Evaluating optimization and reverse engineering techniques on data-intensive systems

Csaba Nagy

Department of Software Engineering
University of Szeged

Supervisor: Dr. Tibor Gyimóthy

Summary of the Ph.D. 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

1 Introduction

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 [15].

In 1981, Pawlak published a paper reporting some of the activities of the Information Systems Group in Warsaw [14]. 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 [16]. Another example from the high-energy physics community, the Large Hadron Collider machine, generates 2 PB of data per second when in operation [7]. These systems are usually referred to as *data-intensive systems* [2, 10–12].

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 [3]: "*... 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.*"

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.

Here, 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?

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) Defining and 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

1.2 Publications

Most of the research results presented in this thesis were published in proceedings of international conferences and workshops or journals. Section 5.1 provides a list of selected peer-reviewed publications. Table 1.1 is a summary of which publications cover which results of the thesis.

No.	Contribution - short title	Publications
I/a.	Extracting architectural dependencies in data-intensive systems	[23]
I/b.	Case study of a large banking system	[20, 25]
II/a.	A reverse engineering framework for Magic	[18, 24, 27]
II/b.	Defining and evaluating complexity measures in Magic	[26]
III/a.	Static security analysis	[21]
III/b.	Optimizing information systems: Code factoring in GCC	[19, 22]

Table 1.1: Relation between the thesis topics and the corresponding publications.

2 Architecture recovery of legacy data-intensive systems

In this part, we introduce the analysis techniques we developed for data-intensive systems with database management systems in the center of their architecture.

First, we present a method for extracting dependencies between source code elements and data tables in data-intensive systems based on data accesses via embedded SQL queries (*Create-Retrieve-Update-Delete, CRUD* dependencies) that we use to investigate safe relations, e.g. for impact analysis or architecture recovery. Next, we describe a case study where we combine the previously introduced methods as bottom-up techniques with top-down techniques (interviews with the developers) for reverse engineering a large legacy data-intensive system written in Oracle PL/SQL.

2.1 Extracting architectural dependencies in data-intensive systems

2.1.1 Extracting embedded SQL queries from source code

A typical way of communicating with an RDBMS is by using SQL queries through a library such as JDBC. ORM technologies like Hibernate are becoming popular too, but they also use SQL statements at a lower level . Hence, most of the reverse engineering methods heavily rely on the extraction or capturing of SQL statements used to communicate with the underlying RDBMS.

In our paper [24] we introduced an extraction technique for analyzing SQL statements embedded in the source code of a special procedural language. The programming style of this language makes the whole system strongly database dependent and it makes use of SQL queries common in the system. The SQL statements to be executed are embedded as strings sent to specific library procedures and their results can 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. This makes our method quite general and suitable 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 2.1). With this simple idea we 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. Whenever the constructed string is considered syntactically correct, it has the key characteristics of the executed SQL command.

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

(a)

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

(b)

Figure 2.1: Sample code of an (a) embedded SQL command (b) and its extracted form where the parameter of the LIKE is determined by a variable.

We noticed that in ForrásSQL, developers prefer to prepare statements as close to their execution place as possible and they prefer to keep SQL keywords in separate string literals. So in most cases, it is possible to substitute a variables with its last defined value within the same control block. In other cases the variable can be replaced with the variable name, as described previously.

The technique has its limitations, but in the ForrásSQL context it worked reasonably well. An additional benefit is that it scales well with large systems. With this technique we identified 7, 434 embedded SQL strings (based on the specific SQL library procedure calls) in a 315 kLOC application and we successfully analyzed 6, 499 SQL statements, which is 87% of all the embedded SQL strings.

2.1.2 Dependencies via data accesses in data-intensive systems

An analysis of SQL queries can be carried out to discover dependencies in the software which arise through the database. Such dependencies can help us track the flow of data or discover explicit or implicit relations among source elements.

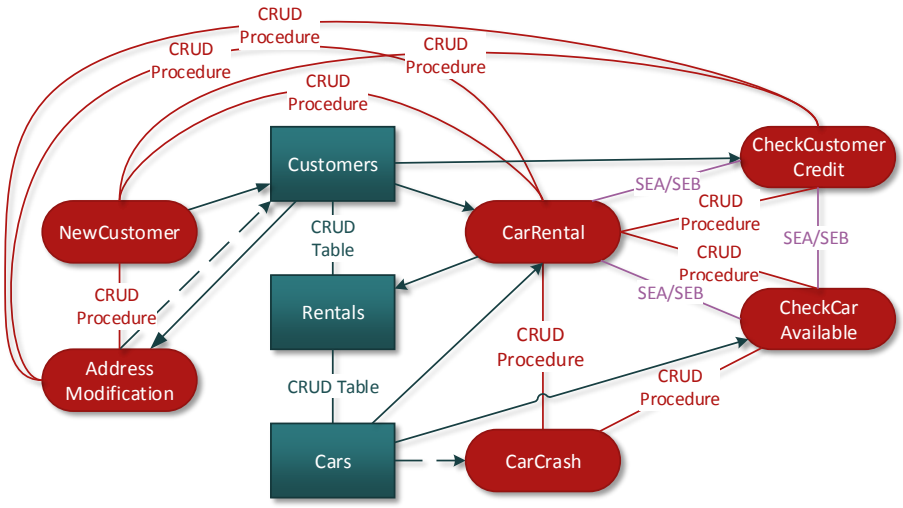

Figure 2.2: Typical *CRUD* and *SEA/SEB* relations between procedures and between tables.

It was shown earlier that *CRUD* matrices are useful tools for supporting program comprehension and quality assessment [1, 17]. In our paper [24], we described the application of a *CRUD*-based Usage Matrix for dependency analysis among program elements (a sample graph representation of *CRUD* relations can be seen in Figure 2.2). In the ForrásSQL system that we analyzed we identified relations among procedures based on table or column accesses and compared these relations to dependencies recovered by *SEA/SEB* relations [6]. The results indicated that the disjoint parts of the relation sets of the two methods were similar in size, and that their intersection was considerably smaller (about 3% of the union). Based on this empirical evaluation, we concluded that neither of the relations was definitely better (safer and more precise) than the other; they were simply different. Thus they should be applied together in situations where a safe result is sought in the context of data-intensive systems.

2.1.3 Own contribution

Designing the SQL extraction algorithm and the relations among source code elements; performing the analysis in the case study and evaluating the results was the work of the author. In addition, the major part of designing the Transact SQL Schema and parsing Transact SQL code was the work of the author too, but designing and implementing the full analysis was carried out as a joint work with co-authors of corresponding paper [23]. For the analysis of ForrásSQL sources, we used the previously implemented ForrásSQL front-end and the analyzers of the Columbus technology, extending SEA/SEB computation for ForrásSQL, which was implemented by the author. Following our contribution, Liu et al. published a similar approach for systems written in PHP [8].

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

In our case study, one of our industrial partners asked us to help them in solving maintenance issues of their huge database system. They had a large Oracle PL/SQL system which had evolved over the years to a system having a database dump with more than 4.1 MLOC (counting just the non-empty and non-comment lines of code, excluding data insertions).

Applying our techniques, we reconstructed an architectural map of their system: we identified high-level components and their related objects during interviews with members of the development team (they had strict naming conventions), and used our previous technique (Section 2.1.2) to identify relations between components (via call and CRUD dependencies). The final results indicated that each of the 26 logical component were related to practically every other component (Figure 2.3). In another part of our analysis, we identified unused

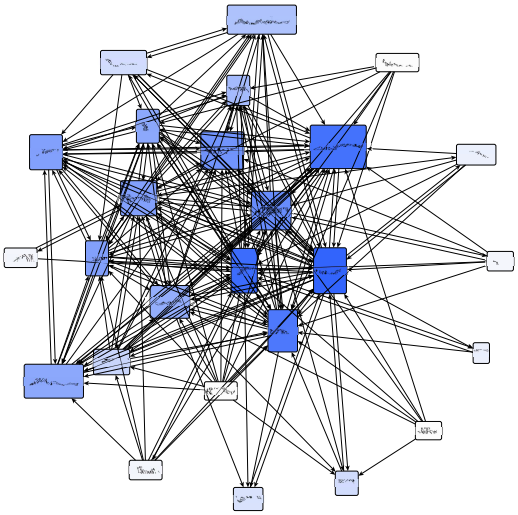

Figure 2.3: Relations among components of a large data-intensive system that evolved in an ad-hoc manner: almost all of the 26 components are related to practically all other component (names distorted).

data objects and objects placed in the wrong components (they logically belonged to other components).

Identified dependencies also supported the elimination of a huge component from the PL/SQL source base, which they re-implemented in Java. With the help of our analysis, they were able to eliminate uncut relations of the unused component. Besides identifying issues in the design of the architecture, using the static code checkers and a clone detector we identified a number of coding issues that needed to be addressed.

2.2.1 Own contribution

To perform our analysis of the Oracle PL/SQL system, we extended the Columbus framework with an Oracle front-end that is able to analyze PL/SQL code and Oracle SQL queries as well. The author's own contribution was mainly in designing the Oracle Schema and in implementing the parser. The author designed the method of retrieving the architecture map of the target system and the method for identifying uncut relations of an unused component. The author carried out an analysis of the target system and discussed results with the developers. The rest of the implementation and the research work was shared work with co-authors of a corresponding publications [20, 25].

3 The world of Magic

Here, we describe how we applied the Columbus methodology to Magic, a fourth generation language. We developed a framework for analyzing quality attributes of Magic applications and extracting architectural dependencies outlined in the previous chapters.

First, we describe our technique for analyzing an application written in Magic by adapting a 3GL reverse engineering methodology called the Columbus methodology. We show that 3GL analysis techniques (e.g. for quality assurance or supporting migration) support the development in this environment as well. Then, we examine a new complexity metric for the Magic language as during the adaptation we realized that 3GL complexity metrics did not fulfill the needs of experienced Magic developers.

3.1 A reverse engineering framework for Magic applications

Fourth generation languages are also referred to as very high level languages. A programmer 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.

Magic, a good example of a 4GL, was introduced by Magic Software Enterprises in the early 80's as an innovative technology to move from code generation to the use of an underlying meta model within an application generator. It was invented for the development of business applications with a special development style that is strongly database oriented. Most of the software elements are related to database entities: fields of records can be simply reached via variables and a task (which is the nearest element we have to a procedure) is always connected to a data table on which it performs operations. Hence, applications developed in Magic can also be treated as data-intensive systems.

With our industrial partner, SZEGED Software Inc., we conducted research work in adapting the Columbus methodology as a 3GL reverse engineering methodology to reverse engineering Magic applications. We investigated how to analyze the source code quality of a Magic application [24] and how to identify architectural relations in such a system (e.g. CRUD relations, as used in the earlier PL/SQL system) [27].

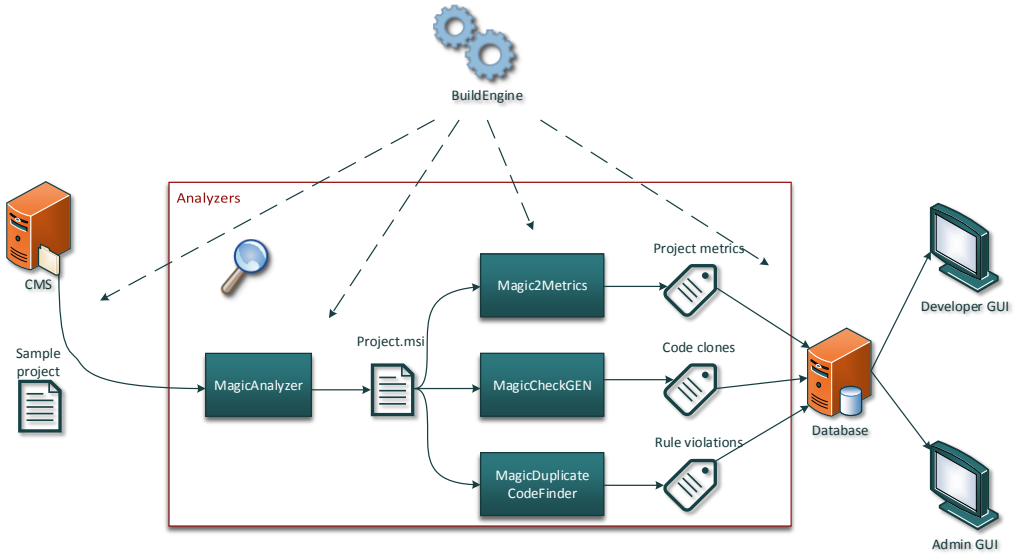

Figure 3.1: Columbus methodology adapted in the Magic environment.

We implemented the whole Columbus methodology for Magic starting from the parsing phases to the computation of product metrics, the identification of coding issues and the extraction of architectural views of the system (see Figure 3.1).

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 3.1: Main characteristics of the system in question.

We showed that existing techniques can be adapted to Magic, but with some differences: (1) quality attributes need to be handled with caution, for example, as the complexity needs to be redefined; (2) the application development environment stores information in its export file which could not be gathered from 3GL code

with a simple computation. For instance, in Magic there is no need to analyze embedded queries to retrieve CRUD relations because data accesses are handled directly via specific variables and this information is readily available in the export file.

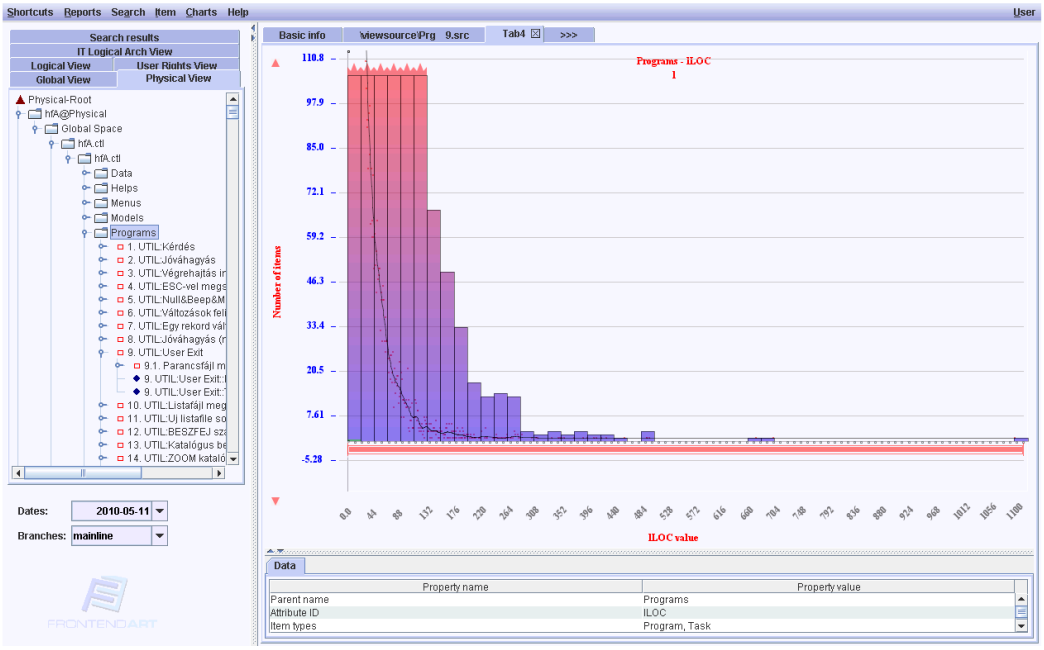

Figure 3.2: Histogram showing the frequency distribution of LLOC metric values of tasks.

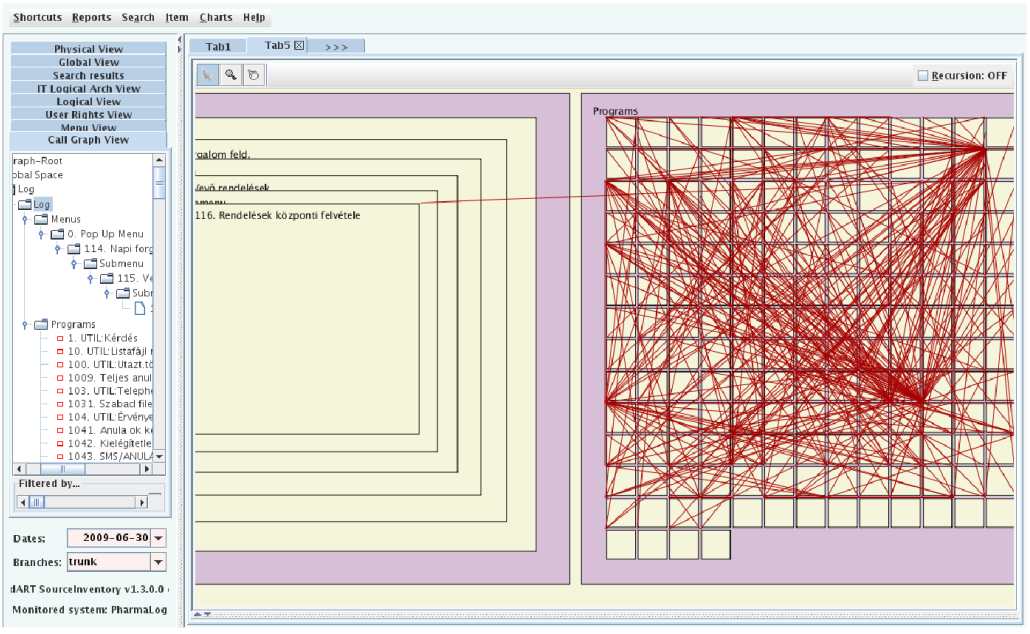

Figure 3.3: Program calls extended with menu accesses.

Thanks to our industrial partner, we had a chance to test our implementation and validate the results in an 'in vivo', industrial context by analyzing a large-scale Magic application. In addition, we received direct feedback from experienced Magic developers. Table 3.1 shows sample metrics of the Magic system that we analyzed while Figure 3.2 and Figure 3.3 show a sample histogram of a selected metric and an architectural view respectively.

The reverse engineering framework served as a good basis for additional quality assurance tasks as well, e.g. in a study we successfully automated the GUI testing of Magic applications based on UI information stored in the constructed ASG [18].

3.1.1 Own contribution

The author's contribution was mainly in designing and implementing the reverse engineering framework for Magic. Although the parser which constructs the ASG was the work of the industrial partner, the author designed the initial versions of the Magic Schema and the APIs to handle it. The author also defined the new metrics for Magic and the different architectural views of Magic applications. The identified coding issues were defined in cooperation with the industrial partner as most of the validation and testing tasks were performed together. The author designed the automatic GUI testing tool published in the corresponding paper [18]. The rest of the work is a shared work of authors of papers [24, 27] and the colleagues who supported these studies.

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

When we investigated the internal structure of Magic programs, we identified key aspects needed to define new metrics and adapt 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. Modifying our measures, we got several experienced developers to participate in an experiment to evaluate different approaches for applying complexity metrics.

We adapted two 3GL complexity metrics to Magic (McCabe's and Halstead's complexity metrics) and, based on the feedback of developers, we implemented a modified version of McCabe's cyclomatic complexity (see Definition 1). With the help of experienced Magic developers, we carried out an experiment to evaluate our approaches and we found no significant correlation between developers ranking and our initial McCabe complexity. In contrast, we found a strong correlation between the modified version of McCabe's complexity the developers' rankings and between Halstead's complexity and their rankings.

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

T: a Task in the Project

LU: a Logic Unit of a Task

TC: a called Task in LU

Def. 1: The definition of McCabe's cyclomatic complexity for Logic Units ($McCC$), Weighted Logic Units per Task ($WLUT$), and the modified McCabe's cyclomatic complexity ($McCC_2$).

Figure 3.4 shows the ranks of Magic program based on their complexity measures. In the figure, the *EC* value (Experiment Complexity) is based on the average rank given by developers, while *HPV* and *HE* are Halstead complexity metrics.

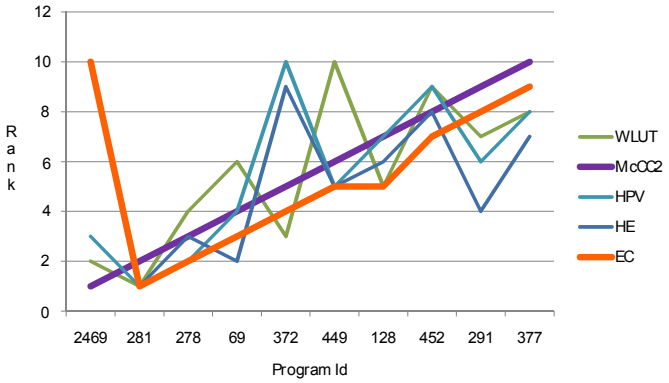

Figure 3.4: The *EC* value (the average rank given by developers) compared with values got from the main complexity metrics. (Weighted Logic Units per Task (*WLUT*), and the modified McCabe's cyclomatic complexity ($McCC_2$), Halstead's *Program Volume* (*HPV*) and *Effort to implement* (*HE*) metrics.)

3.2.1 Own contribution

The author's work was mainly involved in defining the new metrics and conducting the experiment with the developers, although the implementation of the complexity metrics was the work of co-authors of the corresponding paper [26]. We consider the definition of the modified cyclomatic complexity a relevant contribution of the experiment since we defined a new, understandable complexity measure for use by Magic developers.

4 Security and optimization

Now, we will discuss analysis and transformation methods 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; so the results presented here can be applied to applications without databases as well.

First, we present static analysis techniques for applications that work with input data coming from external sources (e.g. from user input or I/O operations). Then we discuss the effectiveness of local code factoring algorithms implemented on different intermediate levels of GCC for optimizing the size of applications written in C or C++.

4.1 Static security analysis based on input-related software faults

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 problems while the program is running. The path which the data travels through can be tracked using *dataflow analysis* 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.

The main steps of our approach (Figure 4.1) 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.

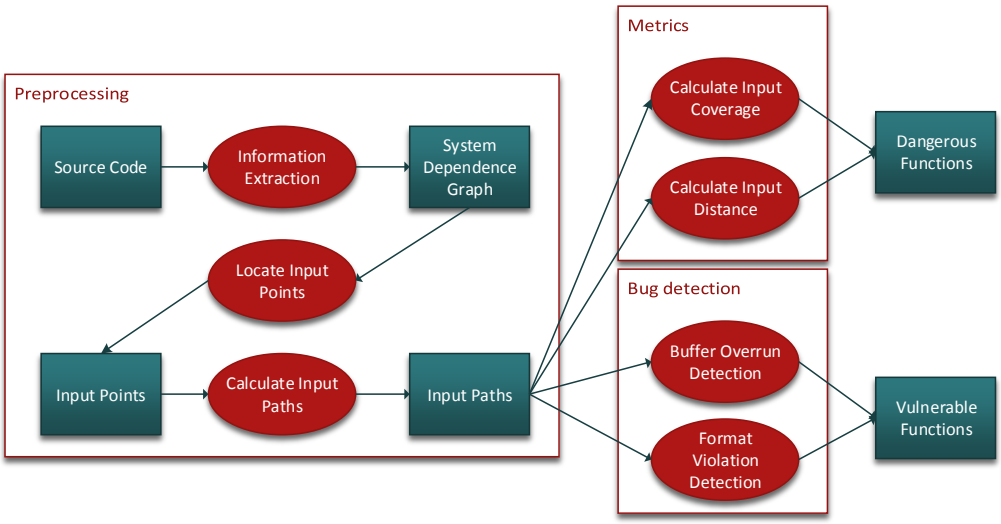

Figure 4.1: An overview of our approach.

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

(a)

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

(b)

Table 4.1: (a) Input operations in Pidgin, (b) and the list of top ten input coverage values of functions.

We presented the results of applying our technique on open source software and described a case study on Pidgin, a popular chat client that we analyzed. We found real faults in Pidgin and in other open source applications. Our approach is novel as it uses input coverage and distance metrics to provide developers with a list of functions that are most likely to contain potential security faults. The Pidgin case study demonstrated the effectiveness of our metrics applied on an application with a total number of 7,173 functions and 229,825 LOCs. Based on our measurements, just 10.56% of the code was found to be related directly to user input and 2,728 functions worked with input data. Some of these values can be seen in Table 4.1.

4.1.1 Own contribution

For the analysis of C and C++ code we used CodeSurfer, which is a product of GrammaTech Inc. The author's own contribution was mainly in designing the algorithms, implementing them as a CodeSurfer plug-in, and evaluating them in the given case study. We consider the approach presented to be a relevant contribution in the area of static security analysis as more studies cited our paper [21].

4.2 Optimizing information systems: code factoring in GCC

Here, we introduce new optimization algorithms implemented at different optimization levels of GCC, the popular open source compiler. The algorithms are so-called code factoring algorithms, a class of useful optimization techniques specially developed for code size reduction. Developers recognized the power of these methods and nowadays several applications use these algorithms for optimization purposes (e.g. 'The Squeeze Project'[1], one of the first projects using this technique).

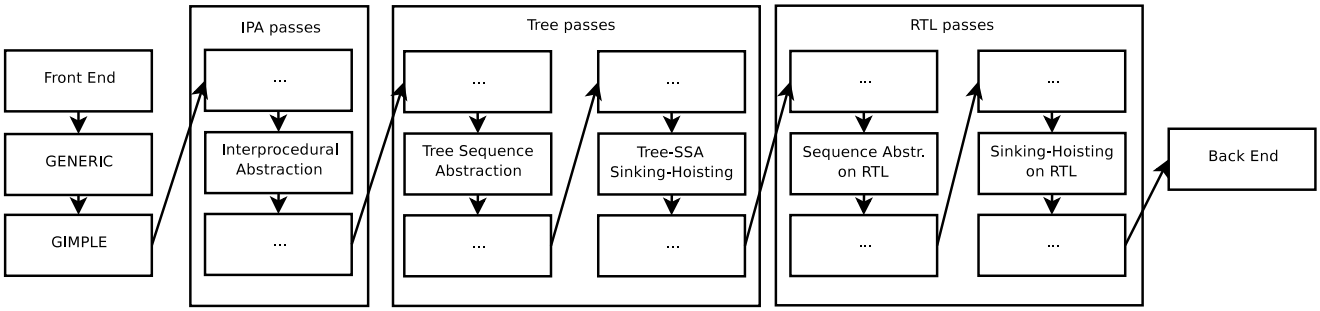

Figure 4.2: An overview of the implemented algorithms.

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. Figure 4.2 shows the order of our new passes.

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 too. 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.

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 that can be turned into procedures. After creating the new procedure, we can simply replace the identical regions with calls to the newly created subroutine. A similar technique can be applied to support *multiple-entry single-exit* (MESE) code fragments as well.

The correctness of the implementation was checked, and the results were measured on different architectures with GCC's official Code-Size Benchmark Environment (CSiBE) as a real-world system (see Table 4.2). These results showed that on the ARM architecture we could achieve a 61.53% maximum and 2.58% average extra code-size saving compared to the '-Os' flag of GCC.

4.2.1 Own contribution

The algorithms were designed based on the previous work published in [9]. The implementation of the sinking-hoisting and the sequence abstraction algorithms on all optimization phases of GCC is a shared work among the

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

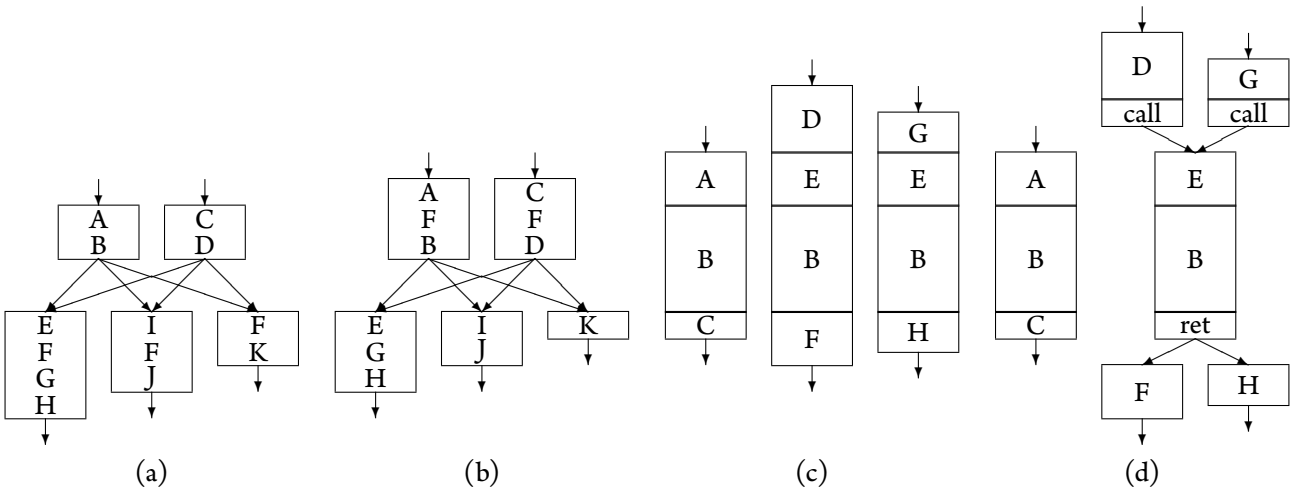

Figure 4.3: Basic blocks with multiple common predecessors (a) before and (b) after local factoring. Abstraction of (c) instruction sequences of differing lengths to procedures using the strategy for abstracting only the longest sequence (d). Identical letters denote identical sequences.

flags	i686-elf			arm-elf		
	size (byte)	avg. saving (%)	max. saving (%)	size (byte)	avg. saving (%)	max. saving (%)
-Os	2900177			3636462		
-Os -ftree-lfact -frtl-lfact	2892432	0.27	6.13	3627070	0.26	10.29
-Os -frtl-lfact	2894531	0.19	4.31	3632454	0.11	4.35
-Os -ftree-lfact	2897382	0.10	5.75	3630378	0.17	10.34
-Os -ftree-seqabstr -frtl-seqabstr	2855823	1.53	36.81	3580846	1.53	56.92
-Os -frtl-seqabstr	2856816	1.50	30.67	3599862	1.01	42.45
-Os -ftree-seqabstr	2888833	0.39	30.60	3610002	0.73	44.72
-Os -fipa-procabstr	2886632	0.47	56.32	3599042	1.03	59.29
all	**2838348**	**2.13**	**57.05**	**3542506**	**2.58**	**61.53**

Table 4.2: Average and maximum code-size saving results. *Size* is given in bytes and *saving* is the size saving correlated to '-Os' in percentage (%).

author and the second author, Gábor Loki of the corresponding paper [22]. The author also carried out the performance measurements and evaluated the results. In addition, the author did the initial work on transforming the ASG of Columbus to the Tree intermediate language of GCC, published in [19].

5 Conclusions

This thesis discusses different techniques for analyzing data-intensive systems and automatically performing transformations on them. Here, we summarize the contributions and conclude the thesis by answering our research questions and elaborate on the main lessons learned.

5.1 Summary of the 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 [24, 27]. 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 [4, 5, 13].

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.

Acknowledgments

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, Dr. 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 and Gábor Novák. 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.

Bibliography

References

[1] 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.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Kaiping Liu, Hee Beng Kuan Tan, and Xu Chen. Extraction of attribute dependency graph from database applications. In *Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference*, pages 138–145. IEEE Computer Society, 2011.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

Corresponding publications of the Thesis

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.