installed on it. CodeSurfer's settings for the analysis were the default settings, but we used `-cfg-edges both` and `-basic-blocks yes` to have more detailed control flow information for further manual inspection.

### Input points

In Pidgin we found 99 input points in total (Table 7.3.2). Most of these were `read` calls used to read a buffer from a file descriptor, but there were many `fread` and `fgets` functions as well. We did not find any dangerous input functions like `gets` or `scanf(``%s'', str)`.

| Name | Occurrences |
|---|---|
| read() | 55 |
| fread() | 12 |
| fgets() | 10 |
| gg_read() | 9 |
| gethostname() | 6 |
| getpwuid() | 2 |
| fscanf() | 1 |
| getenv() | 1 |
| getpass() | 1 |
| char *argv[] | 1 |
| int argc | 1 |

**Table 7.3.2:** Input points in Pidgin. The first column lists the name of input statement, while the second column lists the number of occurrences of the actual statement. (Here, `gg_read()` is an internal function to read data from SSL sockets.)

### Metrics

The total input coverage of the source code is 10.56%, while the mean value of the input coverage for all functions is 9.67%. The function with the highest coverage has an input coverage of 84.62%, while 2,728 functions involve user input. The set of functions with top 10 coverage values is shown in Table 7.3.3.

After manual inspection we noticed that most of these functions were used to clean up the memory after using the input related buffers (their naming conventions show this: `*_free`, `*_destroy`) and we did not find faults in these functions. However, we saw that besides the top functions there are 379 functions with at least 50% coverage and only 65 of them had over 20 lines of code. We did not inspect all of these functions, but we noticed that with our technique we could significantly reduce the number of functions that required inspection during a code review.

The longest distance an input travels from its input point through the dataflow is 100 vertices, which is high compared to the other analyzed projects (the average value of the longest distances for all the projects analyzed was 44.08 and 9 projects of 12 were below 50). In this case, the input statement was a `fgets` system call that read a buffer with a limited size from an input file. Along the path of the same input there was a total of 365 functions involved. This distance is high even

| function | lines | coverage (%) |
|---|---|---|
| yahoo_roomlist_destroy | 12 | 83.33 |
| aim_info_free | 13 | 84.62 |
| s5_sendconnect | 22 | 77.27 |
| purple_ntlm_gen_type1 | 35 | 77.14 |
| gtk_imhtml_is_tag | 91 | 76.92 |
| jabber_buddy_resource_free | 25 | 72.00 |
| peer_oft_checksum_destroy | 8 | 75.00 |
| qq_get_conn_info | 12 | 75.00 |
| _copy_field | 8 | 75.00 |
| qq_group_free | 8 | 75.00 |

**Table 7.3.3:** List of top ten input coverage values of functions in Pidgin.

inside the project itself, as the average of longest distances for different input points (average of $max\{Distance(p_i, f_1), \ldots, Distance(p_i, f_n)\}$ values for all $p_i$) is only 12.98 in Pidgin.

There is another interesting top value in Pidgin related to the function that is related to input data of most input points (it can be calculated by counting $p_i$ points with $Distance(p_i, f_j) > 0$ values for all $f_j$ functions and taking the maximum of these values). In Pidgin we found a function that works with input data coming from 31 different input points in the source code. This function is called `gg_debug` and is used for internal debugging purposes, which explains the high number of related input points since the function is called with string parameters in many different contexts.

### Fault detection

We analyzed Pidgin with the format string detection (Section 7.2.3) and the buffer overflow detection with taintedness checking algorithms (Section 7.2.3). The evaluated algorithms produced only one warning on file `libpurple/protocols/zephyr/ZVariables.c` for a `strcpy` function call. After manual inspection we found that this call was a fault that was related to a buffer overflow. The fault can be seen in Figure 7.3.1.

In the source code snippet in Figure 7.3.1 our fault detection produced a warning for the `strcpy` call in line 136 of function `get_localvarfile`. This function copies the content of the buffer `pwd->pw_dir` into the destination buffer pointed by `bfr` without performing bounds checking. Here, `bfr` is a pointer parameter of this function, but when the function is called from `ZGetVariable`, `bfr` points to `varfile` which is a statically allocated string buffer with maximum of 128 characters (line 28). The content of `pwd->pw_dir` is set in line 132, and it contains the name of the home directory of the current user. If the length of this directory name exceeds 128 characters, the `strcpy` call produces a *segmentation fault*.

```
25: char *ZGetVariable(var)
26:     char *var;
27: {
28:     char varfile[128], *ret;
29:
30:     if (get_localvarfile(varfile))
31:         return ((char *)0);
...
42: }
...
114: static int get_localvarfile(bfr)
115:     char *bfr;
116: {
117:     const char *envptr;
118: #ifndef WIN32
119:     struct passwd *pwd;
120:     envptr = purple_home_dir();
121: #else
...
127: #endif
128:     if (envptr)
129:         (void) strcpy(bfr, envptr);
130:     else {
131: #ifndef WIN32
132:         if (!(pwd = getpwuid((int) getuid()))) {
133:             fprintf(stderr, "Zephyr ....");
134:             return (1);
135:         }
136:         (void) strcpy(bfr, pwd->pw_dir);
137: #endif
138:     }
...
143: }
```

**Figure 7.3.1:** A buffer overflow fault in Pidgin. Vulnerable `strcpy` is in line 136 of file libpurple/protocols/zephyr/ZVariables.c.

## 7.4 Related work

Many static analysis tools have appeared to help companies develop more reliable and safe systems by automating testing, code review, and source code auditing processes during the development cycle. There are different solutions for different languages like CodeSonar tool of GrammaTech and PCLint[2] for C/C++, CheckStyle[3] or PMD[4] for Java or FXCop developed by Microsoft for C# and there are multi front-end solutions like Columbus developed by FrontEndART Ltd. Many of these tools are able to find certain rule violations and they can show potential faults to developers, but only some of them are able to locate security faults. Chess and McGraw published a brief overview of security analysis tools and compared their benefits in a paper [23], while Tevis and Hamilton published a similar comparison for security tools [111].

Since Aleph1 published his paper describing an exploiting technique [4] against buffer over-

[2]http://www.gimpel.com/

[3]http://checkstyle.sourceforge.net/

[4]http://pmd.sourceforge.net/

flow vulnerabilities, researchers have published many methods for detecting these kinds of vulnerabilities. Most of these approaches work with dynamic bounds checking techniques [40, 77, 103, 122]. Static techniques were also published based on integer-range analysis [116], annotation-assisted static analysis technique [45] or on pointer analysis techniques [8, 79]. However, tests and comparisons of these techniques and their related tools indicate that it is still hard to locate these kinds of errors and often these techniques still give many false positive warnings [122]. A comparison of available exploiting, defending and detecting techniques was published in [78].

Focusing on user-related input is an important idea behind static security analyzers and it is also common to work with a graph representation that can be used to track control and data dependencies. Scholz et al. described an approach in [105] to identify security vulnerabilities via user-input dependence analysis. In their technique they mapped a user-input dependency test to a graph reachability problem that can be solved with simple graph traversal algorithms. Hammer et al. presented another notable technique in [59] which is closely related to our work since they perform security analysis based on PDGs. Their work is about information flow control, which is a technique for discovering security leaks in software. This technique is closely related to tainted variable analysis.

Our approach uses an input coverage metric to show developers which functions involve user input. Using coverage metrics is common in testing and there are tools that can measure this value at run time. However, our approach computes this metric statically from the SDG of the program. Our coverage metric can be also viewed as a coupling metric that measures the coupling of functions to the input variables in the source code. A similar idea was presented in [62], where the authors propose a coupling metric to measure how information flows between functions.

## 7.5 Conclusions

Locating security faults in complex systems is difficult and there are relatively few effective automatic tools available to help developers. Here, we presented an approach to automatically locate input-related security faults (buffer overflow and format string vulnerabilities) and help developers locate security vulnerabilities by marking parts of the source code that involve user input. Our technique has three main steps: (1) locate input points, (2) calculate metrics to determine a set of dangerous functions, (3) perform automatic fault detection to identify security faults.

We presented the results of applying our technique on open source software and described a case study on Pidgin as the largest and most popular application we analyzed. We found security faults in Pidgin and in other software analyzed as well. Our fault detection techniques focused on buffer overflows and format string vulnerabilities, which were the most common input-related faults in C source code. Our approach is novel as it uses input coverage and distance metrics to provide developers with a list of functions that are the most likely to contain potential security

faults. The Pidgin case study demonstrated the effectiveness of our metrics. Pidgin has a total number of 7,173 functions and 229,825 LOCs. Based on our measurements, just 10.56% of the code is related directly to user input and 2,728 functions work with input data. Limiting the number of dangerous functions and code that is affected by user input will undoubtedly help to reduce the effort of a code review.

*"Normal people... believe that if it ain't broke, don't fix it. Engineers believe that if it ain't broke, it doesn't have enough features yet."*

Scott Adams

# 8

# Optimizing information systems: code factoring in GCC

GCC (GNU Compiler Collection)[1] is a compiler with a set of front ends and back ends for different languages and architectures. It is part of the GNU project and it is free software distributed by the Free Software Foundation under the GNU General Public License and GNU Lesser General Public License. As the official compiler of Linux, BSDs, Mac OS X, Symbian OS and many other operating systems, GCC is the most common and most popular compilation tool used by developers. It supports many architectures and it is widely used for mobile phones and other handheld devices. When compiling a software for devices like mobile phones, pocket PC's and routers where the storage capacity is limited a very important feature of the compiler is being able to provide the smallest binary code where possible. GCC already contains code size reducing algorithms, but since in special cases the amount of free space saved may be very important, other optimization techniques may be very useful.

In our initial work we transform the ASG of Columbus to the Tree intermediate language of GCC [124]. During this study, we recognized the power of the optimization jobs in GCC, so here, we introduce new algorithms based on code factoring. Developers recognized the power in code factoring methods and nowadays several applications use these algorithms for optimization purposes. One of these real-life applications is called 'The Squeeze Project' maintained by

[1] http://gcc.gnu.org/

Saumya Debray[2], which was one of the first projects using this technique. Another application is called *aiPop* (Automatic Code Compaction software), which is a commercial program released by the *AbsInt Angewandte Informatik GmbH* with 'Functional abstraction (reverse inlining) for common basic blocks' feature[3]. This application is an optimization software package with support for C16x/ST10, HC08 and ARM architectures and it is used by SIEMENS as well.

In this chapter, we will give an overview of code factoring algorithms on different *intermediate representation languages* (IL) of GCC. The main idea behind this approach was introduced in the *GCC Summit* by Lóki et al. [80], and since then techniques have represented a class of new generation in code size optimization. Naturally, we thought that the implementation on higher level IL should lead to better results. We tested our implementation and measured the new results on CSiBE [12], which is the official Code-Size Benchmark Environment of GCC containing about 18 projects with roughly 51 MB of source code in total.

## 8.1 Overview

*Code factoring* is a class of useful optimization techniques specially developed for code size reduction. These approaches seek to reduce size by restructuring the code. One possible factoring method is to make small changes on local parts of the source, which is called local factoring. Another way is to abstract parts of the code and separate them into new blocks or functions. This technique is called *sequence abstraction* or *functional abstraction*. Both cases can work on different representations and can be adapted to these languages.

GCC offers several *intermediate languages* for optimization methods (Figure 8.1.1). Most of the optimizing transformations were implemented on the *Register Transfer Language* (RTL) [49] in previous versions of GCC. RTL is a very low level language where the instructions are described one by one. It is so close to assembly that the final assembler output is generated from this level. Before the source code is transformed to this level, it passes higher representation levels too. First, it is transformed to GENERIC form, which is a language-independent abstract syntax tree. This tree will be lowered to GIMPLE, which is a simplified subset of GENERIC. It is a restricted form where expressions are broken down into a 3-address form.

Since many optimizing transformations require higher level information on the source code that is difficult, or in some cases even impossible to obtain from RTL, GCC developers introduced a new representation level called Tree-SSA [93, 94] that was based on the *Static Single Assignment* form developed by researchers at IBM in the 1980s [35]. This IL is especially suitable for optimization methods that work on the *Control Flow Graph* (CFG), which is a graph repre-

[2]http://www.cs.arizona.edu/projects/squeeze/

[3]http://www.absint.com/aipop/

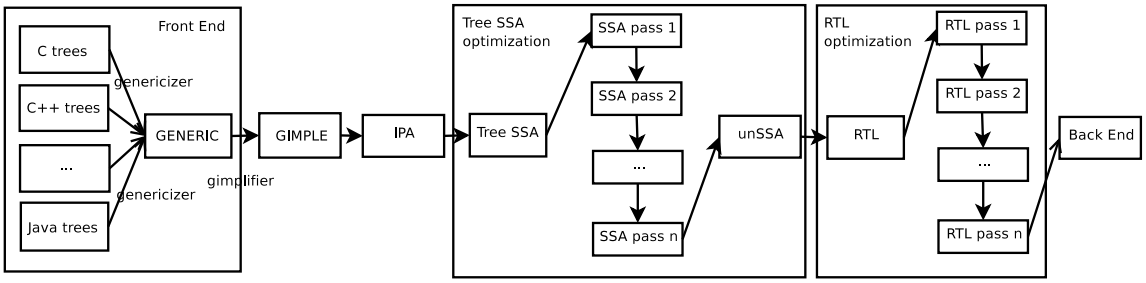



**Figure 8.1.1:** Optimization passes in GCC.

sentation of the program containing all paths that might be traversed during the execution. The nodes of this graph are *basic blocks* where one block is a straight-line sequence of the code with only one entry point and only one exit. These nodes in the flow graph are connected via directed edges and these edges are used to represent jumps in the control flow.

It may be possible to run one algorithm on different representations together (e.g. first on SSA level and later on RTL). Due to the different types of information stored by various ILs, it is possible that after running the algorithm on a higher level it will still find optimizable cases on a lower level as well. For this reason, we got our best results by combining these refactoring algorithms on all optimization levels implemented by us.

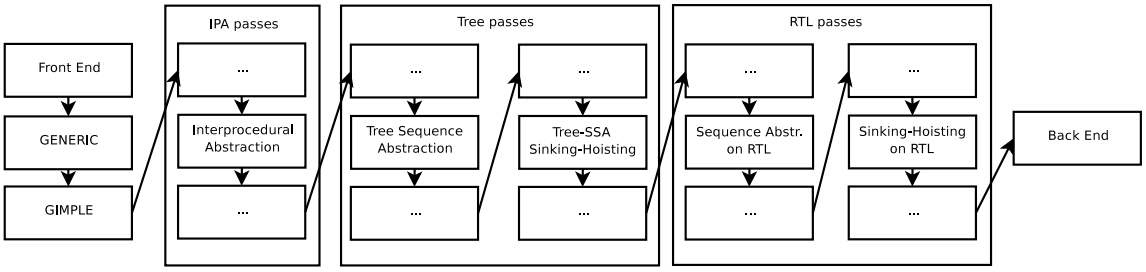



**Figure 8.1.2:** An overview of the implemented algorithms.

We implemented the given code factoring algorithms on the Tree-SSA and on the RTL levels as well, and for the sequence abstraction algorithm we implemented an interprocedural version too. Figure 8.1.2 shows our new passes.

## 8.2 Sinking-Hoisting

The main idea behind *local factoring* (also called *local code motion*, *code hoisting* or *code sinking*) is quite simple. Since it often happens that basic blocks with common predecessors or successors contain the same statements, it might be possible to move these statements to the parent or child blocks.

For instance, if the first statements of an `if` node's `then` and `else` blocks are identically the same, we can easily move them before the `if` node. With this shifting - called code hoisting -

we can avoid unnecessary duplication of statements in the CFG. This idea can be extended to other more complicated cases as not only an `if` node, but a `switch` and a source code with strange `goto` statements may contain identical instructions. Furthermore, it is possible to move the statements from the `then` or `else` blocks after the `if` node too. This is called code sinking, which is only possible when there are no other statements that depend on the shifted ones in the same block.

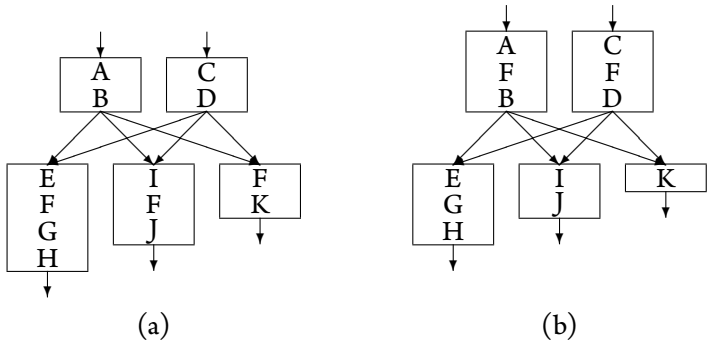



**Figure 8.2.1:** Basic blocks with multiple common predecessors (a) before and (b) after local factoring.

When making this code shift, we must collect basic blocks with common predecessors or common successors (also called same parents or same children) (see Figure 8.2.1). These basic blocks represent a sibling set and we have to look for common statements inside them. If a statement appears in all the blocks of the sibling set and it does not depend on any previous statement, it is a *hoistable statement*; or if it has no dependency inside the basic blocks, it is a *sinkable statement*. When the number of blocks inside the sibling set is bigger than the number of parents (children) it is worth hoisting (sinking) the statement (see Figure 8.2.2).

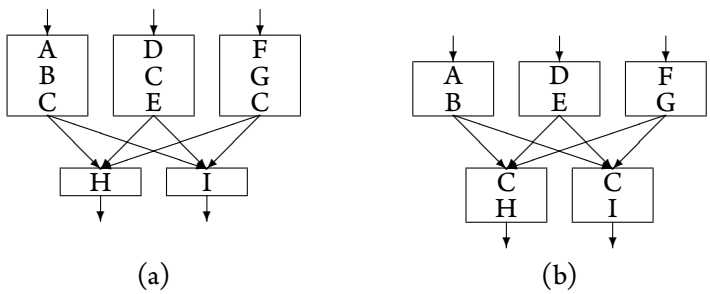



**Figure 8.2.2:** Basic blocks with multiple common successors (a) before and (b) after local factoring.

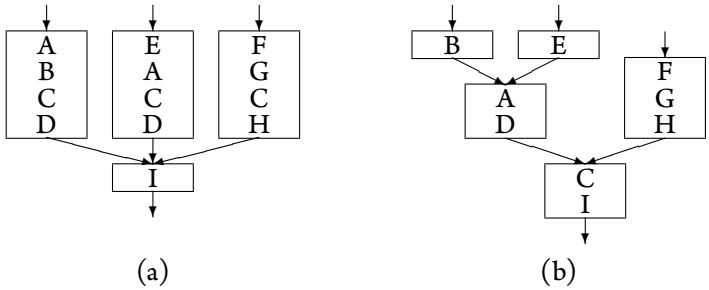



**Figure 8.2.3:** Basic blocks with multiple common successors, but only partially common instructions (a) before and (b) after local factoring.

To get a better size reduction, we can handle a special case of local factoring where there are identical statements not appearing in all the basic blocks of the sibling set. When the number of blocks counting these statements is quite big and we could sink or hoist these statements if desired, it is still possible to simplify the CFG (Figure 8.2.3). For instance, we can create a new block and link it before or after the sibling set, depending on the direction of the shift. By building correct edges for this new basic block we can rerun the algorithm on the new sibling set with the same statements, and move the identical statements to our new basic block. This way the gain may be a slightly less because building a new basic block requires some extra cost for the new statements. However, this way we can have a more efficient algorithm in terms of code size optimization.

### 8.2.1 RTL CODE MOTION

We first implemented this algorithm on the RTL level of GCC's optimization phases. This internal representation language is a low level representation of the source code, very close to assembly. Therefore, when we are thinking of movable statements, these statements are assembly-like statements where we are working with registers, memory addresses, and so on.

*RTL expressions* (*RTX*, for short) are small instructions classified by *expression codes* (*RTX codes*). Each RTX has an expression code and a number of arguments. The number of these small instructions is quite big compared to other representations. Owing to this, the number of potentially movable instructions is the biggest on this level. Although it has an influence on the compilation time too, it is not relevant because the instructions must be compared to each other only for local parts of the CFG where we do code factoring. The asymptotic complexity of this algorithm is $O(n^2)$ (where $n$ is the number of instructions). The reason for this complexity is that the comparison of all instructions is made one by one in the basic blocks of a sibling set in order to find identical statements.

One important advantage of the hoisting-sinking algorithm running on this representation level is the definition for 'identically same' statements. It is evident that equal instructions must

have the same RTX code, but because of the low level of RTL, the arguments must be exactly equal too. To decide if one statement is movable, we have to check its dependencies too. We simply have to check the statements before the current instruction for hoisting and do the same checking with the instructions after the movable statement until the end of the current basic block for sinking.

With this implementation, we managed to achieve a code size reduction of 4.31% (compared to '-Os') on a file in the CSiBE environment. Compiling *unrarlib-0.4.0* for the *i686-elf* target we achieved an average code-size saving of 0.24%. The average value measured on CSiBE was about 0.19% for the same architecture.

### 8.2.2 Tree-SSA code motion

*Tree-SSA* [93, 94] is a higher level representation than RTL. This representation is similar to GENERIC (a language independent code representation), but it contains modifications to represent the program so that every time a variable is assigned in the code, a new version of the variable is created. Every variable has an SSA_NAME that contains the original name for the variable and an SSA_VERSION to store the version number for it. Usually, in control flow branches it is not possible at compile time (only at run time) to decide which previous version of a variable will be taken. In order to handle these situations in SSA form, *phi nodes* [93] are created to help us follow the lifecycle of a variable. A phi node is a special kind of assignment with operands that indicate which assignments to the given variable reach the current join point of the CFG. For instance, consider the use of variable $a$ (from example Figure 8.2.4/a) with version number 9 and 8 in the `then` and `else` cases of an `if` node. Where $a$ is referenced after the join point of `if`, the corresponding version number is ambiguous; and to handle this ambiguity a phi node is created as a new artificial definition of $a$ with version number 1.

The SSA form is not suitable for handling non-scalar variable types like structures, unions, arrays and pointers. For instance, for an $M[100][100]$ array it would be nearly impossible to keep track of 10000 different version numbers or to decide whether $M[i][j]$ and $M[l][k]$ refer to the same variable or not. To overcome these problematic cases, the compiler stores references to the base object for non-scalar variables in *virtual operands* [93]. For instance, $M[i][j]$ and $M[l][k]$ are treated as references to $M$ in virtual operands.

At this optimization level, we should redefine 'identically same statements' because we should not use the strict definition we used before, where we expected from two equal statements for each argument to be equal as well. Due to the strict definition, $a_{x1} = b_{y1} + c_{z1}$ differs from $a_{x2} = c_{z2} + b_{y2}$. However, these kinds of assignments are identical because '+' operand is commutable and SSA versions are not different for our cases. At the SSA level, we should define two statements to be equal when the TREE_CODE of the statements are equal, and if their arguments are variables, their SSA_NAME operands are equal too (version number may differ). We require non-variable

arguments to be exactly the same, but if the statements are commutable, we check the arguments for different orders as well.

In the case of dependency checking, Tree-SSA stores the immediate uses of each statement in a list that we can walk through using an iterator macro. Thanks to this representation, the dependency check is the same as that before in the RTL phase.

```
 int D.1770;
<bb 0>:
 if (a_2 > 100) goto <L0>;
 else goto <L1>;
<L0>:;
 a_9 = b_5 + a_2;
 c_10 = a_9 * 10;
 a_11 = a_9 - c_10;
 goto <bb 3> (<L2>);
<L1>:;
 a_6 = a_2 + b_5;
 c_7 = a_6 * 12;
 a_8 = a_6 - c_7;
 # a_1 = PHI <a_11(1), a_8(2)>;
<L2>:;
 D.1770_3 = a_1;
 return D.1770_3;
```

(a) Before code motion

```
 int D.1770;
<bb 0>:
 a_16 = a_2 + b_5;
 if (a_2 > 100) goto <L0>;
 else goto <L1>;
<L0>:;
 c_10 = a_16 * 10;
 goto <bb 3> (<L2>);
<L1>:;
 c_7 = a_16 * 12;
 # a_14 = PHI <a_16(1), a_16(2)>;
 # c_15 = PHI <c_10(1), c_7(2)>;
<L2>:;
 a_1 = a_14 - c_15;
 return a_1;
```

(b) After code motion

**Figure 8.2.4:** An example code for Tree-SSA form with movable statements.

When moving statements from one basic block to another, we need to pay attention to the phi nodes, the virtual operands and the immediate uses. Typically, a variable of a sinkable assign statement's left hand appears inside a phi node (example Figure 8.2.4). In these situations, after copying the statement to the children or parent blocks we must recalculate the phi nodes. Afterwards, in both sinking and hoisting cases, we must walk over the immediate uses of the moved statements and rename the old defined variables using the new definitions.

The current implementation has a weakness in moving statements with temporary variables. This problem occurs when an assignment with more than one argument is transformed to the SSA form. The problem is that in the SSA form one assignment statement may contain one operand on the right hand side, and when GCC splits a statement with two or more operands, it creates temporary variables that have unique SSA_NAMEs containing a creation id. As we said above, two variables are said to be equal when their SSA names are equal. Consequently, these statements will not be recovered as movable statements. Since these kinds of expressions are often

used by developers, solving this problem in the implementation phase we can get better results in size reduction.

On Tree SSA, with this algorithm the best result we could attain was a 10.34% code saving on a file compiled to the ARM architecture. By compiling the *unrarlib-0.4.0* project in CSiBE for the *i686-elf* target, we were able to achieve an extra code saving of 0.87% compared to '-Os' optimizations and the average code size reduction measured for the same target was about 0.1%.

## 8.3 Sequence abstraction

*Sequence abstraction* (also known as *procedural abstraction*) in contrast to local factoring works with whole *single-entry single-exit* (SESE) code fragments, not just with single instructions. This technique is based on finding identical regions of code which can be turned into procedures. After creating the new procedure we can simply replace the identical regions with calls to the newly created subroutine.

There are well-known existing solutions [32, 38] today, but these approaches can only deal with the kind of code fragments that are either identical or equivalent in some sense or can be transformed with register renaming to an equivalent form. However, these methods fail to find an optimal solution for the cases where an instruction sequence is equivalent to another one, while a third one is only identical with its suffix (Figure 8.3.1/a). The current solutions overcome this problem in two possible ways. One way is to abstract the longest possible sequence into a function and leave the shorter one unabstracted (Figure 8.3.1/b). The second way is to turn the common instructions in all sequences into a function and create another new function from the remaining common part of the longer sequences (Figure 8.3.2/c). This way, we can deal with the overheads of the inserted extra call/return code as well.

Our approach was to create *multiple-entry single-exit* (MESE) functions in the cases described above. Doing this, we allow the abstraction of instruction sequences of differing lengths. The longest possible sequence will be chosen as the body of the new function, and according to the length of the matching sequences we define the entry points as well. The matching sequence will be replaced with a call to the appropriate entry point of the new function. Figure 8.3.2/d shows the optimal solution for the problem depicted in Figure 8.3.1/a.

Sequence abstraction has some performance overheads with the execution of the inserted call and the return code. Moreover, the size overheads of the inserted code must also be taken into account. The abstraction will only be carried out if the gain resulting from the elimination of duplicates exceeds the loss arising from the insertion of extra instructions.

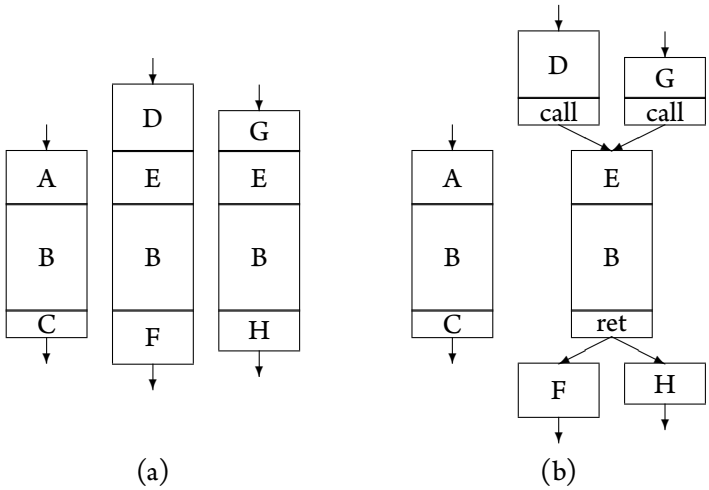



**Figure 8.3.1:** Abstraction of (a) instruction sequences of differing lengths to procedures using the strategy for abstracting only the longest sequence (b). Identical letters denote identical sequences.

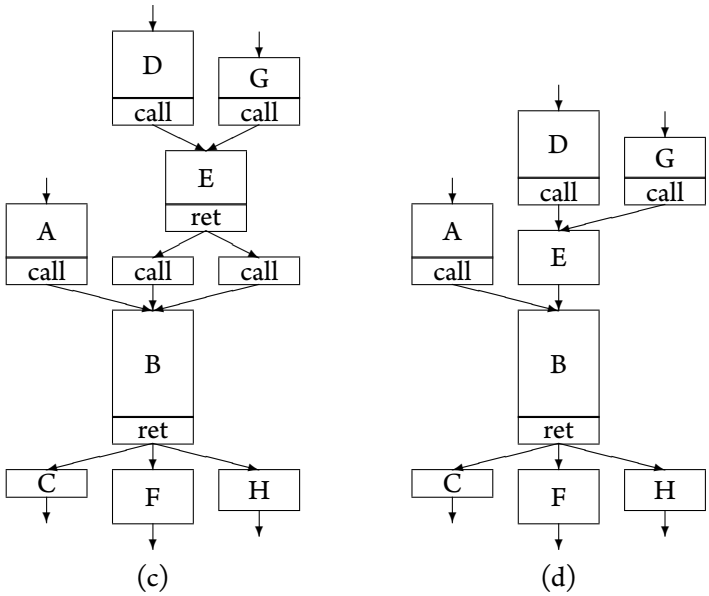



**Figure 8.3.2:** Abstraction of instruction sequences from Figure 8.3.1 of differing lengths to procedures using different strategies (c,d). Identical letters denote identical sequences.

### 8.3.1 Sequence abstraction on RTL

Using the RTL representation algorithms we can optimize just one function at a time. Although sequence abstraction is inherently an interprocedural optimization technique, it can be adapted to intraprocedural operations. Instead of creating a new function from the identical code fragments, one representative instance of them has to be retained in the body of the processed function, and all the other occurrences will be replaced by the code transferring control to the retained instance.

However, to preserve the semantics of the original program, the point where the control has to be returned after the execution of the retained instance must be remembered somehow, hence the subroutine call/return mechanism has to be mimicked. In the current implementation, we use labels to mark the return addresses, registers to store references to them, and jumps on registers to transfer the control back to the 'callers'.

Implementing sequence abstraction in the RTL phase has the benefit of reducing the code size instead of implementing the abstraction on a higher level intermediate language. Most of the optimization algorithms are terminated when the sequence abstraction is initiated in the compilation queue. Those very few algorithms which are executed after sequence abstraction do not have or only have a very small impact on code size. So our algorithm still has a noticeable effect on the output before it is generated by GCC.

The current implementation can only deal with identical statements where the registers and the instructions are exactly the same. For further improvements, with some extra cost it might be possible to abstract identical sequences where the registers may differ.

This implementation has approximately $O(n^2)$ cost of running time. The reason for this is the comparison of possible sequences inside basic blocks with $n$ instructions on the current IL. This cost can be optimized to $O(n \log n)$ using hashtables with fingerprints.

This algorithm gave us maximum code saving of 45.69%, on a source file compiled in CSiBE. Compiling the *libmspack* project of CSiBE for the *arm-elf* target, we got an additional extra code saving of 2.39% compared those for '-Os' optimizations, and our average result in size reduction measured for the same target was about 1.01%.

### 8.3.2 Sequence abstraction on Tree

As a general rule in compilers, one command in a higher intermediate language representation might describe more architecture-dependent instructions in a lower IL. In our view, if the sequence abstraction algorithm can merge similar sequences in a higher IL, it could lead to a better code size reduction.

In Tree IL, there are fewer restrictions than in RTL. For instance, we do not have to worry about register representations. So the algorithm is able to find more sequences as good candidates for abstraction, while in RTL we must be sure that all references to registers are the same in each subsequence.

Unfortunately, the results did not come up to our expectations. The main problem was that the sequence abstraction algorithm is in a very early stage of compilation passes. Other algorithms followed by abstraction may simply undermine the results. This is supported by the fact that after our pass there, we managed to achieve a 9.25% (2.5% on average) code size reduction counted in Tree units. In addition, there are some cases where the abstraction does the merge, but it is not really desirable because one or more sequences are dropped (for example with dead code),

or there is a better solution for code optimization. These cases mostly occur when the algorithm tries to merge short sequences.

Still, for a better performance there are other improvements possible for the current implementation. One of them might be to extend the current implementation with the ability to abstract approximately equal sequences as well. The current implementation realizes abstractable sequences where the statements are equal, but for a few cases it might be possible to handle approximately the same sequences as well. Another possibility is to compare temporary variables as well. It is exactly the same problem as that described before for code motion on the Tree-SSA level (see Section 8.2.2).

This implementation, similar to that for RTL, has a run time cost of about $O(n^2)$ for the same reason.

With this optimization method we managed to achieve a maximum code-size saving of 41.60% on a source file of CSiBE, and by compiling *flex-2.5.31* project of the environment for the *arm-elf* target, we achieved an additional code saving of 3.33% compared to that for '-Os' optimizations. The average result in size reduction measured for the same target was about 0.73%.

### 8.3.3 Procedural abstraction with IPA

The main idea behind *interprocedural analysis* (IPA) optimizations is to produce algorithms which work on the entire program, across procedure or even file boundaries. For a long time, the open source GCC had no powerful interprocedural methods because its structure was optimized for compiling functions as units. The new IPA framework and passes were introduced by the *IPA branch* [68].

We also implemented the interprocedural version of our algorithm. This implementation is very similar to the one we use on Tree, but with one big difference: we can merge sequences from any functions into a new real function.

With this approach we can handle more code size overheads coming from function call API than the other two cases described above. In addition, we are also able to merge the types of sequences that use different variables or addresses, because it is possible to pass the variables as parameters for our newly created function. The procedural abstraction identifies and compares the structure of the sequences to find good candidates for the abstraction, and with this method we are able to merge sequences more efficiently than that using the above-mentioned implementations.

In spite of these advantages, there are disadvantages as well. The IPA is also in a very early stage in compilation passes, so there is a possibility that other algorithms would be able to optimize the candidates better than procedural abstraction (the same cases as in sequence abstraction on Tree). The other disadvantage is that the function call requires many instructions and estimating its cost is difficult. This means that we can only guess the gain we can obtain using a given abstraction,

because determining how many assembly instructions there are in a higher level statement is not possible. This cost computation problem also exists at the Tree level too. The guesses will not be as accurate as the calculation of the gain on the RTL level, and for several cases where we could save code size this may mean that the algorithm does no actual abstraction task.

Our implementation of this algorithm has a cost of running time about $O(n \log n)$, as a comparison of possible sequences is realized using hashtables. This may be compared with the slower $O(n^2)$ implementations for other representations in Table 8.4.3.

With IPA our best result on a source file of CSiBE was a code saving of 59.29% compared to that for '-Os'. By compiling the *zlib-1.1.4* project of the environment for the *arm-elf* target, we got an average code saving of 4.29% and the average value measured for CSiBE was about 1.03%.

## 8.4 Experimental evaluation

For the implementation, we used the GCC source taken from the repository of the *cfo-branch*[4]. This source is a snapshot taken from GCC version 4.1.0. Our algorithms are publicly available in the same repository as well.

We used CSiBE v2.1.1[5] as our testbed, which is an environment developed especially for code-size optimization purposes in GCC. It contains small and commonly used projects like zlib, bzip, parts of Linux kernel, parts of compilers and graphic libraries. CSiBE is a very good testbed for compilers and it can be used not just for measuring code size results, but also for validating compilations.

Our testing system was a dual Xeon 3.0 Ghz PC with 3 Gbyte memory and a Debian GNU/Linux v3.1 operating system. To perform tests on different architectures, we crosscompiled GCC for *elf* binary targets on common architectures like *ARM* and *SH*. For crosscompiling, we used binutils v2.17 and newlib v1.15.0.

The results in Table 8.4.1 and Table 8.4.2 indicate that these algorithms are really efficient methods for code size optimization, but on higher level intermediate representation languages further improvements may be required to get a better performance due to the above-mentioned difficulties. For the *i686-elf* target, by running all the implemented algorithms as an extension to the '-Os' flag, we managed to achieve maximum average code-size savings of 57.05% and 2.13%, respectively. This code-size saving percentage was calculated by dividing the size of the object (compiled with given flags) by the size of the same object compiled with the '-Os' flag. This value was subtracted from 1 and converted to a percentage score by multiplying it by 100.

[4]http://gcc.gnu.org/projects/cfo.html

[5]http://www.csibe.org/

| Flags | i686-elf | | arm-elf | | sh-elf | |
|---|---|---|---|---|---|---|
| | size (byte) | saving (%) | size (byte) | saving (%) | size (byte) | saving (%) |
| -Os | 2900177 | | 3636462 | | 3184258 | |
| -Os -ftree-lfact -frtl-lfact | 2892432 | 0.27 | 3627070 | 0.26 | 3176494 | 0.24 |
| -Os -frtl-lfact | 2894531 | 0.19 | 3632454 | 0.11 | 3180186 | 0.13 |
| -Os -ftree-lfact | 2897382 | 0.10 | 3630378 | 0.17 | 3179622 | 0.15 |
| -Os -ftree-seqabstr -frtl-seqabstr | 2855823 | 1.53 | 3580846 | 1.53 | 3149822 | 1.08 |
| -Os -frtl-seqabstr | 2856816 | 1.50 | 3599862 | 1.01 | 3162678 | 0.68 |
| -Os -ftree-seqabstr | 2888833 | 0.39 | 3610002 | 0.73 | 3166054 | 0.57 |
| -Os -fipa-procabstr | 2886632 | 0.47 | 3599042 | 1.03 | 3160626 | 0.74 |
| **all** | **2838348** | **2.13** | **3542506** | **2.58** | **3123398** | **1.91** |

**Table 8.4.1:** Average code-size saving results. *Size* is given in bytes and *saving* is the size saving correlated to '-Os' in percentage (%).

| Flags | i686-elf | arm-elf | sh-elf |
|---|---|---|---|
| | max. saving (%) | max. saving (%) | max. saving (%) |
| -Os -ftree-lfact -frtl-lfact | 6.13 | 10.98 | 10.29 |
| -Os -frtl-lfact | 4.31 | 3.51 | 4.35 |
| -Os -ftree-lfact | 5.75 | 10.34 | 8.78 |
| -Os -ftree-seqabstr -frtl-seqabstr | 36.81 | 56.92 | 43.89 |
| -Os -frtl-seqabstr | 30.67 | 45.69 | 42.45 |
| -Os -ftree-seqabstr | 30.60 | 41.60 | 44.72 |
| -Os -fipa-procabstr | 38.21 | 56.32 | 59.29 |
| **all** | **57.05** | **61.53** | **60.17** |

**Table 8.4.2:** Maximum code-size saving results for CSiBE objects. *Saving* is the size saving correlated to '-Os' in percentage (%).

Here, we should add that by compiling CSiBE with just an '-Os' flag using the same version of GCC as that used for implementation, we can get an optimized code whose size is 37.19% smaller than that got by compiling it without optimization methods.

By running all the implemented algorithms together, we can get a smaller code saving percentage compared to the sum of percentages for individual algorithms. This difference arises because the algorithms work on the same source tree and the earlier passes may optimize the same cases that would be realized by the subsequent methods applied. This is the reason why for the *i686-elf* target, by running local factoring on RTL level and Tree-SSA, we managed to achieve an average code saving on CSiBE of 0.19% and 0.10%, respectively, while by running both of these algorithms the result was only 0.27%. This difference also confirmed that the same optimization method run

on different ILs may realize different optimizable cases and running the same algorithm on more than one IL will lead to a better performance.

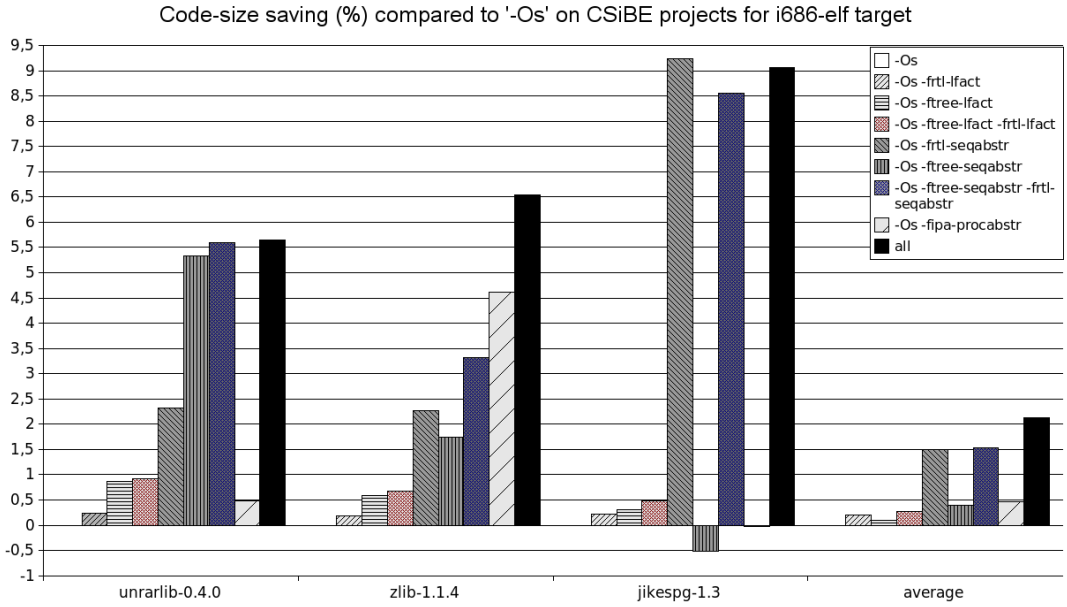



**Figure 8.4.1:** Detailed results for selected CSiBE projects.

The reason for the negative values in the code-saving percentages column (Figure 8.4.1) is exactly the same. Here, even if an algorithm optimizes the source tree, it might undermines the input of another that could do a better optimization job for the same tree. These differences do not mean that the corresponding algorithms are not effective methods, but they usually tell us that later passes may be able to optimize the same input source better.

In the table of compilation times (see Table 8.4.3), two algorithms outdo the others. These are the RTL and Tree sequence abstractions because the current implementation is realized with a running time of about $O(n^2)$. Nevertheless, these algorithms can be implemented with $O(n \log n)$ length of duration and this can result in a relative increase in the compilation time similar to that for interprocedural abstraction, which has a run time of $O(n \log n)$.

We should mention here that the sequence abstraction algorithms affect the running time as well because these methods may add new calls to the CFG. As local factoring does not change the number of executed instructions, these optimizations do not change the execution time of the optimized binaries. We found on our testing system with CSiBE that the average growth in running time compared to that for the '-Os' flag was about 0.26% for tree abstraction, 0.18% for interprocedural abstraction and 2.49% for the execution of all the algorithms together.

| flags | i686-elf | | arm-elf | | sh-elf | |
|---|---|---|---|---|---|---|
| | absolute | relative | absolute | relative | absolute | relative |
| -Os | 259.17 | 1.0000 | 285.97 | 1.0000 | 311.73 | 1.0000 |
| -Os -ftree-lfact -frtl-lfact | 262.73 | 1.0137 | 288.93 | 1.0104 | 333.64 | 1.0703 |
| -Os -frtl-lfact | 264.33 | 1.0199 | 303.27 | 1.0605 | 321.45 | 1.0312 |
| -Os -ftree-lfact | 259.90 | 1.0028 | 343.06 | 1.1996 | 321.97 | 1.0328 |
| -Os -ftree-seqabstr -frtl-seqabstr | 455.53 | 1.7576 | 464.00 | 1.6225 | 609.11 | 1.9540 |
| -Os -frtl-seqabstr | 315.75 | 1.2183 | 521.80 | 1.8247 | 651.13 | 2.0888 |
| -Os -ftree-seqabstr | 303.25 | 1.1701 | 325.76 | 1.1391 | 360.41 | 1.1562 |
| -Os -fipa-procabstr | 284.31 | 1.0970 | 298.79 | 1.0448 | 337.98 | 1.0842 |
| **all** | **342.69** | **1.3223** | **393.36** | **1.3755** | **489.69** | **1.5709** |

**Table 8.4.3:** Compilation time results. *Absolute* is in second (s) and *relative* means the multiplication factor of the compilation time with '-Os'.

## 8.5 Conclusions

From the results presented above, it seems that the sequence abstraction algorithms yielded the best results here. However, the scores indicate that these algorithms did not work as well on higher level abstraction languages as we had initially expected. The reason is that in earlier passes our methods may undermine the code for later passes that could better handle the same optimization cases. Perhaps these cases could be eliminated by compiling in two passes, where in the first pass we recover these problematic cases, and in the second one we optimize only those cases that would not ruin later passes.

As for local code motion we saw that these algorithms produced smaller percentage scores, but worked well together with later passes. The reason is that these methods do transformations on local parts of the source tree and do not make global changes on it. Another benefit is that local factoring has very small or usually no overheads, and the (in)ability to make a good cost computation does not affect the optimization too much.

Another conclusion concerns the running time of the algorithms. The sequence abstraction brought the best results, but with a fairly slow running time. Since this time affects the compilation, we may infer that if the duration of the compilation is really important, we suggest using local code motion as a fast and effective algorithm. Otherwise, when the compilation time is not too important, we can use all the algorithms together to achieve better overall results.

# Part V

# Conclusions

*"People love chopping wood. In this activity one immediately sees results."*

Albert Einstein

# 9 Conclusions

In this thesis we discuss different techniques for the automatic analysis and optimization of data-intensive applications. Now, we summarize the contributions and draw some conclusions. We will answer our research questions and elaborate on the main lessons we learned.

## 9.1 Summary of the thesis contributions

In general, the results presented show that static analysis techniques are good tools for supporting the development of data-intensive systems. The source code of a software system is its best documentation, hence by analyzing the source code we can recover implicit information about the system that might remain hidden if we used other approaches. We also showed that data accesses (e.g. via embedded SQLs) are good sources of architectural information in data-intensive systems. The techniques presented were also applicable to systems written in a special 4GL called Magic. In addition, we were able to identify security issues and perform transformations on different representations of a system.

We should add here, that most of the presented approaches had a real industrial motivation, allowing us to validate our methods in 'in vivo', industrial environment. Innovation projects sponsored by the European Union also rely on the results of our work [129, 132]. In addition, the reverse engineering framework for Magic motivated research studies for the National Scientific

Students' Associations Conference. These studies were later presented at international conferences as well [39, 50, 92].

Now, we will state our initial research questions and answer them one after the other.

RQ1: Can automated program analysis techniques recover implicit knowledge from data accesses to support the architecture recovery of data-intensive applications?

In Chapter 3, we introduced a new approach for extracting dependencies in data-intensive systems based on data accesses via embedded SQL queries (CRUD dependencies). The main idea was to analyze the program logic and the database components together, so as to be able to recover relations among source elements and data components. We analyzed a large financial system written in ForrásSQL with embedded Transact SQL queries and compared the results with those got using Static Execute After/Before relations. The results show that CRUD relations recovered connections among architectural components that would otherwise have remained hidden using other approaches.

We also examined the approach presented in Chapter 4, where we studied a large system developed in Oracle PL/SQL. We performed a bottom-up and top-down analysis on the system to recover the map of its architecture. For the bottom-up analysis, we utilized a technique based on the CRUD relations, while for the top-down analysis we interviewed developers of the system in order to learn more about its development.

The results got from both studies suggest that with the automatic analysis of data accesses in data-intensive systems, can indeed recover detailed information from the source code would probably have remained hidden if we have applied other approaches. Hence, our approach provides a good basis for further analysis including quality assurance approaches, impact analysis, architecture recovery and re-engineering.

RQ2: Can we adapt automatic analysis techniques that were implemented for 3rd generation languages to a 4th generation language like Magic? If so, can static analysis support the migration of Magic applications with automated techniques?

In Chapter 5, we introduce a novel approach for analyzing applications written in Magic. Here, we used the export file of the application development environment as the 'source code' of the application. It is rather a saved state of the development environment, but it carries all the necessary information that can be used to analyze to support quality assurance or migration tasks. Hence, we were able to implement a reverse engineering framework based on the Columbus methodology, a methodology designed for reverse engineering applications written in C, C++ or Java, which are typical 3rd generation languages. With the cooperation of our industrial partner, we demonstrated that our framework was helpful in the development of Magic applications.

In Chapter 6, however, we said that we need to be careful with the adaptation aspects because some attributes of the target language may differ. In particular, here we showed that neither the Halstead metrics nor McCabe metric matched the actual complexity definition of experienced Magic developers; hence a new complexity measure was defined that was more realistic.

In Chapter 5 we also showed that via a static analysis of Magic applications we can gather implicit knowledge that is useful for supporting the migration of Magic applications from previous versions of Magic to newer ones. With the help of our reverse engineering framework, we can recover CRUD relations that can support the redesigning of the database, e.g. by creating foreign keys which were not supported by previous versions of Magic.

RQ3: How can we utilize control flow and data flow analysis so as to be able to identify security issues based on user-related input data?

In Chapter 7, we presented a static analysis technique based on user-related input data to identify buffer overflow errors in C applications. The technique follows the user input from its entry point throughout the data flow and control flow graph and issues warnings if it reaches an error-prone point without any bounce checking. We implemented our approach as a plugin to the Grammatech CodeSurfer tool. We tested and validated our technique on open source projects and we found faults in Pidgin and cyrus-imapd, applications with about 200 kLoC.

RQ4: Can we use local refactoring algorithms in compilers to optimize the code size of generated binaries?

In Chapter 8, we showed that local code factoring algorithms are efficient algorithms for code size optimization. We implemented these algorithms in different optimization phases of GCC and we also implemented hoisting and sinking algorithms on the RTL and Tree-SSA intermediate languages of GCC. The correctness of the implementation was verified, and the numerical results were measured on different architectures using GCC's official Code-Size Benchmark Environment (CSiBE) as a real-world system. These results showed that on the ARM architecture we were able to achieve maximum and average extra code-size savings of 61.53% and 2.58% respectively, compared with the '-Os' flag of GCC.

# Bibliography

## References


[1] Peter H. Aiken. *Data reverse engineering: slaying the legacy dragon*. McGraw-Hill, 1996.

[2] Peter H. Aiken. Reverse engineering of data. *IBM Systtems Journal*, 37(2):246–269, April 1998.

[3] A. J. Albrecht and J. E. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transaction on Software Engineering*, 9:639–648, November 1983.

[4] Elias Levy (Aleph1). Smashing the stack for fun and profit. *Phrack*, 49, November 1996. URL http://insecure.org/stf/smashstack.html.

[5] P. Anderson and T. Teitelbaum. Software inspection using CodeSurfer. In *Proceedings of the Workshop on Inspection in Software Engineering (CAV 2001)*, Paris, France, July 2001.

[6] P. Anderson, D. Binkley, G. Rosay, and T. Teitelbaum. Flow insensitive points-to sets. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:0081, 2001.

[7] Paul Anderson. Codesurfer/path inspector. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, page 508, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th international conference on Software engineering (ICSE 2005)*, pages 332–341, New York, NY, USA, 2005. ACM.

[9] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transaction on Software Engineering*, 22:751–761, October 1996.

[10] Samuel Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1993)*, pages 384–396, New York, NY, USA, 1993. ACM.

[11] Keith Bennett. Legacy systems: Coping with success. *IEEE Softw.*, 12(1):19–23, January 1995.

[12] Árpád Beszédes, Rudolf Ferenc, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács. CSiBE benchmark: One year perspective and plans. In *Proceedings of the 2004 GCC Developers' Summit*, pages 7–15, June 2004.

[13] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.

[14] David Binkley. Source code analysis: A road map. In *Proceedings of the 2007 Future of Software Engineering (FOSE'07)*, pages 104–119. IEEE Computer Society, May 2007.

[15] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.*, 16(2):8, 2007.

[16] Barry Boehm. Some future trends and implications for systems and software engineering processes. *Systems Engineering*, 9(1):1–19, 2006.

[17] Barry Boehm. A view of 20th and 21st century software engineering. In *Proceedings of the 28th international conference on Software engineering (ISE 2006)*, pages 12–29. ACM, 2006.

[18] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, 1st edition, 1981.

[19] Stefan Brass and Christian Goldberg. Semantic errors in SQL queries: A quite complete list. *Journal of Systems and Software - Special issue: Quality software*, 79(5):630–644, May 2006.

[20] Huib van den Brink, Rob van der Leek, and Joost Visser. Quality assessment for embedded SQL. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 163–170. IEEE Computer Society, 2007.

[21] M. Burgin and N. Debnath. Complexity measures for software engineering. *J. Comp. Methods in Sci. and Eng.*, 5:127–143, January 2005.

[22] M. Y. Chan and S. C. Cheung. Testing database applications with SQL semantics. In *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, pages 363–374. Springer, 1999.

[23] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2 (6):76–79, 2004.

[24] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transaction on Software Engineering*, 20:476–493, June 1994.

[25] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[26] Aske Simon Christensen, Anders Müller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Static Analysis Symposium (SAS 2003)*, pages 1–18. Springer-Verlag, 2003.

[27] Anthony Cleve. *Program Analysis and Transformation for Data-Intensive System Evolution*. PhD thesis, University of Namur, October 2009.

[28] Anthony Cleve and Jean-Luc Hainaut. Dynamic analysis of SQL statements for data-intensive applications reverse engineering. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering (WCRE 2008)*, pages 192–196. IEEE Computer Society, 2008.

[29] Anthony Cleve, Jean Henrard, and Jean-Luc Hainaut. Data reverse engineering using system dependency graphs. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 157–166. IEEE Computer Society, 2006.

[30] Anthony Cleve, Tom Mens, and Jean-Luc Hainaut. Data-intensive system evolution. *IEEE Computer*, 43(8):110–112, August 2010.

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

[32] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 139–149, 1999.

[33] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[34] Coverity, Inc. Opensource report, 2008. URL http://scan.coverity.com/.

[35] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[36] K.H. Davis and P.H. Alken. Data reverse engineering: a historical survey. In *Proceedings of the Seventh Working Conference on Reverse Engineering.*, pages 70–78, 2000.

[37] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. Agile parsing in TXL. *Automated Software Engineering*, 10(4):311–336, October 2003.

[38] Saumya K. Debray, William Evans, Robert Muth, and Bjorn de Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[39] Richárd Dévai, Judit Jász, Csaba Nagy, and Rudolf Ferenc. Designing and implementing control flow graph for magic 4th generation language. In *Proceedings of the 13th Symposium on Programming Languages and Software Tools (SPLST 2013)*, pages 200–214, Szeged, Hungary, August 26-27 2013.

[40] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th international conference on Software engineering (ICSE 2006)*, pages 162–171. ACM, 2006.

[41] G.A. Di Lucca, A.R. Fasolino, and U. De Carlini. Recovering class diagrams from data-intensive legacy systems. In *International Conference on Software Maintenance (ICSM 2000)*, pages 52–63, 2000.

[42] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: an analysis of the internet virus of november 1988. In *IEEE Computer Society Symposium on Security and Privacy*, pages 326–343. IEEE Computer Society Press, 1989.

[43] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, 6th edition, 2010.

[44] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 151–162. ACM, 2007.

[45] David Evans, John Guttag, James Horning, and Yang Meng Tan. Lclint: a tool for using specifications to check code. *SIGSOFT Softw. Eng. Notes*, 19(5):87–96, 1994.

[46] Rudolf Ferenc. *Modelling and Reverse Engineering C++ Source Code*. PhD thesis, Doctoral School in Mathematics and Computer Science, University of Szeged, 2004.

[47] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, October 2002.

[48] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[49] Free Software Foundation. GNU Compiler Collection (GCC) internals. http://gcc.gnu.org/onlinedocs/gccint; accessed November, 2013.

[50] Dániel Fritsi, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. A layout independent GUI test automation tool for applications developed in Magic/uniPaaS. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools (SPLST 2011)*, pages 248–259, Tallinn, Estonia, Oct 4-7 2011.

[51] Spyridon K. Gardikiotis and Nicos Malevris. A two-folded impact analysis of schema changes on database applications. *International Journal of Automation and Computing*, 6 (2):109–123, 2009.

[52] Carl Gould, Zhendong Su, and Premkumar T. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 645–654. IEEE Computer Society, 2004.

[53] Florian Haftmann, Donald Kossmann, and Eric Lo. A framework for efficient regression tests on database applications. *The VLDB Journal*, 16(1):145–164, January 2007.

[54] Jean-Luc Hainaut. Introduction to database reverse engineering. Technical report, University of Namur, 2002.

[55] Jean-Luc Hainaut. Legacy and future of data reverse engineering. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009)*, pages 4–4, 2009.

[56] Jean-Luc Hainaut and Anthony Cleve. Dynamic analysis of SQL statements in data-intensive programs. Technical report, University of Namur, 2008.

[57] Jean-Luc Hainaut, Jean Henrard, Vincent Englebert, Didier Roland, and Jean-Marc Hick. Database reverse engineering. In *Encyclopedia of Database Systems*, pages 723–728. Springer, 2009.

[58] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[59] Christian Hammer, Jens Krinke, and Frank Nodes. Intransitive noninterference in dependence graphs. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006)*, pages 119–128. IEEE Computer Society, 2006.

[60] Ramzi A. Haraty, Nash'at Mansour, and Bassel Daou. Regression testing of database applications. In *Proceedings of the 2001 ACM symposium on Applied computing (SAC 2001)*, pages 285–289. ACM, 2001.

[61] Ramzy A. Haraty, Nashat Mansour, and Bassel A. Daou. *Advanced Topics in Database Research*, volume 3, chapter Regression test selection for database applications. Idea Group Inc, 2004.

[62] Mark Harman, Margaret Okulawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of function coupling. *IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution (PMESSE 1997)*, pages 28–32, May 1997.

[63] Jean Henrard. *Program Understanding in Database Reverse Engineering*. PhD thesis, University of Namur, 2003.

[64] Jean Henrard and Jean-Luc Hainaut. Data dependency elicitation in database reverse engineering. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, page 11. IEEE Computer Society, 2001.

[65] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 392–411, New York, NY, USA, 1992. ACM.

[66] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[67] Michael Howard, David LeBlanc, and John Viega. *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill, 2005.

[68] Jan Hubička. The GCC call graph module, a framework for interprocedural optimization. In *Proceedings of the 2004 GCC Developers' Summit*, pages 65–75, June 2004.

[69] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[70] Judit Jász, Árpád Beszédes, Tibor Gyimóthy, and Václav Rajlich. StaticExecute After/Before as a Replacement of Traditional Software Dependencies. In *Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM 2008)*, pages 137–146. IEEE Computer Society, 2008.

[71] Gregory M. Kapfhammer. Towards a method for reducing the test suites of database applications. In *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST 2012, pages 964–965. IEEE Computer Society, 2012.

[72] R. Kazman, S.G. Woods, and S. Jeromy Carriere. Requirements for integrating software architecture and reengineering models: CORUM II. In *Proceedings of the Fifth Working Conference on Reverse Engineering (WCRE1998)*, pages 154–163, 1998.

[73] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[74] Jingfei Kong, Cliff C. Zou, and Huiyang Zhou. Improving software security via runtime instruction-level taint checking. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability (ASID 2006)*, pages 18–24. ACM, 2006.

[75] R. T. Kouzes, G. A. Anderson, S. T. Elbert, I Gorton, and D. K. Gracio. The changing paradigm of data-intensive computing. *IEEE Computer*, 42(1):26–34, January 2009.

[76] D. J. Kuck, Y. Muraoka, and Shyh-Ching Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Trans. Comput.*, 21(12):1293–1310, 1972.

[77] Eric Larson and Todd Austin. High coverage detection of input-related security faults. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[78] Kyung-Suk Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exper.*, 33(5):423–460, 2003.

[79] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. *SIGSOFT Softw. Eng. Notes*, 28(5):317–326, 2003.

[80] Gábor Lóki, Ákos Kiss, Judit Jász, and Árpád Beszédes. Code factoring in GCC. In *Proceedings of the 2004 GCC Developers' Summit*, pages 79–84, June 2004.

[81] Stephen MacDonell. Metrics for database systems: An empirical study. In *Proceedings of the 4th International Symposium on Software Metrics*, pages 99–107. IEEE Computer Society, 1997.

[82] Michaël Marcozzi, Wim Vanhoof, and Jean-Luc Hainaut. Test input generation for database programs using relational constraints. In *Proceedings of the Fifth International Workshop on Testing Database Systems*, pages 6:1–6:6. ACM, 2012.

[83] James Martin. *Application Development without Programmers*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1982.

[84] C.A. Mattmann, D.J. Crichton, J.S. Hughes, S.C. Kelly, and M. Paul. Software architecture for large-scale, distributed, data-intensive systems. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pages 255–264, 2004.

[85] Chris Mattmann and Paul Ramirez. A comparison and evaluation of architecture recovery in data-intensive systems using focus. Technical report, Computer Science Department, University of Southern California, 2004.

[86] Chris A. Mattmann, Daniel J. Crichton, Andrew F. Hart, Cameron Goodale, J.Steven Hughes, Sean Kelly, Luca Cinquini, ThomasH. Painter, Joseph Lazio, Duane Waliser, Nenad Medvidovic, Jinwon Kim, and Peter Lean. Architecting data-intensive software systems. In *Handbook of Data Intensive Computing*, pages 25–57. Springer Science+Business Media, 2011.

[87] Andy Maule, Wolfgang Emmerich, and David S. Rosenblum. Impact analysis of database schema changes. In *Proceedings of the 30th international conference on Software engineering (ICSE 2008)*, pages 451–460. ACM, 2008.

[88] T.J. McCabe. A complexity measure. *IEEE Transaction on Software Engineering*, SE-2(4), dec 1976.

[89] Tom Mens and Serge Demeyer. *Sotware Evolution*. Springer Berlin Heidelberg, 2008.

[90] Richard C. Millham. *Handbook of Research on Innovations in Database Technologies and Applications: Current and Future Trends*, chapter Data Reengineering of Legacy Systems, pages 37–44. IGI Global, February 28 2009.

[91] J. K. Navlakha. A survey of system complexity metrics. *The Computer Journal*, 30:233–238, June 1987.

[92] Gábor Novák, Csaba Nagy, and Rudolf Ferenc. A regression test selection technique for Magic systems. In *Proceedings of the 13th Symposium on Programming Languages and Software Tools (SPLST 2013)*, pages 76–89, Szeged, Hungary, August 26-27 2013.

[93] Diego Novillo. Tree SSA a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, May 2003.

[94] Diego Novillo. Design and implementation of Tree SSA. In *Proceedings of the 2004 GCC Developers' Summit*, pages 119–130, June 2004.

[95] E. Nyary and H. M. Sneed. Software maintenance offloading at the Union Bank of Switzerland. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 1995)*, page 98. IEEE Computer Society, 1995.

[96] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. *SIGSOFT Softw. Eng. Notes*, 28(5): 128–137, September 2003.

[97] Kai Pan, Xintao Wu, and Tao Xie. Generating program inputs for database application testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 73–82. IEEE Computer Society, 2011.

[98] Janos Pantos, Arpad Beszedes, Pal Gyenizse, and Tibor Gyimothy. Experiences in adapting a source code-based quality assessment technology. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*, pages 311–313. IEEE Computer Society, 2008.

[99] Z. Pawlak. Information systems theoretical foundations. *Information Systems*, 6(3):205 – 218, 1981.

[100] William J. Premerlani and Michael R. Blaha. An approach for reverse engineering of relational databases. *Communications of the ACM*, 37(5):42–ff., May 1994.

[101] R. Kelly Rainer and Casey G. Cegielski. *Introduction to Information Systems: Enabling and Transforming Business*. John Wiley & Sons, Inc., 4 edition, January 11 2012.

[102] H. Rottgering. Lofar, a new low frequency radio telescope. *New Astronomy Reviews*, 47 (4-5, High-redshift radio galaxies - past, present and future):405–409, Septepmber 2003.

[103] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.

[104] Diptikalyan Saha, Mangala Gowri Nanda, Pankaj Dhoolia, V. Krishna Nandivada, Vibha Sinha, and Satish Chandra. Fault localization for data-centric programs. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 157–167. ACM, 2011.

[105] Bernard Scholz, Chenyi Zhang, and Cristina Cifuentes. User-input dependence analysis via graph reachability. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 25–34, Beijing, September 2008.

[106] H. Sneed. Migrating pl/i code to java. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR 2011)*, pages 287–296, 2011.

[107] H.M. Sneed and K. Erdoes. Migrating AS400-COBOL to Java: A report from the field. In *Software Maintenance and Reengineering (CSMR 2013), 2013 17th European Conference on*, pages 231–240, 2013.

[108] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4): 410–457, 2006.

[109] Ian Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2011.

[110] María José Suárez-Cabal and Javier Tuya. Using an sql coverage measurement for testing database applications. In *Proceedings of the 12th ACM SIGSOFT 12th international symposium on Foundations of software engineering*, pages 253–262. ACM, 2004.

[111] Jay-Evan J. Tevis and John A. Hamilton. Methods for the prevention, detection and removal of software security vulnerabilities. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 197–202, New York, NY, USA, 2004. ACM.

[112] Frank Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.

[113] M.J.P. Van der Meulen and M.A. Revilla. Correlations between internal software metrics and software dependability in a large population of small C/C++ programs. In *Proceedings of ISSRE 2007, The 18th IEEE International Symposium on Software Reliability*, pages 203–208, November 2007.

[114] A. Van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC 1998)*, page 90. IEEE Computer Society, 1998.

[115] June Verner and Graham Tate. Estimating size and effort in fourth-generation development. *IEEE Software*, 5:15–22, 1988.

[116] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, 2000.

[117] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. *ACM Trans. Softw. Eng. Methodol.*, 16(4), September 2007.

[118] G.E. Witting and G.R Finnie. Using artificial neural networks and function points to estimate 4GL software development effort. *Australasian Journal of Information Systems*, 1(2), 1994.

[119] Jun Xu and Nithin Nakka. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN 2005)*, pages 378–387. IEEE Computer Society, 2005.

[120] Sheng Yu and Shijie Zhou. A survey on metric of software complexity. In *Proceedings of ICIME 2010, The 2nd IEEE International Conference on Information Management and Engineering*, pages 352–356, April 2010.

[121] Stanley B. Zdonik and David Maier, editors. *Readings in object-oriented database systems*. Morgan Kaufmann Publishers Inc., 1990.

[122] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6): 97–106, 2004.

## Corresponding publications of the Thesis

[123] Dániel Fritsi, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. A methodology and framework for automatic layout independent GUI testing of applications developed in Magic xpa. In *Proceedings of the 13th International Conference on Computational Science and Its Applications - ICCSA 2013 - Part II*, pages 513–528, Ho Chi Minh City, Vietnam, June 24-27 2013. Springer.

[124] Csaba Nagy. Extension of GCC with a fully manageable reverse engineering front end. In *Proceedings of the 7th International Conference on Applied Informatics (ICAI 2007)*, January 28-31 2007. Eger, Hungary.

[125] Csaba Nagy. Static analysis of data-intensive applications. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*. IEEE Computer Society, March 5-8 2013. Genova, Italy.

[126] Csaba Nagy and Spiros Mancoridis. Static security analysis based on input-related software faults. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009)*, pages 37–46, Fraunhofer IESE, Kaiserslautern, Germany, March 24-27 2009. IEEE Computer Society.

[127] Csaba Nagy, Gábor Lóki, Árpád Beszédes, and Tibor Gyimóthy. Code factoring in GCC on different intermediate languages. *ANNALES UNIVERSITATIS SCIENTIARUM BUDAPESTINENSIS DE ROLANDO EOTVOS NOMINATAE Sectio Computatorica - TOMUS XXX*, pages 79–96, 2009.

[128] Csaba Nagy, János Pántos, Tamás Gergely, and Árpád Beszédes. Towards a safe method for computing dependencies in database-intensive systems. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, pages 166–175, Madrid, Spain, March 15-18 2010. IEEE Computer Society.

[129] Csaba Nagy, László Vidács, Rudolf Ferenc, Tibor Gyimóthy, Ferenc Kocsis, and István Kovács. MAGISTER: Quality assurance of Magic applications for software developers and end users. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–6, Timisoara, Romania, Sept 2010.

[130] Csaba Nagy, Rudolf Ferenc, and Tibor Bakota. A true story of refactoring a large Oracle PL/SQL banking system. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011)*, Szeged, Hungary, Sept 5-9 2011.

[131] Csaba Nagy, László Vidács, Rudolf Ferenc, Tibor Gyimóthy, Ferenc Kocsis, and István Kovács. Complexity measures in 4GL environment. In *Proceedings of the 2011 International Conference on Computational Science and Its Applications - Volume Part V*, ICCSA'11, pages 293–309, Santander, Spain, June 20-23 2011. Springer-Verlag.

[132] Csaba Nagy, László Vidács, Rudolf Ferenc, Tibor Gyimóthy, Ferenc Kocsis, and István Kovács. Solutions for reverse engineering 4GL applications, recovering the design of a logistical wholesale system. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR 2011)*, pages 343 –346, Oldenburg, Germany, March 1-4 2011. IEEE Computer Society.

# A
# Summaries

## A.1 Summary in English

At the beginning of the 21st century, information systems are not simple computer applications that we sometimes use at work, but large systems with complex architectures and fulfilling important roles in our daily life. The purpose of such systems is to get the right information to the right people at the right time in the right amount and in the right format [101].

In 1981, Pawlak published a paper reporting some of the activities of the Information Systems Group in Warsaw [99]. Their information system was implemented for an agriculture library that had some 50,000 documents. Since then, information systems have evolved as data volumes have steadily increased. There are reports on systems, like those in radio astronomy, where systems need to handle 138 PB (peta bytes) of data per day [102]. Another example from the high-energy physics community, the Large Hadron Collider machine, generates 2 PB of data per second when in operation [75]. These systems are usually referred to as *data-intensive systems* [27, 84–86].

The big data which data-intensive systems work with is stored in a database, typically managed by a database management system (DBMS), where it is structured according to a schema. In relational DBMSs (RDBMS), this schema consists of data tables with columns where the tables usually represent the current state of the population of a business object and columns are its properties.

In order to support the maintenance tasks of these systems, several techniques have been developed to analyze the source code of applications or to analyze the underlying databases for the purpose of reverse engineering tasks, like quality assurance and program comprehension. However, only a few techniques take into account the special features of data-intensive systems (e.g. dependencies arising via database accesses). As Cleve et al. remarked in a study on data-intensive system evolution [30]: "*... both the software and database engineering research communities have addressed the problems of system evolution. Surprisingly, however, they've conducted very little research into the intersection of these two fields, where software meets data.*"

### A.1.1 Goals of the Thesis

In this thesis, we describe studies carried out to analyze data-intensive applications via different reverse engineering methods based on static analysis. These are methods for recovering the architecture of data-intensive systems, a quality assurance methodology for applications developed in Magic, identifying input data related coding issues and optimizing systems via local refactoring. With the proposed techniques we were able to analyze large scale industrial projects like banking systems with over 4 million lines of code, and we successfully retrieved architecture maps and identified quality issues of these systems.
Our results have been grouped into 6 contributions divided into three main parts according to the research topics. In the remaining part of the thesis summary, the following contribution points will be presented:

I Architecture recovery of legacy data-intensive systems

(a) Extracting architectural dependencies in data-intensive systems

(b) Case study of reverse engineering the architecture of a large banking system

II The world of Magic

(a) A reverse engineering framework for Magic applications

(b) Evaluating complexity measures in Magic as a special 4th generation language

III Security and optimization

(a) Static security analysis based on input-related software faults

(b) Optimizing information systems: code factoring in GCC

### A.1.2 Contributions

In general, the results presented showed that static analysis techniques are good tools for supporting the development processes of data-intensive systems. The source code of a software system is its best documentation, hence by analyzing the source code we can recover implicit information about the system that might remain hidden if we used other approaches. We also showed that data accesses (e.g. via embedded SQLs) are good sources of architectural information in data-intensive systems. The techniques presented were also applicable to systems written in a special 4GL called Magic. In addition, we were able to identify security issues and automatically perform transformations to optimize the codesize of a system.

We should add here, that most of the presented approaches had a real industrial motivation, allowing us to validate our methods in 'in vivo', industrial environment. Innovation projects sponsored by the European Union also rely on the results of our work [129, 132]. In addition, the reverse engineering framework for Magic motivated research studies for the National Scientific Students' Associations Conference. These studies were later presented at international conferences as well [39, 50, 92].

RQ1: Can automated program analysis techniques recover implicit knowledge from data accesses to support the architecture recovery of data-intensive applications? We introduced a new approach for extracting dependencies in data-intensive systems based on data accesses via embedded SQL queries (CRUD dependencies). The main idea was to analyze the program logic and the database components together; hence we were able to recover relations among source elements and data components. We analyzed a financial system written in ForrásSQL with embedded Transact SQL or MS SQL queries and compared the results with Static Execute After/Before relations. The results show that CRUD relations recovered new connections among architectural components that would have remained hidden using other approaches. We further investigated the approach and studied a large system developed in Oracle PL/SQL. We performed a bottom-up and top-down analysis on the system to recover the map of its architecture. For the bottom-up analysis, we used a technique based on the CRUD relations, while for the top-down analysis we interviewed the developers.

We conclude that the automatic analysis of data accesses in data-intensive systems is a feasible approach for recovering information from the source code which might otherwise have been hidden using other approaches. Hence, these techniques provide a good basis for further investigation like quality assurance approaches, impact analysis, architecture recovery and re-engineering.

RQ2: Can we adapt automatic analysis techniques that were implemented for 3rd generation languages to a 4th generation language like Magic? If so, can static analysis support the migration of Magic applications with automated techniques? We introduce a novel approach for analyzing applications written in Magic. Here, we treat the export file of the application development environment as the 'source code' of the application. This export file is rather a saved state of the development environment, but it carries all the necessary information that can be used to support quality assurance or migration tasks. With it, we were able to implement a reverse engineering framework based on the Columbus methodology, a methodology designed for reverse engineering applications written in C, C++ or Java, which are typical 3rd generation languages. With the cooperation of our industrial partner we show that the implemented framework is helpful in the development of Magic applications.

As regards the quality attributes of the application, we show that neither the Halstead complexity measures nor McCabe's complexity measure fit the complexity definition of experienced Magic developers; hence a new complexity measure is defined here.

We also showed that via static analysis it is possible to gather implicit knowledge that is useful for supporting the migration of Magic applications from earlier versions of Magic to later ones. With the help of our reverse engineering framework we can recover CRUD relations that can support the redesign of the database, for example by creating foreign keys which were not supported by previous versions of Magic.

RQ3: How can we utilize control flow and data flow analysis so as to be able to identify security issues based on user-related input data? We present a static analysis technique based on user-related input data to identify buffer overflow errors in C applications. The technique follows the user input from its entry point throughout the data-flow and control flow graph and warns the user if it reaches a error-prone point without any bounce checking. We implemented our approach as a plugin to the GrammaTech CodeSurfer tool. We tested and validated our technique on open source projects and we found faults in Pidgin and cyrus-imapd as applications with about 200 kLoC.

RQ4: Can we use local refactoring algorithms in compilers to optimize the code size of generated binaries? We show that local code factoring algorithms are efficient algorithms for code size optimization. We implemented these algorithms in different optimization phases of GCC and we also implemented hoisting and sinking algorithms on the RTL and Tree-SSA intermediate languages of GCC. The correctness of the implementation was verified, and the results were measured on different architectures using GCC's official Code-Size Benchmark Environment (CSiBE) as a real-world system. These results showed that on the ARM architecture we were able to achieve maximum and average extra code-size saving of 61.53% and 2.58% respectively, compared with the '-Os' flag of GCC.

## A.2 Summary in Hungarian

Napjaink információs rendszerei már nem egyszerű alkalmazások, amikkel egy-egy fontosabb feladatot oldunk meg. Ma már hatalmas méretű, összetett architektúrájú rendszerekkel dolgozunk, amik részei a mindennapjainknak, ott vannak a táblagépeinken, okos telefonjainkon, mindenhol. Ezeknek a rendszereknek a célja, hogy a helyes információt a megfelelő embereknek pontos időben és formában juttassák el [101].

Pawlak 1981-ben megjelent cikkében ír a Varsói Egyetem Információs Rendszerek Munkacsoportjának eredményeiről [99]. Tanulmányában bemutat egy információs rendszert, ami egy könyvtári rendszer és mintegy 50.000 dokumentumot kezel. Azóta az információs rendszerek rengeteget fejlődtek, és a kezelt adatmennyiség is jelentősen megnőtt. Ismerünk olyan rendszereket a rádiócsillagászatból, amik napi 138 PB (peta byte) adatot kezelnek [102]. Jól ismert a részecskefizika világából a CERN 2008-ban átadott Large Hadron Collider részecskegyorsítója is, ami másodpercenként 2 PB adatot kezel [75]. Az ilyen rendszereket a jelentős adatterhelés miatt *adat-intenzív rendszerek*nek nevezzük [27, 84–86].

A hatalmas mennyiségű adat, amit az adat-intenzív rendszerek kezelnek, általában egy *adatbázis*ban kerül eltárolásra, amit egy *adatbázis-kezelő rendszer* (*database management system*, DBMS) kezel valamilyen adat *séma* szerint rendszerezve. *Relációs DBMS*-ekben (RDBMS) ez a séma táblákat tartalmaz, amik általában egy entitást jelölnek különböző tulajdonságokkal, amiket a tábla oszlopai tárolnak.

Az ilyen rendszerek karbantartásának támogatására több módszert is kidolgoztak mind a forráskód, mind pedig az adatbázis elemzésének segítségével is. Kevés olyan módszer van viszont, ami valóban figyelembe veszi az adat-intenzív rendszerek sajátosságait (pl. adatelérésen keresztüli függőségek vizsgálata). Ahogy Cleve et al. megjegyzik azt az adat-intenzív rendszerek evolúcióját vizsgáló tanulmányukban [30]: *"mind a szoftver, mind az adatbázis rendszerek fejlesztői keresik a megoldásokat a szoftver evolúció problémáira. Mégis, meglepően kevés kutató munka vizsgálja a két területet együttesen, ahol a szoftver és az adat találkozik."*.

### A.2.1 Tézis célkitűzései

Jelen tanulmányban adat-intenzív rendszerek visszatervezési módszereit vizsgáljuk statikus elemzési módszerekkel. Olyan módszerekkel foglalkozunk, amik a Cleve et al. által is felvetett módon, a szoftver és az adat komponensek együttes vizsgálatával nyernek ki rejtett kapcsolatokat adat-intenzív rendszerekből. A kinyert információ segítségével megoldást keresünk adat-intenzív rendszerek architektúrájának feltérképezésére; egy speciális negyedik generációs nyelvben, Magicben fejlesztett alkalmazások minőségbiztosítására; input adat okozta biztonsági hibák felderítésére; valamint információs rendszerek optimalizálására lokális refaktoring műveletek segítségével. A bemutatott módszerekkel nagyméretű, ipari rendszereket elemzünk, egyebek mellett egy több, mint 4 millió soros banki rendszer esettanulmányát is bemutatjuk, ahol a rendszer architektúra térképét állítjuk elő automatikus eszközökkel, illetve minőségproblémákat tárunk fel benne.
Az elért eredményeinket hat tézispontban foglaljuk össze, amelyek az alábbiak:

I Örökölt, adat-intenzív rendszerek architektúrájának visszatervezése

  (a) Architekturális függőségek feltérképezése adat-intenzív rendszerekben

  (b) Nagyméretű, örökölt rendszerek architekturális problémáinak vizsgálata

II A Magic világa

(a) Magic alkalmazások visszatervezését támogató elemzőcsomag kifejlesztése

(b) Új komplexitás metrikák definiálása és kiértékelése Magic rendszereken

III Biztonsági elemzés és optimalizálás

(a) Felhasználói input okozta biztonsági hibák felderítése

(b) Információs rendszerek optimalizálása: kód faktoring a GCC fordítóban

### A.2.2 Eredmények összefoglalása

Összességében, az eredmények azt mutatják, hogy statikus kódelemző módszerekkel hatékonyan lehet támogatni az adat-intenzív rendszerek fejlesztési folyamatait. Egy alkalmazás legjobb dokumentációja a forráskód, a forráskódot elemezve ezért olyan implicit információt nyerhetünk a rendszerről, ami más módszerek számára rejtett maradhat. Megmutatjuk, hogy az adatelérések (pl. beágyazott SQL utasításokon keresztül) ilyen rejtett függőségeket hordoznak, ugyanakkor jó forrásai architekturális kapcsolatoknak. A bemutatott módszerek alkalmazhatóak Magic-re is, mint egy speciális negyedik generációs programozási nyelvre. Mindemellett, egy statikus elemzési módszert mutatunk be felhasználó input okozta biztonsági hibák felderítésére, és optimalizációs eljárásokat ismertetünk a kódméret csökkentésére.

Fontosnak tartjuk megjegyezni, hogy a bemutatott eredmények általában valós, ipari motivációs igényt elégítenek ki, aminek eredményeként kidolgozott módszerek tesztelését is ipari környezetben végezhettük el. A kutatási munkák eredményeire ezért a külső hivatkozások mellett Európai Uniós támogatással megvalósuló, innovációs projektek is támaszkodnak. Emellett a Magic rendszereken elért eredmények több szakdolgozatnak és TDK munkának az alapját is adták, amelyek nemzetközi konferenciákon is bemutatásra kerültek.

1) Lehetséges-e automatikus forráskód elemzési módszerekkel, adatelréseket vizsgálva, információt kinyerni, ami segíthet egy adat-intenzív rendszer architektúrájának feltérképezésében? Bemutattunk egy új módszert adat-intenzív rendszerek architekturális kapcsolatainak kinyerésére (*CRUD* kapcsolatok), ami az adatelréseket vizsgálja a beágyazott SQL utasítások elemzésével. Az ötlet alapja, hogy a program alkalmazás oldalát és az adatbázist együttesen elemezzük, felderítve ezzel olyan függőségeket, amik adatbázis használat miatt jöhetnek létre. Egy nagyméretű, pénzügyi rendszert vizsgálunk, amit ForrásSQL nyelven fejlesztettek Transact SQL és MS SQL utasításokat beágyazva a kódba. A kinyert kapcsolatokat a Static Execute After/Before kapcsolatokkal vetjük össze, aminek az eredményeként azt tapasztaljuk, hogy a *CRUD* kapcsolatok olyan függésekre mutatnak rá, amiket más módszerek nem ismernek fel. Ezt a módszert használjuk ezért egy későbbi tanulmány során is, ahol egy nagyméretű Oracle PL/SQL rendszer architektúráját térképezzük fel.

A tanulmányokból kiderül, hogy automatikus elemzési módszerekkel vizsgálva az adatelréseket olyan hasznos információt nyerhetünk egy rendszerről, amit más módszerekkel nem tudnánk felderíteni. A technika alkalmazása ezért javasolt lehet olyan elemzéseknél mint pl. a hatásanalízis, architektúra visszatervezés vagy minőségbiztosítás.

2) Adaptálható-e egy harmadik generációs nyelvekhez kifejlesztett automatikus elemzési módszer egy negyedik generációs nyelvre, mint amilyen a Magic? Amennyiben igen, úgy lehetséges-e statikus kódelemzéssel támogatni egy Magic alkalmazás újabb verzióra történő migrálását? A disszertációban bemutatunk egy újszerű módszert Magic alkalmazások statikus elemzésére. Ebben a módszerben az alkalmazásfejlesztő környezet mentését tekintjük az alkalmazás 'forráskódjának'. Ez a mentési állomány ugyan nem tekinthető a hagyományos értelemben vett forráskódnak, mégis minden fontos információt tartalmaz az alkalmazás felépítéséről. Erre támaszkodva, egy teljes elemző eszközcsomagot fejlesztünk a Columbus módszertanból kiindulva, amit célzottan C, C++ és Java nyelven fejlesztett alkalmazások visszatervezésére terveztek. Az ipari partnerünk segítségével megmutatjuk, hogy a kifejlesztett eszközcsomag jól használható Magic alkalmazások visszatervezéséhez.

Megmutatjuk, hogy az ismert komplexitás metrikák közül a McCabe és a Halstead komplexitás metrikák sem tükrözik a fejlesztők komplexitás elképzelését, ezért Magic-re egy új komplexitás metrikát javaslunk.

Azt is megmutatjuk, hogy statikus kódelemzéssel felfedezhetőek olyan kapcsolatok az alkalmazásban (pl. CRUD relációk, táblák közötti külső kulcs kapcsolatok), amik jelentősen segíthetik egy Magic alkalmazás migrálását egy korábbi verzióról egy újabb verzióra.

3) Hatékonyan használhatóak-e a vezérlési és adatfolyam elemzések a felhasználói input okozta biztonsági hibák felderítéséhez? Bemutatunk egy olyan elemzési módszert, ami a vezérlési és adatfolyam elemzéseket felhasználói input okozta biztonsági hibák felderítéséhez használja, C nyelven íródott alkalmazásokban. A módszer a különböző I/O műveletekből származó adatot követi nyomon az adatfolyamban, és jelez, ha a vezérlés olyan, hibára érzékeny ponthoz jut, ahol nem lett leellenőrizve korábban az külső forrásból érkező adat. A módszert GrammaTech CodeSurfer pluginként implementáljuk és nyílt forráskódú rendszereken teszteljük. A közel 200.000 kódsoros Pidginben és cyrus-imapd-ben is rámutatunk tényleges hibákra a segítségével.

4) Milyen mértékben lehet csökkenteni kód faktoring algoritmusok segítségével egy fordító által előállított binárisok méretét? Kód factoring algoritmusok családjába tartozó lokális faktoring és sequence abstraction algoritmusokat implementálunk a GCC különböző optimalizációs szintjein, hogy azt vizsgáljuk milyen kódméret csökkenés érhető el az algoritmusok segítségével. Az algoritmusokat a GCC hivatalos, kódméret mérésre kialakított tesztkörnyezetén teszteljük (Code-Size Benchmark Environment, CSiBE). A méréseket több architektúrára is elvégeztük, amik közül az ARM architektúrán a legmagasabb kódcsökkenés 61.53% volt, az átlagos pedig 2.58%-os az egyszerű '-Os' kapcsolóval összevetve, ami jelentős mértékű csökkentésnek tekinthető.

# B
# Magic

## B.1 Magic Schema

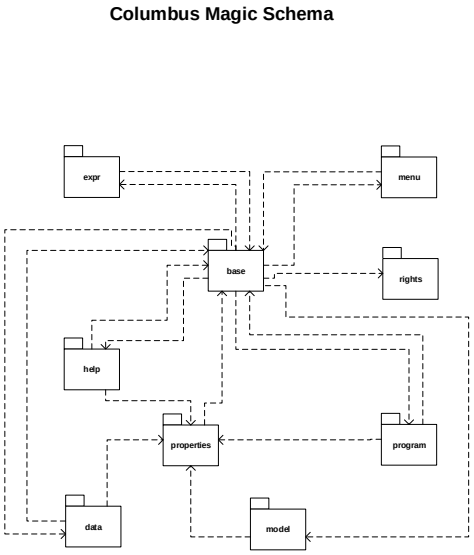



**Figure B.1.1:** Magic Schema - packages and their relations

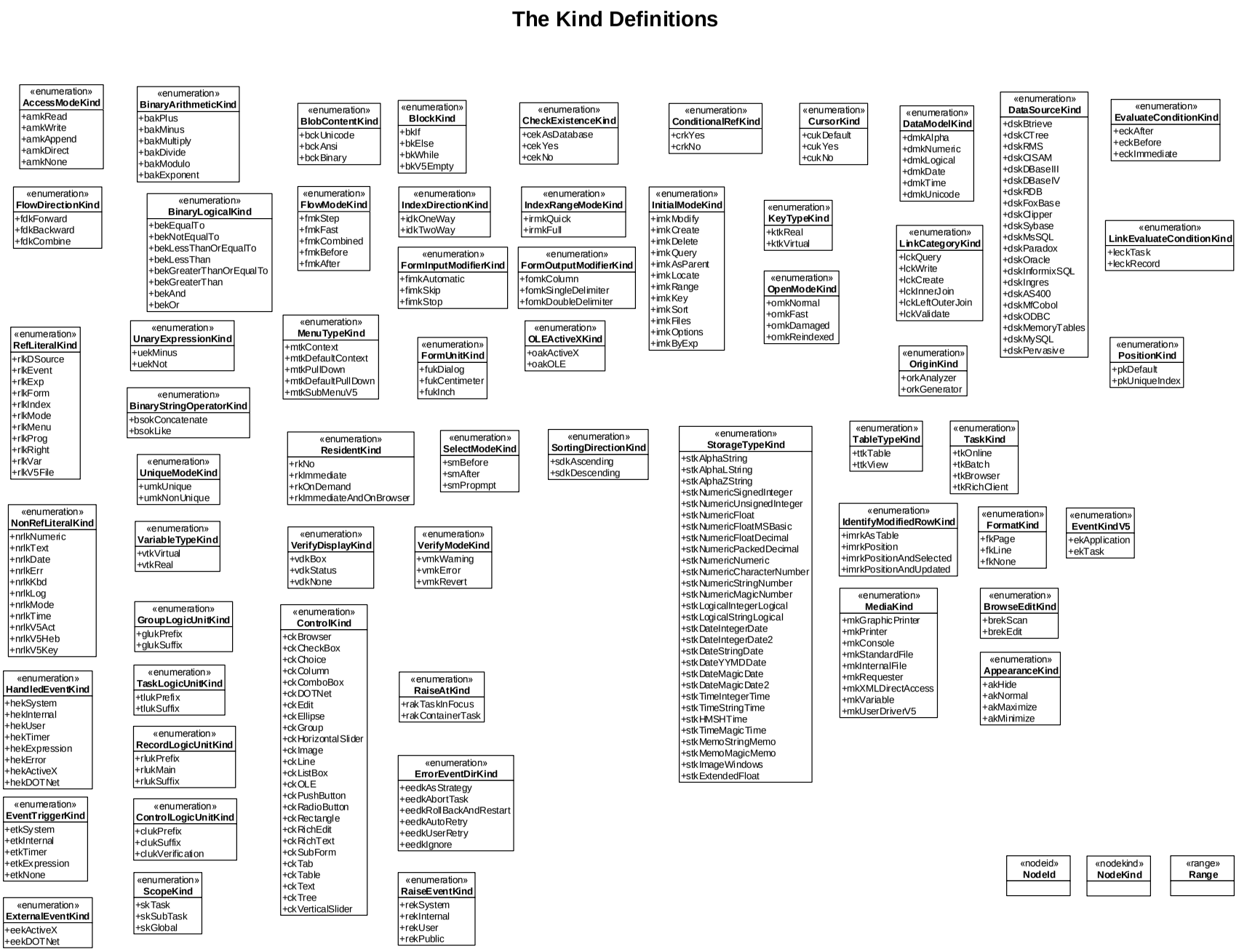



**Figure B.1.2:** Magic Schema - The Kind Definitions

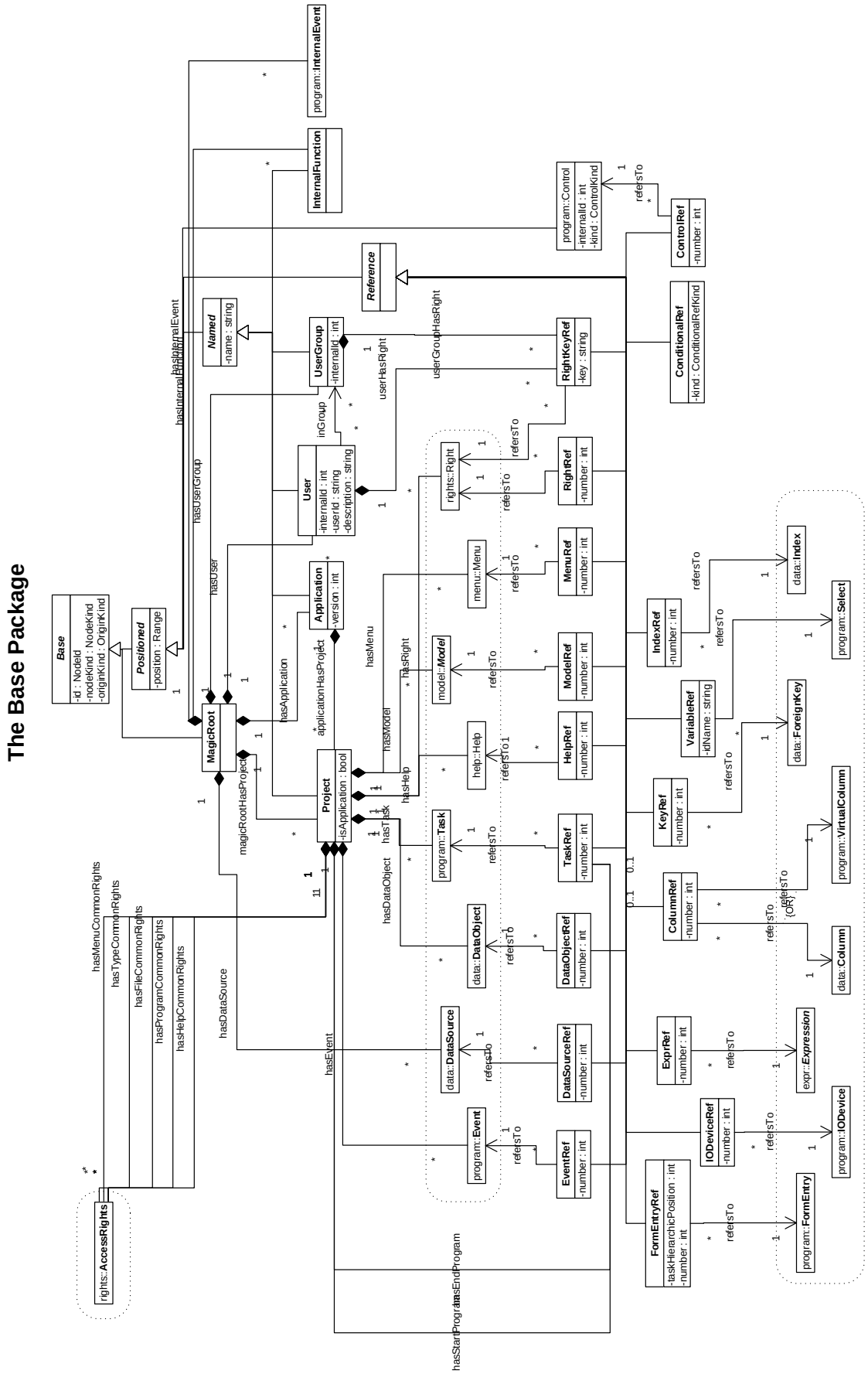



**Figure B.1.3:** Magic Schema - The Base Package

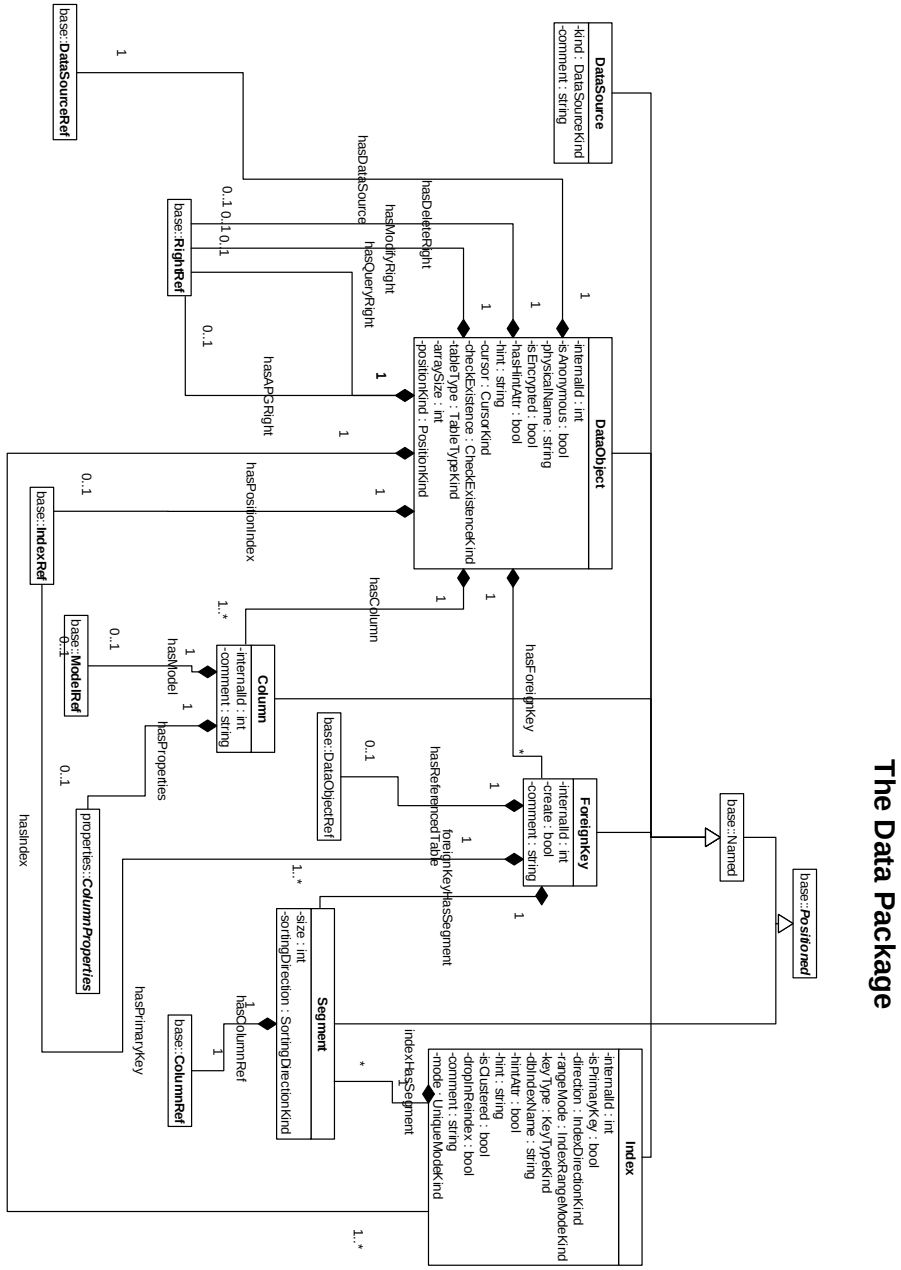



**Figure B.1.4:** Magic Schema - The Data Package

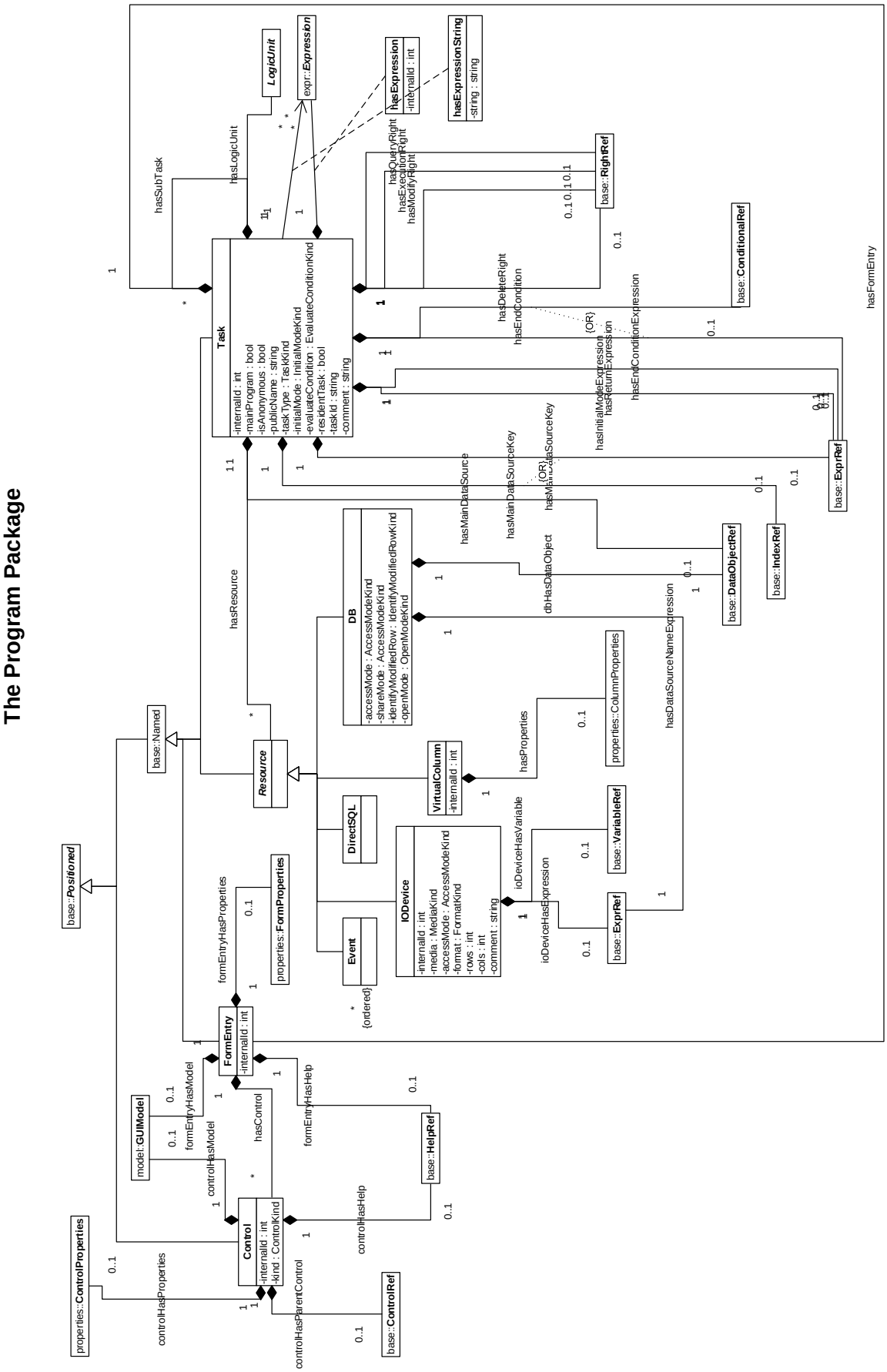



**Figure B.1.5:** Magic Schema - The Program Package

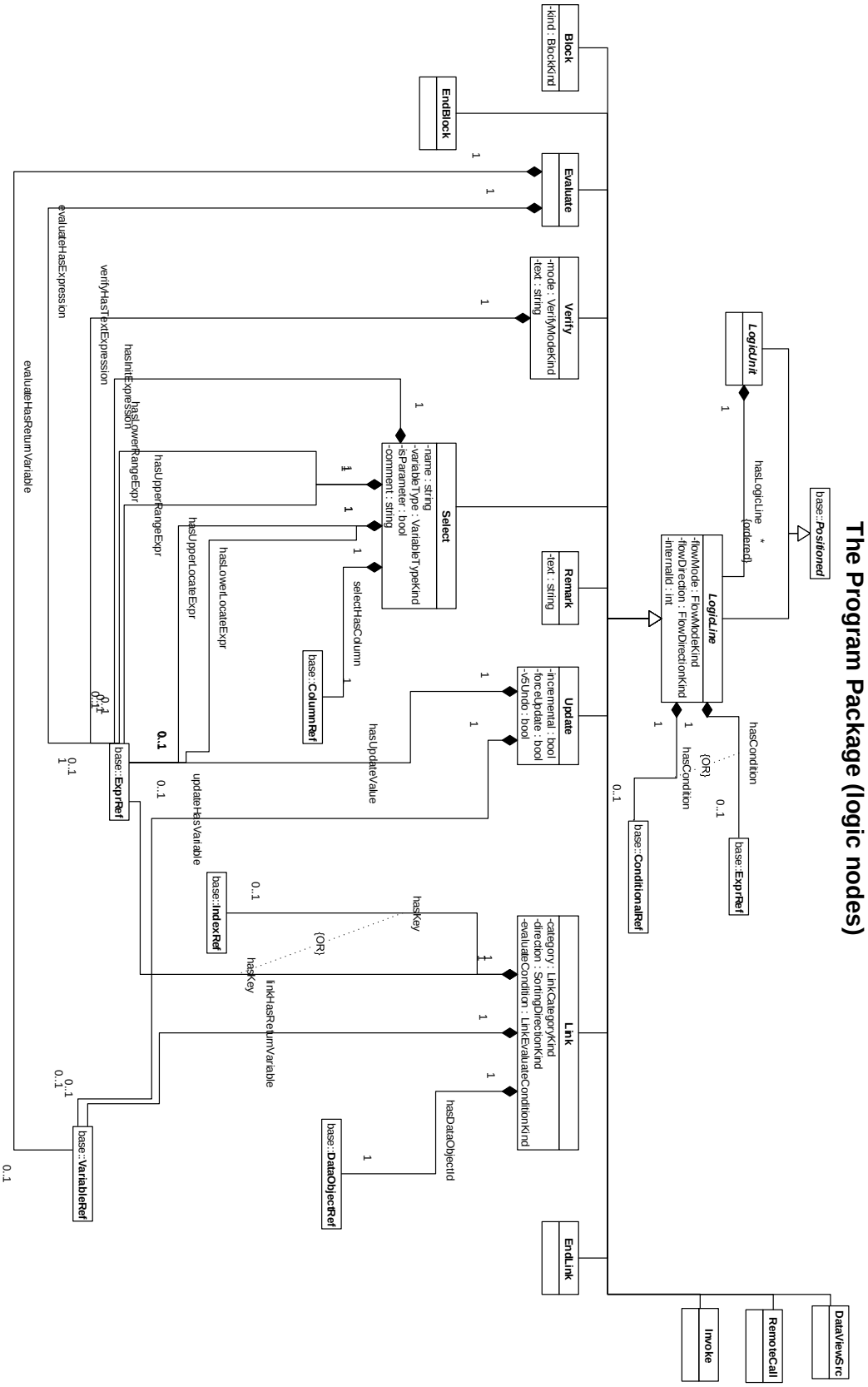



**Figure B.1.6:** Magic Schema - The Program package (logic)

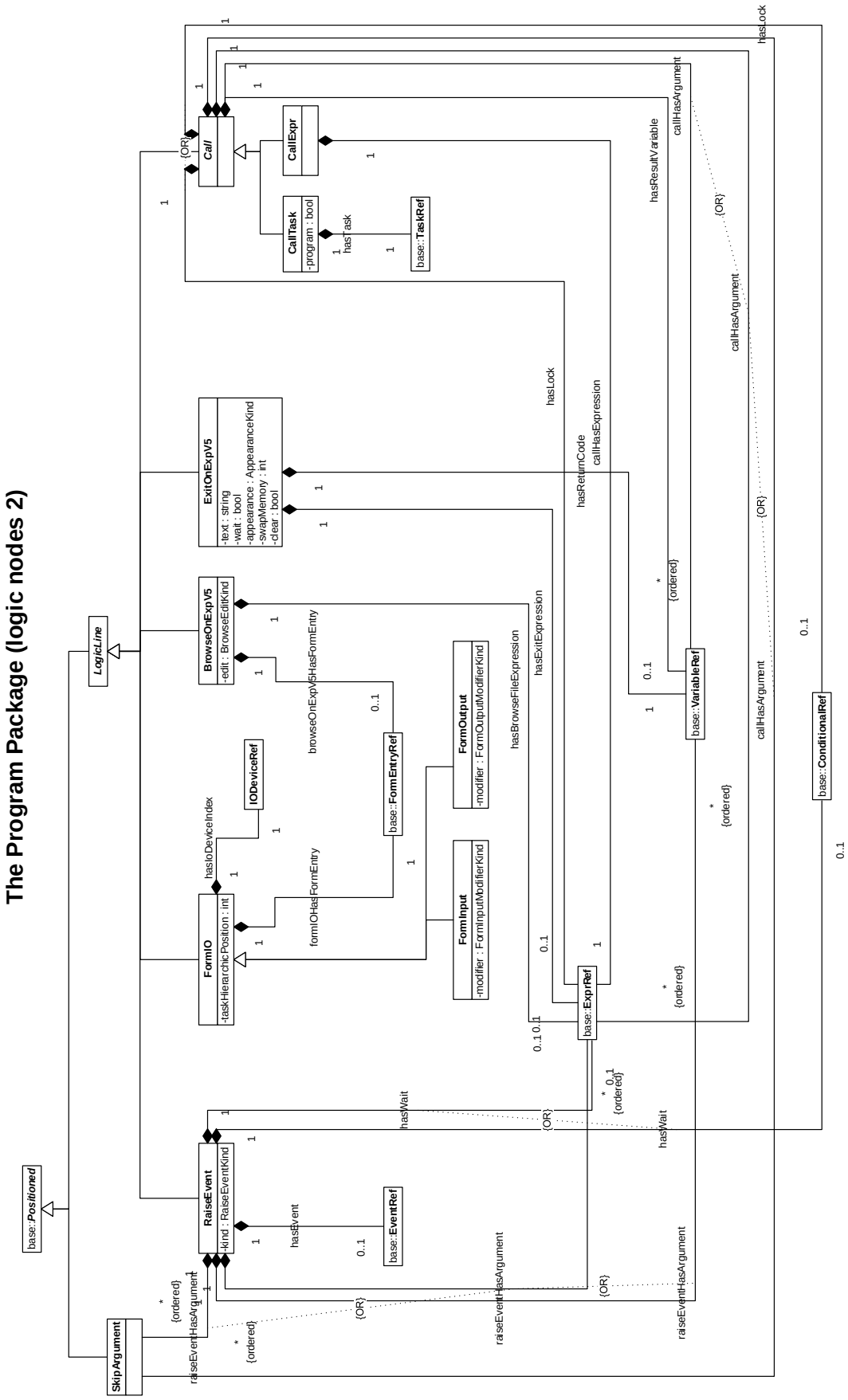



**Figure B.1.7:** Magic Schema - The Program package (logic2)

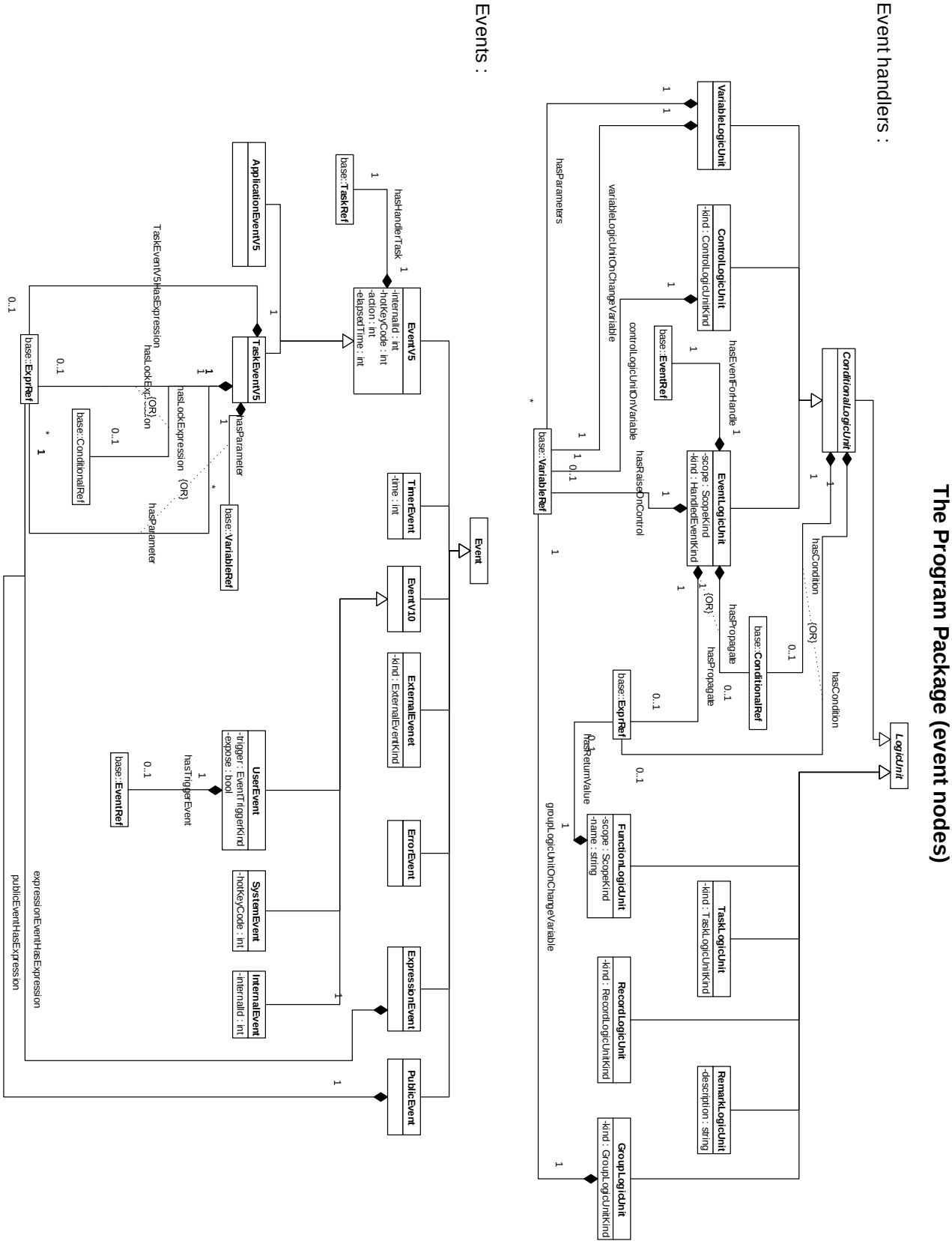



**Figure B.1.8:** Magic Schema - The Program package (events)

## The Model Package

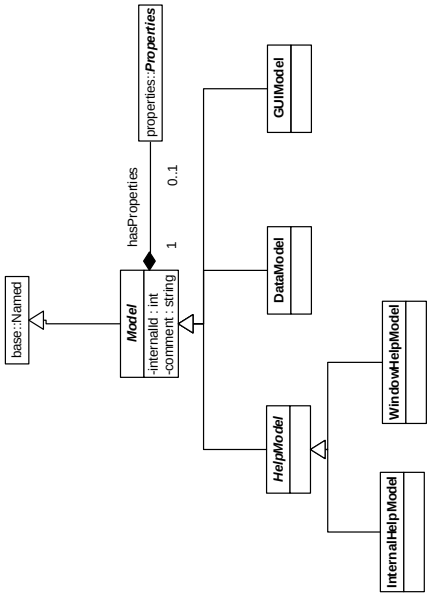



**Figure B.1.9:** Magic Schema - The Model package

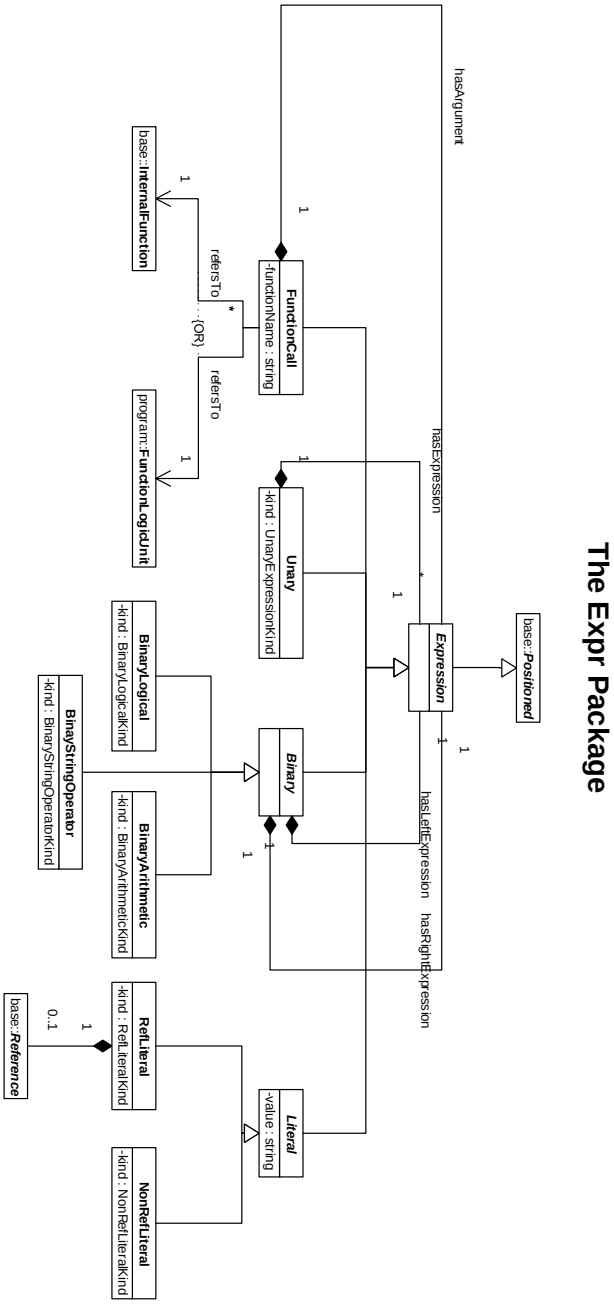



**Figure B.1.10:** Magic Schema - The Expr package

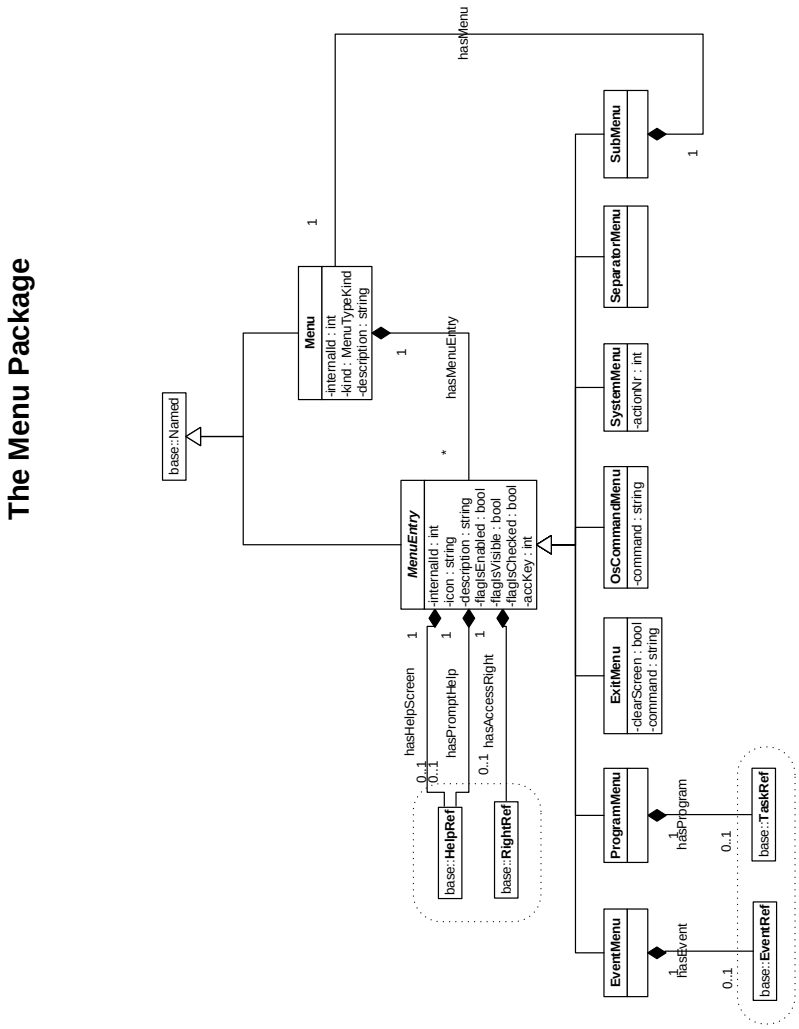



**Figure B.1.11:** Magic Schema - The Menu package

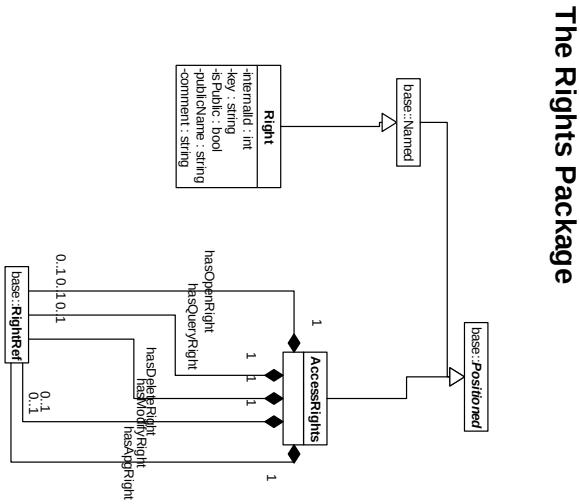



**Figure B.1.12:** Magic Schema - The Rights package

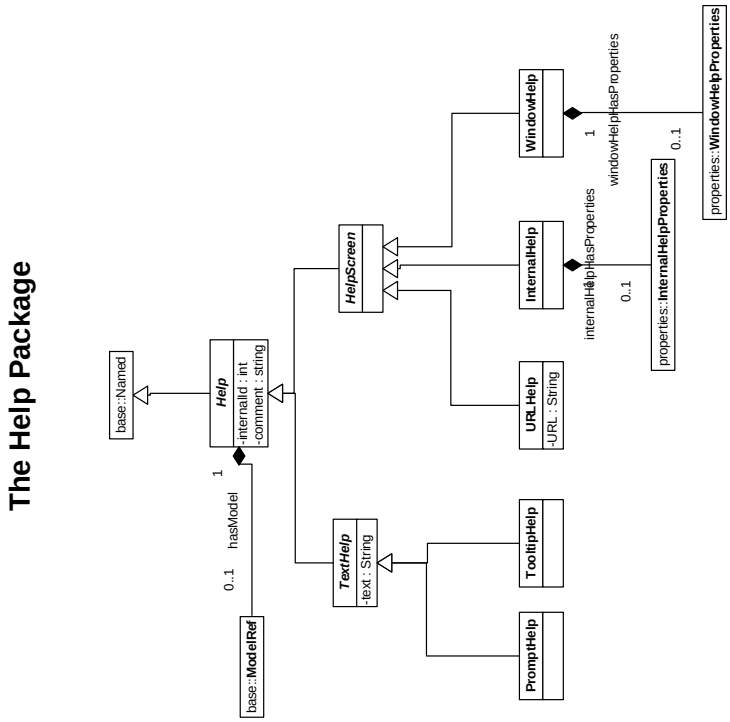



**Figure B.1.13:** Magic Schema - The Help package

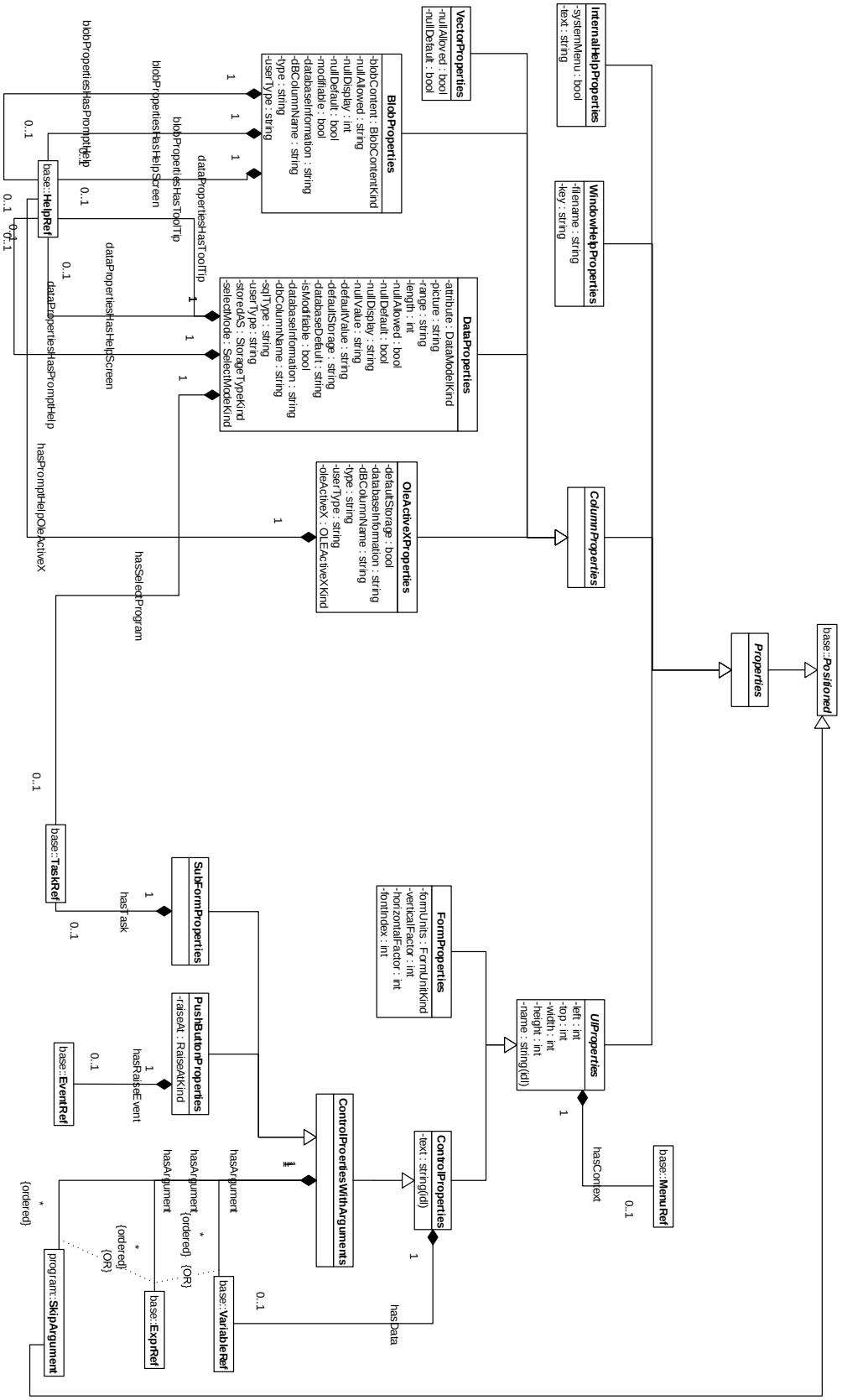



**Figure B.1.14:** Magic Schema - The Properties package

## B.2 Magic metrics

| Name | Description |
|---|---|
| TNLU | Total number of logic units |
| TNME | Total number of menu entries |
| TNPJ | Total number of projects |
| TNPR | Total number of programs |
| TNT | Total number of tasks |
| TNnRLL | Total number of non-remark logic lines |
| TNnRLU | Total number of non-remark logic units |

**Table B.2.1:** Size metrics for Magic applications.

| Name | Description |
|---|---|
| MNP | Maximum number of parameters in a task |
| NA | Number of applications |
| NC | Number of columns |
| NDO | Number of data objects |
| NDS | Number of databases |
| NDSU | Number of databases used in the project |
| NE | Number of expressions |
| NH | Number of helps |
| NI | Number of indexes |
| NIS | Number of index segments |
| NLL | Number of logic lines |
| NLU | Number of logic units |
| NM | Number of models |
| NME | Number of menu entries |
| NOA | Number of ancestors |
| NPJ | Number of projects |
| NPR | Number of programs |
| NPV | Number of parameter variables |
| NPVL | Number of parameter variables locally defined |
| NR | Number of rights |
| NRV | Number of real variables |
| NRVL | Number of real variables locally defined |
| NT | Number of tasks |
| NVV | Number of virtual variables |
| NVVL | Number of virtual variables locally defined |
| NnRLU | Number of non-remark logic units |
| TNC | Total number of columns |
| TNDO | Total number of data objects |
| TNDSU | Total number of data sources used |
| TNE | Total number of expressions |
| TNI | Total number of indexes |
| TNLL | Total number of logic lines |

**Table B.2.2:** Size metrics for Tasks in a Magic application.

| Name | Description |
|---|---|
| CBT | Coupling between tasks |
| CBTDO | Coupling between tasks and tables |
| NCR | Number of column references |
| NII | Number of incoming invocations |
| NOI | Number of outgoing invocations |

**Table B.2.3:** Coupling metrics for a Magic application.

| Name | Description |
|---|---|
| McCC | McCabe's cyclomatic complexity |
| NL | Nesting level |
| WLUT | Weighet complexity of logic units per task |
| NII | Number of incoming invocations |
| NOI | Number of outgoing invocations |

**Table B.2.4:** Complexity metrics for a Magic application.

## B.3 Magic export files

```
VRSN=561

PRG={
HDR={DESC="Pr¢ba",RSDNT=N,SQL=N,PRK=1,POS=M},
RSRCE={
FLD={DESC="Virtual v ltoz¢",ATTR=A,RSRCE={MDFY=Y,TRNS=I,DEF=N,CTRL=E,CTRL_STP="
0.00000",NUL_ALW=N}}},
DTLS={
KEY={MOD=N},
DB={},
FLW=O,DEL=N,END=N,EMOD=B,
MOD={MOD=M},
LCT={DIR=A},
RNG={DIR=A},
FLG={SLCT=N,CNF=N,SLOC=R,SFAIL=S,LSTRG=O,SUFIX=N},
SIDE_WIN={
RPR={RPR=N}},
WIN={OPN=Y,CLS=Y,FGND=Y,FLIP=N,
DTLS={RTRN=N}},
BOX={DX=78,DY=21,DIR=V},
UPD={DATE="09/02/04",TIME="09:22:32"}},
FLW={
BRK={TRNS=U,ERROR=A,PRK={VIEW=7}
VIEW={
RMRK="Megjegyz⊠s",
SLCT={NAME="A",CNF=N,FLD=1,MOD=V,FLW={CND=Y,MOD=S,DIR=C}},
STP={EXP=1,MOD=E,FLW={CND=Y,MOD=B,DIR=C}},
LNK={DIR=A,CLS=5,MOD=N,FLW={CND=Y,MOD=B,DIR=C}},
END_LINK,
BLOCK={END_BLK=7,FLW={CND=Y,MOD=B,DIR=C}},
END_BLK,
CALL_TSK={MOD=T,LOCK=Y,FLW={CND=Y,MOD=S,DIR=C}},
EXE={EXP=1,FLW={CND=Y,MOD=B,DIR=C}},
UPD={FLD="A",EXP=2,MOD=N,ABRT=Y,FLW={CND=Y,MOD=B,DIR=C}},
WRT={MOD=A,IO={},DSP={},FLW={CND=Y,MOD=B,DIR=C}},
READ={MOD=C,IO={},DSP={},DLMR=32,FLW={CND=Y,MOD=B,DIR=C}},
SCN={MOD=E,EDIT=S,FLW={CND=Y,MOD=S,DIR=C}},
EXT={TXT="",CLR=Y,FLW={CND=Y,MOD=S,DIR=C}}}},
BRK={TRNS=N,ERROR=A,PRK={}}},
DSP={NAME="Pr¢ba",MOD=H,COLOR=4,
FRM={Y=1,DX=80,DY=23,COLOR=3,CHAR=D,OPN=N,SYSMNU=N},
BLOCK={DX=78,DY=21,COLOR=1},
ACTV={DX=78,DY=21}},
EXP="A=''",
EXP="''"}
```

**Figure B.3.1:** An example export file from Magic v5.

## B.4 Magic screen shots

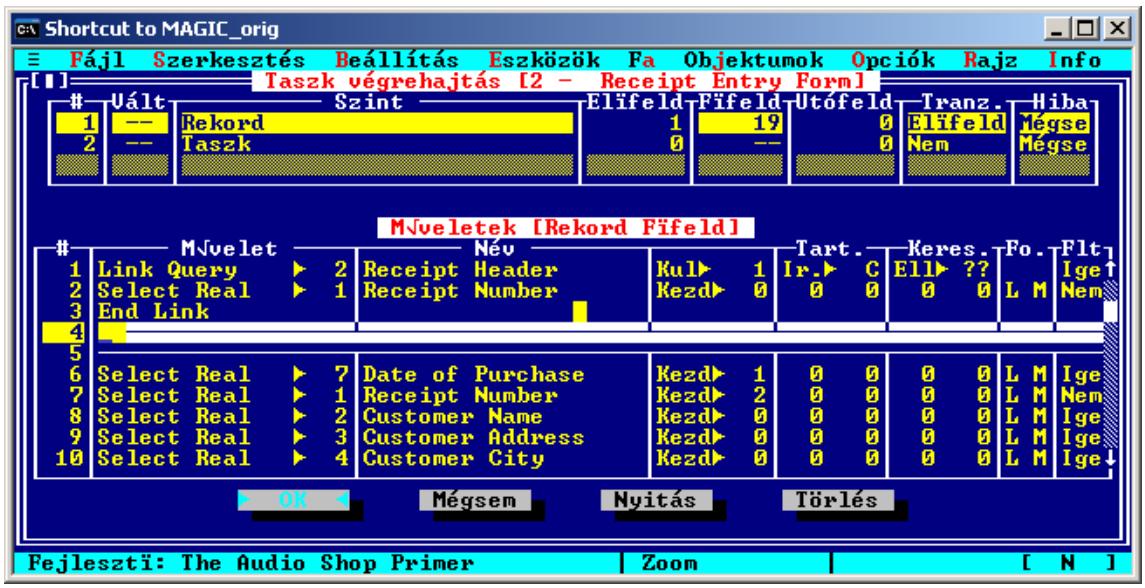



**Figure B.4.1:** An example screen shot of the Task editor of Magic v5.

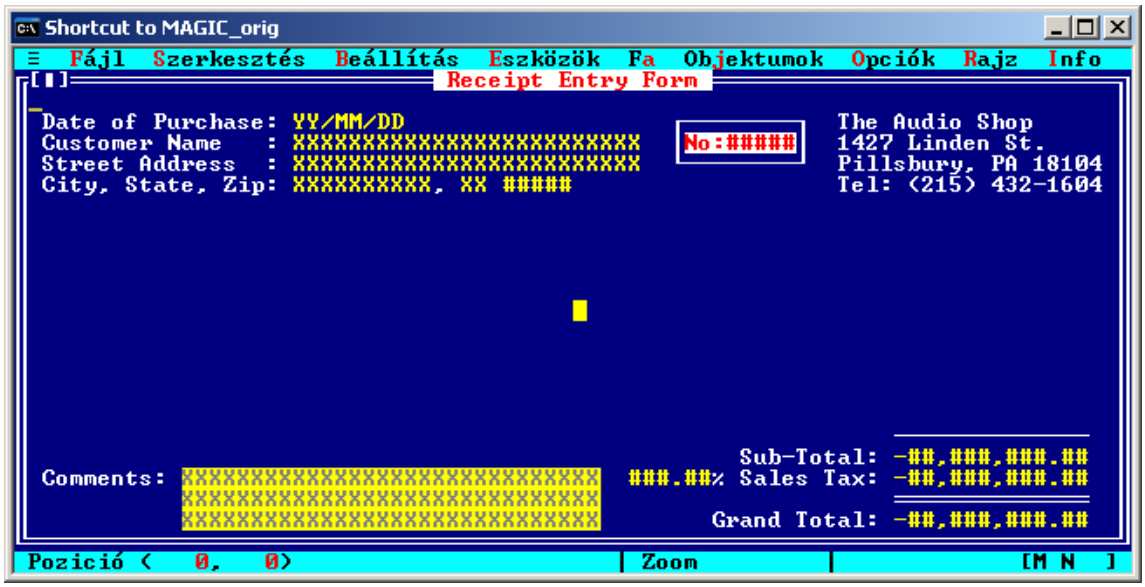



**Figure B.4.2:** An example screen shot of the Form editor of Magic v5.

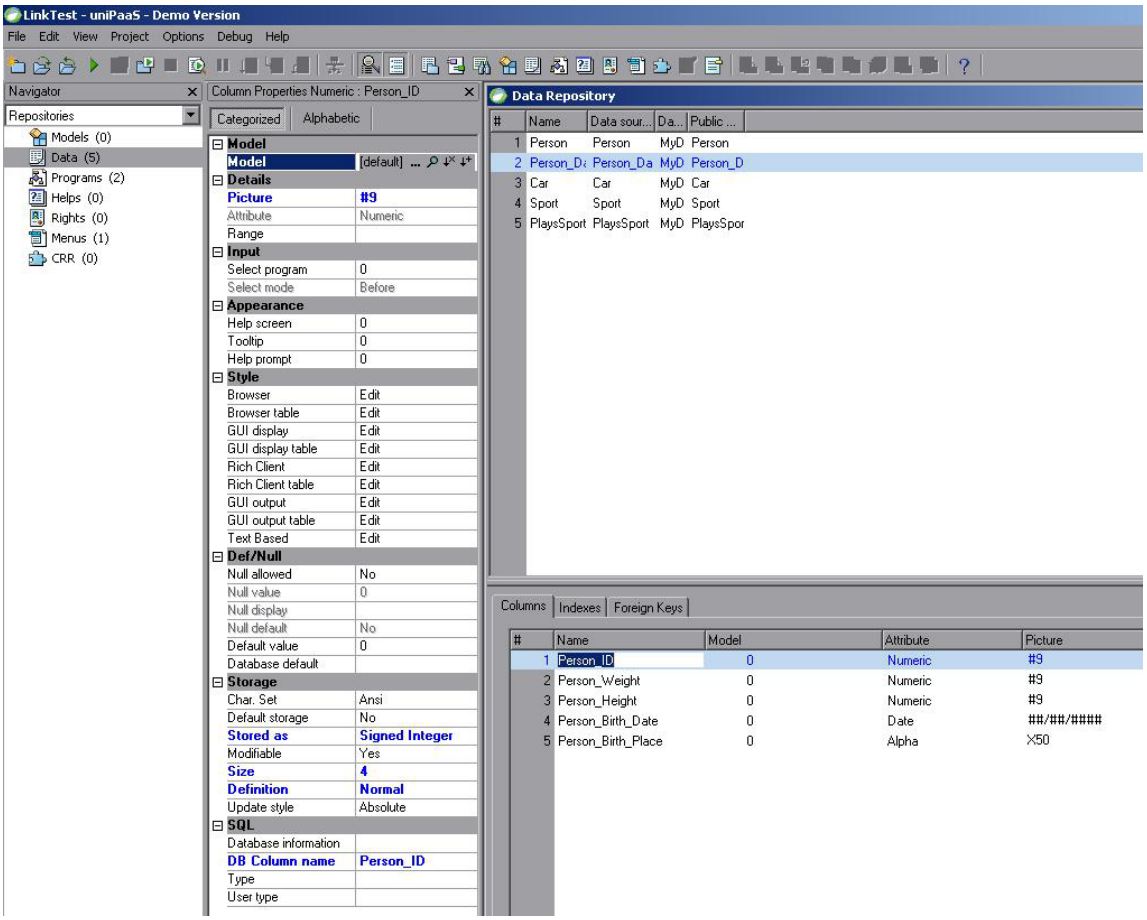



**Figure B.4.3:** An example screen shot of the Data Table editor of uniPaaS.

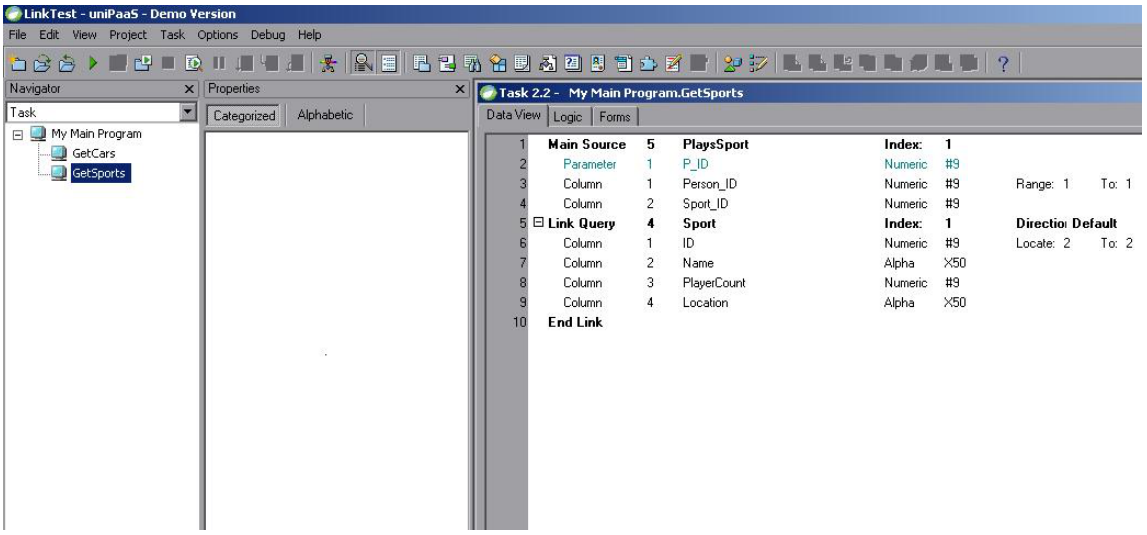



**Figure B.4.4:** An example screen shot of the Data View of uniPaaS.