Utilization of Underlying Semantic Information in Textual Data

András Kicsi

Department of Software Engineering
University of Szeged

Szeged, 2022

Supervisor:

Dr. László Vidács

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
OF THE UNIVERSITY OF SZEGED

University of Szeged
Ph.D. School in Computer Science

"It's still magic even if you know how it's done."

— Terry Pratchett, A Hat Full of Sky

Preface

Ever since my childhood, I knew that the world is a wonderful place where one can find magic in nearly everything, only the right viewpoint is needed. This can even apply to a doctoral thesis. I can promise, however, that whatever viewpoint the reader adopts, Magic will still be encountered through these pages in one way or another.

I was always fascinated with stories. They make us think, motivate us, and each of us stars in many little stories of our own every single day. Our stories consist of a cavalcade of exciting words, and together they form our personality. We are going to embark on a journey spanning pharmaceutical supply system features, finding targets for software tests, and even dealing with human spine disorders, but the underlying theme is always the versatility and usefulness of words. Modern technology enables us to teach computers to work with these, extract their meaning, or even use them to construct their own understanding. And this is a wonder by itself, even if we can explain it.

It is my most sincere conviction that other people are what truly make our life worth living. I am incredibly grateful that I have the good fortune of knowing so many great people. I would like to extend an extra special thanks to my family, who have given me a brain and a heart to face the world, and have continuously supported me through it. I would also like to thank my love and my wonderful friends, who truly give merit to my life. Thus, if we assess life in such a way, then mine is truly blessed.

There are so many who helped me along the journey of my studies that I couldn't possibly name them all. My utmost gratitude goes out to my supervisor, László Vidács, who, in my opinion, is the best supervisor I could have ever wished for, and not just for my doctoral studies, but also my work. I will be forever grateful for all the guidance and care he provided through the years. I do not consider myself easily inspired by outside sources, and yet, to this day, he can always manage this. I am exceptionally thankful for my amazing co-authors and my dear colleagues who greatly contributed to the success of our research. To name a few of them, I would like to thank Viktor Csuvik, Klaudia Szabó Ledenyi, Péter Pusztai, and Ferenc Horváth without whom I would have much less to write about now, as their devoted work was indispensable for my own. I would also like to thank Tibor Gyimóthy for providing me with an offer for doctoral studies at the Department of Software Engineering, and many interesting research opportunities ever since. It was a privilege to conduct my studies in such company.[1]

András Kicsi, 2022

[1]My work was also supported by the ÚNKP-21-4-1 and ÚNKP-22-4-1 New National Excellence Program of the Ministry for Innovation and Technology from the Source of the National Research, Development and Innovation Fund.

Contents

List of Tables

List of Figures

"Once you learn to read, you will be forever free."

— Frederick Douglass

1 Introduction

If we look at almost any aspect of civilization, we will find textual information indispensable. Written text has always been the primary source of knowledge, but more importantly, the leading promoter of human organization and cooperation. The more organized side prevailed in nearly any conflict throughout the entirety of history, and organization and cooperation led to our current advances, including our modern society and technology. Never has been more information recorded in textual form than in our present time.

Parallel to this, there has never been a better time to process the information in natural language text. Modern information technology has the capacity to process hundreds of sentences in the blink of an eye and do it with a memory that is vastly superior to the human mind's.

The current work delves into the extraction of information from various textual sources and the possible utilization of such information, often including its combination with other already established methods.

The thesis discusses three main topics: *I. Feature-Extraction in 4GL Systems*, *II. Textual Methods in Aiding Test-to-Code Traceability* and *III. Machine Understanding of Radiologic Reports*. These topics emphasize the importance of underlying semantic knowledge in text and aim to extract and utilize this information from various artefacts. The thesis consists of these three main parts, which correspond to the three thesis points. The methods, experiments and results of the thesis have also been the subject to many of the author's previous publications, of which 16 should be mentioned here. Their contribution to the specific thesis points of the thesis is summarized in Table 1.1.

№	[77]	[67]	[65]	[78]	[68]	[75]	[31]	[30]	[71]	[76]	[66]	[70]	[73]	[72]	[69]	[74]
I.	♦	♦	♦	♦	♦											
II.						♦	♦	♦	♦	♦	♦					
III.												♦	♦	♦	♦	♦

Table 1.1: Thesis contributions and supporting publications

The thesis is structured as follows. The next chapter, Chapter 2, briefly lays down some of the most important concepts spanning multiple parts of this thesis.

The **first part**, which describes the results of our work in the feature-extraction of fourth generation language (4GL) product line adoption, consists of a chapter on this topic, Chapter 3, providing an understanding of what a product line is and why it is challenging to build one from variants. The chapter's main contributions include a feature extraction technique based on call graphs and information retrieval, several new feature level metrics for fourth-generation languages and a community-based feature analysis method.

The **second part**'s only chapter, Chapter 4 investigates textual methods in the interest of finding the classes under test for unit tests. It provides a glance at the importance and current state-of-the-art techniques of test-to-code traceability and establishes that most of them use some forms of textual methods as part of their solution. The chapter aims to provide a detailed analysis of the most commonly used textual techniques and their possible improvements.

The **third part**, dealing with the machine understanding of Hungarian spinal reports, also has only one chapter. Chapter 5 describes how radiologists compose reports and the motivation behind their analysis. The chapter showcases our experiments in identifying three entity classes through machine learning means and extracting their connections via linguistic analysis, eventually constructing a structured representation. The chapter also elaborates on report composition and the importance of correcting lexical errors in order to map the gathered entities to an ontology, resulting in proper identification. It also highlights our solution for the automatic correction of radiologic reports.

Chapter 6 sums up the thesis, while in appendices A and B, brief summaries of the thesis are shown in English and in Hungarian, respectively. The appendices, furthermore, contain brief summaries on the thesis points, as well as the author's contributions and publications.

1.1 Contributions

The ideas, figures, tables and results included in this thesis were published in scientific papers (listed at the end of the thesis). In a nutshell, the author is responsible for the following contributions:

Chapter 2.: The author implemented the information retrieval feature-extraction solution and took part in the planning and coordination of the experiments, including the combination possibilities, new metrics, community detection, and evaluation. He took part in the combination effort between static analysis and information retrieval, both in implementation and the analysis of the results. The author implemented a basic feature-extractor with graphical interface based on information retrieval for the initial approach, and later performed the analysis of the variants, and planned the validation procedure.

Chapter 3.: The author implemented a Latent Semantic Indexing based solution for recovering traceability links and the evaluation code relying on naming conventions and also manual data. He also implemented recovery techniques based on various naming conventions and conducted experiments with them. The author planned and coordi-

nated the manual annotation of the TestRoutes dataset and also the Stack Overflow experiments. He also took part in the evaluation and explanation of various other results and the planning of all of the published experiments.

Chapter 4.: The author laid the groundwork and coordinated the manual annotations, and took a big part in the later refinement of the data. He took part in the planning of all aspects of the machine understanding method and coordinated its implementation. The author planned the functional evaluation and its guidelines. He took a big role in the evaluation and explanation of the results and their implications.

"Maybe stories are just data with a soul."

— Brené Brown

2 Background

The following chapters of the thesis present three distinct topics and the author's role in them. Even though the chapters deal with different domains, there are some underlying concepts that will inevitably come up. Let us address these in a really brief overview first.

Artificial Intelligence and Machine Learning

For a long time, humanity has been obsessed with thinking machines. Computers serve us is every second of our daily life with highly advanced capabilities that were unimaginable just a few years ago. Artificial intelligence (AI) has proven to be able to outperform the human brain at many tasks. Long before the currently fashionable artificial neural networks' debut, there have already been highly advanced heuristic and machine learning algorithms capable of incredible feats. Even these, however, were highly reliant on quality data. Heuristic methods have to be constructed along empirical evidence, and training data is one of the cornerstones of machine learning. In modern machine learning solutions, data is everything. As of now, there is still room for human creativity, and there are countless new solutions that are waiting to be invented. Data is everywhere around us, and it's up to researchers and industrial practitioners to extract the information and use it with care.

Natural Language Processing

Natural languages are the languages that are developed naturally in use. All of us use them on a regular basis which provides our most essential means of communication. Natural languages differ from formal ones as they tend to have more exceptions, irregularities, and as a consequence, complexity. Our current topics mainly concern two of such languages, English and Hungarian. These are highly different. Hungarian is a morphologically rich language with vast amounts of possible suffixes, while this is much less so for English text. The two also highly differ in their sentence structure. Thus, invariability between them is unattainable as of now for almost any automated method aiming to extract information.

Natural language processing (NLP) is the process of analysing the natural language text with various goals. NLP is widely used in several domains, and recent advances really highlight its capabilities. AI assistants can understand our speech without difficulty, and chatbots are capable of expressing human-like behaviour near-flawlessly. Natural language conveys semantic meaning, and thus its processing can bridge some structural boundaries set by domain-specific restrictions such as source code syntax. One specific field of NLP is called information retrieval (IR), which deals with the extraction of valuable information from elements of a text. Our current topics all revolve around information retrieval in one way or another.

Latent Semantic Indexing

LSI [34] is an older technique that has been used as mainstream in many tasks of semantic analysis. It is often considered one of the base techniques of many software engineering research approaches that rely on information retrieval. It relies on semantic information of text handled as vectors and produces a more compact form of vectors that results in the semantically more similar documents obtaining less distance in their vectors. LSI uses singular value decomposition to achieve this task.

Doc2Vec

Doc2Vec was introduced by Google's developers [105] and can be considered an extension of Word2Vec, a word embedding technique commonly used in various machine learning approaches in recent years. It operates with vector representations of words that are transformed to a lower number of dimensions via neural networks. The hidden layer has fewer neurons than the input and output layers, and the weights of the hidden layer provide the word embedding output we need. Thus, similarly to LSI, a more compact representation is constructed. Doc2Vec differs from Word2Vec only in also adding a unique identifier for each document to the input layer, thus distinguishing different documents (like sentences, articles or software code methods), which permits a word to have a different meaning in different contexts.

Long Short-term Memory

Long short-term memory (LSTM) [54] is an artificial neural network used in artificial intelligence and deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections. This makes it possible for the LSTM to process entire data sequences. BiLSTM is the bi-directional version of this model. This consists of two LSTMs: one taking the input in a forward direction and the other in a backwards direction. BiLSTMs increase the amount of information available to the network, improving the context available to the algorithm.

Conditional Random Fields

Conditional random fields (CRF) [84] is a model suitable for sequence learning, which also provides a solution to the label bias problem. CRF estimates the conditional probability of a sequence.

Bidirectional Encoder Representations from Transformers

Bidirectional encoder representations from transformers (BERT) [36] is a transformer-based model that applies the popular attention mechanism to textual context. The model takes into account the words that occur before and after the tokens when representing them. The model can be fine-tuned by adding a single output layer, thus achieving state-of-the-art results on several natural language processing tasks. More specific tasks (downstream tasks) are built into the process, thus resulting in separate fine-tuned models. As a result, the big advantage of BERT is that the difference between the architecture of the pre-trained and the downstream model is minimal. Depending on its size, BERT contains different amounts of encoder layers and a bidirectional self-attention head. Simply put, the architecture of BERT is a set of transformer encoding layers stacked on top of each other.

Static Source Code Analysis

Software code is text structured according to the particular syntax of the language used. The analysis of source code holds a great number of obvious benefits like quality assurance, better compilers, and advanced coding practices. It has been practised and researched for many years. In contrast to dynamic analysis, static analysis does not require the software to be actually run during its examination. Such analysis can involve the construction of an abstract syntax tree (AST), or on an even higher abstraction level, an abstract semantic graph (ASG). By adding function calls as edges, call graphs (CG) can also be constructed. These artefacts serve as potent tools for analysis and play at least some part in many of our current topics.

Evaluation Metrics

There are some universal metrics our results will be displayed with. Let us discuss them here briefly.

Precision is the proportion between correctly detected or retrieved results (relevantResults) and all detected or retrieved results (retrievedResults). It computes as

$$precision = \frac{relevantResults \cap retrievedResults}{retrievedResults}$$

It basically describes what proportion of our results (retrieved tests for traceability or tokens classified as disorders for radiology understanding, for example) was correct.

Recall is the proportion between the correctly detected or retrieved results and all the results that should have been detected or retrieved. It computes as

$$recall = \frac{relevantResults \cap retrievedResults}{relevantResults}$$

It basically describes what proportion of the real results (retrieved tests for traceability or tokens classified as disorders for radiology understanding, for example) was indeed detected or retrieved.

F1-score (or just F-score in our context) is a measure that combines precision and recall is the harmonic mean. This is usually a good indicator of how well a method performs.

Accuracy describes how many of the decisions were on point. In a binary decision case, it factors in true positive (TP), false positive (FP), true negative (TN), and false negative (FN) decisions and computes as

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

In a multi-label classification case, accuracy is computed as

$$accuracy = \frac{CorrectClassifications}{AllClassifications}$$

Part I

Feature-Extraction in 4GL Systems

"Any sufficiently advanced technology is indistinguishable from magic."

— Arthur C. Clarke

3 Feature-Extraction of Magic Applications

3.1 Overview

Let us imagine that we have built a software system. Our imaginary system deals with pharmaceutical wholesaler logistics. We have a client with specific needs and requirements that the software fulfils and improves the client's daily business. We get compensated accordingly, maintain the system, provide regular updates, and address the client's arising needs.

There are, however, more businesses with similar needs. Why would we need to start over and build another system from the basics? Software reuse is a field of software engineering that is widely used in nearly every industrial setting. The same software could be sold to different clients. This is popularly called a clone-and-own [41] model. The parallel versions of the system are called variants.

This seems like an effortless way to make money with little work. Challenges may arise, however. An oversimplified example is illustrated in Figure 3.1. In our imaginary example, our first client had the pharmaceutical supplies packaged into boxes, and warehouse workers managed their storage by manual labour. Our second client wants something similar, but machines heavily aid its storage. New features have to be implemented. The clients can require feature models that are vastly different in some aspects. In a clone-and-own setting, this means that the new features are just added to the previously existing code. What about our old client? Do we also enable these features for them as part of an update? They may not want to pay for it if it does not match their needs, understandably. And what shall we do with the unused features? Delete it from the second variant? As time passes, the feature models resemble ever-growing trees that are very hard to consolidate. Soon, various updates have to be implemented differently for each variant, and we will have created very resource-intensive maintenance necessities for ourselves. It is easy to imagine what complications would arise, should we also consider making variants for even more distinct clients, such as grocery stores. One possible remedy for this predicament is the adoption of software product lines (SPL). This would result in one large system, the product line, instead of many variants. Features could be enabled or disabled according

to each client's needs, and essential updates could get to every affected client. This way, the feature model becomes a more sophisticated tool that can enhance further business with minimizing unnecessary effort.

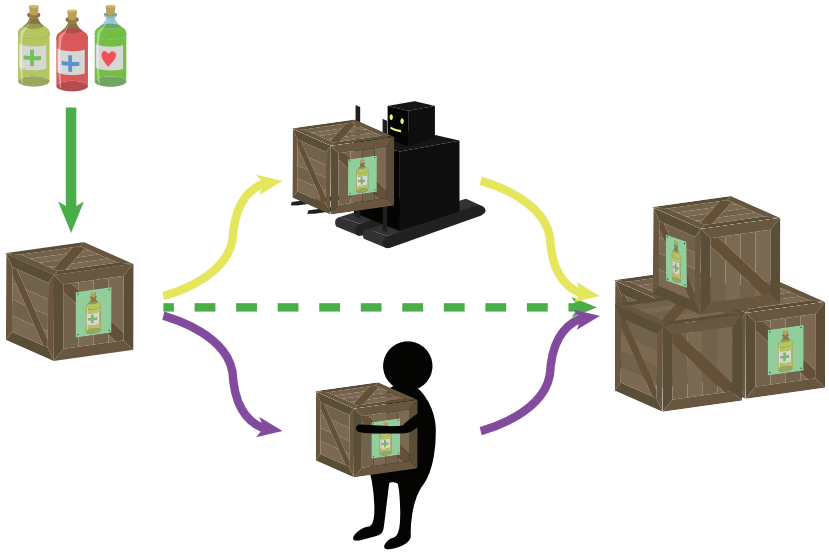

Figure 3.1: A simple example for different features

Maintaining parallel versions of software satisfying various customer needs is challenging. Many times, the clone-and-own solution is chosen because of short term time and effort constraints. As the number of product variants increases, a more viable solution is needed through systematic code reuse. A natural step towards more effective development is the adoption of product line architecture [26]. Product line adoption is usually approached from three directions: the proactive approach starts with domain analysis and applies variability management from scratch. The reactive approach incrementally replies to the new customer needs when they arise. When there are already several systems in production, the extractive approach seems to be the most feasible choice. During the extractive method, the adoption process benefits from systematic reuse of existing design and architectural knowledge [81]. An advantage of the extractive approach, in general, is that several reverse engineering methods exist to support feature extraction and analysis [60, 11, 38].

In the course of an industrial project, we conducted research in such an environment. It was a product line adoption project where the new architecture was based upon an existing variant, and the specific functions of the others were being merged into the final architecture. In principles, this is closest to the extractive approach. Our subject was a legacy high market value, wholesaler logistics system adapted to various domains in the past, using the clone-and-own method. It was developed in a fourth-generation language (4GL) technology, Magic [97]. The product line architecture was built based on an existing set of products developed in the Magic XPA language. This provided some unique challenges. Although there is reverse engineering support for usual maintenance activities [108, 110], the special structure of Magic programs made it necessary to experiment with targeted solutions for coping with features. Furthermore, approaches used in mainstream languages like Java or C++ also needed to be reconsidered for systems developed in 4GLs as they tend to have their unique elements and structure. For instance, in Magic, there is no coding in the traditional sense. The developer instead sets up the user interface, and data processing units in a development

environment and the flow of the program follows a well-defined structure.

The focus of our work was the feature identification and analysis phase of the project. This is a well-studied topic in the literature for mainstream languages [11], but this is much less deeply explored for 4GL environments. Our method started from a set of very high level features provided by domain experts and used information extracted from the existing program code. The information retrieval (IR) approach to feature extraction as a single method turned out to be rather noisy in Magic 4GL system analysis [77]. Hence the extraction was performed by combining and further processing call graph information on the code with the textual similarity between code and high level features. Essentially, the method is working simultaneously with structural (syntactic) and conceptual (text-based) information, similarly to what have been previously proposed for traditional object-oriented systems [5]. While most related literature deals with object-oriented systems, our goal was to aid the product line adoption of an existing 4GL system. This brought not only a distinction from the language perspective but also a less general and more industrially motivated viewpoint. While similar methods could be used in object-oriented environments, these often had to be modified in our case. IR-based methods have fewer problems dealing with the different paradigms. With structural information, however, it can be challenging to achieve the same results as with more simply structured systems. Besides combining the textual extraction with a structural one, our research produced an effective method for filtering the data from both of these sources as well. This can result in information more suitable for performing the SPL adoption for various stakeholders of the project, including domain experts and architects. In summary, the contributions of this chapter are the following:

- A method for feature extraction by combining syntactic and textual information and using filtering results of the two sources.
- Several new metrics for 4GL feature extraction that previously did not exist for the feature level.
- Feature analysis methods by applying community detection algorithms to match features with call graph communities.
- A description of the application of the approach in an industrial setting in 4GL environment during a product line adoption project.

The main goal of our work was to aid our industrial partner in its ongoing product line adoption task. While our results may have the potential to be used in the case of other 4GL languages or other feature extraction work, we did not attempt to introduce a completely foolproof and fully versatile approach for every situation. The chapter is structured as follows. Section 3.2 introduces our current task and defines our approach to tackle the problem. We overview the related work in Section 3.3. Section 3.4 provides more detailed information on our methods for feature extraction and presents our various result sets. Section 3.5 introduces several new metrics and demonstrates their usability in the analysis of four variants of the same system. Section 3.6 describes our experiments on call graph communities. Since we aimed to facilitate the adoption process, an analysis of the project's progress can be found in Section 3.7. We evaluate our various feature extraction outputs and the community detection's information value in Section 3.8 with the help of two research questions that we seek answers to.

Section 3.8 provides some discussion on the results, while we conclude the chapter in Section 3.10.

3.2 Feature Extraction and Abstraction of Magic Applications

3.2.1 The Structure of a Magic Application

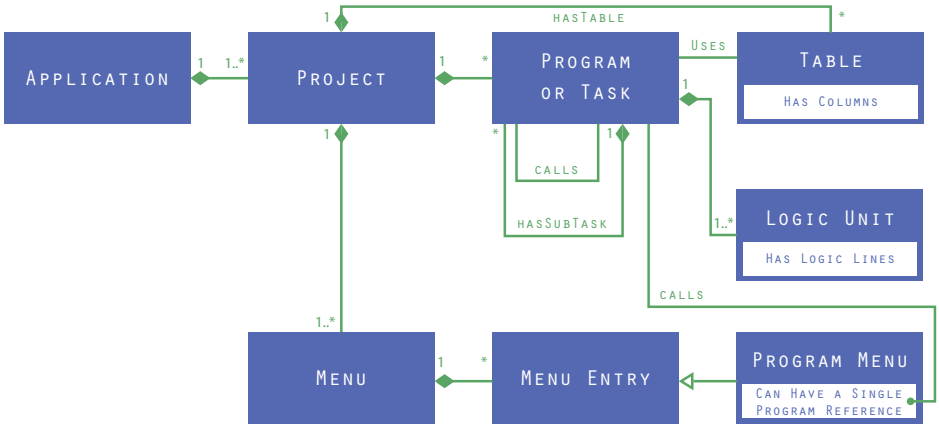

Figure 3.2: Illustration on the elements of a Magic application

Being a fourth-generation language, Magic does not completely follow the structure of a traditional programming language. It has a lot of different elements. In this subsection, we seek to introduce the only ones necessary for the understanding of our approach. Figure 3.2 presents these elements.

A software written in Magic is called an *application*. These can be built up from one or more projects. In turn, each project can have any number of *tasks* containing the software's actual logic. The tasks branching directly from a project are called *programs*. These can have their own subtasks and be called anywhere in the project like methods in a traditional programming environment, but their subtasks can only be called by the (sub)task containing them. Any task or subtask can access data through *tables*.

It is also possible for a program to be called through *menus*, which are controls designed to provide user intervention and usually start a process by calling programs. Having sufficient information on menus, we used these as a base for the call graph in structural feature extraction, deriving calls from menus.

3.2.2 Product Line Adoption in a Clone-and-own Environment

The decision of migrating to a new product line architecture is hard to make. Usually, there is a high number of derived specific products, and the adoption process poses several risks and may take months [27, 25, 23]. The subject system of our analysis is a leading pharmaceutical wholesaler logistics system that was started more than 30 years ago. Meanwhile, almost twenty derived variants of the system were introduced at various complexity and maturity levels with independent life cycles and isolated

maintenance. Our industrial partner is the developer of market-leading solutions in the region, which are implemented in the Magic XPA fourth-generation language.

Our work was part of an industrial project aiming to create a well-designed product line architecture over the isolated variants. The existing set of products provided an appropriate environment for an extractive SPL adoption approach. Characterizing features is usually a manual or semi-automated task, where domain experts, product owners and developers co-operate. Our aim was to aid this process by automatic analysis of the relation of higher-level features and map program level entities to features.

The 4GL environment used to implement the systems requires different approaches and analysis tools than today‘s mainstream languages like Java [108, 49]. The developers work in a fully-fledged development environment by customizing several properties of programs. Magic program analysis tool support is not comparable to mainstream languages. Hence this was a research-intensive project.

The feature extraction process was challenging since the 19 product variants were written in 4 different language versions, Magic V5, Magic V9, UniPaaS 1.9 and XPA 3.x. In the case of the oldest Magic V5 systems, there is a high demand for the migration to a newer version. UniPaaS 1.9 introduced considerable changes in the language by using the .NET engine for applications. Most systems are implemented in this version. The latest Magic XPA 3.x line of the language lies close to the uniPaaS v1.9 systems. Each variant was between 2,000 and 4,000 Magic programs in size with a high amount of code in common. Magic is a data-intensive language, which is clearly reflected in the program code as well, the variants containing a maximum of 822 models and 1,065 data tables.

3.2.3 Feature Extraction Approach

During product line adoption's feature extraction phase, various artefacts are obtained to identify features in an application [13]. This phase is also related to feature location. The analysis phase targets common and variable properties of features and establishes the reengineering phase. This last phase migrates the subject system to the product line architecture.

In this research project, our aim was feature extraction and analysis. Our inputs were the high-level features of the system and the program code. We applied a semi-automated process as in the work of Kastner et al. [60]. Domain experts collect high-level features from the developer company. The actual task is to establish a link between features and the main elements of the Magic applications. Although there exists a common analysis infrastructure for reverse engineering 4GL languages [109, 108, 110], the actual program models differed here.

Figure 3.3 illustrates our current approach to feature extraction. We assigned several elements for each high-level feature, and this information was aiding the work of developers and domain experts working on the new product line architecture. This is the feature extraction phase, which can be seen at the center of the figure with the green feature nodes, and the yellow program nodes are organized in graphs with their connections displayed. During the assignment, we mainly relied on structural information attained on call dependency by constructing a call graph of the programs of a variant. This resulted in a high number of located elements, crucial for the development of product line architecture. However, the large amount of data can be hard to grasp in its entirety. We combined this method with information retrieval, which can

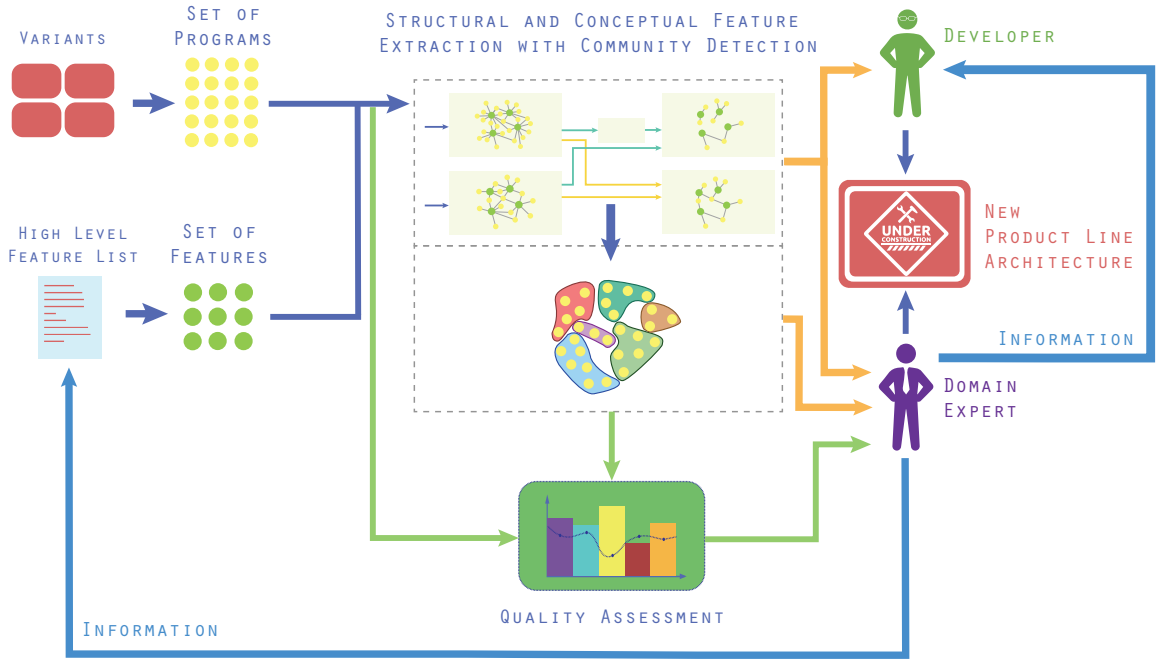

Figure 3.3: An illustration of feature extraction and analysis as part of product line adoption

also make it easier to cope with a 4G language by utilizing conceptual connections and is successfully applied in software development tasks, such as in traceability scenarios for object-oriented languages [99]. A comprehensive overview of NLP techniques – including latent semantic indexing (LSI), the technique we chose – is provided by Falessi et al. [40].

To further assist the work of domain experts, we analyzed communities based on the call graph of the entire system. This can open the way for comparison between the view of domain expert features with code level communities that reflect their implementation. Community detection algorithms address large networks and have already been used for software engineering problems like handling dependencies [48] and support modularization [106]. However, to our knowledge, they haven't been used for product line research previously.

In 2017 [77], we presented our first LSI-based approach for feature extraction. LSI is already known to be capable of producing good quality results combined with structural information [5]. Our works also introduced more than forty new metrics [64] in total for 4GL feature extraction, which included properties based on complexity, coupling, size, and similarity. The advances listed above are elaborated in the following sections.

3.3 Related Work

The literature of reverse engineering 4GL languages is not extensive. By the time the 4GL paradigm has arisen, most papers coped with the role of those languages in software development, including discussions demonstrating their viability. The paradigm is still successful, but only a few works are published about the automatic analysis and modeling of 4GL or specifically Magic applications. The maintenance of Magic appli-

cations is supported by cost-estimation and quality analysis methods [153, 158, 109]. Architectural analysis, reverse engineering and optimization are visible topics in the Magic community [49, 116, 110, 108], and after some years of Magic development, migration to object-oriented languages [94] as well.

SPL has a widespread literature, and over the last decade, it has gained even more popularity. Researchers have tackled all three phases of feature analysis (identification, analysis, and transformation). A mapping study on research works on feature location can be read in [11].

Software product line adoption is a time-consuming task. Several semi-automatic approaches have been proposed [150, 10, 50] to accelerate this activity. Reverse engineering is a popular approach that has recently received increased attention from the research community. With this technique, missing parts can be recovered, feature models can be extracted a set of features, etc. [150, 88]. Applying these approaches, companies can migrate their system into a software product line. However, changing to a new development process is risky and may have unnecessary costs. The work of Krüger et al. [82] supports cost estimations for the extractive approaches and provides a basis for further research.

Efficient community detection algorithms have already been developed which can cope with extensive graphs with millions of nodes and potentially billions of edges [18]. Originally, community detection was primarily applied on graphs that represent complex networks (e.g. social, biological, technological) [43], and have also been suggested for software engineering problems [48]. Besides applications in testing and dependency analysis, software modularization [106] is also addressed by community algorithms. Although modularization is related to reuse and a natural extension of the approach is to use them for feature analysis purposes, we were not aware of previous work on features and community algorithms in the 4GL context. Graph-based methods, in general, usually yield good results and scale well to large systems. For example, in [160] Xue et al. introduced a model differencing technique to detect evolutionary changes to product features. In our process, they only relied on feature sets, while our work points out that the relationships with communities can also be valuable information.

Feature models are considered first-class artefacts in variability modelling. Haslinger et al. [50] presented an algorithm that reversed engineers a feature model for a given SPL from feature sets that describe the characteristics each product variant provides. She et al. [135] analyzed Linux kernel (which is a standard subject in variability analysis) configurations to obtain feature models. LSI has been applied for recovering traceability links between various software artefacts, even in feature extraction experiments [161, 4, 37]. The work of Marcus and Maletic [99] is an early paper on applying LSI for this purpose. Eyal-Salman et al. [38, 37] used LSI to recover traceability links between features and source code with about 80% success rate, but experiments were done only for a small set of features of a simple Java program. The main contrast between Eyal-Salman et al.[37] and our methods is that instead of using family models and test cases to refine LSI results, we utilized the information of call graphs and communities for this purpose. IR-based solution for feature extraction was combined with structural information in the work of Al-msie'deen et al. [5]. Further research dealt with constraints in a semi-automatic way [12] both for functional and even nonfunctional [136] feature requirements. For an overview of analysis methods in product line research, see Thum et al. [146].

Substantial research on similar fields already pointed out that features are not

independent of each other, and the structure of source code resembles the structure of features [160, 79, 122]. Building on this intuition, metrics can be defined [122] based on structural similarity and can be used in a feature location method within an iterative context-aware approach. Besides these metrics, communities also provide relevant information about the structure of the program code. In further contrast, our approach considers the internal structure of software for determining the relevance of the program elements to the features.

Several studies [149, 151, 62, 120] have shown that variability analysis across different software levels is able to come up with promising results. For example, in [62] the authors presented an approach for commonality and variability analysis across multiple software projects at different levels of granularity. They evaluated their work on 19 C/C++ operating systems, while our approach was tested in the Magic programming language. Feature levels are also a novelty in variability analysis.

Static methods individually are commonly used for the detection of features [79, 60, 120], but the combination of those is not a common practice in the field. Hybrid approaches [4] combine two or more types of analysis to use one type of analysis to compensate for the limitations of another, thus achieving better results. Dynamic analysis[79, 117, 98] is utilized in some of the related research to analyze execution scenarios by using methods that collect and analyze information about the execution of the artefacts. Search-based strategies [93, 149] apply algorithms from the optimization field, such as genetic algorithms in the localization features.

Measuring the complexity of software at the source code level is approached from many directions. First, and still popular complexity measures (McCabe [103], Halstead [47], Lines of Code [8]) were surveyed by Navlakha [111]. A survey that sums up complexity measures was published by Sheng Yu et al. [167]. In the mainstream programming language context, there are papers available that analyze the correlation between certain complexity metrics. For instance, Meulen et al. [152] showed that there are solid connections between LOC and HCM, as well as between LOC and CCM in C/C++ programs. For other 4GLs, there have been some endeavors to define metrics to measure the size of a project [153], [158], [96]. There are also some industrial solutions to measure metrics in the 4GL environment. For instance, *RainCode Roadmap*[1] for Informix 4GL provides a set of predefined metrics about code complexity (number of statements, cyclomatic complexity, nesting level), concerning the structured query language, and about lines. In the world of Magic, there is a tool for optimization purposes too called Magic Optimizer[2] which can be utilized to perform static analysis of Magic applications. It does not provide metrics, but it can locate potential coding problems.

As it is visible, product line adoption is a widespread area, and the evaluation of the research results is often a challenging task. In a traditional programming language environment, several approaches were introduced to overcome these obstacles. For example, in a similar work [53] where structural and lexical information were combined, researchers used the well-known precision and recall metrics to compare their approach against state-of-the-art techniques. This method is similarly used in several works [155, 6, 161, 101]. Although the evaluation process is qualitative and seems appropriate, it is not applicable in every case [7, 123, 124] which is even more valid for non-traditional programming languages like Magic. Applying existing approaches in 4GL presents

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

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

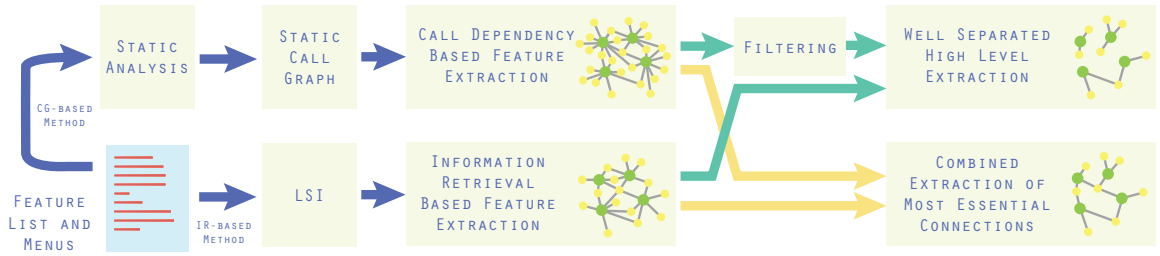

Figure 3.4: A more detailed view on the feature extraction process

several obstacles since the entire environment is very different. The structure of these systems also differs from the conventional projects. Finding an open-source project is a very hard task. It is clear that the evaluation in these cases should be unique, and new approaches are needed.

Several existing approaches can be adapted to the 4GL environment, although none of the papers we encountered cope with Magic product line adoption directly.

3.4 Feature Extraction Experiments

3.4.1 Overview

In this section, we present our feature extraction methods based on call dependency and textual similarity, as well as the combination and possible filtering options. Figure 3.4 illustrates the processes described in this section. Our static analysis is specific to the Magic language. One of our subject system variants was selected by domain experts to be used as a starting point for product line adoption. It was a specific variant involving 4,251 programs, 822 models and 1,065 data tables. The new product line started from this variant. The capabilities of the other variants were being built into the product line during the adoption process. A feature list has been provided for us, structured in a tree format, consisting of three levels which had 10, 42 and 118 unique elements, respectively. This list was being refined in an iterative manner. From these, the upper level is used throughout the thesis to display our results. The features of this level are listed in Figure 3.5. The numbers shown here are in accordance with the numbers in the labels in our later graph examples.

1 – Manufacturing	6 – Administrator interventions
2 – Interface	7 – Supplier order management
3 – Access management	8 – Invoicing
4 – Quality control	9 – Master file maintenance
5 – Stock control	10 – Customer order reception

Figure 3.5: The higher level features of the system

3.4.2 Approach

Feature Extraction Using (Task) Call Dependency

This approach relies on the call dependencies between programs and tasks. To construct a call graph from these dependencies, we used the process illustrated in Figure 3.6. The figure represents a minimalistic example of a Magic application. Squares mean tasks and programs, while other program elements like projects, logic units, logic lines, etc., are shown as circles. The abstract semantic graph is constructed from the source code provided by our static source code analysis tool. As the next step, we added the call edges to the graph by examining Magic elements that operate as calls between tasks and programs. Finally, in the last two steps of the process, we eliminated some nodes and edges from the graph, keeping only the necessary ones *i.e.,* call edges, tasks, and programs. From the CG, we obtained the features by running a customized breadth-first search algorithm from specific starting points determined by menu entries. The domain experts assembling the feature list know the menu structure well. Hence we considered the given feature-menu connections a good starting point for call graph construction. A graph representation of the CG based results can be seen later on the left side of Figure 3.8.

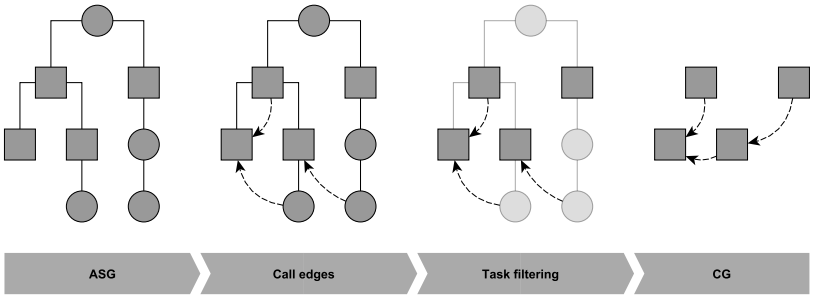

Figure 3.6: The process of calculating the call graph

Textual Similarity

By the textual nature of information retrieval techniques, they are less susceptible to various difficulties some other techniques face. In our case, one of the biggest problems is that the systems processed use different versions of the Magic language. Thus, IR can contribute to a more versatile approach. For this purpose, we used the Latent Semantic Indexing technique [34] to measure textual similarity. A more complete summary of our feature extraction work with LSI is presented in [77]. Similarly to the CG technique, we also utilized information retrieval to determine connections between features and programs of the system. Though the structural information obtained from the call graph is more thorough, it can be more fitting for the developers rather than domain experts. With purely structural information, it is hard to separate along features. Having many programs laying the groundwork for any single feature, it is hard to grasp the overall aim. This conceptual analysis separates more agreeably according to the semantics of the feature. Hence, it can be more valuable for domain experts. A graph representation of textual similarity results can be seen later on the right side of Figure 3.8.

3.4.3 Results and Further Possibilities

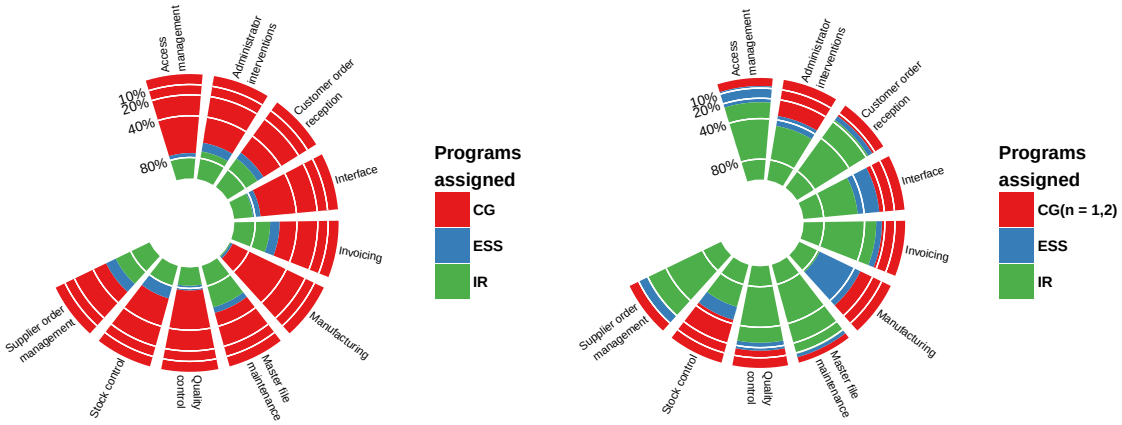

Figure 3.7: The size of our result sets for each feature with each technique. Results without filtering shown on the left, results with filtering on the right. (For interpretation of the references to color in the text, the reader is advised to observe the web version of [68].)

The two methods we used for program assignment to features use fundamentally different strategies for achieving their results. Consequently, the results themselves also show a significant difference, overlapping only partially. The set of programs for the techniques presented are shown on the left side of Figure 3.7. Each slice of the diagram represents a top-level feature, and its colors indicate the number of programs detected by each technique. IR represents the result set of the information retrieval technique, CG represents the pairs attained by call graph, while ESS represents the set of programs considered most essential, detected by both techniques. The left side of Figure 3.9 shows the number of programs assigned in each set. The abbreviations match the ones explained at the previous figure.

The call graph dependency technique provides a high number of programs for each feature, as displayed on the left side of Figure 3.8. These relations are based on real calls of the code itself, which can be considered a credible source of information. It is important to note that we only used static call information, thus at runtime, not every call occurs inevitably. Developers need to work with programs, hence they are required to have some readily available information on all of them. The call information, which this technique uncovers, is likely to benefit their basic understanding of the programs. Still, it also presents a problem of coping with a large amount of data not really distinguishable in any manner.

The conceptual method produces fewer programs for each feature, as can be seen on the right side of Figure 3.8 where we show a graph representation of the IR-based results. Further examination of random cases revealed that even considering this, a significant amount of noise presents itself. Textual similarity works with very little information in these cases. Hence it is likely for similar wording or more general words like "list" to produce misleading matches, occurring in the text of many features. So this method can connect radically different parts of a system both in the aspect of features and the system structure. Thus, we considered this conceptual method less precise.

Figure 3.8: Graph visualization of the set of results obtained by the call graph (Left) and the information retrieval (Right) technique

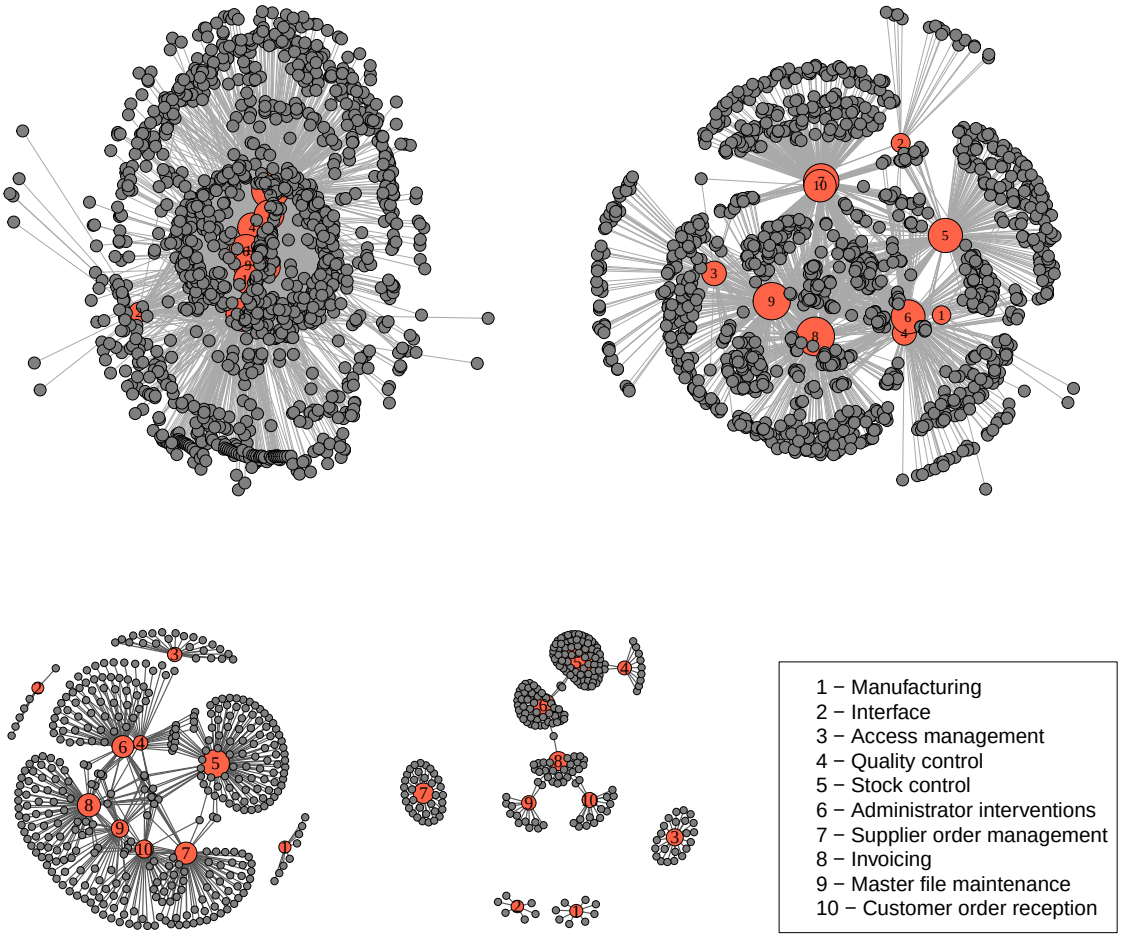

Figure 3.9: Graph visualization of the set of programs deemed most essential. Results shown on the left, results with filtering on the right

Looking at only the intersection of the connections found by these two techniques, we found that this set of connections considers both the structural and conceptual information, producing only connections that are indeed present on both levels. This results in a clearer, more straightforward set of connections, which contains the most essential findings of the two techniques.

As we could see before, the structural information produces many matches for each feature, and we observed a considerable overlap between features. We decided to clear these matches too with a filtering technique applied to the structural information output, which filters out less specific programs. The filtering technique works with a number n, which denotes the maximal number of features a program can connect before it is considered less specific and is filtered out from the program set of features. This removes the programs with less information value and results in even simpler groups of programs for each feature. We have to note that less specific programs are often no less important, but their relations are harder to comprehend, which was our focus in the current case. On the right side of Figure 3.7 and Figure 3.9, the results of the

common structural and conceptual connections of this filtered approach can be seen, featuring only programs with a maximum of two connections. It is apparent from the graph that features are much better separated, providing a suitable high-level glance at the background of features without a lot of technical details, ideal for top-level understanding.

Examining the graphs, several interesting conclusions can be made. For example, feature number 7 behaves like any other feature considering the purely conceptual or purely structural viewpoint. Its common graph provides a clearer picture, apparently connecting through a group of more general features to a large number of other features. However, in the filtered case, it is nicely separated with a group of unique programs specific to the feature.

3.5 Metrics for 4GL Feature Extraction

We can consider features as sets of Magic programs that take part in their implementation. These sets usually have overlaps since programs can be used by more than one feature. In this section, our results computed on the top-level features are presented, which involves ten features as previously seen in Figure 3.5 and represent the system's primary functions. Their dimensions after extraction can be observed in Table 3.1 and Figure 3.10, the values representing the number of programs implementing each feature, working with more than 2,000 programs in total. Four variants of the subject system were considered.

Table 3.1: The recovered number of programs for each feature of the variants with call-graph (CG) and information retrieval (IR) based extraction

Variant	CG-V1	CG-V2	CG-V3	CG-V4	IR-V1	IR-V2	IR-V3	IR-V4
Manufacturing	49	48	47	405	12	13	12	12
Interface	5	5	5	68	36	43	34	22
Access management	13	83	12	421	37	44	36	125
Quality control	152	146	146	441	60	82	60	113
Stock control	348	352	339	769	208	225	209	312
Administrator interventions	198	196	190	647	202	392	205	312
Supplier order management	156	155	156	466	206	235	201	335
Invoicing	272	267	265	602	278	299	274	394
Master file maintenance	70	68	66	467	266	299	259	374
Customer orders	294	290	288	457	193	208	190	276

The four variants under consideration will be mentioned simply as V1, V2, V3 and V4. These are all real variants of the system that were under use by customers of our industrial partner. To our knowledge, V4 differs from the other variants significantly, while the others share a great number of programs and support a very similar set of functions but still vary somewhat in the specifics. Table 3.2 displays the size and properties of these four variants. From this data, it is even more clear that V4 is the largest variant, even though V2 has the most tasks. V1 and V3 are very similar according to both our experiences and their characteristics.

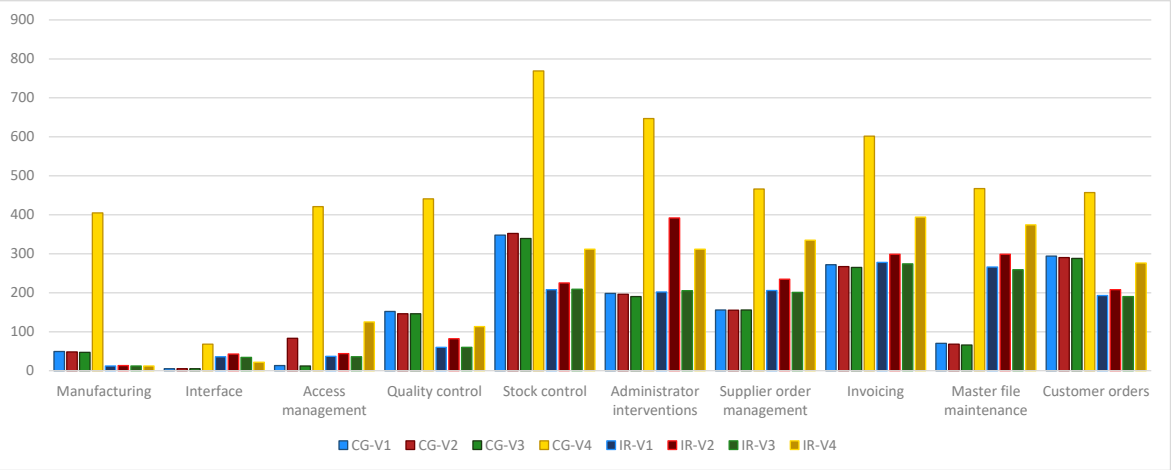

Figure 3.10: Number of programs for each feature with call-graph (CG) and the information retrieval (IR) based extraction

Table 3.2: The characteristics of the variants under analysis

Variant	Logic Lines	Tasks	Programs	Data Objects
V1	366,328	13,365	2,001	699
V2	467,823	25,457	2,719	703
V3	355,604	13,151	2,001	697
V4	518,304	18,291	4,251	1,065

3.5.1 Definitions

In the following, we define several metrics that we deemed suitable for measuring the properties of features. Some of our proposed feature level metrics are extensions of already defined metrics on the program level. In many cases, these can be summed up or averaged to get suitable measurements for feature level too. In other cases, we define new ways that didn't exist on the program level. Feature level metrics can also be utilized to analyze the whole system itself or the progress of feature extraction. The proposed size and similarity metrics are much simpler than our proposed coupling and complexity metrics, thus only a short summary is displayed for these.

Size

Since at the feature extraction phase we worked on assigning programs to specific features, the most straightforward metrics to think of here is the number of programs assigned to each feature. On the other hand, the number of features assigned to each program can also be measured. These metrics provide a basic understanding of the size of features and the importance of the specific programs in their operation. These can be crucially important since we can get a picture of the complications resulting from changes implemented in a single program. Since a Magic application can have more than one project, the number of these is also important on the feature level. In our case, the applications always had one single project.

Programs also access data objects to gain the data needed to perform their tasks. Reliance on data objects can also be important. One such metric is the total number of

data objects used. Since data objects are located in data sources and consist of columns and can even have indices, the number of these are also measured. In our case, there are three different data sources, of which most features tend to use all three.

Similarity

Feature similarity can be measured for each pair of features. Computing these metrics to features of the same variant can also have its uses, for example, if we are contemplating merging some features on the lower level. Still, the central importance lies in computing difference between the same feature of two different variants. Since we can handle features as program sets, the simplest approach is the number of common programs or the absolute difference in the number of programs. The average number of programs or the proportion of the size of the smaller feature relative to the larger one can also provide a quick and easy glance at their similarity. These metrics are based on feature size, and while they can indicate similarity, we also introduced some textual methods. The number of unique words in the features' programs also tends to correlate with size but works at a more conceptual level. We can also calculate the proportion of common words. This, for instance, reveals that V4 differs from the rest of the variants greatly, which corresponds to our previous knowledge. Textual matching can also be applied as only partial matching, as when one term appears inside another. We computed both the number and proportion of these sub matches. For these textual metrics, textual preprocessing was applied to the natural language segments of the programs.

Coupling

Coupling metrics for features represent the differences in their internal and external references. A high number of valid external references can possibly imply a bad modular design.

Coupling Between Features (CBF): One of the most straightforward coupling metrics can be the number of features a feature in question is calling. This number is computed by inspecting the programs of the features. Since features can be viewed as sets of programs, there is a call between features if there is at least one call between the set of programs of both features. Coupling between features measures only the outgoing calls from each feature. It can be viewed as a metric that lets us know how many other features a single feature depends on. Being aware of dependence can be valuable for instance if we decide not to provide a feature for a customer, the other features that depend on this can potentially be hindered.

Coupling Between Programs Inside Features (CBTIF): While coupling between features measures the outgoing calls from each feature, knowing the inside structure of a feature can be just as important. For this, we can count the calls with both participating programs inside the same feature. This is coupling between programs inside a feature, which does not represent a number of features but a number of calls.

Sum of Coupling Between Programs and Data Objects (SCTBDO): Feature coupling can also be viewed at the data object level. The number of utilized data objects is a metric that already exists for each program. It is helpful in measuring the extent of data used by each program. The total of this number for each feature can give us the sum of coupling between programs and data objects. This can be interesting information on the feature level since we can see how much each feature relies on stored

data, thus getting a more complete picture of the feature itself.

Program Clarity (PC): Another piece of interesting information can be derived from the number of features a single program is connected to. Suppose a program is only connected to a sole feature. Then, it can be more easily maintained with less consideration of the subsequent changes in functionality because the change only affects one feature. Additionally, if the customer decides not to require a specific feature, the programs of this feature can be excluded. Program clarity, a number indicating this condition, represents the percentage of programs that are unique to a single feature.

Complexity

The quantification of software complexity is a basic idea that is widely used throughout software development. Complexity can be defined in many ways, and this is the same with feature complexity. Since a feature is handled as a set of programs that aims to reach a common goal, the complexity of the feature is derived from the complexity of its programs.

As established in the paper of Nagy et al. [109] in the Magic context, the widely used McCabe complexity measure is not really suitable for representing the real complexity of a program. On the other hand, Halstead complexity metics [47] correlated well with the opinions of the experienced developers involved in the study. To compute Halstead complexity metrics, we use the following values:

- 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

Halstead Volume (HV): It represents the magnitude of information inside a feature, more precisely the bits required for its code. This can also be interpreted as a measure describing the amount of information a reader of the code must attain to completely understand the feature itself. Since a feature can involve many programs, this number is usually very high. As it can be seen from its formula, this measure involves all operators and operands in the feature. It computes as follows:

$$HV = (N_1 + N_2) * log_2(n_1 + n_2)$$

Halstead Difficulty (HD): It can be used to measure fault sensitivity. The important factors in this property are the number of distinct operators inside a feature and the ratio of all and distinct operands. Both of these properties result in more fault sensitivity. It computes as follows:

$$HD = \left(\frac{n_1}{2}\right) * \left(\frac{N_2}{n_2}\right)$$

Other metrics that can accurately measure complexity on feature level include Halstead Vocabulary, which describes the sum of all distinct operators and distinct operands in a feature, Halstead Effort (HE), which describes the product of volume and effort and represents the developer effort of the code. Henry-Kafura Complexity

(HKC) can also be valuable, which works with the inner and outer calls of the features, practically building complexity on coupling information. It computes as:

$$HKC = (N_1 + N_2) * (\text{fan-in} * \text{fan-out})^2$$

During our work we also computed these measures.

3.5.2 Experiments

In the remaining part of the section, we present a number of results and discuss the meaning of the data retrieved. Experiments were done on all top-level features of all four variants on both call graph (CG), and information retrieval (IR) based feature extraction outputs. We present the results in graphic format. Due to space limitations and for the elimination of monotony, we only display the results found most notable.

We can note that through all variants, the IR technique seems to provide more stable numbers, while the CG technique usually shows significant differences between variants and even features of the same variant. This does not mean that the CG technique would be inferior in any way, and from our previous knowledge, we are aware that the output of the IR based extraction contains a large amount of noise as a result of short feature names which have served as queries for LSI. The seemingly more stable results can even be a consequence of the noise itself. On the other hand, the CG-based extraction can produce a variable number of programs for each feature and each variant. These calls found by the call graph technique are present in the system and provide a less conceptual grouping. It is also important to note that with the call graph technique, we found a high number of more general programs that are connected to nearly every feature. Even considering these differences, we can find that the results of the metrics still move along very similar curves in the case of both extraction techniques.

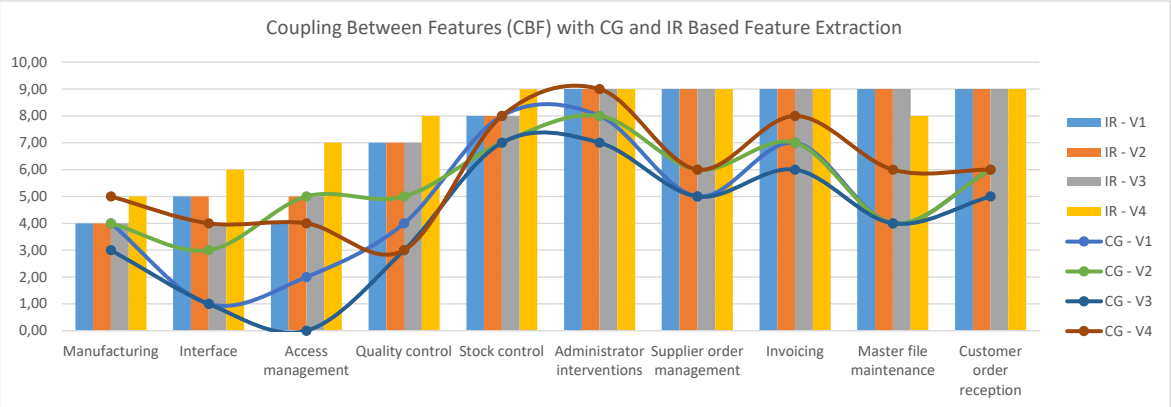

Figure 3.11: Coupling Between Features at each variant with call graph and information retrieval based feature extraction

The results of the proposed Coupling Between Features metric can be seen in Figure 3.11. As in the following figures, the columns represent the results of measurements done on IR-based extraction, while the lines represent the results of the extraction based on CG. The maximum of feature coupling is 9 since there are 10 features on the top level. Hence this is the actual maximum number of features another feature can call. As it can be seen in the case of IR-based extraction, many features achieve this with a minimal coupling of 4 overall. With CG-based extraction, on the other hand, only one

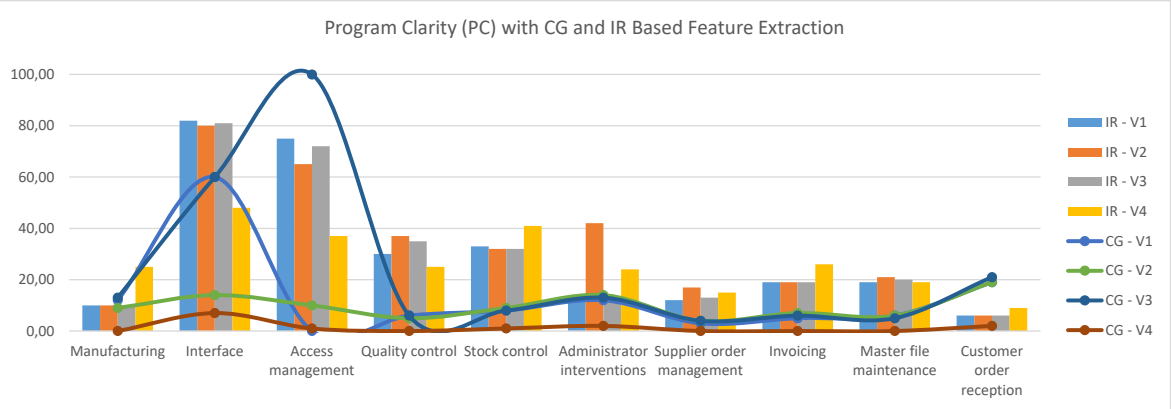

Figure 3.12: Program Clarity at each variant with call graph and information retrieval based feature extraction

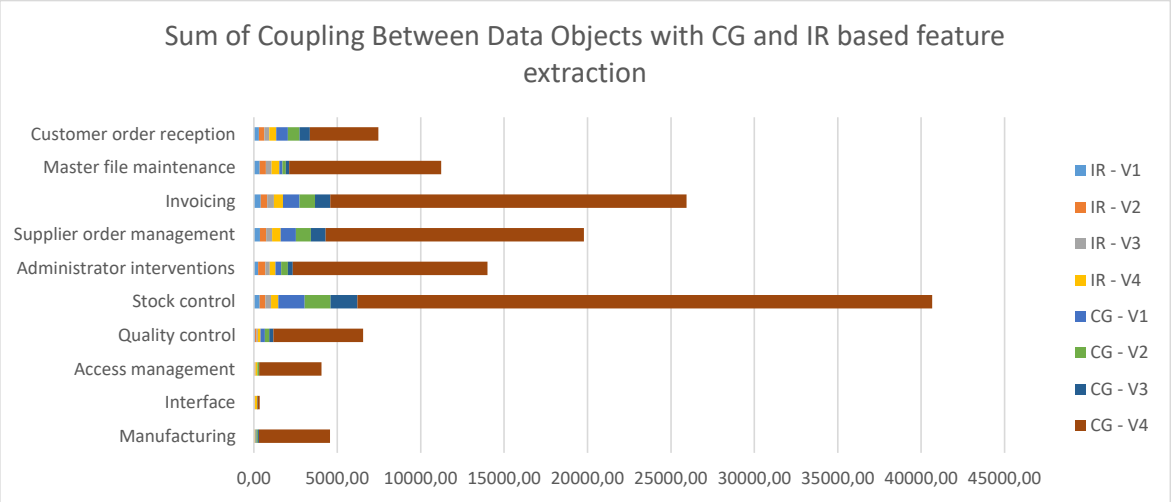

Figure 3.13: Sum of Coupling Between Data Objects with CG and IR based feature extraction

feature, Administrator interventions of V4, reaches this high level while the minimal coupling is at the Access management feature of V3, which appears to be calling no other feature at all. In the IR case, the high values are caused by the already mentioned noise as well as other factors like how general the concept of each feature is. It is apparent that in this case, the features achieving the lower coupling values are also the same that had the lowest number of programs, but this can also be a consequence of the more specific text of the feature that the IR based extraction could benefit from. Access management seems to be the most diverse feature in both cases, with different values at nearly every variant. This is probably the consequence of varying customer requests and needs about user permissions. It can also be noted that though V4 is significantly larger than the other variants, its coupling values are only slightly higher. It is also visible here that CG and IR seem to move along similar curves.

Figure 3.12 represents our results of Program Clarity. High clarity means that there are more programs of the feature that only contribute to that single feature. This metric is somewhat the opposite of CBF since it measures the self-reliance of features. This can also be seen from the results, CG-V3 achieves the highest clarity, which also had the lowest coupling, and particularly with IR, we can see that the values

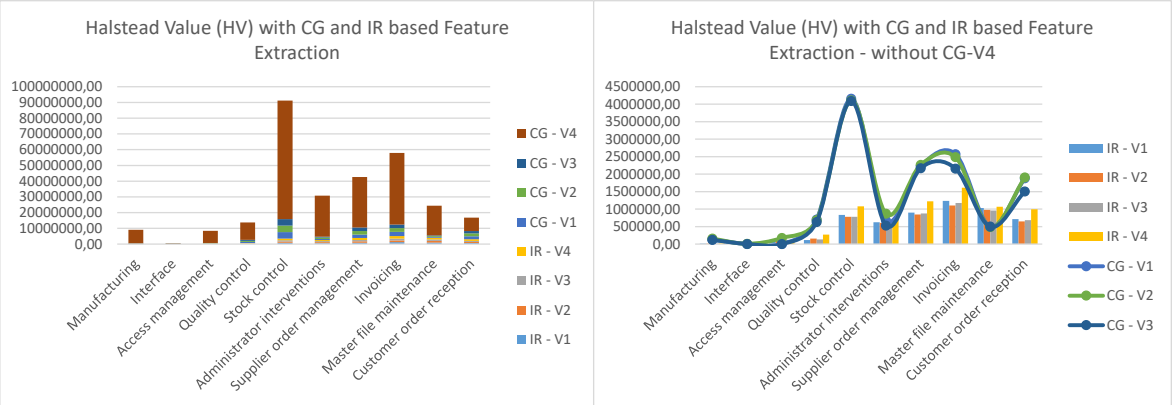

Figure 3.14: Left: Halstead Value with CG and IR based feature extraction. Right: Halstead Value with CG and IR, disproportionally large values filtered out for easier analysis

seem to be quite on the opposite side of the scale at each feature. Still, some interesting exceptions are present, like the values of Quality control in the CG case, which was at a medium level considering coupling and one of the lowest at clarity. The highest values are of the Access management feature of V3 and the Interface of V1 and V3. These features all consist of a low number of programs with only 12 and 5 programs at CG, which can contribute to high clarity, but this raises questions about the Interface feature of V2, which contains only five programs but achieves a much lower value.

The Sum of Coupling Between Data Objects results are presented in Figure 3.13. This metric is meant to measure each feature's reliance on stored data. Since it sums the values of programs, it is logical for larger systems to achieve greater values. This is the exact case that seems to happen, seeing that V4 dominates every single feature. It is interesting, although this only seems to happen in the CG case. In the case of IR, the highest values are also usually achieved by V4 but to a significantly less extent. It is apparent that in most cases, CG achieves higher values than IR. This can stem from the fact that Magic is a highly data-intensive language, and there are a lot of programs that manipulate data for a feature. Since the main goal of these programs can be data object interaction, there may be less text for information retrieval to build upon. Hence these programs are overlooked. CG, on the other hand, is aware of the calls themselves, which are made in case of data reliance and discovers these programs easily.

Considering complexity, we can see similar trends as with SCBDO, CG-V4 dominates the results in every case and seems disproportionally large on the figures. Figure 3.14 represents the results of Halstead Value, which measures the information value of features. On the left side, we can see that CG-V4 takes up most of the space, indicating that V4 is the most complex variant of these four, having a large amount of non-trivial code and can be much harder to understand in its entirety. On the right side, we filtered out CG-V4 to have a chance to get a better look at the values of the other cases. As it can be seen, CG usually produces programs with higher complexity. This can also be a consequence of IR overlooking a number of programs with complex logic or data manipulation that have less lexical information value and are much more meaningful on the data or logic side. We can also note here that while IR's value remains relatively low for every feature, IR still seems to follow CG's results, just with much lesser values.

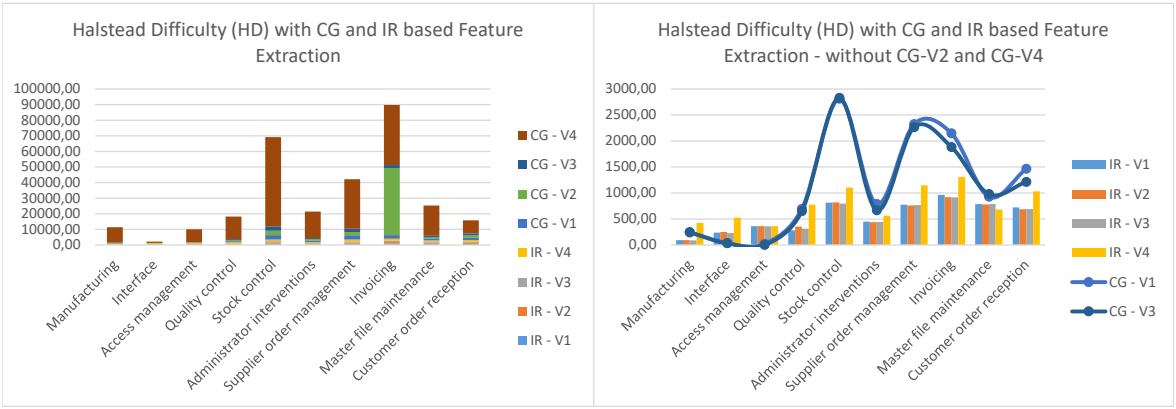

Figure 3.15: Left: Halstead Difficulty with CG and IR based feature extraction. Right: Halstead Difficulty with CG and IR, disproportionally large values filtered out for easier analysis

Figure 3.15 shows the results of Halstead Difficulty. This metric measures fault sensitivity. As we have already seen in the case of HV, the CG based extraction of V4 produces a set of programs with very high complexity. This can also be seen here, on the left side of the figure. Surprisingly, there is also one feature, Invoicing of CG-V2, that achieved the same magnitude of HD value. This is a fascinating matter since the number of programs extracted here, 267, is very close to the values of V1 and V3 and much less than V4's 602 programs, even if Invoicing is usually one of the most complex features in each case. One possible explanation for this can come from the number of tasks of the variants. As we can see from Table 3.2, V2 has a very high number of tasks, significantly higher than any other variant, while its number of programs falls somewhere in between. This has to mean that V2's programs contain more subtasks than the programs of other variants. Since we could see from our previous metrics that V2 nearly always achieves lower values than V4, we could wonder how this difference in program sizes failed to influence any of the metrics. The answer could be that a major amount of the extra tasks inside V2 contribute to the Invoicing feature providing more functions upon specific customer requests. Since HV is not exactly high in this case, this can mean that while the feature did not gain much complexity, it became much more fault sensitive. Thus this feature could be hard to maintain and deserves consideration of refactoring. On the right side of the figure, we also filtered out CG-V4 and CG-V2 for an easier glance at the rest of the variants. We can see that these results usually move along the same curves as HV.

Finally, these metrics are not only capable of revealing meaningful information about systems, features and outputs of feature extraction methods but can also be combined in several ways to attain even more understanding. Some metrics are dependent on the number of programs or tasks in a feature, which in some cases can be beneficial, but in others, it can hide some significant differences. To eliminate this, we can divide the metric by the number of programs or tasks inside a feature. For example, we could do this to any complexity metric to get the average complexity of programs or tasks of each feature or to SCBDO to get average reliance on data. These can paint a much different picture. For instance, Henry-Kafura Complexity provides similar data to HV with a difference of some IR values becoming significantly higher. On the other hand, if we divide it by the number of tasks, we get an average HKC value of tasks in each feature. This is illustrated in Figure 3.16. From this figure, it

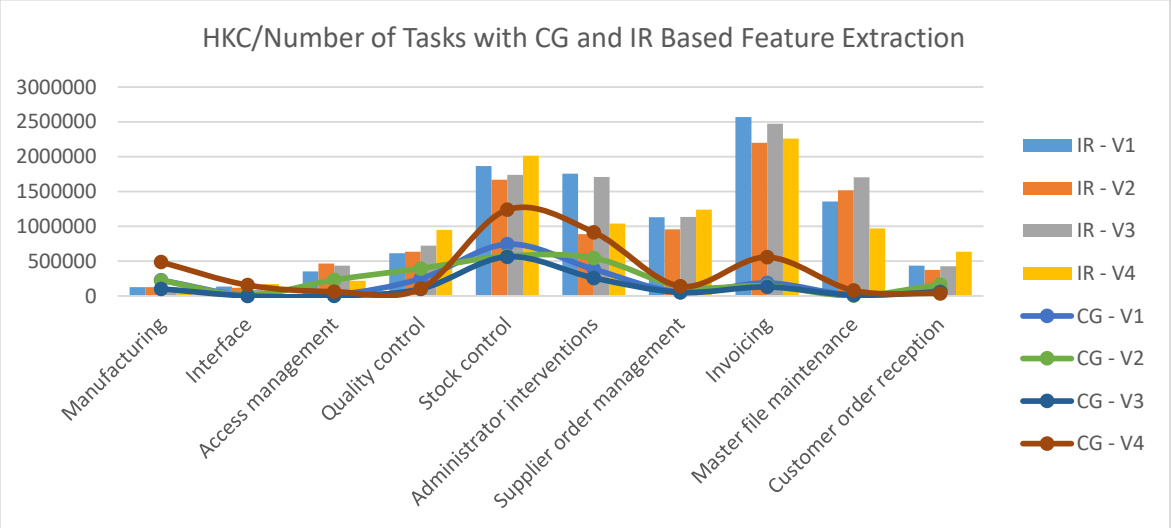

Figure 3.16: Henry-Kafura Complexity in proportion to the number of programs at each variant with CG and IR based feature extraction

is visible that IR values are usually higher, most probably because of the high number of external calls we have seen at the CBF metric. Even considering this, it is apparent that IR and CG values behave similarly, meaning that the features tend to behave the same way with both feature extraction methods on the task level.

Considering these findings, feature level metrics are likely to be suitable for the analysis of features in variants of Magic systems. Properly utilizing these, we can come to realizations that can greatly aid not just product line adoption but also ease the future maintenance of a system.

3.6 Feature Analysis using Call Graph Communities

3.6.1 Overview

Community detection provides a grouping of programs with denser inner connections. As already established, feature detection also results in a grouping of the programs, just as community detection. This means that we can compare these two sets of results in a relatively easy way. This comparison can be useful in many ways, enabling us to reach new information about the feature model and even refine the feature extraction process's output. This section presents our experiments with various community detection algorithms and investigates their possibilities in aiding product line adoption when combined with the previously constructed outputs of feature extraction.

3.6.2 Approach

Communities are groups of nodes located in a graph. They are more densely connected internally than to the rest of the graph. They are highly researched topics in network science and contribute to scientific and industrial research. Groups of densely connected nodes exist in almost every network. Since communities are not strictly defined, several different correct groupings of graph nodes can exist. Many community

detection algorithms are applied, often even specialized to a specific field, like social networks.

In our current analysis, we consider the programs of the system to be the nodes of the graph and apply the community detection to them. The edges of the graph are the calls the programs make extracted by static analysis. The previously detected features are not indicated in the graph and play no part in the community detection process. As the call edges are denser inside the communities, we can expect that these programs work together more closely and thus perform similar tasks. This is the exact property that characterizes features also. Therefore we could expect significant overlaps in communities and detected features.

There is a key value in community detection, modularity, which is a scale value between -1 and 1 that represents the density of the edges inside communities compared to the edges outside communities. Theoretically, optimizing this value results in the best possible groups of nodes. Computing this entirely is complex, thus in practice, practitioners rely on heuristic algorithms. During our experiments, we considered the following community detection algorithms provided by the R software environment:

Edge Betweenness: Edge betweenness scores are the number of shortest paths that pass through an edge. The algorithm removes the edges in decreasing order of these scores.

Fast Greedy: Each node starts as an individual community, and they are merged together in a locally optimal manner, considering the largest potential increase in modularity.

Leading Eigenvector: In each step, the graph is split into two groups in a way that maximizes modularity. This is determined by the leading eigenvector of the modularity matrix computed on the graph.

Louvain: A multi-level algorithm that consists of repetitions of greedily assigning locally optimized small communities and then considering these assigned groups as individual nodes.

Walktrap: This method starts random walks from the nodes, and because these tend to leave communities less often, it decides community merges according to them. It has a parameter t which represents the number of steps in each random walk.

To quantify the results of these community algorithms, we decided to use some metrics that can contribute to the understanding of the output. One obvious metric is the number of communities assigned. Different algorithms produced a varying number of communities on the same data. We experimented with forcing a previously defined number of communities. This approach did not do well since it often produced many tiny and a few oversized communities that separated the graph very badly and provided no real information value. We computed the following metrics for further investigation:

NoC: Number of communities.

MFC: The maximal coverage of a single feature.

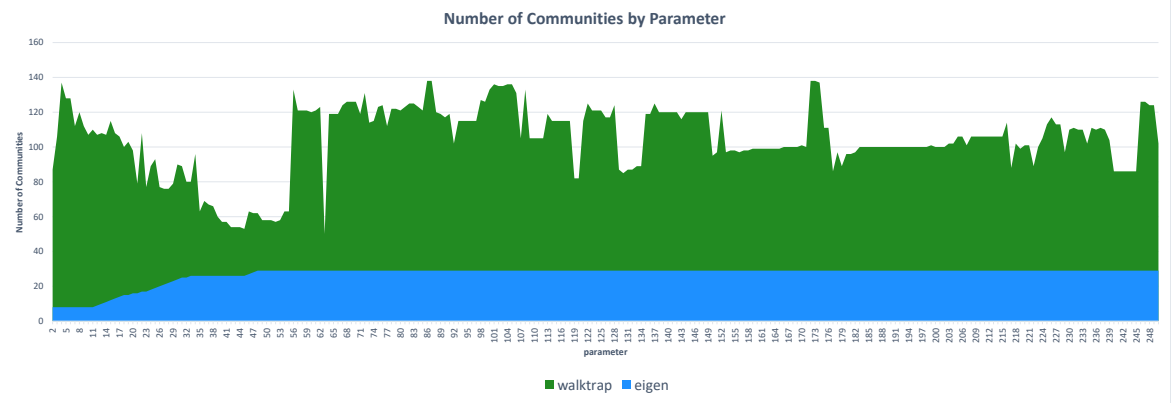

Figure 3.17: Number of communities at various parameter settings of the Walktrap and Leading Eigenvector methods

NoCSC: Number of communities with significant coverage. We consider coverage significant if the community covers at least 10% of the programs of a feature.

AFC: The average percentage of programs of features covered by communities with at least 10% coverage

3.6.3 Results

Comparison of Community Algorithms

Table 3.3: Results of our analysis with the five community algorithms for top level features

Algorithm	NoC	MFC	NoCSC	AFC
Edge Betweenness	123	31.81%	6	45.84%
Fast Greedy	44	31.81%	7	57.09%
Leading Eigenvector	29	45.35%	9	61.75%
Louvain	32	36.36%	10	52.91%
Walktrap (t=40)	57	40.91%	6	56.63%

Table 3.3 presents the results of the metric values measured on the top level of features. Each row represents the results of a different community algorithm. For Walktrap, we have chosen a *t=40* walk length, which according to the results of our experiments, provided the most suitable number of communities. The Leading Eigenvector method also works with a parameter. In this case, we have chosen *29* for similar reasons. Figure 3.17 shows the number of communities at each value of the parameters of these techniques listed from 2 to 250. A large number of communities can present too much granularity, especially for comparison with the top level of features since we have only ten features on this level. Having a greater number of features can make a large number of communities beneficial. The results of the table indicate that Edge Betweenness detects a significantly larger amount of communities than the other algorithms. These are usually very small with a few larger communities. The MFC value represents the broadest coverage of a single feature, meaning that, for example,

in the Edge Betweenness case, there was a community that covered 31.81% of a single feature, which was the largest value in this scenario. As it is visible, this number can vary greatly through different algorithms. This number, however, only describes the coverage of a single community rather than a representation of all. NoCSC, on the other hand, provides an exact number on this property, representing the number of communities that cover at least 10% of a feature. This is not a large value in any case. The lowest values were achieved by the algorithms with the highest NoC values, which is not surprising since these qualities can be somewhat opposite. Since there is more granularity, the smaller groups are bound to cover less from each feature. AFC represents the total coverage achieved by each feature on average, considering communities with at least 10% of a feature covered. This number moves around 55%, with Edge Betweenness differing significantly, most probably also due to the high granularity. This means that, on average more than half of the programs of a feature are located in communities containing a significant part of that feature. This can suggest that the main functionalities of the features are mostly achieved in communities specialized to that single feature or some larger communities which cover an important part of several features. Figure 3.18 illustrates the rate of coverage for each feature by communities that represent at least 10% of the feature's activity.

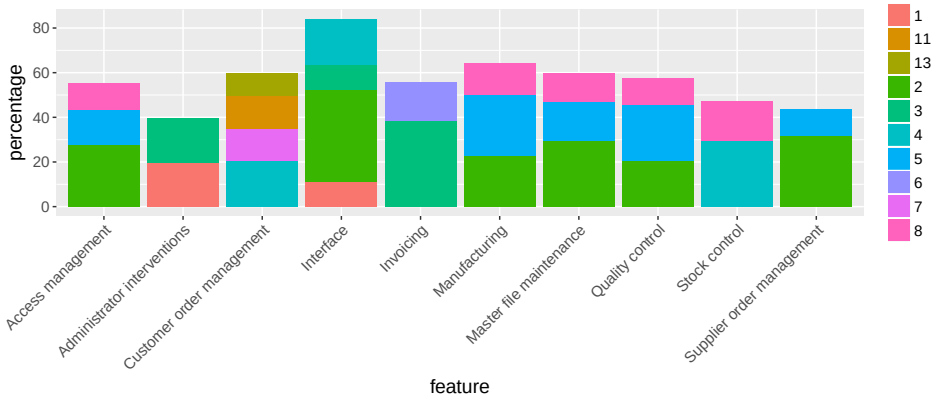

Figure 3.18: The proportions of communities determined in case of each feature at top level with Walktrap (t=40) detection and a 10% feature coverage filtering

Feature Analysis using Communities

Domain experts can play a significant part in the adoption of a new product line architecture. To their work, complete knowledge of the features is invaluable. Aiding this, community detection can highlight previously unknown aspects of the systems. As community detection is based on actual call edges of the call graph, the programs attached to each other are indeed meant to be interacting during the proper run of the system. With this knowledge, we can assume that the programs working towards the same goal should have more calls to each other, which makes communities an ideal and easy way to form groups of these programs. Possessing knowledge of groups of programs inside the system can also aid the testing process greatly. Testing the product line usually tests the functionalities the system provides. In other words, it tests the features themselves. While the features are identified with proper feature extraction, testing is still done on the code itself. Consequently, testing a single feature can involve programs that domain experts do not foresee. Community detection can provide a way

for domain experts to form realistic expectations of the system's involved elements, thus representing an additional valid and not overly complex perspective about the system.

Of our five chosen community algorithms, Walktrap was chosen because it produced a manageable number of communities that still seemed relatively unanimous. It has to be remarked that all community detection algorithms produced similar results, and we did not see any significant differences that could not be influenced by the number of communities identified. One important advantage of Walktrap can be that it can produce a variable number of communities. Changing its parameter, we can get more, which can improve the situation on lower feature levels where more features are present, which are better represented with more, smaller communities rather than a few medium-sized ones. A detailed graph illustrating top-level feature results with Walktrap communities can be seen in Figure 3.22 (non-printed version recommended) or the web version of [68]. The current section also provides some analysis on these. Each node represents a program, and the edges are the calls the programs make according to the call graph. The colors of nodes represent the features that they were assigned to in the CG-based feature extraction. Note that these graphs are generated automatically from our outputs. This display is not complete since there are programs with more than one features, and we could only color the nodes according to one of these. However, the names of all assigned features appear on the nodes, so there is no information loss. The location of the nodes represents the communities as each community forms a circle of its programs. We use these same notations in all the following community examples.

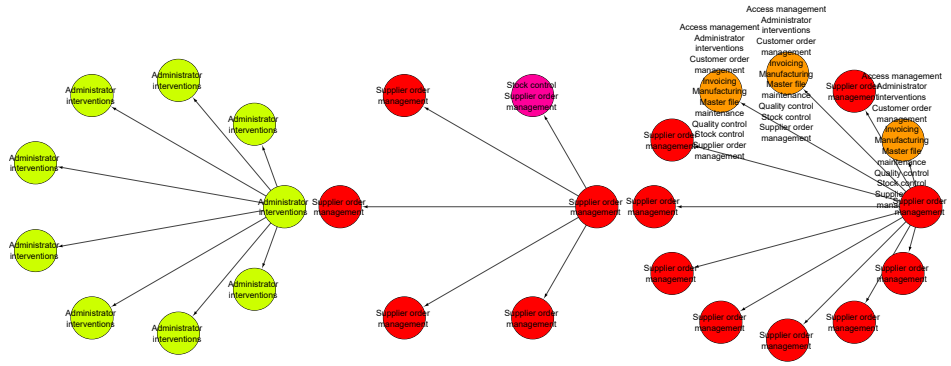

Figure 3.19: Three communities with a well distinguishable goal. Each node represents a program of the system, while its color marks its feature affiliation. See the node text for cases of more than one features assigned. The edges are the calls the programs make while the circular groups represent their assigned communities

Generally, we can say about the whole graph that various communities are present. As with every community detection method, we can observe some really large communities that usually involve programs of a lot of different features. On the other hand, we can see many smaller ones with less variance. For example, the three communities presented in Figure 3.19 are not particularly large, but each can be considered as belonging mainly to a single feature. It is apparent that the left community is part of Administrator interventions, the central community belongs to Supplier order management, as does the community on the right, but it also takes part in the work of several other features. Orange nodes tend to be more general in the whole graph, supporting a lot of features, with only four communities having programs that only

deal with Access management. This can mean that Access management is usually more interwoven with other features. The same can be said about Manufacturing to an even greater extent. Administrator interventions, on the other hand, owns a great number of communities exclusively. While small differences inside a community can be present, each community detection algorithm produces a high number of communities that are very similar to these, centered around different features. At the current level, almost every top-level feature has at least one community it is very dominant in, except two of the features, Manufacturing and Master file maintenance. Interface only seems to appear in communities with a lot of Access management programs which can suggest its heavy reliance on Access management.

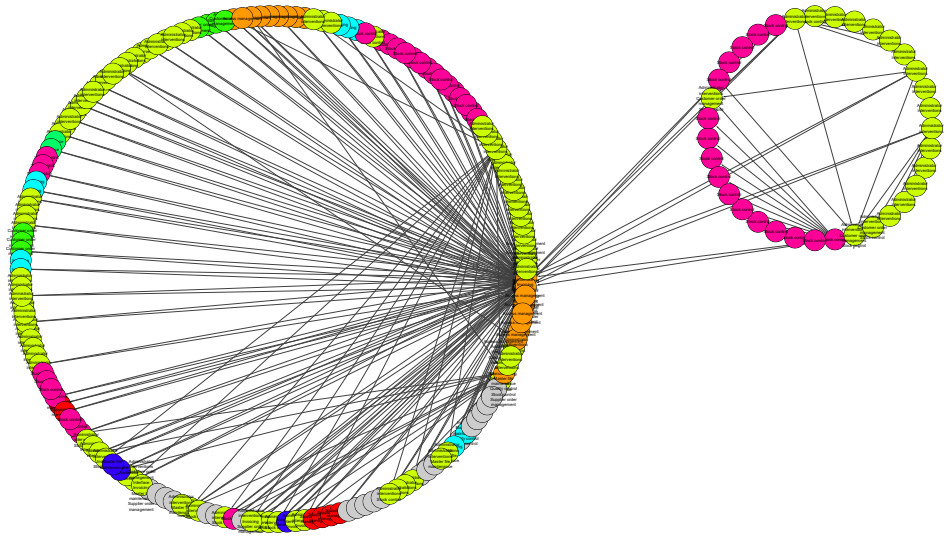

Figure 3.20: A larger, more general community involving a lot of features and a medium one with two main features

In Figure 3.20 we can see another example taken from the graph. We can see a large community that consists of programs of many features and a medium-sized one with two dominant features, making several calls to programs of the large community. The figure only contains the edges that have both endpoints in these communities, but it is already apparent that there are two programs inside the large community targeted by a vast amount of program calls. If we take a look at the full picture in Figure 3.22, we can see that there are still many more call edges to these programs from various communities. Since these programs represent many features, this suggests that they are some of the most essential programs that almost every feature relies on. They can also be the cause for the detection of such a large community since a lot of programs rely solely on them. It is also interesting that while targeted by a lot of calls, they do not seem to make any. This implies that they provide some service that is routinely needed rather than playing a vital controlling part.

We can also see some grey nodes that feature extraction did not assign any features to. Communities could also extend feature extraction outputs since if we encounter featureless programs inside a community with a highly dominant feature, we can suspect that it serves the same feature.

Applying community detection algorithms on the programs of a Magic application, we have found that the communities overlap with our feature extraction output. Apart from a few large, more general communities, we can detect several clusters of programs

that are specialized to achieve a single feature. Our investigation showed that most top-level features are represented by at least one community of their own. While sometimes different specialized communities also arise or some disappear, the same conditions can be observed with all five community detection algorithms involved in the experiment.

Analysis at deeper level of features

The results presented above only involve the top-level features domain experts have provided us. These features can be further detailed to a second feature level, and the analysis can be executed here also. This can result in a more comprehensive picture. A further advantage of community detection on a lower level is that the algorithms used do not depend on feature level, and they tend to identify more communities than the number of top-level features at our disposal. Usually, there are a lot of very small communities. These tend to perform subtasks of the same task and also belong to the same feature. Since they are small, it would be a fair assumption that they may represent lower level features working together for the same smaller goal, contributing to the top-level feature. A comparison of lower-level features and these smaller communities can highlight or disprove this. Additionally, there are larger communities that are likely to involve more of the lower level features. These can be seen as programs working more closely together. Thus, community detection can highlight valuable information at lower feature levels about both lower and top-level features. With this, we can gain more information about the system's actual inner working, which is worthy of examination. Level 2 features can also present more interesting information because the top level of features is often considered too general for actual work, while second level features draw a more detailed picture of the actual functions the features provide. Since there are only 10 features at the top level while there are 49 on the second, community detection at this level can also lead us to valuable observations.

The results of second feature level community detection are fully available in Figure 3.23. Three of its communities are presented here in Figure 3.21. The feature names in these figures were omitted because of the length of their qualified names and their large quantity. Instead of their names, we have displayed numbers representing their level 2 features. While a lot of programs still represent only one feature, it is not rare here for one program to belong to tens of features. At the second level, it is observable that specialized communities are still present in large numbers. Programs that only serve one feature tend to occur together, mostly with only a few programs that also work on some additional features. The communities presented in the example are actually the same as the ones we have seen in Figure 3.19, now with the second level of features. Apparently, no significant changes have occurred. From one perspective, this is not surprising since the call edges themselves did not change, only the feature classification of programs. This means that the communities are bound to remain the same, only consist of programs belonging to different features. The programs belong to the same top-level feature as before, now assigned to a more specific subfeature. This means that adopting a more specific level of subfeatures did not change the inner variability of these communities. The same circumstances can be observed in general as most of the specialized communities retained this quality.

On the other hand, two major differences can be seen from the top level. One of these is that programs belonging to a large number of top-level features seem to have gained even more subfeatures in this scenario. This means that these programs take part in even more features. The interesting aspect of this is that there seems to be no

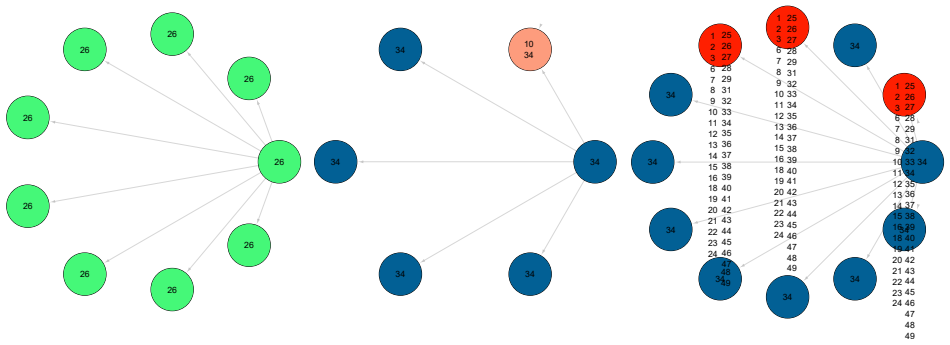

Figure 3.21: Three communities from the second level feature extraction

real middle ground. A program either supports just a selected few features or almost every second level feature. According to this, we could easily divide programs into two categories, specialised and general programs.

The other difference is that the large communities seem to have gained even more variance. They contain a significantly higher amount of different features, which means that their features consist of more subfeatures. Considering their size, this is not surprising, yet now it is apparent that while these subfeatures rely more on programs of other subfeatures, the programs of the more specialised communities tend to work mostly in separation. This can be an important distinction and can provide information of value in many cases. For example, if we are contemplating a change, we can see the size of unintended impacts a change can cause for each subfeature. Specialised communities tend to be more isolated, hence they can be more easily changed or turned off as a feature. This can be crucial both for product line adoption and for maintenance reasons.

Adopting a deeper level means that the number of features increases significantly, potentially dividing specialised communities and producing more general ones. As we have found, this is rarely the case. These communities seem to retain their good quality of being largely dominated by a single feature, even on lower levels. As the number of features rose from 10 to 49, we experienced no significant change in the rate of specialised versus more general communities. This can even verify the feature model itself since the programs belonging together according to the feature list seem to also work more closely together in reality.

3.6.4 Matching of communities with 1st level features

Communities and matching high level features are shown in Figure 3.22.

3.6.5 Matching of communities with 2nd level features

Communities and matching second level features are shown in Figure 3.23.

3.7 Insights Into the Progress of SPL Adoption

In this section, we provide information on how the project progressed through time and examine the effects of our work. The system and the feature list were constructed iteratively, hence we can compare their versions. We have multiple versions of both at

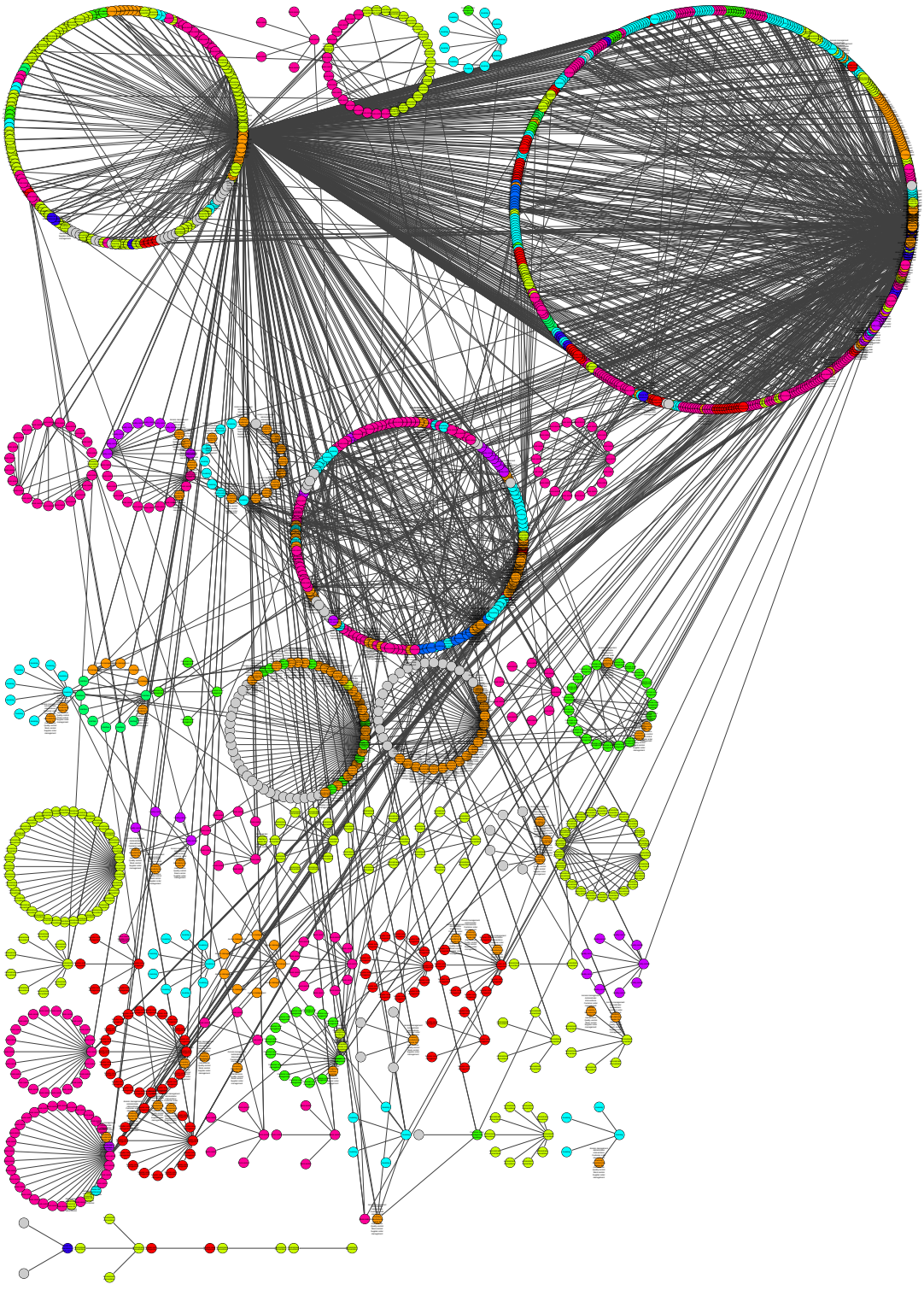

Figure 3.22: Matching communities with high-level features. Each node represents a program of the system, while its color marks its feature affiliation. See the node text for cases of more than one features assigned. The edges are the calls the programs make while the circular groups represent their assigned communities

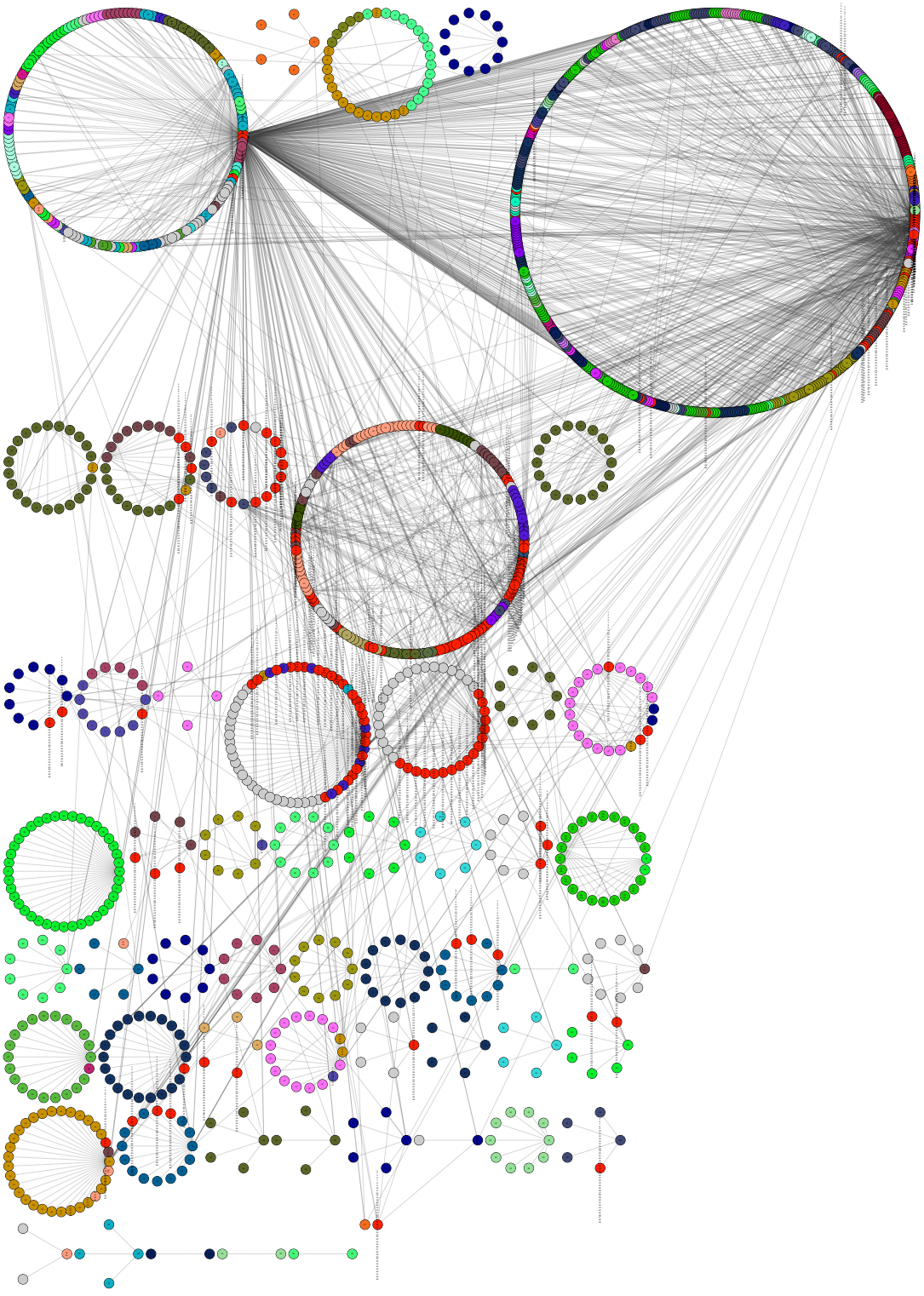

Figure 3.23: Matching communities with second level features. Each node represents a program of the system, while its color marks its feature affiliation. See the node text for cases of more than one features assigned. The edges are the calls the programs make while the circular groups represent their assigned communities

our disposal, with four separate versions of the system under construction and seven stages of the feature list. We are going to refer to the system versions now as SV1 to SV4 and the feature list versions as FV1 to FV7.

Let us look at the changes made on the feature list first. The rates of feature additions and removals can be seen in Figure 3.24 where the circles correspond to the transitions between versions. Each circle consists of four tracks that represent the changes on their specific feature level. The levels are in increasing order from the center, level 1 represents the broadest categorization of features, while level 4 is the most detailed view. We depicted the menus here as a fourth level since they were manageable accordingly through most of our work. As the list was being developed, features were added to or removed from each list level. Though there is no guarantee that a new iteration necessarily produced a better feature list, domain expertise is still the most reliable source of information in these cases. The changes presented here are the modifications the domain experts deemed necessary.

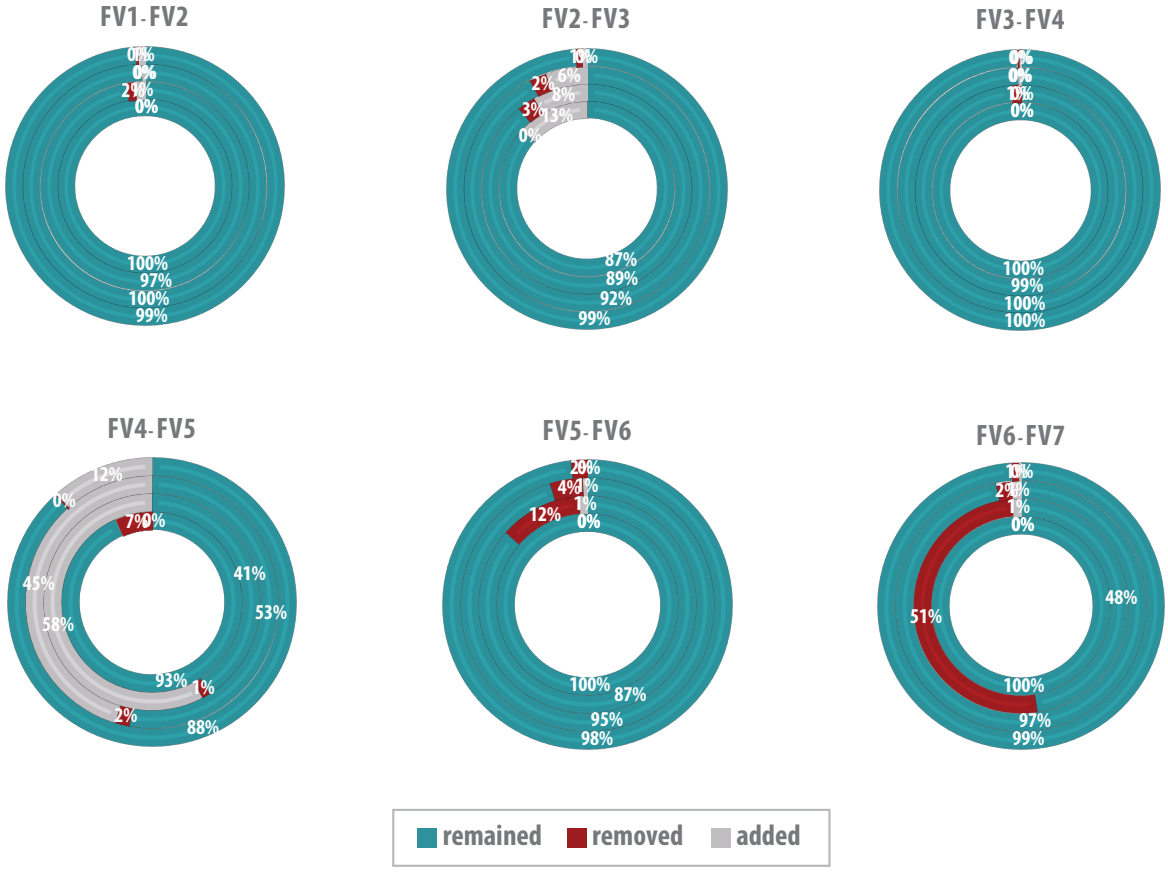

Figure 3.24: The changes made at each transition of feature list versions

Let us take a more detailed look at the figure by addressing some of the major changes. The FV2 to FV3 transition introduced a few new features, the most interesting of which are two new level 1 features. In the FV4 to FV5 transition, a great number of features were added. Level 2 got 94 new features, level 3 got 223, while 98 new menus were introduced. This represents a major update to the feature model, and the new additions make up a significant part of the new feature list. Some deletions also happened, most notably, one first-level feature has been removed. In the FV5 to FV6 transition, we can see the removal of 20 level 2, 20 level 3 features, and 15 menus

as well. The number of level 2 features was much more severely decreased in the FV6 to FV7 transition where 72 features, more than half of its total (including many of the relatively new ones), have been removed.

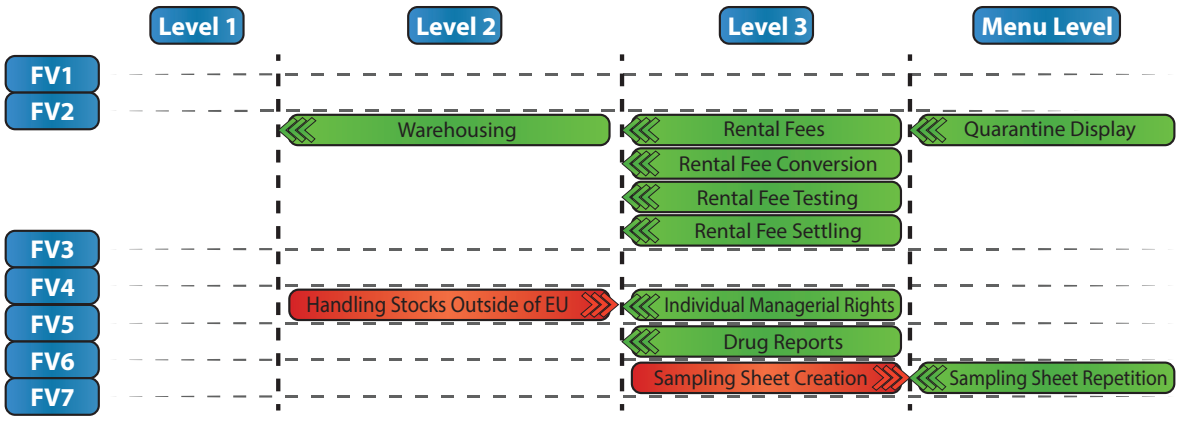

Figure 3.25: A summary of feature level transitions in the seven feature list versions, the green blocks with arrows on the left represent promotions while the red blocks with arrows on the right represent demotions.

Apart from complete additions or removals, a significant rate of these changes stems from splitting up an original feature or merging two of them. We have also experienced several more special cases where features got "promoted" or "demoted" to another feature level, including even the level of menus. We examined these changes and have detected 9 feature promotions in total, while 2 of the features got demoted. None of the features transitioned two levels or moved twice. Figure 3.25 illustrates all of these feature movements that happened during the feature list version transitions currently referenced.

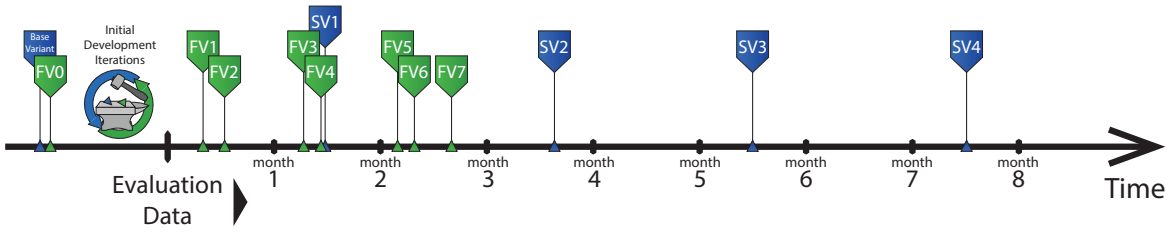

Figure 3.26: The timeline of feature list and system versions currently referenced

Let us now address the versions of the system under construction. We had access to four different versions of the system for our experiments. The versions of the feature list and the versions of the systems had different milestones. Hence these two sets of data are not completely synchronized. As the building of the system itself follows the feature list, the feature list should always be considered more up to date than the system's current state. Figure 3.26 illustrates the progress of the project during the time the feature list and system versions examined in this section were created. We can also see a variant depicted as a base variant. This is the chosen variant the product line was being built upon. There is also a feature list version with the name FV0. This is the feature list we did our experiments on and what we chiefly use in the previous and future sections. There have been several other versions of both feature lists and systems, these can be seen in the figure as the initial development iterations. This

period also introduced considerable changes, even the first level of the feature list was expanded with three new features. The timeline shows that the feature list milestones followed each other quite rapidly, while the development generally took up more time.

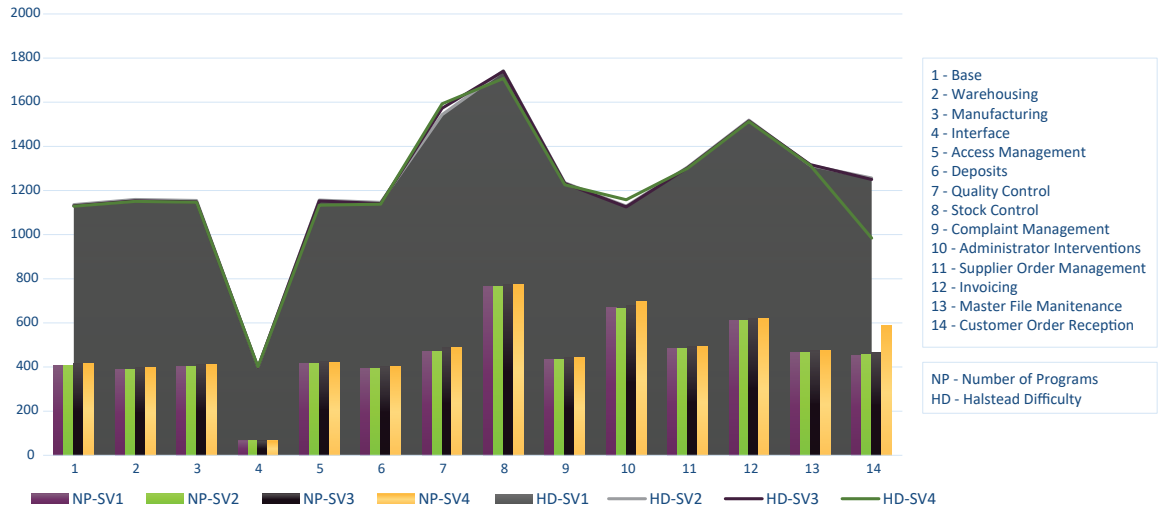

Figure 3.27: A comparison of the four versions of the system regarding their number of programs (NP) and their complexity (HD)

Considering system versions, the properties of each currently referenced milestone can be seen in Figure 3.27. The columns represent the number of programs of each feature, thus display information about the size of a feature. At first glance, we can see that no extraordinary changes were made during this time, but we can detect some small changes at each version transition of the system at every single feature. The largest differences present themselves at the latest version change, particularly in the case of the Customer order management feature and much less prominently at the Administrator interventions feature, where it experienced a significant rise in the number of programs. This number tells us a lot about which features have been modified through time, but it is not complete since the programs themselves can be subjects to modifications as well. Thus the consequences of the changes do not necessarily show up in the numbers. That is why we also displayed the complexity of these features in each version, the lines of the figure represent these. In Section 3.5, we have described our complexity measurement techniques regarding these features. The section presented the Halstead Difficulty metric for features in Magic systems, which is basically an aggregation of the difficulty metrics of all programs of a feature. This metric reflects complexity as a measurement of fault sensitivity. From the figure, we can see that considering the complexity, the most prominent change also happened at the last version transition of the system and - like we have seen at the feature sizes - also at the Customer order reception feature, where there is a major decrease in the complexity of the feature. This means that the then newly added 125 programs are generally less complex and fault sensitive than the feature's previous programs. Several other minor changes and tendencies are also visible. The size of nearly every feature seems to be in a very small growth, while the complexity seems to show a very slight decrease through time in the case of most features.

As it is visible from the timeline (Figure 3.26), the system versions follow the feature list changes much more slowly. At the time of our publications, the development was

still underway. It is nonetheless apparent that the feature list and system versions were heading in the same direction. While comparing them, we can see that in the earlier versions (see Figure 3.5), there are several features (e.g. Feature Complaint management and Deposits at the top level) that are not even represented in the call graph of the system (see Figure 3.8) because they do not even make a single call. Comparing the latest examined versions, however, we can detect much less of these only nominally existing features. This change is certainly welcome since it indicates that the system's progression indeed followed the feature list relatively well.

3.8 Evaluation

In the current section, we aim to evaluate our methods by comparing our results to ones given by human experts working for our industrial partner. We involved two developers and two domain experts working on the project and asked them to fill out our questionnaires based on their expertise. These experiments were conducted retrospectively on the feature version identified as FV0 and the base variant, both referenced in previous sections. For the evaluation, we propose the following research questions:
RQ1: *To what extent our structural and conceptual information based feature extraction methods contribute to the correct mapping of features?*
RQ2: *What additional, not inherently available feature coupling information do communities reveal?*

3.8.1 Feature Extraction Outputs

For RQ1, we have devised the following evaluation process: We have chosen 18 programs at random from each of our main feature extraction outputs and also outside all of the outputs. We shuffled these programs into a single list summing up to 72 programs. This process was performed with three different, randomly chosen features. Thus three lists were assembled with 216 programs overall. The programs were represented both by their names and their internal identifiers, so the developers could also look them up from the system itself. We asked two developers familiar with the system variants to separately judge whether each program is connected to the feature it was listed under. The developers were only provided the programs listed for each of the three features and knowledge on the possible choices they could make, with no further explanation of the size or number of different program sets involved.

The results gathered with this questionnaire are displayed in Figure 3.28. We can see the developer opinions grouped into four separate diagrams by the three chosen features and an overall sum. The first developer's impressions are displayed on the left side of each diagram, while the opinions of the second developer are presented on the right. IR depicts the output of our information extraction based approach, CG represents the output based on the call graph, while ESS depicts the programs extracted both by the IR and CG processes and are considered most essential for comprehension. The programs represented by the Unrelated set are chosen from outside of these sets.

The sets contained 18 programs each and are all disjoint from each other. Maintaining these criteria, the 18 programs were chosen randomly from each set. We have allowed -1 as an option for the case a developer could not come to a conclusive answer.

Thus there are several cases where the sum of our three differently colored columns does not come up to 18. We still have at least 15 evaluated programs for each case.

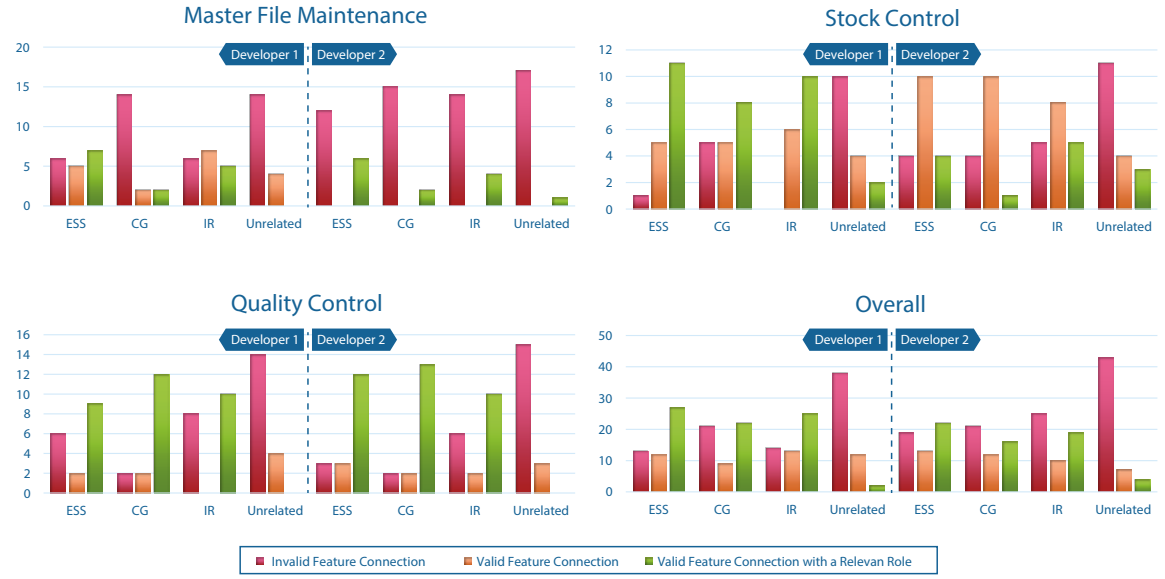

Figure 3.28: Evaluation of our feature extraction outputs according to two developers

It is visible that in most cases, the developers found our combined results the most relevant of the sets provided, the ESS set displaying both the highest green and lowest red columns in the majority of cases. It is also clearly visible that the Unrelated set containing programs not featured in our outputs scored the worst in every case. The only divergence from the success of the ESS results comes up at the Quality Control feature, where their results seem to be highly surpassed by the CG values. There are significant differences between the answers given by developers. This partly stems from their own definitions of a program being connected to a feature and being relevant. For example, it can also be seen that, in general, Developer 1 seems to consider the IR results significantly more valid and relevant than Developer 2.

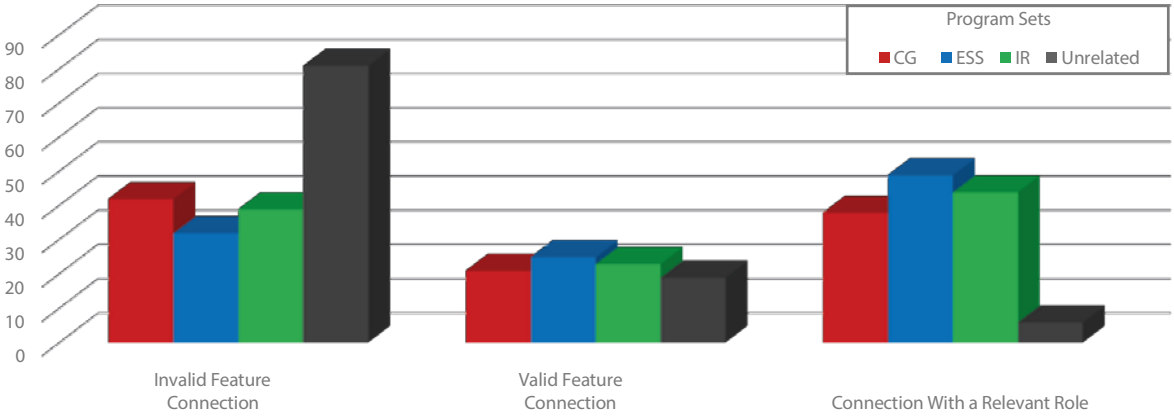

Figure 3.29: The sum of evaluation answers given by the developers

In Figure 3.29, we summarize the answers the developers provided on these program sets. We can see that the developers generally considered the combined results to be the best as this set scored the least in invalid connections and scored highest in both

valid feature connection categories. We can see a similar difference between CG and IR as well as IR usually performed a little better than CG, though as already noted, this preference was not equal among the two developers. The Unrelated set scored the worst in every case, which is not surprising but still demonstrates that our outputs hold valid information and can be used as a reference.

Answer to RQ1: Based on the evaluation results, our outputs hold valid feature extraction information in overall at least 60% of matches based on the opinion of developers. Our output combining structural and conceptual information contains the more relevant programs of a feature, with more than 70% valid matches with over 45% also being relevant overall. At least a third of these combined matches were relevant in every single case.

3.8.2 Communities

Seeking an answer to our second research question, we also performed manual evaluation. Since our community detection based output aims to contribute mainly to the work of domain experts, we asked two domain experts working on the project to provide answers regarding the mutual dependency of features. We listed the 10 high-level features introduced in Section 3.4 and asked the experts to fill out a table in accordance with their views on how likely it is that a change occurring in either of two features would lead to a need for the necessity of also changing the other feature. The possible answers were zero for unlikely, one for uncertain and two for certain.

Figure 3.30 presents the results of the questionnaire, the answers given by the two domain experts are marked red and blue. The color is darker if the domain experts have thought the dependency certain and lighter if they found it uncertain. The white fields represent the choice for considering it unlikely.

To make the community data measurable, we decided to quantify to what extent two features belong to the same communities. We computed their relative weight inside each community compared to other features, and we normalized this with the size of the community. The sum of these results gives our community dependency value which can be divided into multiple categories. This is explained in the equation below where C is the set of communities, F is the feature set, w_1 is the first feature's weight, and w_2 is the second feature's weight. This resulted in a number between 0 and 3 in each case.

$$CommunityDependency = \sum_{c \in C} \left(\frac{1}{size(c)} \frac{w_1^2 + w_2^2}{\sum\limits_{f \in F} w_i} \right)$$

The Community Connections part of Figure 3.30 shows how this number of each feature pair compares to the average of the opinions of the two domain experts. The black fields note the cases where the community-based result showed a lower connection probability level while matching connectivity levels are represented by green fields. The yellow and orange fields represent the cases where the communities suspect a higher chance of connection. Orange is a more significant difference.

While the information possessed by domain experts is invaluable in the adoption process, there is a difference in their opinions, which is significant in some cases like the connections of Stock Control and Invoicing or Stock Control and Supplier order management. Since the domain experts have no ready knowledge of every single program, these differences are inevitable. While this assessment only involved the highest

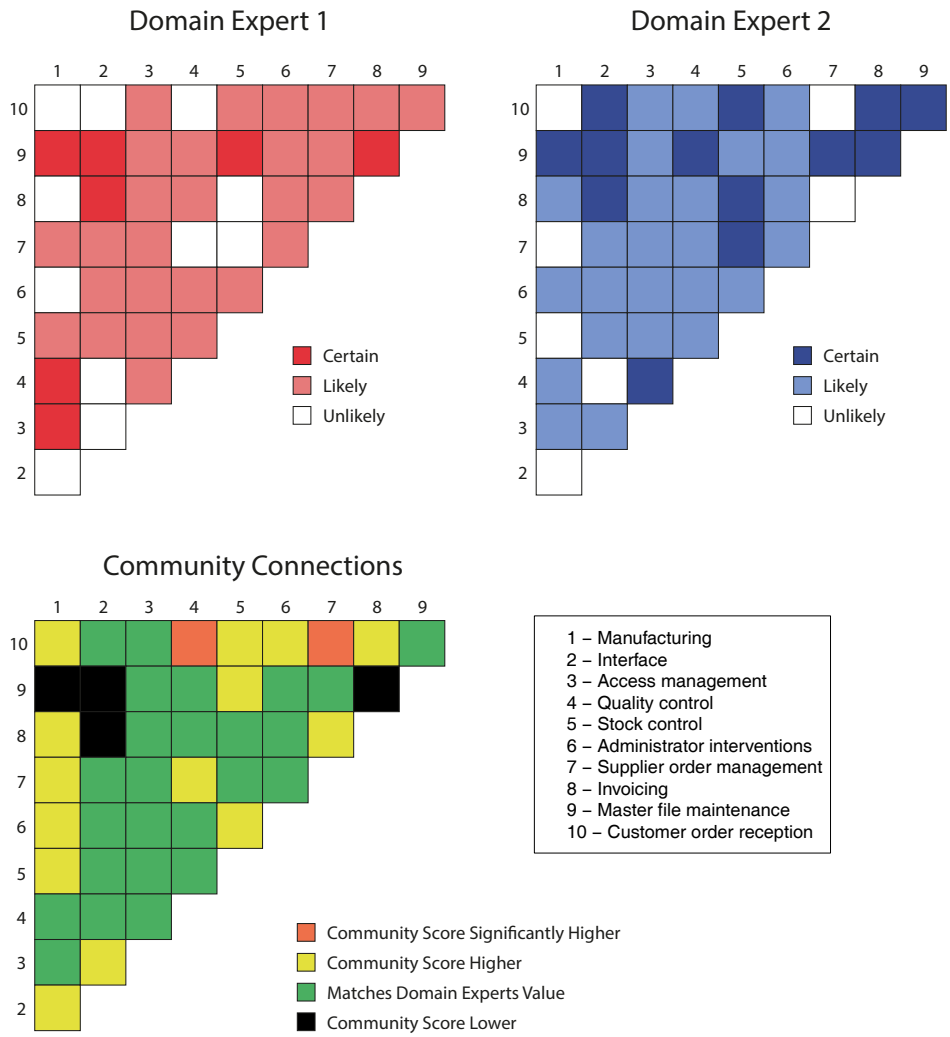

Figure 3.30: The answers given by domain experts and a comparison of their average with community-based assessment

level of features, which are very general, we can imagine how much harder this task could be on a level with tens or even hundreds of features.

Furthermore, it is visible that the green fields are in the majority, which can mean that the community detection output is similar to the average of domain expert views, thus holds real and potentially useful information about the dependencies between features. We note that the average of the votes of domain experts in the figure is based on the rounded down value of the average. Rounding up, we would get more black fields but also even more green fields. This indicates that if domain experts take a more cautious approach, the data of the communities can match even better with their views. Our results also fall closer to the answers of the domain experts individually than they do to each other, the absolute difference between them being 24 while the community dependency shows 20 and 22 absolute difference from them.

Although we can see that the communities can contain similar information to what the domain experts use, they also build on structural information as they are working on the call graph of the system. This means that the graph contains additional information that highlights a more structural level, and while calls inside the system do not occur necessarily in every case, it is taking them into account, even for a quick inspection.

Answer to RQ2: According to the evaluation, the call graph communities represent real feature coupling information that approximates domain expert knowledge well while relying only on structural information, thus can contribute to the decisions of a domain expert by providing another informed viewpoint.

3.9 Discussion

In this section, we overview the possible uses of our methods presented. Figure 3.31 highlights how various feature extraction techniques can be used to help in building the new product line architecture.

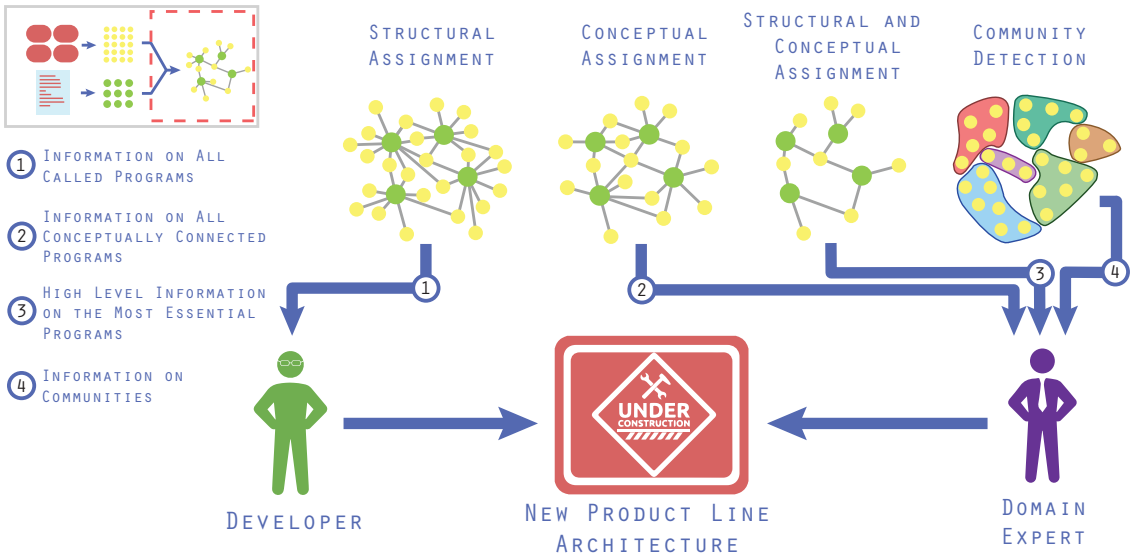

Figure 3.31: The possible ways of usage of the results of various feature extraction techniques in helping product line adoption

- Structural Extraction - Provides a detailed, widespread analysis. It is good for developers since they are required to have knowledge of all of the programs called by a feature.
- Conceptual Extraction - For domain experts, on the other hand, all called programs can be too much. This approach introduces conceptual dependencies but may contain too much noise for smooth work.
- Combination (ESS) - Grasps the essence of features, more fit for domain experts. While constructing the new architecture, the domain experts need to judge properly which parts of the variants should be adopted. In this decision-making process, the results of this combined extraction highly decrease complexity, and it can also facilitate test planning in the future.
- Community detection - Identifies program communities over the call graph that work closely together while having less outer relations.
- Community matching - Matches program communities with features in the code. It can be used to find differences between the view of the domain experts and the actual implementation of features.

As our evaluation pointed out, the developers found the programs of our result sets as a source of feature mapping information much more preferable to the other, unrelated programs.

Besides these, we mention some other possible ways to use the results. Firstly, connections are not necessarily observable through the calls of the system. Programs can, for instance, connect by accessing the same data objects. This means that not every connection will present itself on the call graph. These, however, can be found via conceptual feature extraction since it is likely that programs using the same data are conceptually connected to the same feature. This is why the programs detected by the conceptual extraction and not discovered via structural information can still be valuable. This seems to be verified by the developers also, since, in our evaluation, we found that information retrieval based results scored even higher in their regard than the call graph based results. However noisy, conceptual data still represents a ready source of the semantically connected programs to each feature. Hence it can also be a useful information source.

Additionally, both the structural and information retrieval based methods can be tailored according to our intent by filtering out the more general programs of the call graph, which provides the possibility to form even better-separated program sets and by changing semantic similarity thresholds in the conceptual method.

As our evaluation pointed out, communities hold valid information about features, and similar opinions can be extracted from it as relying on domain expertise while working on a more structural level. Although community detection and matching can aid the understanding of features and the current state of the system, they can also be useful in the future at a maintenance stage. Additional possible uses even involve the refinement of the feature model and easily comprehensible tracking of the evolution of programs. Some appropriate setting of communities could also be optimized for a recommendation system of possible splitting or merging of features.

We made our results available to our industrial partner at the time of the construction of the new SPL architecture. The work is finished, and the project was concluded successfully. The results of our experiments were utilized in the process.

Our project required us to remain within the field of 4GL languages, but some of our methods could also lead to viable approaches in more traditional languages like Java or C++. Since call graphs can be constructed for these languages also and tend to contain a lot of natural language text, our method can be adapted to work in more traditional conditions. Magic and 4GL languages, in general, are more data-intensive, and in our specific advantageous case, features were readily connected with menu elements making direct calls inside the software. However, a similar menu to class position can also be rather easily achieved in a lot of Java or C++ applications due to the confinements of many frameworks.

3.10 Conclusions

The thesis point presented an industrial project, which dealt with a software product line adoption process. We concentrated on the feature extraction and analysis aspects of the project, which are fundamental parts of the effort because further architecture redesign and implementation are and will be based on this information. For feature extraction, we used two approaches: one based on computing structural information in the form of call-graphs, and the other extracted from the textual representation of

high-level feature models and the code. The combination of the two pieces of information had to be performed and processed in such a way that the resulting models are most useful for project participants. Experimental results show that the final models are significantly more comprehensible and hence directly usable (in various forms) by domain experts, architects and developers. Our work also introduced several new metrics for the feature level of product line adoption of 4GL systems that provide new insights on the size, similarity, coupling and complexity of features. The high-level view of domain experts is not necessarily in line with the actual implementation. We introduced community detection methods on the call structure of the system to match communities with feature code originated from domain experts. We evaluated our results based on the analysis of the system variants and a comparison with human expertise and shown that our various outputs present knowledge that matches well with the thinking of experts but can still highlight additional insights. The proposed method, including the associated toolset, was used by our industrial partner during this effort, which resulted in a successful project. Although the approach was implemented in Magic, a 4GL technology, we believe that the fundamental method could be suitable for other more traditional paradigms as well after the necessary adaptations.

The thesis point provided new solutions for 4GL feature extraction in product line adoption, including a combination of semantic and structural information, several adopted or new metrics, and a community-based mapping of features. The methods were found to be valid by domain experts and were used during the product line adoption of a real logistical system.

The author considers the followings as his main contributions:

- The author implemented the LSI-based solution for mapping features according to semantic information.
- The author took part in planning the combination possibilities of the CG and IR techniques.
- The author devised some of the new metrics, and he took part in the evaluation of the systems based on the new metrics.
- The author took part in the design and evaluation of the community-detection experiments.
- The author maintained contact with the industrial partner during the work, and took significant part in the project's documentation.
- The author coordinated several steps in the implementation, including the combination, filtering, evaluation, calculation of new metrics, and community-detection steps of the work.
- The author took part in the evaluation work, the planning of the evaluation process, and the analysis and explanation of the results.

Part II

Textual Methods in Aiding Test-to-Code Traceability

"'Would you tell me, please, which way I ought to go from here?'
'That depends a good deal on where you want to get to,' said the Cat.
'I don't much care where–' said Alice.
'Then it doesn't matter which way you go,' said the Cat."

— Lewis Carroll, Alice in Wonderland

4 Test-to-Code Traceability

4.1 Overview

The creation of quality software usually involves a great effort on part of developers and quality assurance specialists. The detection of various faults is usually achieved via rigorous testing. In a larger system, even the maintenance of tests can be a rather resource-intensive endeavor. It is not exceptional for software systems to contain tens of thousands of test cases each serving a different purpose. While their aims can be self-evident for their authors at the time of their creation, they bear no formal indicator of what they are meant to test. This can encumber the maintenance process. The problem of locating the parts of code a test was meant to assess is known as test-to-code traceability.

Proper Test-to-Code traceability would facilitate the process of software maintenance. Knowing what a test is supposed to test is obviously crucial. For each failed test case, the code has to be modified in some way, or there is little point to testing. As this has to include the identification of the production code under test, finding correct test-to-code traceability links is an everyday task, automatization would be beneficial. This could also open new doors for fault localization[80], which is already an extensive field of research, and even for automatic program repair [159], greatly contributing to automatic fixes of the faults in the production code.

To the best of our knowledge, there is no perfect solution for recovering the correct traceability links for every single scenario. Good testing practice suggests that certain naming conventions should be upheld during the testing, and one test case should strictly assess only one element of the code. These guidelines, however, are not always followed, and even systems that normally strive to uphold them contain certain exceptions. Thus, the reliability of recovery methods that build on these habits can differ in each case. Nonetheless, the method of considering naming conventions is one of the easiest and most precise ways to gather the correct links.

In its simplest form, maintaining naming conventions means that the name of the test case should mirror the name of the production code element it was meant to test, its name consisting of the name of the class or method under test and the word "test"

for instance. The test should also share the package hierarchy of its target. In a 2009 work of Rompaey and Demeyer [132] the authors found that naming conventions applied during the development can lead to the detection of traceability links with complete precision. These, however, are rather hard to enforce and depend mainly on developer habits. Additionally, method-level conventions have various other complicating circumstances.

Other possible recovery techniques rely on structural or semantic information in the code that is not as highly dependent on individual working practice. One such technique is based on information retrieval (IR). This approach relies mainly on textual information extracted from the source code of the system. Based on the source code, other, not strictly textual information can also be obtained, such as Abstract Syntax Trees (AST) or other structural descriptors. Although source code syntax is rather formal and most of the keywords of the languages are given, the code still usually contains a large amount of unregulated natural text, such as variable names and comments. There are endless possibilities in the naming of variables, functions, and classes. These names are usually quite meaningful. While source code is hard to interpret for humans as natural language text, machine learning (ML) methods commonly used in natural language processing (NLP) could still function properly.

Compared to a small manual dataset, Rompaey and Demeyer [132] found that lexical analysis (Latent Semantic Indexing - LSI) applied to this task performed with 3.7%-13% precision while the other methods all achieved better results. Thus, it is known that IR-based methods most probably do not produce the best results in the test-to-code traceability field by themselves. However, they are still in constant use in current state-of-the-art solutions. Using textual methods may not be the single best way to produce valid traceability links, but modern approaches still employ them in combination with other techniques. The textual methods used in these systems are usually less current, most solutions simply rely on matching class names or the latent semantic indexing (LSI) technique as part of their contextual coupling. Thus, finding better performing textual methods can improve these possible combinations as well, having the potential of major contributions to the field. Our findings [30] show that improved versions of lexical analysis can significantly outshine the previously mentioned underwhelming results, raising their average precision to over 50%.

To investigate the benefits of ML models in this topic and to point out their distinction from simple naming conventions, our experiments were organized along the following research questions:

- **RQ1:** How generally are naming conventions applied in real systems?
- **RQ2:** Is there a way to further improve test-to-code traceability results relying on modern information retrieval methods?
- **RQ3:** How well do various text-based techniques perform compared to human data?

Our goal was to recover test-to-code traceability links for tests based on only the source files, currently focusing on Java systems and JUnit tests. To do so, a suitable input representation has to be generated. From this, an artificial intelligence model is to be trained for the search for the most similar test-to-code match. The chapter is organized as follows. Related work is overviewed in Section 4.2. Section 4.3 presents

diverse background information, including our various approaches to input generation and traceability link recovery. Our evaluation procedure and the sample projects are also described in this section. An evaluation on eight systems follows in Section 4.4 with the discussion of these results in Section 4.5. Some threats to validity are addressed in Section 4.6, while the chapter concludes in Section 4.7.

4.2 Related Work

Traceability in software engineering research typically refers to the discovery of traceability links from requirements or related natural text documentation towards the source code [9, 100]. Based on the study of Borg et al. [19], most of the traceability evaluations have been conducted on small bipartite datasets containing fewer than 500 artifacts, which is too few to real external validity. While data limitations still persist, the current chapter's evaluation is conducted on eight software systems, using different oracles. While test-to-code traceability is not the most widespread topic amongst recovery tasks, several well-known approaches aim to cope with this problem. Still, as yet, none of them has provided a perfect solution for the problem [61, 132, 75, 126, 31, 30]. The current state-of-the-art techniques [125] rely on a combination of diverse methods - i.e. techniques based on dynamic slicing and contextual coupling. The use of textual information is common in these techniques. Our currently discussed work took a closer look at various textual similarity techniques, and combinations of these resulted in promising recovery precision.

In a recent work [157], authors presented TCtracer, a tool which combines an ensemble of new and existing techniques and exploits a synergistic flow of information between the method and class levels. The tool observes test executions and creates candidate links between these artifacts and the ones under test. It then assigns scores (which are used to rank the candidates) to the candidate links. These scores are calculated using the combination of eight test-to-code traceability techniques including four string-based techniques, two statistical, call-based techniques, Last Call Before Assert (LCBA) which relies on the last method call before the assertion statement, and Naming Conventions (NC). Although this and our work share many common factors, there are significant differences. First of all, our technique does not rely on information based on test-execution. Secondly, the two rankings are fundamentally different: our work relies on IR techniques (and refine these using various approaches, with an initial static analysis), while White et al. calculate the ranking scores based on formulas defined in the paper. We also researched different ways of representing the source code.

The utilization of structural information has also occurred in other works [118, 127, 125]. In their 2015 work, Ghafari et al. [44] also employed structural information. Here, the main goal was to identify traceability links between test cases and methods under test, which is still not a mainstream topic in the field, as most methods aim for production classes. The proposed approach correctly detects focal methods under test automatically in 85% of the cases. Bouillon et al. leveraged failed test cases to find the location of errors in source code [20]. To link the tests to the production code, they built the static call graph of each test method and annotated each test with a list of methods which may be invoked from the test. The use of structural information also occurs in other extraction methods, feature extraction for instance, where it was shown that its combination with LSI is capable of producing good results [39]. In our work,

structural information is used in several source code representations. Call information was also utilized, even though it was extracted only from the text. Even so, it was found a valuable addition as a filter.

Like LSI, TF-IDF is also a text-based model commonly used in the software engineering domain. This technique was, for instance, used by Yalda et al. [162] to trace textual requirement elements to related textual defect reports, and by Hayes et al. [51] in the after-the-fact tracing problem. In requirement traceability, the use of TF-IDF is so widespread, that it is considered a baseline method [143]. Text-based models are still very popular in the requirement traceability task also, they are incorporated in several recent publications [42, 52]. Our experiments covered LSI and TF-IDF as standalone techniques and also as a refinement for Doc2Vec, which was shown in our previous work [30] to produce higher quality results.

In our findings, the use of document embeddings resulted in the highest precision values. Word2Vec [105] gained a lot of attention in recent years and became a very popular approach in natural language processing. Calculating similarity between text elements using word embeddings became a mainstream process [102, 148, 156, 46, 163, 113]. Doc2Vec [104] is an extension of the Word2Vec method dealing with whole documents rather than single words. Although not enjoying the immense popularity of Word2Vec, its use is still prominent in the scientific community [173, 32, 154, 35].

The use of recommendation systems is widespread in the field of software engineering [80, 128, 129]. Presenting a prioritized list of most likely solutions seems to be a more resilient approach even in traceability research [75, 31].

Because of the numerous benefits of tests, developers tend to create a lot of them even though it is challenging to determine what new tests have to be added to improve the quality of a test suite. Since 100% coverage is often infeasible, several new approaches have been proposed for interpreting coverage information. For instance, Huo et al. [55] introduced the concepts of direct coverage and indirect coverage, that address these limitations. In addition, several other challenges are present in general software testing [15], like coherent testing, test oracles and compositional testing. The more challenges are solved, and the more the community understands about testing in general, the better test-to-code traceability results can become [119]. Our research also aimed to shed some light on class and method naming habits which can lead to a better understanding of testing in real-life software systems.

Although natural language based methods are not the most effective standalone techniques, state-of-the-art test-to-code traceability methods like the method provided by Qusef et al. [125, 127] incorporate textual analysis for more precise recovery. Jin et al. in [45] presented a solution that uses deep learning and word embeddings to incorporate requirements artifact semantics and domain knowledge into the tracing solution. The authors evaluated their approach against LSI and VSM (Vector Space Model). They found that their neural network approach only outperforms these when the tracing network has a large enough training dataset which is hard to obtain. Other works also explore the use of word embeddings to recover traceability links [45, 46, 172, 165]. Our current approach differs from these in many aspects. To begin with, we make use of different similarity concepts and further refine these with structural information. Next, our document embeddings are computed in one step, while in other approaches this is usually achieved in several steps. Finally, our models were trained only on source code (or on some representation which was obtained from the source code), and there was no additional natural language corpus.

4.3 The Proposed Method

The goal of the approach is to investigate textual techniques for the sake of improving test-to-code traceability. This approach could improve the performance of existing techniques on this specific problem and also serve as the groundwork for future works on test-to-code traceability. To achieve this, let us grasp the process in Figure 4.1. The input is a software system, which consists of Java source files. The output is a ranked list for each test case with the production classes that are likely to be a target of the test. The input files are transformed in such a representation, that is more suited to machine learning than raw source code. Three techniques are trained to measure the similarity between test and code classes. Class information is also obtained from the source files like the list of imported packages and the methods defined in a class. Each model produces a list of similar code classes. These results are susceptible to providing faulty results because of the probabilistic nature of ML techniques. Thus, these lists are filtered with the class information which was obtained earlier.

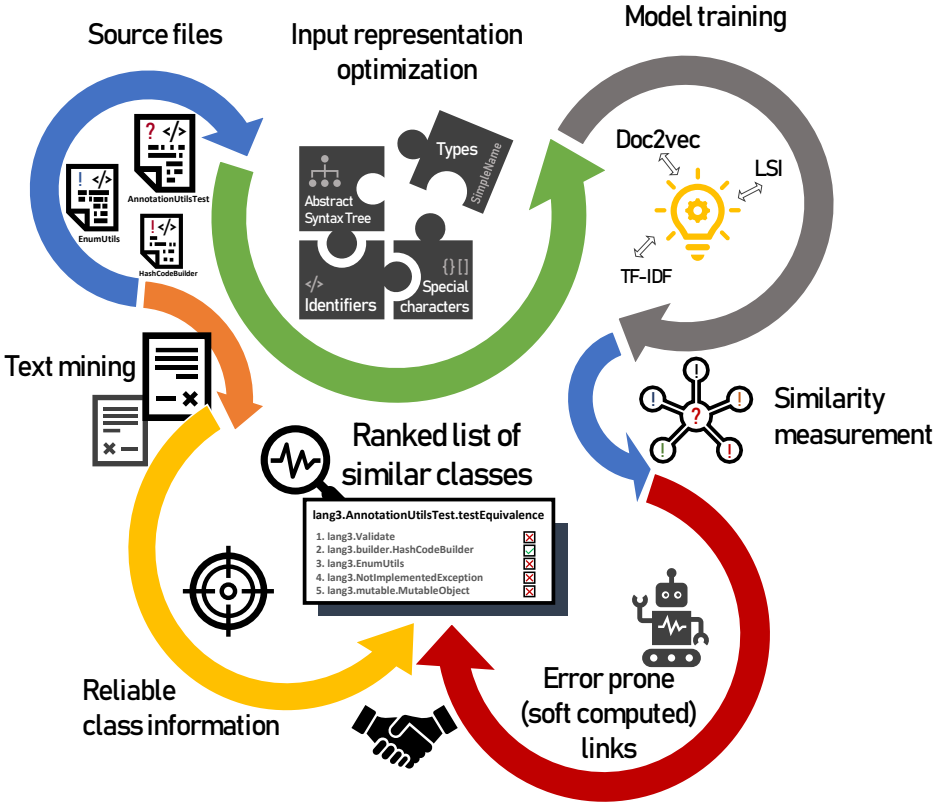

Figure 4.1: A high-level overview of the proposed process

Our research strived to achieve a comprehensive evaluation of three text-based techniques on the test-to-code traceability problem rather than simply providing a new method. Thus, our results were evaluated on eight real programs and also using a variety of source code representations and settings. Our work aimed to show that LSI itself performs better than it had been previously perceived by the research community [75], investigated the question of source code representations in the task, and also found that Doc2Vec can significantly outperform LSI [31] while a suitable combination of the textual similarity techniques could provide even better results [30].

Even though our experiments involved systems written in the Java programming language, the applied IR-based techniques mainly use the natural language part of the

code, making the approach semi-independent from the chosen programming language. Nevertheless, language invariability cannot be guaranteed. These methods also depend on the habits of the developers. The naming conventions and the descriptiveness of the language of the natural text factors in a great deal in textual similarity. This is why it is also crucial that the developers possess sufficient education and experience to produce sufficiently clean source code. Furthermore, as it is visible with our current systems under test, systems with similar properties can produce vastly different results with the same methods.

Our experiments feature the extraction of program code from the systems under test using static analysis, obtaining different input representations, distinguishing tests from production code, textual preprocessing, and determining the conceptual connections between tests and production code. During our experiments, the Gensim [1] toolkit's implementation was used for all three textual methods. The initial static analysis that provides the text of each method and class of a system in a structured manner is achieved with the Source Meter [141] static source code analysis tool.

The proposed approach recommends classes for test cases starting from the most similar and also examine the top 2 and top 5 most similar classes. Looking at the outputs in such a way makes it a recommendation system, which provides the most similar parts of production code for each test case. Examining not only the most similar class but the top_N most similar ones has the benefit of highlighting the test and code relationship more thoroughly.

The proposed approach was evaluated on eight medium-sized open source projects written in Java, a further overview of these systems is available in Subsection 4.3.9. In this work, the models are not trained on plain source code, the feasible input representations are introduced in the next section. Our most comprehensive and latest findings on text-based methods in test-to-code traceability can be found in [66]. The techniques used through our work follow in brief summary. Some of the the discussed methods like LSI and Doc2Vec have already been defined in Chapter 2, but they are also briefly included here for completeness's sake.

4.3.1 Latent Semantic Indexing

LSI is a relatively old algorithm and there are also some previous findings on its uses on this specific problem. It builds a corpus from a set of documents and computes the conceptual similarity of these documents with each query presented to it. In our current experiments, the production code classes of a system were considered as the documents forming the corpus, while the test cases were considered as queries. The algorithm uses singular value decomposition to achieve lower dimension matrices which can approximate the conceptual similarity.

4.3.2 Doc2Vec

Doc2Vec is originated from Word2Vec [104], which is an artificial neural network that can transform (*embed*) words into vector space (*embedding*). The main idea is that the hidden layer of the network has fewer neurons than the input- and output layers, thus forcing the model to learn a compact representation. The novelty of Doc2Vec is that it can encode documents, not just words, into vectors.

4.3.3 Term Frequency-Inverse Document Frequency

TF-IDF is a basic technique in information retrieval. It relies on numerical statistics reflecting how important a word is for a document in a corpus. The frequency value is a metric that increases each time a word appears in the document but is offset by the frequency of the word in the whole corpus, highlighting the most specific words for each document.

4.3.4 Result Refinement with $ensemble_N$ Learning

In our early works [31, 30, 75] our experimental analysis led us to the conclusion that different ML techniques capture different similarity concepts. This means, that each examined technique can provide useful information, while generally, the desired code class appears close to the top of every similarity list. Thus, it should be possible to refine the obtained results a technique provides with another list that comes from a separate technique. The algorithm is very simple: only those code classes remain which are present in both similarity lists. Since every code class is ranked in the lists, we limit the search to the top N most similar ones, this way the algorithm will drop out the classes from the first list which are not amongst the top_N links of the second.

4.3.5 Soft Computed Call Information

Since the listed techniques do not take class information into account, an additional simple filter can also be added. The following assumptions should be true in most cases: (1) the package of the class under test should either be the same as the test's or it should be imported in the test and (2) a valid target class should have a definition for at least one method name that is called inside the body of the test case. These criteria still do not guarantee a valid match.

Methods and imports are obtained from the Java files using regular expressions. These may differ for different programming languages along their different syntax.

4.3.6 Extended Naming Convention Extraction

The above presented techniques all result in a filtered list of soft computed links - i.e. there is no guarantee, that those are correct. Naming conventions, however, are known to produce traceability links with very high precision [132]. If a project lacks these good naming practices, naming conventions simply cannot be used in finding the correct matches. In this final approach, the naming convention is observed first. If it is applicable, it is accepted. Otherwise, the results of an IR-based approach (LSI, Doc2Vec, etc.) are considered.

4.3.7 Optimal Input Representation

It is evident that the exact contents of the input are of crucial importance. In this subsection, we briefly overview the representations of code snippets (classes or methods) used for the traceability experiments. A code representation is the input of a machine learning algorithm that computes the similarity between distinct items. Abstract Syntax Trees (AST) were utilized to form a sequence of tokens from the structured source code. An AST is a tree that represents the syntactic structure of the source code,

```
boolean contains(Object target) {
    for (Object elem: this.elements) {
        if (elem.equals(target)) {
            return true;
        }
    }
    return false;
}
```

Figure 4.2: An example method declaration, from which the AST of Figure 4.3 was generated

without including all details like punctuation and delimiters. For instance, a sample Abstract Syntax Tree is displayed in Figure 4.3 which was constructed from the source code of Figure 4.2. To better understand the advantages and best possible methods of using the AST, we have chosen experimented five different code representations, of which four relies on AST information. The five representations are described below. These were constructed according to our previous work and are some of the most widely used representations in other research experiments [156], constructed along the work of Tufano et al. [148].

SRC

Let us consider the source code as a structured text file. Textual methods are often used in the context of natural language processing. These techniques include the tokenization of sentences into separate words and the application of stemming. With natural language, the separation of words can be quite simple. In the case of source code, however, we should consider other factors as well. For instance, compound words are usually written by the camel case rule, and names can also be separated by punctuation. The definition of these separators are one of the main design decisions in this representation. For the current work, words were split by the camel case rule, by white spaces and by special characters that are specific to Java ("(", "[", "."). The Porter stemming algorithm was used for stemming. This approach notably does not use the AST of the files, making it a truly only text-based approach.

TYPE

To extract this representation for a code fragment, an Abstract Syntax Tree has to be constructed. This process ignores comments, blank lines, punctuation, and delimiters. Each node of the AST represents an entity occurring in the source code. For instance, the root node of a class (CompilationUnit) represents the whole source file, while the leaves are the identifiers in the code. In this particular case, the types of AST nodes were used for the representation. The sequence of symbols was obtained by pre-order traversal of the AST. The extracted sequences have a limited number of symbols, providing a high-level representation.

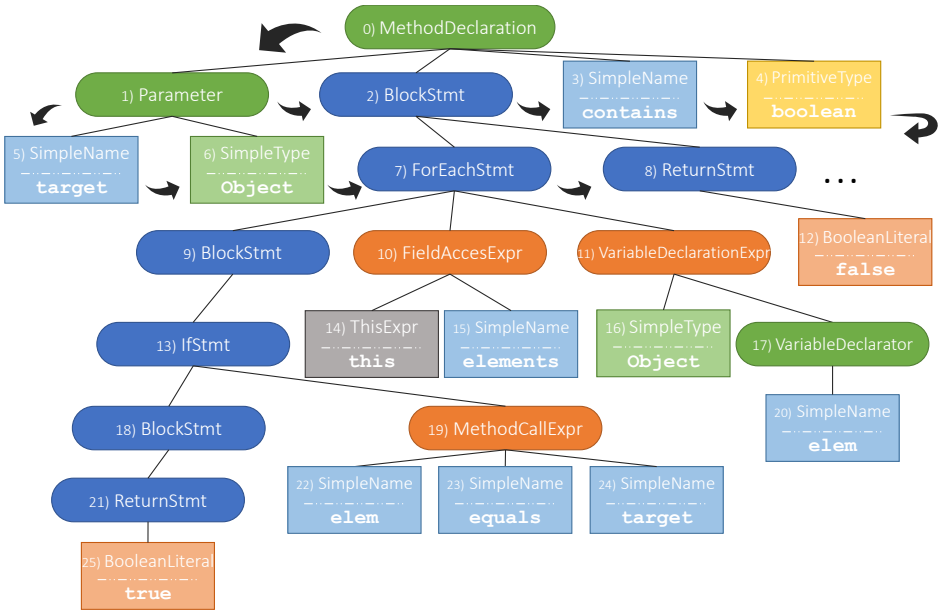

Figure 4.3: An Abstract Syntax Tree, generated from the example of Figure 4.2. The numbers inside each element indicate the place of the node in the visiting order. Leaves are denoted with standard rectangles (note that here the value and the type is also represented), while intermediate nodes are represented by rectangles with rounded corners

IDENT

Every node in the Abstract Syntax Tree has a type and a value. The top nodes of the AST correspond to a higher level of abstraction (like statements or blocks), their values typically consist of several lines of code. The values of the leaf nodes are the keywords used in the code fragment. In this representation, these identifiers are used by traversing the AST tree and printing out the values of the leaves. The values of literals (constants) in the source code also might occur here, these are replaced with placeholders representing their type (e.g. an integer literal is replaced with the *<INT>* placeholder, while a string literal with *<STRING>*). The extracted identifiers contain variable names. In the current experiments, they were split according to the camel case rule popularly used in Java.

LEAF

In the previous two representations, distinct parts of the AST were utilized to get the input. This approach takes both the types and node values into account. Just as before, a pre-order visit is performed from the root. If the node is an inner node then its type, otherwise (when it is a leaf) its value is printed. This representation captures both the abstract structure of the AST and the code-specific identifiers. Considering the latter, these can be very unique and thus very specific to a class or a method.

SIMPLE

The extraction process is very similar to the previous one, except that in this case only values with a node type of *SimpleName* are printed out. These nodes occur very often, they constituted 46% of an AST on average in our experiments. These values

correspond to the names of the variables used in the source code while other leaf node types like literal expressions or primitive types hold very specific information. Note that in the IDENT representation, the replacing of literals eliminated the AST node types of literal expressions. Only the modifiers, names, and types remained, thus becoming similar to this representation. With this representation, however, we do not exclude the inner structure of the AST.

4.3.8 Evaluation Procedure

In their 2009 evaluation, Van Rompaey and Demeyer [132] found that the naming conventions technique produced 100% precision in finding the tested class at each test case it was applicable to. The authors used a human test oracle consisting of 59 randomly chosen test cases altogether. These can be considered too few measuring points for proper generalization, but nevertheless, it is visible that naming conventions can identify the class under test in the overwhelming majority of the cases. Naming convention pairs can also be extracted automatically from method, class, and package names. Thus, one of our evaluation methods relies on the naming conventions technique.

Since naming convention habits may influence this, our approach was also evaluated on a human test oracle described in [76]. TestRoutes is a manually curated dataset that contains data on four of our eight subject systems, Commons Lang, Gson, JFreeChart and Joda-Time. It is a method-level dataset that classifies the traceability links of 220 test cases (70 from JFreeChart, 50 from each of the others). This information is also suitable for class-level evaluations, as this is a relaxed version of the same problem. The dataset lists the methods under test as focal methods (there can be multiple focal methods for a test case), as well as test and production context. Our current focus is on the classes of these focal methods. For JFreeCart and Joda-Time, the dataset specifically targeted test cases that were not covered with simple naming conventions, this will also be evident at our results. For the other systems, the dataset contains data on randomly chosen test cases.

The TestRoutes data was annotated by a graduate student familiar with software testing. The tests were not executed during the annotation process. The annotator worked in an integrated development environment, studied the systems' structure beforehand, and maintained regular communication with the researchers, addressing the arising concerns. The collected traceability links were inspected and validated by a researcher, with another researcher also verifying the links of at least ten test cases of each system.

A relatively simple yet sufficiently strict set of rules was applied in the naming convention based evaluation. Our NC-based evaluations were based on package hierarchy and exact name matching. This is further detailed in Subsection 4.4.1, where this particular naming convention ruleset is referred to as PC (package + class).

With such an evaluation, it is only possible to find one pair to each test case correctly. Our methods produce a list of recommendations in order of similarity. Every class is featured on this list. Thus, with our current evaluation methods, the customary precision and recall measures always coincide, which necessarily means that the F-measure metric would also have the same value. This is in accordance with the evaluation techniques commonly used for recommendation systems in software engineering. Because of this equality, we shall refer to our quantified results in the chapter as precision only.

4.3.9 Sample Projects

Our results were evaluated on multiple software systems and with multiple settings. These involved the following open-source systems: ArgoUML is a tool for creating and editing UML diagrams. It offers a graphic interface and relatively easy usage. Commons Lang is a module of the Apache Commons project. It aims to broaden the functionality provided by Java regarding the manipulation of Java classes. Commons Math is also a module of Apache Commons, aiming to provide mathematical and statistical functions missing from the Java language. Gson is a Java library that does conversions between Java objects and Json format efficiently and comfortably. JFreeChart enables Java programs to display various diagrams, supporting several diagram types and output formats. Joda-Time simplifies the use of date and time features of Java programs. The Mondrian Online Analytical Processing (OLAP) server improves the handling of large applications' SQL databases. PMD is a tool for program code analysis. It explores frequent coding mistakes and supports multiple programming languages.

The versions of the systems under evaluation, their total number of classes and methods, and the number of their test methods are shown in Table 4.1, while Figure 4.4 visually reflects these numbers. It has to be noted that several methods of the test packages of the projects have been filtered out as helpers since they did not contain any assertions.

Table 4.1: Size and versions of the systems used

System	Version	Classes	All Methods	Test methods
ArgoUML	0.35.1	2,404	17,948	554
C. Lang	3.4	596	6,523	2,473
C. Math	3.4.1	2,033	14,837	3,493
Gson	2.8.0	757	2,467	924
JFreeChart	1.0.19	953	11,594	2 239
Joda-Time	2.9.6	522	9,934	3,779
Mondrian	3.0.4.11371	1,626	12,186	1,546
PMD	5.6.0	1,608	9,242	825

4.3.10 Mining Stack Oveflow for Traceability Links

Another notable experiment on our part was the employment of information retrieval in attempting to link test and production code snippets on the well-known Stack Overflow [2] site. Our experiments used the data provided by Baltes et al. [14].

The goal of our experiment was to see how information retrieval methods perform in cases where vast amounts of data are present and to see whether they can still provide valid results. LSI and Doc2Vec were featured in these experiments.

During the experiments, we filtered out the posts that were old versions or did not deal with the Java language and only extracted the questions and accepted answers for each post. The resulting 1.5 million posts were submitted to textual preprocessing, consisting of the extraction of methods via regular expressions, separating test and code methods, and filtering out extremely short methods. Thus, 33,115 test methods and 608,067 production methods were extracted from the whole dataset. The methods

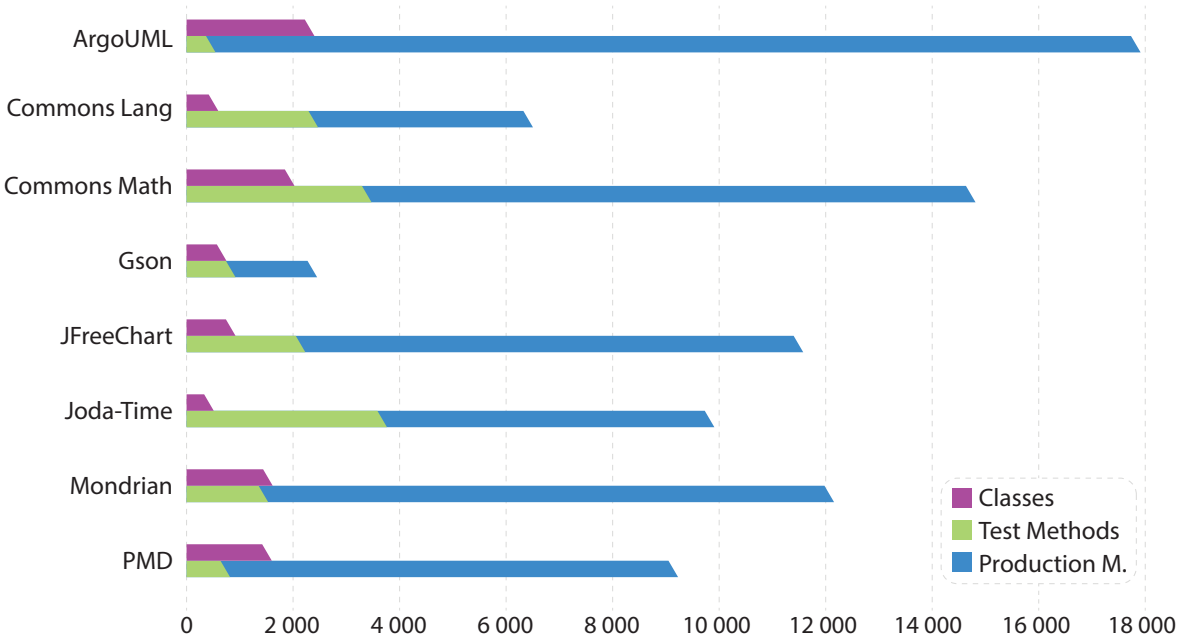

Figure 4.4: Properties of the sample projects used

underwent lower casing, camel case splitting, and stopword filtering also. As this was found to exceed our computation capacity, our experiments with Doc2Vec were performed with the snippets of 100,000 randomly chosen posts, while LSI used all available methods.

4.4 Results

The current section evaluates the various approaches described in the previous section, featuring the results obtained from different representations and learning settings. Various naming convention possibilities are overviewed with their applicability values determined via automatic extraction. Next, our experiments with the $ensemble_N$ approach are presented, where the best N value has been sought on NC-based and manual traceability links. The traceability approaches are also compared to each other based on both NC and manual evaluation.

We note that production methods containing less than three tokens in their method bodies were filtered out since trivial and abstract production methods are not likely to be the real focus of a test.

4.4.1 Applicability of Naming Conventions

Naming conventions for tests are a vague term that can mean a multitude of various practices. The conventions are usually agreed upon by the developers and written guidelines rarely even exist. They can also be only considered a mere good practice, and their use varies by teams or even individuals. As there can be various conventions, and their use is different in most systems, relatively vague criteria are needed to detect them in a versatile manner. Let us consider a few general criteria for our examination. These are presented in Figure 4.5. There are, of course, other possible criteria, including abbreviations or some other distinction for tests except the word "Test". Still, these seem to be the most intuitive and most popular naming considerations.

Let us consider some of the possible combinations of the listed criteria components. Figure 4.6 presents these. Some other viable combinations can also exist, which did not seem suitable for the unique distinction of test-code pairs. The criteria are ordered by strictness in a descending manner. While the stricter criteria produce more distinction between pairs, they are less versatile and are harder to uphold. Table 4.2 presents the extent to which the naming conventions were found to be applicable to the evaluated systems.

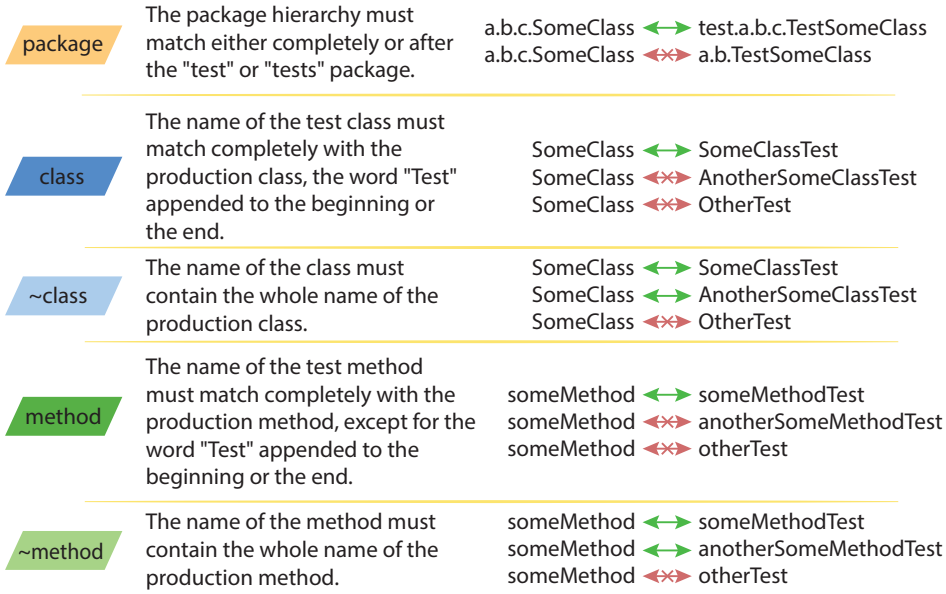

Figure 4.5: Various possible naming convention criteria components

Table 4.2: The applicability of the naming conventions technique using different approaches

Naming Criteria	ArgoUML	Commons Lang	Commons Math	Gson	JFreeChart	Joda-Time	Mondrian	PMD
PCM	14.91%	17.04%	12.50%	1.74%	32.60%	3.60%	6.57%	7.93%
PM	20.73%	19.80%	16.85%	2.18%	38.95%	10.09%	9.04%	8.43%
PCWM	19.82%	56.67%	32.19%	9.59%	49.53%	23.78%	11.51%	15.86%
PWCWM	21.27%	66.73%	37.52%	9.59%	50.92%	59.65%	12.09%	16.48%
PWM	33.45%	70.79%	45.42%	15.47%	58.64%	74.42%	31.01%	25.53%
M	28.91%	19.96%	21.16%	3.05%	40.15%	11.38%	12.22%	11.90%
PC	60.18%	84.58%	75.07%	26.14%	96.47%	36.80%	17.82%	58.36%
C	64.00%	84.58%	75.07%	27.89%	96.47%	37.30%	20.81%	66.91%
WM	74.00%	80.00%	81.53%	60.24%	61.28%	78.55%	73.34%	58.36%
PWC	75.09%	99.11%	88.06%	28.87%	97.05%	98.04%	21.52%	61.09%
WC	80.55%	99.11%	91.42%	44.77%	97.41%	98.04%	35.96%	72.12%

As it is visible from the results of the table, there is a significant jump in the applicability at the PC naming convention variant, which considers package hierarchy and an exact match to the name of the class. While the extent of the increase of applicability varies between systems, it is apparent that most of them produce only very few traceability links when method names are also considered. As our experiments featured class level test-to-code traceability, our further results are going to use PC as the default naming convention.

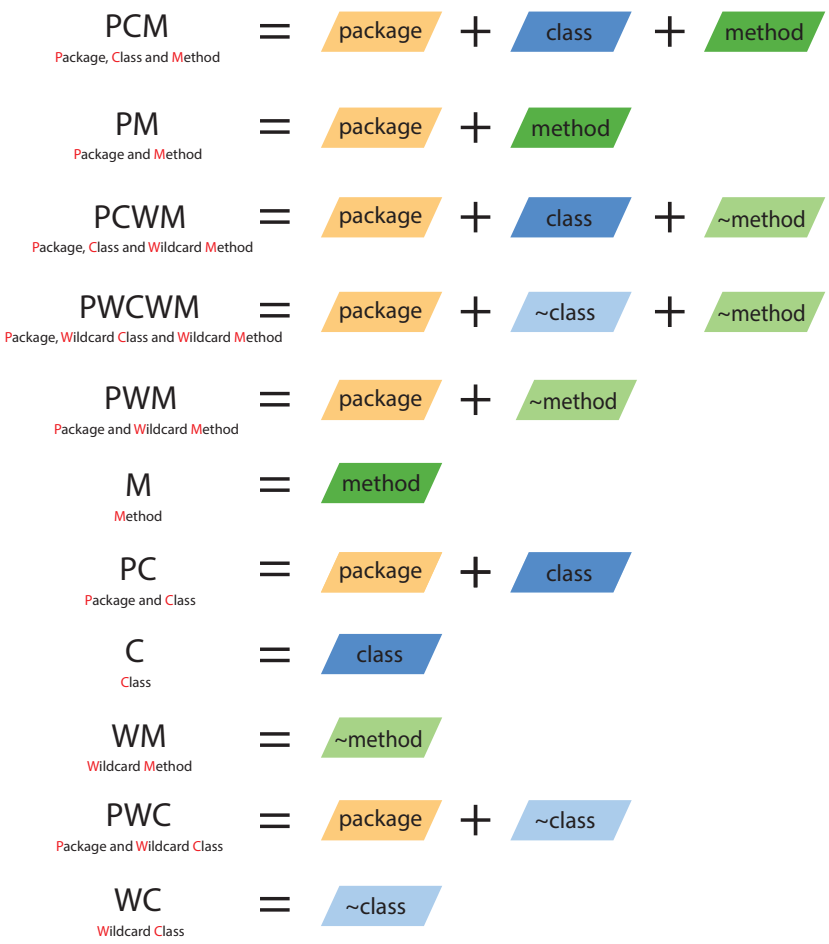

Figure 4.6: Some of the possible naming convention criteria in descending order of restrictiveness

4.4.2 Ensemble Experiments

Figure 4.7 and Figure 4.8 show the results of our $ensemble_N$ learning approach. As one can see in the figures, the experiments were carried out using different N values: 50, 100, 200, and 400. These values only influence the size of each similarity list. If N is relatively big, then the filtering on the original similarity list (which originates from Doc2Vec) will not drop out many entries since many of the elements are present in the other two lists. In contrast, if N is a small number, the filtering is stricter since every similarity list contains only a limited number of entries. The previous argument can be further elaborated: if N is *big*, the resulting similarity list is going to rely mostly on the original one, while if it is *small*, the approach makes better use of the information from the other two approaches.

First, let us consider Figure 4.8, which visualizes the results from the sample projects measured via automatic naming convention extraction. The small flags on the top of the bars indicate the highest values for each system in their category (top_1, top_2, top_5). The flag's color is the same as its bar; a white flag means that the highest values are equal. Remarkably, no case was encountered where there were two or three highest values. In this experiment, the different source code representations are also considered. Looking at the figure, it is apparent that most of the flags appear at the IDENT representation. It is also worth mentioning that at the top_1 results, the

$ensemble_{50}$ approach seems to produce the highest values. Considering multiple recommendations (top_2 and top_5), the situation is less obvious: $ensemble_{100}$ also seems to provide good results. $Ensemble_{400}$ seems to be less precise. It was prevalent only in the case of Mondrian using the SIMPLE representation. The results on the manual dataset also reinforce this finding. In Figure 4.7, almost every flag belongs to the $ensemble_{50}$ approach, except in two cases, when it produced the same value as the others.

Figure 4.7: Results of the $ensemble_N$ learning approach measured on the manual dataset

4.4.3 NC-based Evaluation

Table 4.3 shows the top_1 results of different machine learning approaches, evaluated via naming conventions. Cells of color teal indicate the highest values for each system within a method, while cells of color violet indicate the overall top values. For the $Ensemble_N$, only those cases are listed where N = 50, since this setting seemed to be the most beneficial (for further discussion see Subsection 4.4.2). The listed approaches

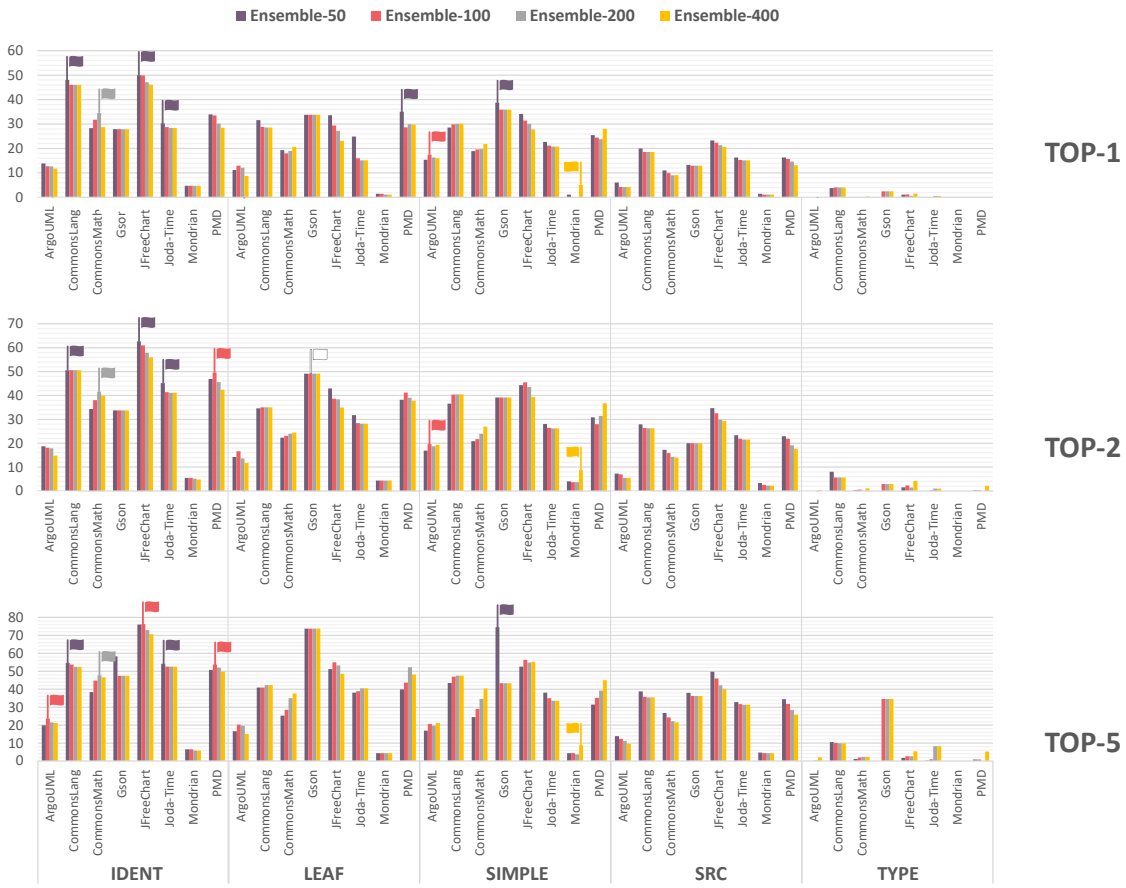

Figure 4.8: Results of the $ensemble_N$ learning approach using NC-based evaluation

correspond to the ones introduced in Section 4.3. *[approach]+**CG*** refers to filtering with our soft computed call information described in Section 4.3.5.

4.4.4 Evaluation on Manual Data

The results measured on the manual dataset are shown in Table 4.4. Similarly to the previous table, teal indicates the highest values within a method, while violet highlights the overall highest value. The left part of the table shows precision values of top-1 matches, while on the right side of the table, the top-5 results are listed. The top-5 results are always equal or higher than the top-1 numbers since there are more than one similar matches considered during the evaluation. Here, the results of different approaches are compared to the dataset's data, which contains manually curated traceability links on four of our subject systems. In this table, two additional rows are introduced. The first row shows the applicability of the naming convention (that was denoted PC previously). These numbers depict the conventions' applicability to the specific test cases in the dataset, rather than the whole system. If naming conventions should be considered accurate, this value would intuitively correspond to the precision that could be achieved without any additional IR-based approach, only relying on the names. The last line's title contains the *NC* addition. Our method here first attempts to detect the link using naming conventions, and if it fails, the suggestion of Doc2Vec is considered. If the resulting precision values would be lower than before,

Table 4.3: Top-1 results featuring the different text-based models trained on various source code representations, evaluated using naming conventions. - highest value in a row - highest value in a column

Method	Representation	ArgoUML	C. Lang	C. Math	Gson	JFreeChart	Joda-Time	Mondrian	PMD
Doc2Vec	IDENT	19.63%	82.16%	50.00%	45.83%	49.22%	41.43%	66.42%	37.15%
	LEAF	18.43%	61.00%	33.01%	47.92%	25.10%	20.79%	65.33%	42.04%
	SIMPLE	24.77%	67.91%	33.78%	47.50%	30.69%	26.26%	65.33%	34.82%
	SRC	7.85%	31.32%	15.46%	16.67%	22.64%	22.30%	21.53%	15.92%
	TYPE	0.60%	4.36%	0.78%	2.92%	2.36%	5.18%	0.00%	0.00%
LSI	IDENT	32.93%	66.08%	19.42%	30.83%	33.29%	35.04%	22.99%	19.96%
	LEAF	14.80%	23.11%	3.63%	7.08%	9.63%	16.12%	11.31%	7.80%
	SIMPLE	15.71%	21.48%	3.47%	4.37%	13.15%	6.69%	4.38%	11.46%
	SRC	19.64%	54.64%	24.36%	14.17%	21.48%	28.20%	31.02%	22.29%
	TYPE	0.00%	0.48%	0.65%	4.58%	0.00%	0.50%	0.00%	0.00%
TF-IDF	IDENT	35.95%	73.62%	35.78%	35.00%	45.65%	48.71%	73.72%	24.63%
	LEAF	32.63%	70.94%	37.33%	38.33%	48.93%	47.77%	66.79%	23.99%
	SIMPLE	28.70%	69.49%	33.08%	30.00%	44.30%	47.77%	72.26%	23.57%
	SRC	27.79%	51.51%	28.68%	18.75%	25.19%	31.08%	50.73%	22.29%
	TYPE	0.00%	0.48%	0.65%	4.58%	0.00%	0.50%	0.00%	0.00%
Ensemble-50	IDENT	13.89%	48.00%	28.27%	27.92%	50.00%	30.22%	4.75%	33.97%
	LEAF	11.18%	31.61%	19.29%	33.75%	33.56%	24.89%	1.45%	35.03%
	SIMPLE	15.41%	28.54%	18.93%	38.75%	34.01%	22.73%	1.01%	25.48%
	SRC	6.04%	20.00%	11.02%	13.33%	23.24%	16.33%	1.46%	16.35%
	TYPE	0.00%	3.79%	0.12%	0.00%	1.11%	0.00%	0.00%	0.00%
Doc2Vec+CG	IDENT	45.01%	83.14%	61.04%	85.83%	62.82%	43.02%	68.61%	54.14%
	LEAF	42.29%	72.66%	45.65%	44.17%	56.48%	34.10%	73.72%	52.23%
	SIMPLE	41.69%	71.89%	51.41%	52.92%	58.06%	24.32%	74.09%	51.80%
	SRC	32.33%	54.48%	29.62%	35.42%	37.36%	23.38%	47.08%	36.31%
	TYPE	3.63%	17.61%	13.22%	42.08%	10.46%	16.04%	41.24%	15.92%

that would either mean that the dataset is incorrect, or the naming conventions were misleading. It is also clear that if the results of this approach and the plain NC approach were equal, then the IR-based addition would be unnecessary. Eventually, none of these concerns were found to be reflected in Table 4.4. In fact, this approach produced the best results in almost every single case.

4.4.5 Mining StackOveflow for Traceability Links

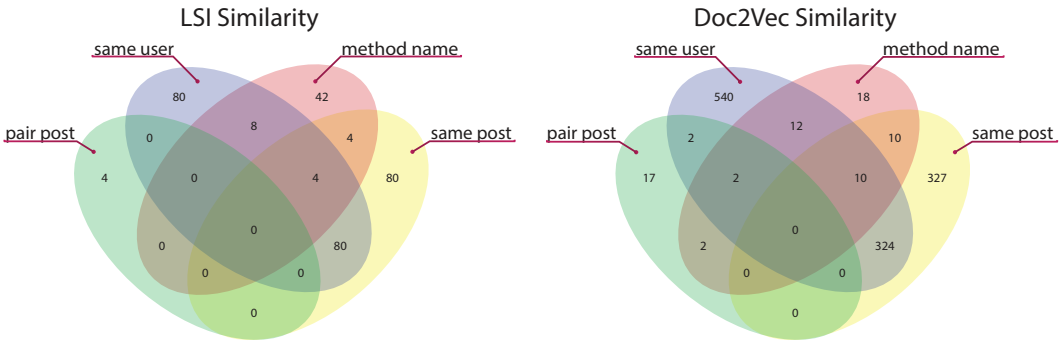

Figure 4.9: A Venn diagram about our Stack Overflow matches

We also briefly mention the results of the Stack Overflow experiment. As there are no real systems to speak of when only looking at code snippets, an accurate evaluation cannot be achieved due to the lack of information. There are, however, some indicators

Table 4.4: Top-1 and top-5 results featuring the different text-based models and the applicability of NC on each project. Models were trained on 5 different source code representations. - highest value in a row - highest value in a column

Method	Representation	Top-1				Top-5			
		C. Lang	Gson	JFreeChart	Joda-Time	C. Lang	Gson	JFreeChart	Joda-Time
NC	-	76.00%	26.00%	0.00%	0.00%	76.00%	26.00%	0.00%	0.00%
Doc2Vec	IDENT	58.00%	15.69%	15.49%	32.00%	62.00%	25.49%	15.49%	48.00%
	LEAF	30.00%	13.73%	11.27%	20.00%	52.00%	21.57%	15.49%	52.00%
	SIMPLE	15.69%	17.65%	14.08%	16.00%	48.00%	21.57%	16.90%	52.00%
	SRC	16.00%	9.80%	12.68%	32.00%	42.00%	29.41%	30.99%	54.00%
	TYPE	4.00%	1.96%	11.27%	4.00%	22.00%	3.92%	11.27%	10.00%
LSI	IDENT	34.00%	17.65%	4.23%	10.00%	68.00%	5.64%	5.63%	44.00%
	LEAF	12.00%	7.84%	4.23%	2.00%	34.00%	23.53%	5.63%	28.00%
	SIMPLE	4.00%	5.88%	4.23%	2.00%	30.00%	23.53%	5.63%	24.00%
	SRC	34.00%	17.65%	12.68%	20.00%	70.00%	37.25%	23.94%	58.00%
	TYPE	4.00%	0.00%	0.00%	0.00%	8.00%	64.71%	0.00%	14.00%
TF-IDF	IDENT	30.00%	19.61%	4.23%	46.00%	76.00%	31.37%	5.63%	70.00%
	LEAF	30.00%	19.61%	4.23%	44.00%	76.00%	33.33%	5.63%	70.00%
	SIMPLE	28.00%	21.57%	4.23%	44.00%	72.00%	33.33%	5.63%	72.00%
	SRC	38.00%	19.61%	23.94%	12.00%	78.00%	43.14%	25.35%	68.00%
	TYPE	4.00%	0.00%	0.00%	0.00%	8.00%	64.71%	0.00%	14.00%
Ensemble-50	IDENT	44.00%	13.73%	4.23%	6.00%	52.00%	23.53%	4.23%	10.00%
	LEAF	13.73%	13.73%	4.23%	10.00%	38.00%	19.61%	4.23%	14.00%
	SIMPLE	14.00%	15.69%	4.23%	2.00%	40.00%	19.61%	4.23%	8.00%
	SRC	7.84%	11.76%	11.27%	12.00%	36.00%	17.65%	28.17%	22.00%
	TYPE	2.00%	1.96%	0.00%	2.00%	8.00%	1.96%	0.00%	2.00%
Doc2Vec+CG	IDENT	58.00%	64.71%	16.90%	24.00%	76.00%	80.39%	23.94%	64.00%
	LEAF	54.00%	54.90%	18.31%	20.00%	72.00%	78.43%	33.80%	66.00%
	SIMPLE	50.00%	56.86%	25.35%	26.00%	76.00%	78.43%	45.07%	64.00%
	SRC	50.00%	56.86%	36.62%	32.00%	78.00%	82.35%	66.19%	74.00%
	TYPE	42.00%	47.05%	11.27%	6.00%	62.00%	74.51%	28.17%	24.00%
Doc2Vec+CG+NC	IDENT	76.00%	64.71%	16.90%	24.00%	86.00%	72.55%	23.94%	64.00%
	LEAF	78.00%	64.71%	18.31%	20.00%	84.00%	70.59%	33.80%	66.00%
	SIMPLE	78.00%	66.71%	25.35%	26.00%	84.00%	76.47%	45.07%	64.00%
	SRC	80.00%	66.67%	36.62%	32.00%	88.00%	78.43%	66.19%	74.00%
	TYPE	74.00%	64.71%	11.27%	6.00%	78.00%	76.47%	28.17%	24.00%

that can imply connection and were not used by the textual methods. For example, test and production snippets submitted by the same user or even belonging to the same post are more probable to be indeed connected. A manual inspection showed that even though a high amount of faulty matches were produced, both LSI and Doc2Vec can produce traceability links that seem highly accurate even for human observers, and even if they could not be easily recovered by other indicators. Figure 4.9 showcases some of our results in a Venn diagram. The areas show our various indicatives of success. Same user means the two methods were submitted by the same user. Same post means that they were featured in the same post, while pair post means that one of the methods was submitted in a question while the other in the accepted answer. Method name means that the test method shares the name of the production method, with an added Test token to either the start or the end. It is visible that Doc2Vec seems to have found much more of the results that seem relevant. Note that LSI worked with significantly more data. Thus, we also performed the experiment with the same subset for LSI, receiving even worse matches with the indicators.

4.5 Discussion

4.5.1 Naming Conventions Habits

The previous section displayed some of the most common naming convention techniques and some data on how frequently they seem to have been utilized in the systems currently under our investigation.

Let us consider an example of how a perfect match would look like viewing all three of package hierarchy, class name, and method name. One such example for Commons Math is illustrated in Figure 4.10, which shows a test case (T) and the production method it is meant to test (P). If every single test case related to its method under test with such simplicity, test-to-code traceability would be a trivial task. Unfortunately, as it is visible from our applicability results, this is very far from reality.

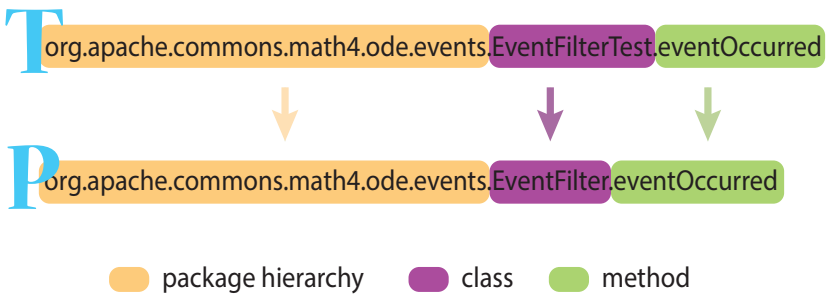

Figure 4.10: A trivial naming convention example from Commons Math

According to our results, the names of test *methods* are much less likely to mirror the names of their production pairs correctly. Although our experiments only dealt with eight open-source systems, it is highly probable that the developers of other systems also tend to behave similarly in focusing more on class-level naming conventions. WM (wildcard method) is obviously full of noise and accidental matches and cannot be considered seriously. PM (package+method) is a much more precise option but as it is visible it was found to be used in about every fifth case. One obvious reason for this can be that it is significantly harder to convey all the necessary information in method names. Production method names should be descriptive and lead to an easy to understand and quick comprehension of what the method does. This is also true about the names of test cases, they should also refer to what functionality they are aiming to assess. Consequently, the names of the test cases would become rather long if they always aimed to contain both the name of the method or methods under test and also provide additional meaningful information about the test itself. It can also be tough to properly reference the method under test on method level by naming conventions only. Polymorphism enables the creation of several methods with identical names that perform similar functionalities with different parameters. These should be tested individually, and test names can have a hard time distinguishing these. The inclusion of parameter types can be a possible solution as performed in Commons Lang for example, at the test case *test_toBooleanObject_String_String_String_String*, testing the production method *toBooleanObject* that gets four String parameters. Our manual investigation shows that test methods are indeed more likely to be named after the functionality they mean to test rather than after single methods even if they only test one method. One method can also be tested by multiple test cases. Thus this is not a

very surprising circumstance. It is apparent that naming conventions on the method level have to be more complicated, and their maintenance necessitates more work on the part of the developers. Thus, method-level naming solutions are likely to be a less valuable option in method-level test-to-code traceability. On the other hand, method-level traceability still requires proper class-level traceability. Thus, names should still be helpful.

Talking about *classes*, production class names seem to be mirrored more often in their test classes' names. This, however, still can be a highly unsteady habit depending on the system. While in Mondrian, production class names are only present in test names once in every fifth case, the same applies to 2160 of the 2239 test cases of JFreeChart. Thus, not surprisingly, it is evident that naming conventions depend on developer habits. The remaining test cases of JFreeChart were also examined, these are overwhelmingly cases where a specific type of charts or other higher-level functionalities are tested, and the test classes are named after these. These cases often depend on multiple production classes providing lower-level functionality.

Mirroring the *package-hierarchy* of the production code while composing tests is also a good practice. The little difference between the C (class) and PC (package+class) values in Table 4.2 shows that developers are very likely to uphold this. This convention is likely to be even more popular than naming matches. Package hierarchy matches are easier to maintain than names and are more convenient as they do not really require additional work from the developers. Even if multiple methods or classes are under tests, their packages only rarely differ. It could also be seen as another level of abstraction. Package hierarchy can only provide very vague clues about traceability links but can be suitable for the elimination of some of the false matches or at least presenting a warning sign about some matches. From the difference of matches found with C-PC (class versus package + class) and WC-PWC (wildcard class versus package + wildcard class), our manual investigation provided less conclusive results.

On the one hand, many systems contain some seemingly arbitrary exceptions to this rule that were most probably due to some design decision or modification in the production code that the structure of the tests has not followed yet. Gson, for example, has a "gson" package in its test structure that contains similarly named test classes to the "internal" package of the production code. Another example can be given from PMD, where there is an extra "lang" package in the hierarchy of the production code, that is not found at the test structure, even though all prior packages match. These are far from system-wide decisions as seen from the NC applicability percentages, but can be hard to detect by automatic means. On the other hand, some faulty matches do exist when not considering the package hierarchy. This can be seen at ArgoUML for instance, the "Setting" class of the production code can match with a lot of test classes if only names are taken into account, many of these would be faulty matches as the tests refer to different settings. Thus, matching packages is also far from universal in real-live systems. Like any other convention, developers make exceptions even if they visibly strive to uphold them at different parts of the code.

Without the insights from the developers of a system, our analysis had to judge their choice and usage of naming conventions based solely on the names themselves. This, of course, can be sufficiently accurate but presents the danger of not managing to grasp the whole system of conventions they used, which can vary. Still, our findings should provide a relatively accurate picture of how naming conventions are used in real-life testing solutions.

Answer to RQ1: Although serious differences can be observed between systems, method-level naming conventions are either complicated or entirely abandoned in most cases, which means that their usefulness is negligible in a general extraction algorithm. Class-level naming conventions seem to be better regarded by developers, and there is a visible effort to uphold them. Our findings show them to be suitable for general use in automatic extraction algorithms. Matching package hierarchies do not provide precise results but seem to be at least as commonly used as class-level conventions. They are likely to be suitable for filtering out false-positive results in algorithms.

4.5.2 Traceability Link Recovery Technique Improvements

It is apparent at first sight that the teal cells are overwhelmingly located in the first rows in Table 4.3. Indeed, the IDENT source code representation seems to be prevalent: it reaches the highest values in 37 cases out of 48 (which is 77%). The violet cells appear only in the last vertical segment of the table, at the Doc2Vec+CG approach.

On the one hand, the $Ensemble_{50}$ approach produced better results than standalone techniques (Doc2Vec, LSI, TF-IDF) and the soft-computed call graph information even improved upon this. On the other hand, Doc2Vec supplemented with this call information resulted in the highest precision values. How is this even possible? The most probable explanation is that $Ensemble_N$ is a filter technique: the resulting similarity list is a reduced one compared to the original (especially when N is a relatively small number). Thus it can happen, that even before applying CG, the $Ensemble_{50}$ already dropped out some of the correct links.

According to the results of the table, IDENT seems the most precise approach. The only exception is the Mondrian project, where the SIMPLE representation appeared to perform best. The difference between IDENT and SIMPLE, however, is not remarkable. It is also worth mentioning that where IDENT is not predominant, SIMPLE was found best in 5 cases out of 11. Subsection 4.3.7 already stated that IDENT and SIMPLE are quite similar. This is also reflected in the results. In contrast, TYPE seems to produce weaker results with every single approach. LEAF is also less precise, probably because its inner structure shares a significant part with TYPE (LEAF is essentially the combination of IDENT and TYPE). From this, a conclusion can be made that the TYPE information of an AST holds less important information for the text-based test-to-code traceability task.

Answer to RQ2: Our inspections concluded that Doc2Vec seems to be the best-performing standalone technique in the field. Although combinations of different techniques can also boost the results, the textually extracted soft-computed call information is likely to boost IR-based approaches even more. In a scenario of combined techniques, call graphs can be a valid filter even for textual connections.

4.5.3 Performance on Manual Data

Compared to the NC-based evaluation, the results captured on the manual dataset are less easy to interpret. As it is visible in Table 4.4, not every violet cell appears in the last row, only most of them. Let us first analyze the top-1 results which are shown on the left side of the table. At 3 out of 4 systems, the highest precision values are reached using Doc2Vec combined with the call information and naming conventions. The only exception is Joda-Time, where TF-IDF seems to be prevalent.

In our earlier work [30], a similar case has already been noted, where TF-IDF also provided the highest precision values. Even in these experiments, however, TF-IDF results are found to be much more variable than others, and this individual case is most probably a result of chance. It is visible, however, that Doc2vec+CG still seems to have produced high precision values, and that applying naming conventions can further boost the approach. In the case of JFreeChart and Joda-Time, the results did not improve despite the added naming convention pairs. This is not surprising since as it is visible, the naming conventions were not applicable for any methods of these systems (as the dataset contained specifically such links by intention). It can, therefore, be stated that IR-based approaches can successfully supplement naming conventions while still maintaining their useful properties.

Looking at the right side of Table 4.4, the precision values are higher than before. It is quite self-evident since here the text-based models have a broader range to guess for the correct matches. While observing top-1 results, the best performing technique was not unanimous. Here, the highest precision values are all located in the last segments. While top-1 results varied in their precision, JFreeChart and Joda-Time having lower results than the other two systems, even a small number of additional candidate links has significantly contributed to the correctness of the matches. By further experiments, it was found that when considering top-10 or even top-20 results, a 100% match would not be uncommon either, though searching through a list of 10 artifacts is not a likely behavior of real-life developers. Thus, their everyday use of these would not be viable. Five results, however, could still make a simple recommendation system.

By studying the manual database, it can be observed that in the cases of projects where proper naming conventions were used, the traceability links can be extracted relying only on this information. However, for those projects which lack these good programming practices, IR-based techniques can find the correct links in a significant part of the cases. Comparing the results of the final Doc2Vec approach and NC itself, the precision values are higher by 28% on average. Even in the cases where some more complex, system-specific naming conventions were utilized, IR-based methods can provide great assistance. While there are likely to be some special cases in the practical use of every naming convention, the use of good programming practices can also improve the performance of text-based techniques which still rely on names in a more versatile manner.

From these results, the choice of an ideal input representation is a more elusive question than in the previous case. While at the NC-based comparison, the IDENT representation seemed prevalent against others, here the SRC representation seems to have produced the highest precision values. It is worth mentioning that among the representations that rely on AST information, SIMPLE performed best. At the Top5 values, it is also clear that the SRC representation won. The good results of SRC are also advantageous since SRC is a purely text-based representation. Since the Doc2Vec+CG+NC method features call information extracted via regular expressions, this is a viable option even without static analyzer tools. While both IDENT and SRC are shown to contribute valuable data, this difference between their relative performance on thousands of test cases of eight systems and the manual data on 220 test cases of four systems makes it harder to believe in a single best simple representation. Since the possible mistakes in the automatically gathered NC data and the less than ideal size of the manual dataset can both contribute to less than precise results in this respect, further research is still necessary for the question of best representation.

Answer to RQ3: According to the manual data, Doc2Vec achieves the best results in most cases. In exceptional cases, however, other text-based techniques can still outperform it. The use of naming conventions and call information also tends to improve the results further. Naming conventions, if existing, are highly precise and can be supplemented with other IR-based techniques to achieve a more versatile text-based approach.

4.5.4 Implications

Our experiments show some simple implications for those who research and aim to build new test-to-code traceability solutions.

While naming conventions are reliable tools and are very precise if applied, they are harder to implement on the method level, and the source code generally contains fewer such connections that could be extracted via simple rules. However, packages and class names can imply the connections rather well, even for this level, even if method names are not as informative. Thus, naming conventions can be extremely useful on every level of traceability link extraction, and new extraction methods would most probably benefit from considering them.

Doc2Vec seems an important upgrade to the more mainstream semantic similarity techniques. While it is still somewhat more resource-intensive than LSI, the difference is not prominent, and just like the other techniques, Doc2Vec is also capable of providing real-time results of most similar parts of code for a test case. Thus, if a single textual technique should be considered, Doc2Vec seems to be the right choice.

The combination of Doc2Vec and other techniques can produce even better results. However, as it was seen that applied as filters, other textual techniques can drop out some of the valuable data, and even if they performed better together in separation, this combination could negatively impact the overall cooperation with other, non-semantic techniques. Call information, even if just gathered via regular expressions, tends to boost these techniques greatly. Combination with call graphs obtained via static or dynamic analysis could undoubtedly result in even better precision, as seen in current state-of-the-art solutions where the LCBA (last call before assert) technique is considered one of the most reliable methods.

Source code representations are less conclusive at the current time. IDENT seemed best in our previous and current NC-based evaluations, but manual data shows that SRC contributes most to the correct extraction. Thus, additional experiments are still required to announce a clear best representation. Nevertheless, these two are the most likely options.

4.6 Threats to Validity

Although our experiments were conducted with the intention of providing a large-scale evaluation and a relatively deep comprehension of current textual methods, some threats to the validity of the derived conclusions still have to be mentioned. While naming conventions are considered a very precise source of information, they have clear limitations. Thus, our automatically-collected evaluation data may contain some errors and is likely to miss at least some valid links. Although manual data is usually considered best, naming conventions enabled us to assess hundreds of tests for each system and even thousands for most. On the other hand, our manual dataset used for

the evaluation is limited in size. Thus, noise in the data could cause discrepancies in the results. This could be tackled by the inclusion of additional manual data, which will hopefully be more widely available in the future.

Our experiments only covered systems written in the Java language. This is a significant limitation as Java differs greatly from several other popular programming languages. Even the structure of the code can show severe differences. Popular naming conventions can vary in these circumstances, new viable combinations could be constructed, and others could become less relevant. This also reflects a great amount on the source code representations. Even the text and even variable names could be susceptible to such a difference. On the other hand, textual methods, building on semantic information rather than program structure, are still the most likely to retain their properties this way.

The experiments were conducted on JUnit tests. The JUnit framework is one of the trail-blazers of current software testing and is extremely popular among developers. Still, it is easy to see that other tests could perform differently when subjected to the experiments. Even in this, however, semantic information should be the least affected as it does not rely on a specific structure or specific forms of assertion statements.

Similarly to the difference in programming languages, the size of the systems could also influence the results. Our systems under evaluation are all medium-sized open-source systems. There is no guarantee that small or large systems would perform the same way, even though the question of proper traceability is probably easier for small systems. The same questions can arise about the domains of the systems, which could also affect traceability. It is visible that systems vary significantly in their properties. An average value of precision is thus hard to pinpoint, it is easier to compare techniques to each other. Our experiments covered more than 1.25 million code lines to provide a large-scale investigation.

Our experiments with naming conventions and even the source code representations represent the options we found most viable. There might be many more naming conventions that could be applied to some systems with great success, even with automated extraction. As there are usually no descriptions about naming conventions for software systems, finding these and judging their usefulness is highly complex. Our experiments considered some of the most simple and widely used conventions. There seems to be a balance between complete precision and easy usage in naming conventions. Our experiments also attempted to investigate this, building our subsequent experiments on a middle way that seemed widely applicable but still precise for our current level.

4.7 Conclusions

The current chapter showcased our experiments with the textual aspect of aiding test-to-code traceability. Two mainstream techniques, reliance on naming conventions and information retrieval were investigated, new ideas, experiments and observations were given on their possible improvements and combination opportunities. The chapter presented an in-depth investigation of the naming convention habits of developers via experiments with eight open-source systems and nine possible combinations of generalizable and simple rules. This experiment revealed that package and class level conventions are generally followed with at least a moderate effort, but method level conventions, although present in every system, are less generally upheld. Besides our

evaluation on manual data, an automatic extraction was also used for further evaluation, relying on package and class level conventions. The six investigated traceability link extraction methods were evaluated with five different source code representations. From these, the identifier-centric (IDENT) representation that utilizes abstract syntax trees came out on top in the overwhelming majority of the cases during the naming convention based evaluation but the text-centric (SRC) representation proved more precise when compared to the limited amount of manual data. Call information retrieved via regular expressions was found to contribute significantly to the results when used as a filtering technique for Doc2Vec. Although the use of LSI and TF-IDF also seems a good candidate for the same purpose, the combination of Doc2Vec and the call information was found to produce the best results. While properly defined and upheld naming conventions can yield extremely precise traceability links, their use is still limited. Automatic recovery of naming conventions, however, can very easily benefit from the addition of other text-based techniques, together constituting a versatile semantic technique that can still be used in combination with other mainstream methods.

The thesis point investigated textual methods in aiding test-to-code traceability and provided solutions to extracting the tested production classes for test cases. The results were evaluated with the naming conventions of eight real systems and manual data on four systems. Naming conventions were shown to reflect package and class information consistently, which is not true for methods. Our of Doc2Vec, LSI and TF-IDF, Doc2Vec was shown to be the best standalone textual method in recovering traceability links and that it could provide a viable method in combination with call information and naming conventions.

The author considers the followings as his main contributions:

- The author implemented an LSI-based solution for recovering traceability links.
- The author implemented the evaluation code for multiple published experiments, relying on naming conventions. He took part in the evaluation and the planning of new configurations.
- The author coordinated the composition of the TestRoutes dataset, maintained contact with the annotator and provided insights into his work, also manually confirming the results and correcting some of the faulty annotations.
- The author implemented the recovery techniques based on various naming conventions and conducted experiments with them.
- The author took part in the planning of the experiments and coordinated most of them.

Part III

Machine Understanding of Radiologic Reports

"Declare the past, diagnose the present, foretell the future."

— Hippocrates

5 Machine Understanding of Radiologic Reports

5.1 Overview

Computer-based aiding methods are an extremely important aspect of modern healthcare [91] [90] [59]. It is hard to find any specific procedure that is entirely independent of information technology. State-of-the-art software products support surgical modeling, the search for anomalies in image data, and even the creation of clinical reports. In the field of radiology, for instance, doctors regularly utilize dictation software that is often already integrated into their adopted information systems.

An examination involving computer imagery like magnetic resonance imaging (MRI) scans usually involves considerably more than the simple scanning procedure. The scans have to be interpreted by specialists, radiologists in our case, who can understand the images, and look for various malformations and pathological disorders. Even as computer-backed solutions are increasingly helpful in assisting this task, human specialists are still needed to overrule the machine learning algorithms. Moreover, the output of the whole process is still natural language text often written in the native language of the radiologist.

It is easy to see that any subsequent interpretation of the reports has to deal with natural language text, even though the images themselves can also be accessible. The human observations that are backed by medical knowledge and experience are still contained only in textual form. Thus, machine understanding of this text is still a crucial task [86].

The process of creating radiologic reports is illustrated in Figure 5.1. The completed scans get transferred to the radiologists, who, according to their domain knowledge, judge the various negative and positive facts seen in the image. Radiologists either dictate or write down their observations, composing the report itself, and also write an opinion based on this, which usually overlaps with the text of the report. In many cases, the specialist has a choice of using a dictation software or typing the text manually either themselves or by dictating to a radiographist assistant. In either case, the

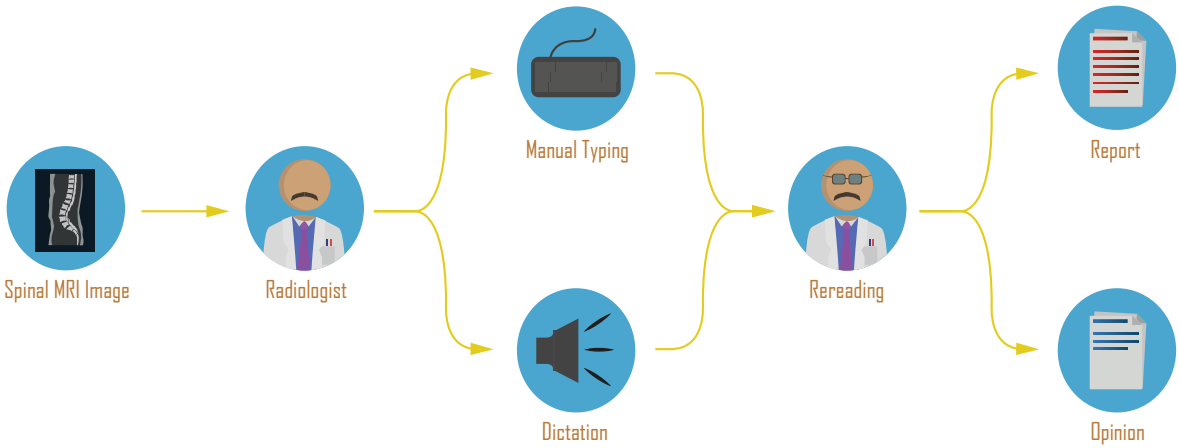

Figure 5.1: The workflow of a radiologic examination

report has to be reread and corrected to remedy the faults committed by the recording medium.

Precise machine understanding could contribute to this process considerably. In real-time, it could automatically check the consistency of the report and the opinion, a graphic visualization could provide a quick glance of what the radiologist might have missed, compare the current report with the results of the previous examinations and even warn the radiologist if the meaning of a sentence is unusual. Valuable statistics could be generated, even from archive data, aiding financial administration of hospitals. Further possibilities lie in the standardization of reports that could enable easier subsequent processing and bring the reports from different institutions to a common ground, contributing to patient-oriented health system improvements [131, 130]. While these are potentially beneficial uses, machine understanding of these texts could provide a pathway for automatic report generation, which is an even worthier goal that would open new doors for more precise and objective reports.

Modern deep learning methods can analyze medical images with high precision. The major setback of such an approach is that these methods tend to require immense amounts of data to work correctly. As the creation of such data requires medical expertise, its gathering presents vast time and resource costs. Traditionally, such data is gathered by radiologists manually annotating tens of thousands of MRI images, precisely pointing out the problems and their locations. Reports are similarly composed by radiologists and represent a very similar, if less graphic source of information. The expert opinion is already contained within. Thus its proper extraction can contribute valid training data even for such an image-based automatic diagnostic tool, saving resources. The current chapter deals with the extraction of such useful information from the text of the reports.

The chapter presents our efforts to extract various entities and their connections from Hungarian radiologic reports. This process involves the classification of various anatomical locations, disorders, and properties via machine learning trained on 487 manually annotated reports. The connections are determined based on language models, and the data is visualized in an easy-to-comprehend manner. The chapter also describes a method that is used for the spelling correction of the reports and shows how it contributes to the overall end results. Section 5.2 presents some of the relevant related work, while Section 5.3 describes our methods briefly. Section 5.4 displays some of our results and also provides discussion. Section 5.5 concludes the chapter.

5.2 Related Work

Words in sentences follow distinct patterns. Sequence tagging is a task where the class type of an individual element can depend on the class type of neighboring elements. BiLSTM-CRF models are widely used architectures in these types of tasks [95]. In the medical domain, BiLSTM-CRF and its derivative architectures are very popular in drug name recognition (DNR), clinical concept extraction (CCE) and adverse drug event recognition (ADER) tasks [92, 56, 24, 170]. In recent years, these networks have been widely used and further improved for named entity recognition tasks from Chinese medical reports [58, 57]. Yin et al. used features extracted by convolutional neural networks to enrich the semantic information of the characters and applied a self-attention mechanism to capture the dependencies between characters [166]. Li at al. implemented attention mechanism into their BiLSTM-CRF architecture, which enabled their model to capture more useful context information and alleviated the problem of missing information caused by long distances between related elements in the sequence [87]. Cai et al. suggested that named entity recognition on Chinese medical reports can be improved by making entity boundary detection more accurate [22]. They proposed the utilization of part-of-speech (POS) tags using a BiLSTM-CRF architecture with a self-matching attention layer. Zhao et al. used a lattice LSTM-CRF system with adversarial training [171].

BERT [36] (Bidirectional Encoder Representations from Transformers) is a language representation model presented by Google in 2018, which performs various pre-training tasks on unlabeled data. The model takes into account the words that occur before and after the tokens when representing them. The model can be fine-tuned by adding a single output layer, thus achieving state-of-the-art results on several natural language processing tasks. BERT was shown to outperform previous models in several tasks [145], even from the medical field. Bressem et al. [21] compared BERT-based solutions on 3.8 million radiologic reports of the chest region. Syed et al. [144] used BERT for classification in a limited domain, in combination with part-of-speech tags and character embeddings. Soares et al. [140] used BERT for learning relations between entities, which is also a possible future goal for our solutions. Cohan et al. [28] used a BERT-based model for sequential sentence classification to promote document-wide understanding. huBERT [112] is the first publicly available BERT model in Hungarian. huBERT is able to achieve better results on several language processing tasks than the multilingual BERT model when dealing with the Hungarian language.

In an overview study concerning textual error correction [83], Kukich classified the breachable difficulties into tree categories, nonword error detection, isolated word error correction, and context-dependent error correction. Nonword detection can be achieved by n-gram analysis looking for unusual character patterns or by a dictionary-based method that looks for each occurring word in an extensive dictionary. The majority of the isolated word correction algorithms consider edit distance as the primary indicator of correctness. According to a study [33], 80% of misspellings are covered with these four common mistakes: a letter added to a word, a letter missing from the word, a single wrong letter or two adjacent letters of the word becoming transposed. These all contain a single mistake that usually results in a nonword. In the case of context-dependent errors, a word is exchanged with another, that does not fit the context. The filtering of these mistakes is usually achieved by statistic language models. Methods aiming to detect misspellings are not new in scientific literature even in the medical field,

covering vaccine safety reports [147], queries on medical portals [29], mammographic reports [107] and other various medical artifacts [85] of specific fields. Medical nonword detection and correction methods tend to use specific dictionaries and often rely on bigram [107], trigram [121], or n-gram [164] models, they also tend to use edit distance as a way to achieve possible corrections. There have also been other approaches like machine translation [114] [115] or noisy channel methods [63]. Significant advancements have also been made already in non-English report corrections like French [134] and Persian [164] reports, as well as some efforts for the correction of Hungarian language reports [137] [138] [139] [72].

An accurate and automatic NER model on clinical texts opens the way to many possibilities. Such exciting applications could be automatic opinion generation, smart statistics generation or visual summarization of the medical records. Our goal is to develop a system that can automatically understand free-text radiologic reports and visualize them in real-time. In our work, BiLSTM-CRF and BERT-based classification methods were used to extract medical named entities effectively, while other components of our system grouped the corresponding entities. A framework for a visualization system was also constructed to visualize entities and their relations in a tree-like structure. This kind of machine learning based, real-time visualization systems are relatively scarce in the literature [133] and to the best of our knowledge they are entirely non-existent for the Hungarian language. An earlier version of our training solution was published previously in Hungarian [70] as well as various supplementing solutions [69, 72, 73] that contributed to the progress of the process.

5.3 Methods

The current section describes the methods used in our experiments. 487 anonymized reports have been manually annotated by a radiologist according to our classification system, this serves as our training data. Our artificial intelligence methods rely on linguistic analysis conducted via the Magyarlánc [174] tool, which provides syntactic, morphologic, constituent, and dependency analysis for general Hungarian texts with high accuracy. Let us overview the steps in our process these in the current section.

5.3.1 Annotation

Our annotation system incorporates a few simple entities that need to be classified. These are the anatomical locations, disorders, and properties. These three classes tend to cover most of the meaningful words found in typical radiologic reports. An example of our system, converted to English for better understanding, can be seen in Fig. 5.2. Note that the reports are at no point translated to English in our process. An entity can consist of multiple words. A term was considered an anatomical location if it describes a specific part of the human body such as "L2" or "disc", or even as a part of a disorder itself like "disci" in "hernia disci". These entities are relatively typical and have a smaller vocabulary than the others. Disorders are the various pathologies observed by the radiologist like "hernia" or "dehydration". Positive or neutral statements also belong here such as "intact" or "status idem". The aspects under observation like liquid content or height are also considered parts of our system's disorders, as these specify the disorder. Disorders can be easily confused with properties such as in the case of "deforming" in our example. Properties are usually describing the stage or degree of a

disorder, or in some cases, specify its precise location. Some examples include "3 mm", "right", and "significantly". Properties clearly have the largest vocabulary since they are much less reliant on the medical terminology than the other elements.

The annotation itself was conducted in the Brat [142] annotation tool by a radiologist and covered 487 Hungarian reports at the current phase. Brat is a tool that facilitates annotation by an easy-to-use web-based user interface, it is highly configurable, and annotation results can be downloaded in a relatively simple, tab-separated format. Note that it is a platform for the annotation, all the annotations were conducted manually. Our annotation system is the result of several meetings between the radiologist, linguists and computer scientists. A thorough set of guidelines was available for the radiologist during the annotation. Since the initial annotation, we have made several attempts to increase the quality of the data. A second radiologist annotator has also performed the annotation, these results were consolidated, and the rules were specified further according to the new interesting cases. A helper solution was also developed [69] to help in improving the consistency of the data during a later inspection by the initial radiologist.

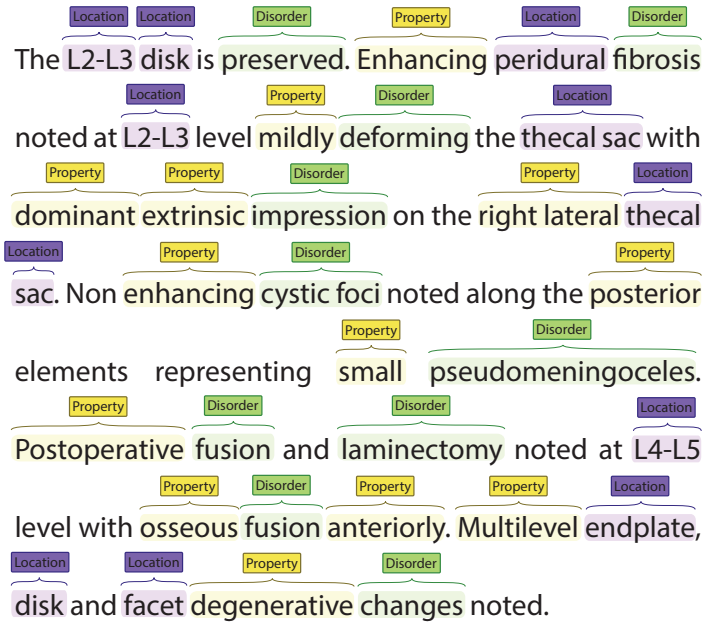

Figure 5.2: An English language illustration of our annotation system

The most basic information of our current annotated dataset can be observed in Table 5.1. Begin tags were used to distinguish the tokens at the start of entities, while Inside tokens note every subsequent tokens that are still parts of the entity. As it is visible, only roughly about every fourth token was found to be an inside token. Note that our later steps in the process still merge some of these entities, as for example the text "L.V disc" was annotated as two separate entities according to our annotation system.

5.3.2 Classification

Our classification model is essentially a named entity recognition (NER) model. In its first iterations it was based on a BiLSTM-CRF (bi-directional long short-term memory [54], conditional random fields [84]) architecture similar to the one published

Table 5.1: Annotation statistics in our current version of the 487 report annotations

	Locations	**Disorders**	**Properties**
Begin-	9,047	7,944	3,487
Inside-	3,952	1,954	1,321
Total	12,999	9,898	4,808

by Ma et al. [95]. Since Hungarian is a morphologically rich language, character level embedding was also utilized in our solution, as suggested by Ling et al. [89]. Apart from word and character embeddings, additional predictive features such as lemmas, part-of-speech (POS) tags and part-of-sentence tags were also utilized in the model. The first layers of our NER model were embedding layers, mostly initialized with random weights. For the textual inputs, the corresponding embedding matrix is initialized with pre-trained word vectors (trained on the Hungarian Wikipedia). In a regular forward-pass, the integer encoded feature sequences were first passed through their corresponding embedding layers where each feature (word, lemma, POS tag, part-of-sentence tag) was mapped to a dense vector representation. The character level vector representations of words were generated by an additional BiLSTM network. The main BiLSTM layer took the concatenated vector representations of all the features (word, lemma, POS tag, part-of-sentence tag, character) as input. The BiLSTM layer's output was passed through a densely-connected feed-forward layer, on top of which a CRF layer performed the final sequence tagging. See [74] for further details. This model is still referenced in the current for the current thesis, as much of our data was obtained using this version of the classification.

In our newer experiments, BERT [36] was found to perform significantly better than our previous model. We used the BertForTokenClassification model implementation to obtain our classification. During this, the weights of a pre-trained BERT model are loaded, which in our case was the huBERT [112] model, a pretrained model for the Hungarian language specifically. Even though the reports tend to use a lot of Latin language and the jargon is highly medical, which is not in the scope of huBERT, the Hungarian model still performed better in our experiments. The model was trained on the data of 487 annotated reports mentioned previously, also used by the BiLSTM-CRF model. In our experiments, a linear layer was added to the hidden output (hidden state output) of the last layer of BERT, which is responsible for assigning the tokens to the classes. The current classification process is visualized in Figure 5.3.

Our classification also utilizes a list of regular expressions, which are used to override some more typical cases of classification. This is added to the output of the machine learning model. Vertebra levels ("L.V"), for example, should always be classified as anatomical locations, and they can be reliably found by regular expressions.

5.3.3 Automatic Correction of the Text

Regardless of the best intentions of the medical experts, some faults still remain in the reports. A text riddled with minor errors is bound to create a problem or at least a nuisance for anyone aiming for automatic understanding. These mistakes can impair the training phase of machine learning, resulting in faulty models and making

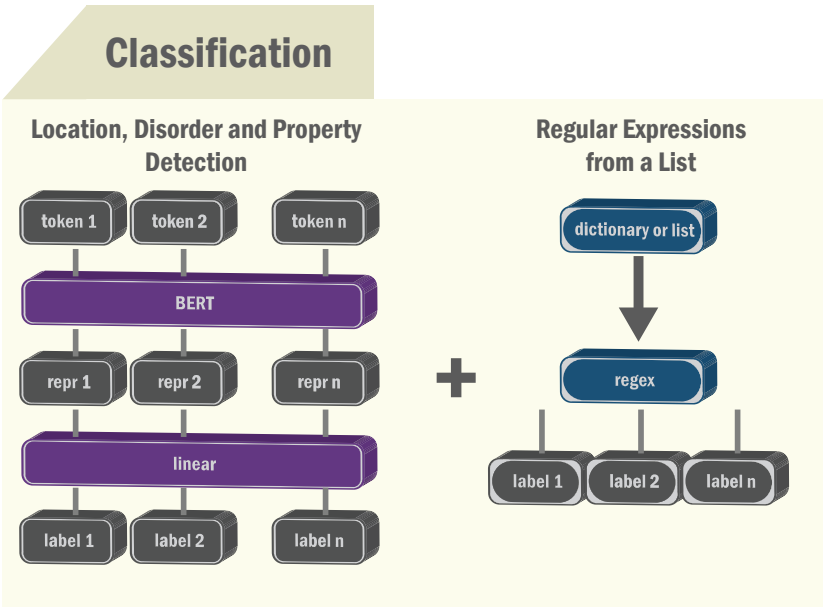

Figure 5.3: An illustration of our classification of locations, disorders and properties

the proper identification of words or their linguistic analysis a futile task. Manual correction of these errors in finished reports is troublesome, and it is not feasible for a real live system. During our experiments on finalized real reports, a manual analysis of the text of 487 reports with combined length of almost 48 thousand words revealed that such mistakes are present in the majority of cases. Of the 487 reports, 295 (60.57% of the reports) included at least one error. These, of course, can be of various natures. A display of the results of our analysis can be seen in Figure 5.4. Mistakes concerning punctuation can impede the work of tokenizers and constituent analyzers. In the case of medical reports, these usually take the form of missing or unnecessary commas and full stops. Words that are correct by dictionary but appear out of the semantic context of their sentence or without the matching conjugation or suffixes are also typical, these are referenced as "context" in the figure. But as it is visible, the great majority of these errors are misspellings or wrong accentuation marks that result in unrecognizable words. These can be found in 166 (34%) of the 487 reports analyzed. From a certain viewpoint, this particular sort of error is less unfortunate as these mistakes stand out properly and thus can be detected with a degree certainty. We focus on the correction of these cases. These cases are often called nonwords [83] and also have typical characteristics [33]. The high number of misspells in radiologic reports is also already established internationally [168].

Morphologically rich languages, such as Hungarian, present new obstacles when aiming to correct mistakes automatically. Let us consider a short example of a sentence from the field of spinal MRI reports shown in Figure 5.5. The first line presents an English sentence, the second line is its Hungarian translation with identical meaning. Suffixes are much more intricate in the Hungarian language, their occurrence and variability are extremely high compared to the English conventions and they change the base of the word much more often. Latin words occur regularly in both Hungarian and English medical texts, and these also get the appropriate suffixes dictated by the rules of the language of the report. These are highlighted with red in the example, while non-Latin word suffixes are highlighted with green. These hybrid words present significant

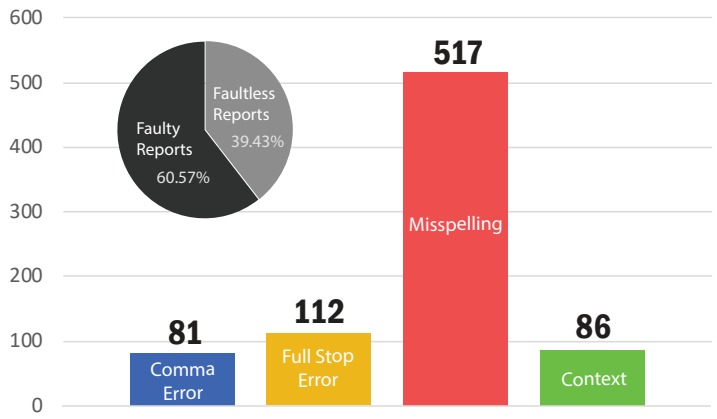

Figure 5.4: A results of the manual correction of the 487 reports.

obstacles for a spellchecking method as they cannot be found in any Hungarian or Latin dictionary, they are basically new words that could be correct according to the rules of the language and sometimes even come from some inner conventions of the medical field like in the case of "sacroliliacalis", which does not conform completely to any known Hungarian suffix. Even Hungarian lemmatizers that are extremely reliable under normal circumstances can not cope well with medical texts. The third line of the figure shows the result of a lemmatization with the Magyarlánc[174] linguistic analysis tool. As it is visible, it does not know the Latin words and leaves their suffixes untouched. Some not especially uncommon words even produce bad lemmas like the shortening of the word "myelon" to "myel". While it is visible that English suffixes for Latin words are also present in medical context, these are relatively easily handled as there are just a few possible combinations. In Hungarian, tens of words could be constructed for each Latin word that can be seen as having an appropriate suffix in its context.

There are sclerotic changes with anterior effusion of the sacroiliac joints bilaterally.

A sacroliliacalis ízületekben bilaterálisan scleroticus elváltozások láthatók anterior ömlennyel.

lemmas A sacroliliacalis ízület bilaterálisan scleroticus elváltozás látható anterior ömleny.

Figure 5.5: An medical text example with highlighed suffixes

Our solution was implemented with the Hunspell [1] spellchecking software which is also used by many popular information technology systems, such as Firefox, Chrome, Libre Office, Photoshop and Eclipse. It is also widely applied in the academic environment, it has been adapted for a great variety of languages like English [16] [3], Arabic [169] or Esperanto [17]. The system works with a built-in dictionary with conjugations and other rules. It finds misspelled words in general texts with high accuracy. Analysing radiologic reports with Hunspell, however, marked an exceptionally large number of errors due to the reports containing a lot of Latin words and medical nomenclature. These are, not surprisingly, not part of the original Hunspell dictionary. Naturally, this does not qualify the system itself, these faulty alerts stem from the non-conventional, and in many cases, not even really proper use of the language, which is nonetheless frequently used and well-understood by radiologists. A human reader's

[1]Hunspell's repository: http://hunspell.github.io

perception is not affected by these words as doctors use these rather consistently and regularly.

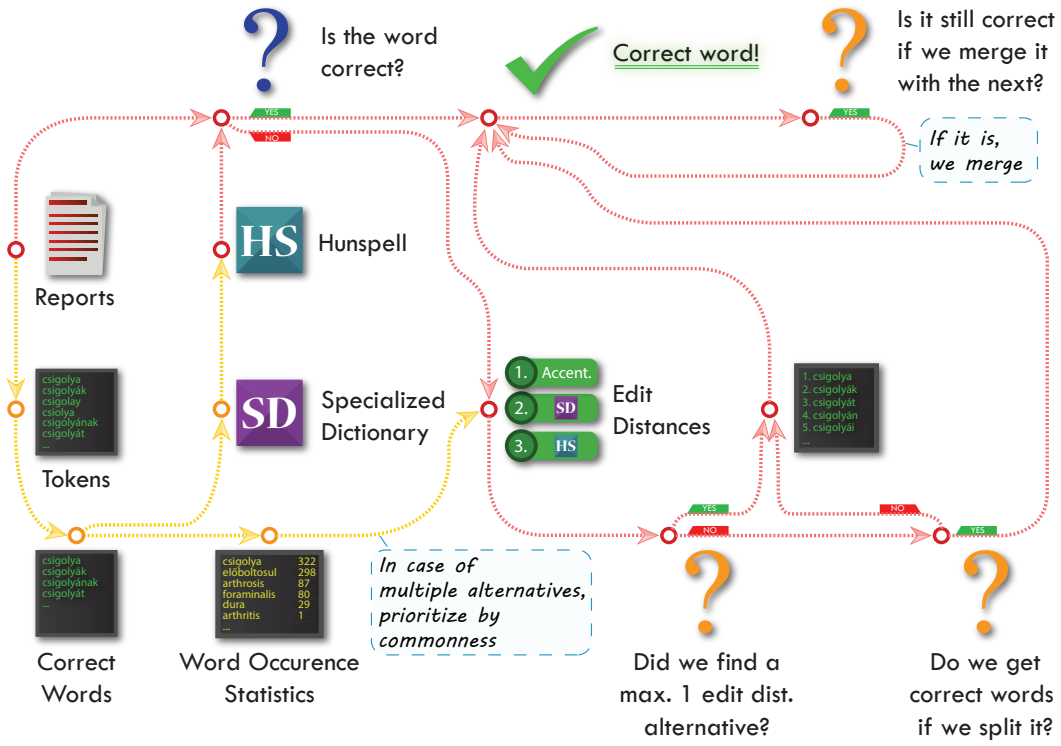

Figure 5.6: The proposed automatic method for detection and correction of misspellings

Thus, additional dictionaries and prioritization rules are needed to correctly handle the text of medical reports besides the Hunspell system. A complete overview of our method is shown in Figure 5.6. Besides the 487 reports mentioned earlier, additional reports were also manually spellchecked to improve the evaluations. Thus, a total number of 882 reports have been used for optimization purposes. Let us consider the yellow lines of the figure first. In its original form, Hunspell suffers heavy difficulties in coping with Latin words and medical terminology. From 5649 reports on our disposal, all the tokens were selected which were classified as an anatomical location, disorder or property by a BiLSTM-CRF classifier (that can also label unknown words based on the context) introduced in the previous subsection. The tokens of the entities recognized by the BiLSTM-CRF system were manually analyzed one by one in lexicographic order, the incorrect tokens were removed. The BiLSTM-CRF classification is only needed here to filter out some of the generic words that are not specialized to the field. If no such reliable classifier would be present, another viable solution could have been the listing of all tokens without the ones already present in Hunspell's built-in dictionary. The suspicious but not obviously faulty tokens were collected in a temporary set. These dilemmas were solved by a radiologist's manual judgment. The resulting specialized dictionary currently contains 6,150 tokens of which 914 can also be found in Hunspell's original dictionary. This dictionary is not only used as a supplement of Hunspell's own dictionary, but also in prioritizing the corrections. This is achieved by ordering the list of the words of the dictionary by the frequency of token occurrence in the considered reports. The usage of this list is showcased later. The specialized dictionary is given to Hunspell, which uses the data to judge the correctness of a word. If the token is correct,

there is nothing much to do. In this case, an attempt is made to merge the word with its consecutive to counter redundant whitespaces between words. For example, the spelling of the correct word "csigolyatest" (vertebral body) can be split into "csigolya" (vertebra) and "test" (body) which are both acceptable, but the radiologist probably meant to write them together. Thus, for each correct or corrected word, it is checked once whether it can still form a meaningful word with its neighbor, and merged with it if deemed necessary.

If the token is incorrect, that is to say, it can be found in neither Hunspell's dictionary or the specialized medical dictionary, an attempt is made for its correction. By default, Hunspell provides an edit distance based prioritized list of the possible fixes. This list has been less than suitable in numerous cases, mainly due to the specific medical terminology. The prioritization has been redesigned to consider accent defects first, for example, the word "előboltosulás" has higher precedence than "elboltosulás" if the original word was misspelled as "eloboltosulás". If no such alternative is found, the correction is based on edit distance. Terms in the specialized dictionary receive higher precedence than the ones in Hunspell's built-in general dictionary. If the edit distance does not indicate a single best correction, the token frequency list is considered, which was constructed earlier. This case is relevant for situations where there are multiple meaningful choices, that could otherwise only be solved by manual consideration. For instance, the word "arthrotis" can be corrected to both "arthritis" and "arthrosis". In this case, the more common "arthrosis" is preferred, which is much more viable in a spinal report.

After producing a viable correction, it is checked whether it is exactly one edit distance apart from the original word. These are the most common misspellings as only one letter was missed or mixed up. In the cases where the algorithm finds at least one such word, it goes straight to the prioritization list, and from these, the most likely one will be the recommended automatic fix. If such a correction is not possible, then instead of suggesting something very different, the algorithm first tries to split the word into two, or in case it was still unsuccessful, into three parts. This way, "előboltosulódiscus" can be divided into "előboltosuló" and "discus" which are two correct words. If after such a split all the resulting words are meaningful, this is automatically considered the right correction. In the cases where a valid split cannot be found, the prioritized list based on the edit distances is considered again, as constructed earlier with Hunspell. In almost every case, a prioritized list is produced as the output of our whole process, suitable even to use as a recommendation system.

5.3.4 Negations

The detection of negations is based on the use of the information provided by the Magyarlánc analyzer and custom rulesets. It relies on the constituent analysis feature, as the clauses of a sentence can be extracted from this information. A clause of a sentence usually deal with naming a single disorder, or in most cases of negation, the absence of it. With some exceptions, our negation-handling module deems a disorder to be negated if a negation (such as no, nor, neither in English) is found to be in the same clause.

5.3.5 Identification

To have a good grasp on the meaning of a report, it is not enough to just label the tokens. The basis of the identification is our primitive ontology, which is written by our sets of identifiers. Size information of these sets is displayed in Table 5.2. Of course, many other names can be associated with an entity, and we can also process them through the handling of synonyms, these are not calculated into the data of the table.

Table 5.2: Size of our various identifier sets

ID scope summary	Specific ID types	Number of identifiers
Locations		87 (about 786 total)
	Vertebra levels	29
	Vertebra suffixes	26
	Not assigned to verterbrae	32
Disorders		274
	Pathologies	233
	Normal conditions	16
	Aspects	25

Most of the anatomical locations (discs, for example) also specify the vertebra levels as they can be encountered at all or almost all levels. This implies a hierarchy which also appears in our identifiers. The vertebra level of the anatomical location is the base of the identifier, while the more precise notation is separated by an underline character. For example, the disc at the fifth lumbar vertebra's level gets the L05_D identifier. As the table shows, 29 vertebra levels and 26 suffixes are present in the ontology at the current time. This means that these identifiers can represent 754 different anatomic locations. Note that not every single one of these is valid as, for example, there is no disc at the L.I. level, but this information is not built into our method.

Disorders do not conform to such easily extractable hierarchies. By our notation, three separate groups exist. Firstly, there are disorders which indeed name harmful conditions, referred to as pathologies in the table. These are the most simple and numerous. While synonyms can cause complications, their handling is quite straightforward. Their identifiers start with the E_ prefix. The P_ prefix notes those disorders that refer to normal conditions, such as the discus being intact or the height of the vertebra being normal. The third notable prefix, ASP_, is used to distinguish disorders that only provide meaning when accompanied by either an E_ or a P_ disorder. These are various aspects of the anatomical locations, for example, their height or water content. While these aspects are not disorders by themselves, they are crucial in the understanding of their accompanying disorder.

5.3.6 Connections

The automatic extraction of connections utilizes the constituent parser of the linguistic analyzer and our classification's output. Almost all of the properties can be attributed to a disorder rather than an anatomical location. Many cases like in "compressed L5 disc", where this presumption seems faulty, are the result of flawed classification (as

"compressed" should be a disorder itself here). Thus, only two kinds of connections are determined among different classes, one between disorders and locations and the other between properties and disorders. Our system also attempts to merge the locations and the disorders that belong together (like in the case of "L2-L3 level mildly deforming the thecal sac" where the proper location would be "L2-L3 thecal sac", these can also be viewed as connections.

Our method uses several predefined rules to cope with the task of these automatic assignments. Our method relies heavily on sentence parsing and the clauses of the sentences as the entities that belong to the same clause are extremely likely to be connected semantically. The following rules were constructed:

- Disorder-Location: The system first considers only the clause of a location. The preceding disorders are prioritized first; if more than one exists, they always receive the same treatment. If none exists, the system looks for rightmost ones. If no such disorder is found, the system broadens the search to the whole sentence, but only considers the words on the left of the location. Coordinations are also considered, the locations that are coordinated with "és" ("and"), "vagy" ("or"), or a comma always receive the same connections.

- Property-Disorder: If a disorder is proceeded by any properties inside its clause, they gain connections. Otherwise, the whole left side of the sentence is considered.

- Location-Location: Some of the locations like "disc" or "endplate" are not entirely specific, they need a vertebra or at least a region to achieve precision. A set of such locations was assembled manually, in their cases, a suitable vertebra or vertebrae are attempted to be found inside their sentence.

- Disorder-Disorder: Similarly to the previous problem, aspects like "height" do not convey meaningful disorders but lend specificity to others. Since they are also very typically worded with little variety, in the same manner, a list of these was gathered, and potentially suitable other disorders are found inside their sentence.

Some of the semantic connections inside the text are referenced implicitly. For example, the sentence might not repeat the vertebra level that was mentioned in the previous sentence. Various typical terms can allude to this, such as "at the same level". These terms can have a variety of meanings, some directed to the previous sentence or clause, some to the next sentence. These terms are detected, and their directions are determined according to a custom ruleset constructed along the examples found in real reports. These are factoring into how our heuristic approach determines connections, and factor into our later evaluation of connections.

We note that the Hungarian sentence structure differs significantly from that in English. Furthermore, the sentences of radiologic reports tend to have relatively simple structures. Thus, such rules can be correct in the overwhelming majority of the cases.

The detected entities are then displayed in a structured manner, their connections highlighted by grouping via frames. The result always forms a tree structure. An illustration of our visualization system adapted to English can be seen in Fig. 5.7. This is a visualization of the report seen previously in Fig. 5.2. Note that this example was explicitly constructed as an illustration. Our system is not suitable for processing English reports which vary significantly from Hungarian reports in both terminology and sentence structure. The detected and identified anatomical locations could also be

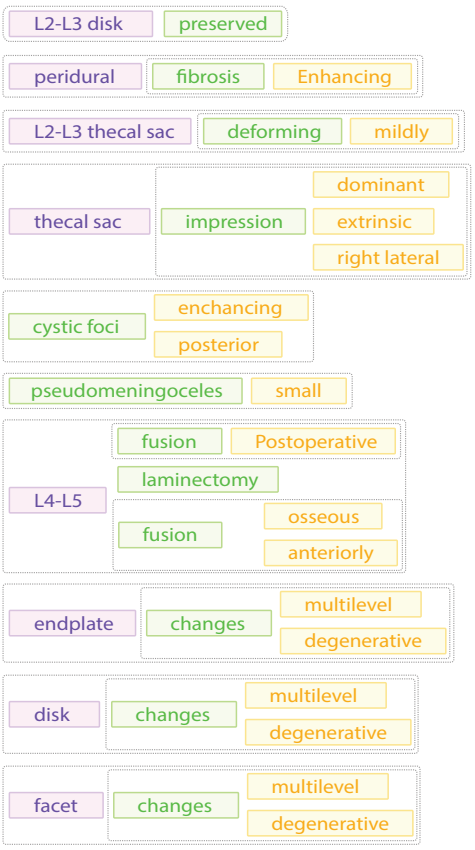

Figure 5.7: An illustration of our structured visualization of the text seen in Fig. 5.2

linked to regions of the human spine. At the current time, this is performed with a rule-set assigning anatomical locations to regions of a rather simple schematic illustration of the human spine.

Our current method is displayed in Fig. 5.8 with a Hungarian example. The 487 reports were manually annotated, and were used for the fine-tuning of the BERT identification model. When the system receives a new report, its text and linguistic features are given to the model for prediction. The model performs the classification of the locations, disorders and properties. The resulting entities are then submitted to our connection extraction, which use sentence and constituent parsing of Magyarlánc and predefined rules to determine the probable connections. This results in a tree visualization where the different entities are color-coded and displayed in a visual format in which the connected entities are grouped together. The process is performed automatically and runs extremely quickly, making it suitable for real-time display of reports during typing. Misspell correction is also present at the input channels. Negations are not featured in the example.

5.4 Results and discussion

5.4.1 Classification and Connections

Our classification is currently performed using BERT, but our earlier method that most of our work is based on was established with BiLSTM-CRF. We compared the two models, the results of which are shown in Table 5.3. The BERT model using

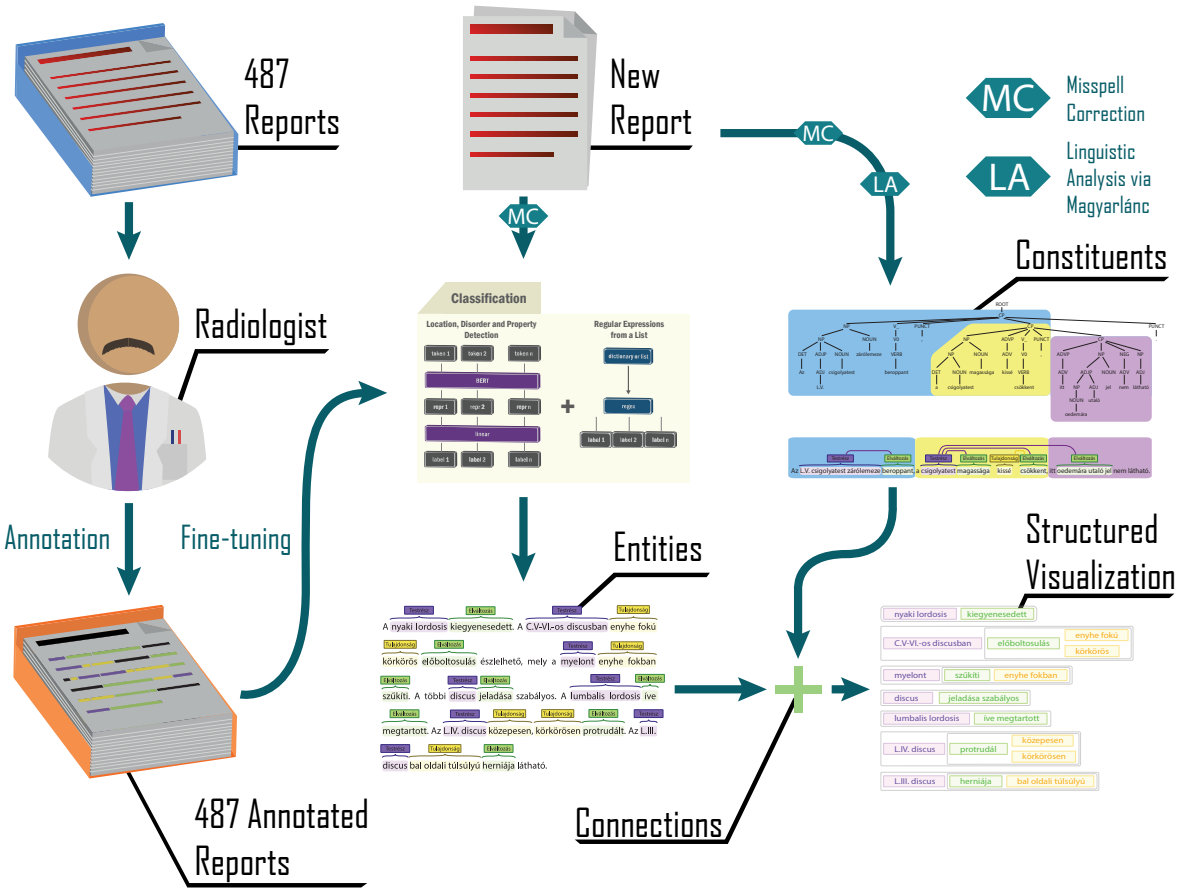

Figure 5.8: Our proposed method for automatic understanding and visualization

only word representations, trained on the same training data, was able to achieve almost 2% better accuracy and F-score results than the BiLSTM-CRF model. During the evaluation, 70% of the data of the 487 reports was used for training (or fine-tuning), 10% used as a validation set, and 20% as the test set. The comparison does not feature the previously mentioned overrides that were performed with regular expressions. The results, however, already feature the corrections that are discussed in the next subsection.

Table 5.3: A comparison of our previous BiLSTM-CRF model and our new BERT-based entity classification

	Accuracy	**F-score**
BertForTokenClassification	96.98	96.85
BiLSTM-CRF	95.10	95.09

It can be noted that locations tend to offer the best results while properties seem to lag behind in both cases. This can be partly due to the visible difference in sample size, but an even more likely cause is the size of the classes' vocabulary. Properties can take up a wide variety of forms while locations use a fairly limited set of terms.

In the 487 reports, the model detected 7,794 disorders, 6,358 locations, and 3,442 properties. Our system assigned 11,016 connections, of which 6,924 were Disorder-

Location, 3,382 were Property-Disorder, 425 mergings of locations, and 285 mergings of disorders. During the manual evaluation the connections were found to be precise. Some mistakes were detected where the classification itself was faulty or in cases of rare, exceptionally complex sentences.

5.4.2 Spelling Correction

The current evaluation features the BiLSTM-CRF model. Linguistic features were also utilized, extracted by the Magyarlánc [174] analysis tool. Upon detection, the entities are identified and their semantic connections are also revealed using our rule-based method that also works with Magyarlánc data.

Summing up our detection results with a micro-average, the model's F1-score increased by 0.35%. The results were obtained through tenfold cross-validation measured three times with different random seeds, the numbers represent the average of these. The corrected data performed better in each of the three runs. In the case of the identification, the quality indicator can be the number of anatomical locations and disorders that our process could not identify at all. In the original data of 487 reports, 505 of 6,358 anatomical locations and 521 of 7,794 disorders were left unidentified. After corrections, these numbers decreased to 488 anatomical locations and 332 disorders. Thus, it is visible that the automatic misspell correction had a big impact on the success of the identification task. No negative effect was observed here. Thus, according to our results, machine learning classification methods are likely to be positively affected (although to a little extent in our case) by the automatic misspell correction, while rule-based methods are likely to benefit even more (more than 20% less unidentified entities in our system).

Comparing the results to the manual dataset, the dataset contained 36 corrections that were not marked by our system, most of these were missing dots or dashes which on the token level cannot be detected. During the inspection of the results, 5 cases were found that were real misspellings and were not detected by the system. In 17 cases, the same word was corrected differently by the manual correction and our system. Of these, there were 7 inconsistencies that we found to be genuinely faulty automatic corrections. The correction method detected 328 errors that the manual check did not. Our analysis found that a total of 4 corrections of the detected elements were unjustified, which in themselves are unusual but probably intentionally spelled the way they appeared ("VA", "VB", "radici", "gerinccsatorna- és"). Further 3 faulty words were correctly detected but were given bad corrections. Thus, of the 328 detected errors, 324 can also be considered a valid fault detection result by human observers. The original human annotator also agreed with these results.

As a motivating example, a well-known text editor's Hungarian spellchecker was tried out on the same example displayed in Figure 5.9, which detected all misspelled words in the text but was able to correct only two of them, and also marked six other words as faulty which are very common in the radiologic terminology. This or very similar text editors are widely used during clinical reporting. The bottom part of the figure displays an easily comparable summary of the corrections the tools made.

Thus, we determided that the automatic correction method can surpass the human observer in the detection of valid errors (more than 30% higher number of real errors in our case). While the fault detection and correction results are significantly worse in case of an unknown text, extensions to the dictionaries can produce corrections that

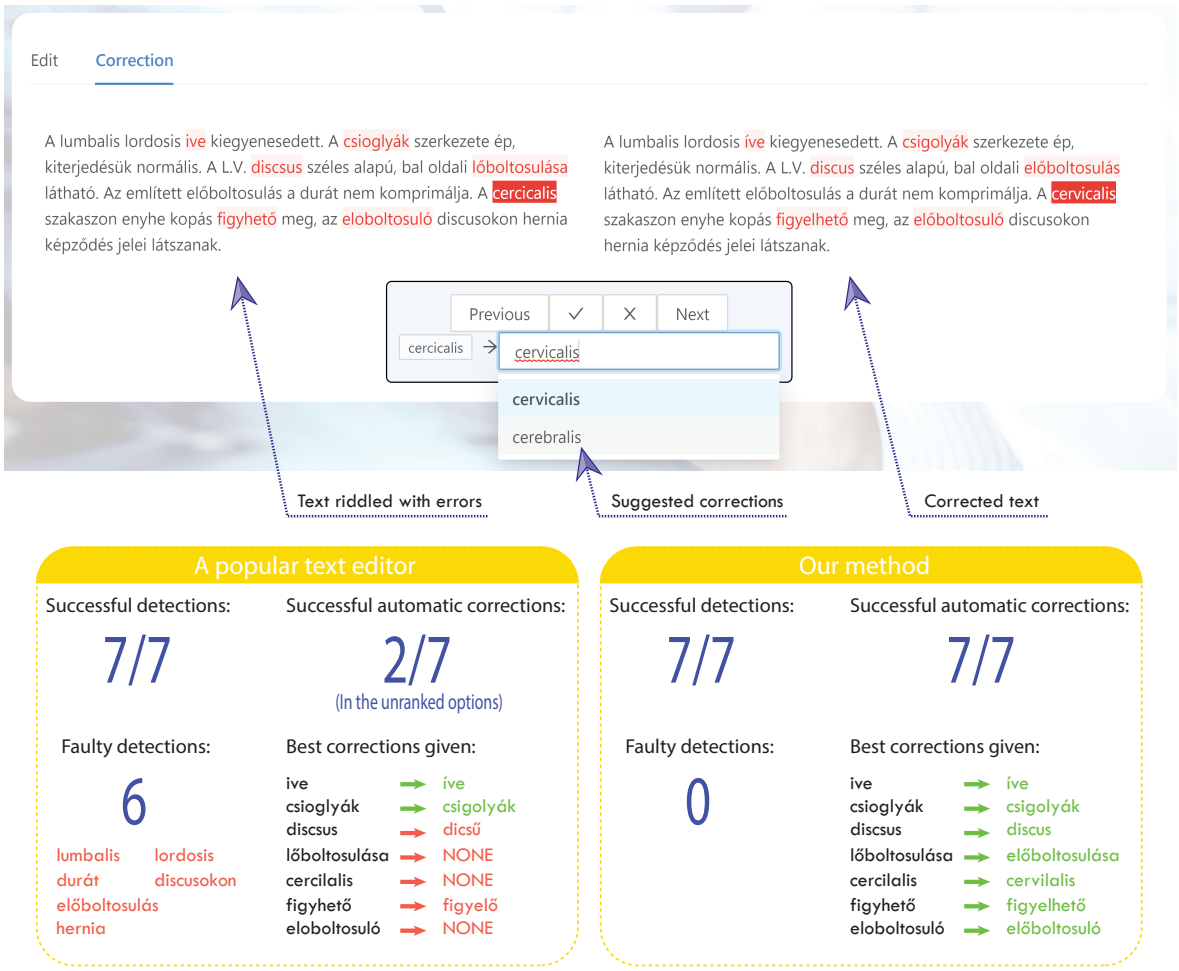

Figure 5.9: Our automatic spelling correction tool for radiologic reports with evaluation of the example text compared to a traditionally used text editor

are overwhelmingly correct (more than 97% valid in our case) without the need of additional adjustments.

It is important to note that our system has been optimized only for the correction of Hungarian radiologic spinal reports. It cannot be declared that in other domains similar results could be guaranteed. Our present system works with medical terminology of a rather small vocabulary. Similar results might easily be produced on other restricted medical fields, adding new dictionaries to the same process.

5.4.3 Functional Evaluation

Since most of the process is hard to measure objectively, we present a complex functional evaluation of how accurate radiologists found our system. This distinguishes three layers of the machine understanding workflow for better differentiation of various errors. Each category had a set of rules for determining the number of possible points as well as the achieved result. Since the evaluation also concerns the integration of these layers, and each category builds upon the results of the previous one, the so inherited errors were handled as correct inputs in the next layer. Thus, the evaluation investigates how the next layer handles them according to the rules, regardless of their correctness. This decision was necessary because this provides a better picture of the

performance of layers in themselves rather than every score being dependent on each of its predecessors. We note that this does not usually improve but also can worsen the scores, as, for example, "L.V." could not possibly get a good disorder identifier and stays without an identifier in the cases it was classified as such. The three categories are as follows:

Classification - The distinction between the various entities marked with different text colors (corresponding to their classification, similarly to the example of Figure 5.2) in the text of the report. The maximum and the obtained score is determined on a conceptual level rather than on a token level. Thus "L.V. discus" can receive two points, one for the proper purple color of "L.V." and one for "discus", even though simple text colouring does not grant such an obvious separation.

Connection - The results of the classification contain the entities that are connected on a conceptual level, such as "L.V.", "discus", or "hernia". At this point, the tree structure is constructed, in which the connected entities are assigned to each other. Thus from the simple sentence segment, "L.V. discus hernia látható.", the connected "L.V." and "discus" location entities should be assigned together as "L.V. discus" in a purple node, and the disorder "hernia" should be in a separate, green node of the tree, assigned to this node. The maximum score is determined by how many nodes the annotator deemed to be necessary. The actual score is the number of nodes that are both correct in their content, and in case of disorders and properties, were assigned to the correct location or disorder, respectively.

Identification - The location and disorder nodes contain entities that should be assigned at least one identifier from our ontology. Since this is a textual matching process, mistakes are usually due to faults in the ontology itself or to faults in our matching rules. Negations also factor in this layer. The maximum score is determined as the number of location and disorder nodes. The actual score is determined as the number of location and disorder nodes that got their proper identifiers, as well as their proper negation state. Thus "L.V. discus" should receive the identifier for the L.V. disc, while "L.II.-L.IV." should receive the identifiers for L.II., L.III., L.IV., and L.V. Also, if "hernia" was properly identified as herniation but got no negation when it should have, the node does not receive the score.

The evaluation was conducted by three separate radiologists (R1, R2, and R3) on the sane 20 reports, according to detailed evaluation guidelines. We note that at this point, none of the three evaluating radiologists took part in the annotation or the construction of the system or its learning data. We also note that this evaluation used the BERT-based implementation of classification. The results are visible in Table 5.4. The average and overall results of the table represent the average of all the scores the radiologists determined as maximum and as achieved scores during evaluation. Thus, for example, the connection category has a smaller impact on the overall results as there were substantially fewer points to earn with them in total than the other categories. The maximum achievable scores determined by R1, R2 and R3 were 2,021, 2,040, and 1,939, respectively, while the achieved scores were 1,972, 1,994, and 1,861, respectively. As also visible from the scores, the radiologists deemed our process good in visualizing Hungarian spinal MRI reports.

Table 5.4: The results of the understanding scores according the three radiologists

Task	R1	R2	R3	Average
Classification	97.61%	97.23%	98.47%	97.75%
Connection	96.22%	98.29%	91.37%	95.32%
Identification	98.48%	97.92%	96.75%	97.72%
Overall	97.57%	97.74%	95.97%	97.12%

5.5 Conclusions

The chapter described our efforts in creating an automatic framework for the machine understanding of Hungarian spinal region reports. Our initial goals were the classification of anatomical locations, disorders and their properties in the free-form text of the reports. This was achieved via BiLSTM-CRF, and later via BERT, trained on 487 manually annotated reports. The classification produced an F1-score value of 96.85. The detected entities are connected to each other based on sentence and constituent parsing and pre-defined rules specific to the Hungarian reports' sentence structure. Identification was performed through a simple ontology involving hundreds of identifiers for anatomical locations and disorders.

Our automatic misspell correction can facilitate this machine understanding process, contributing to both our classification based on machine learning (improved by 0.35%) and identification based on language models (correct identifiers for more than 20% of the previously unidentifiable phrases). The automatic method detected 38% more misspells than a human annotator, and while the success of the process is highly dependent on the quality and size of the dictionaries used, the process itself performed with great precision.

Our process provides a representation of the meaning of the reports, displaying the mentioned anatomical locations, disorders and properties, highlighting their semantic connections. In a functional evaluation, three radiologists evaluated our machine understanding and visualization process and found that it provided correct results in 95.97% to 97.74% of the cases.

Our future plans concern the refinement of our current data, including more novel solutions in the labelling and connection-determining process and their connection to the original MRI images themselves. While this is still ongoing research, some new applications have already arisen. An automatic understanding of the reports could facilitate the generation of training data for automatic analyzer tools aiming to detect disorders on MRI images. As part of a current project, our solution is used for such a task. Another evident use of our work is the facilitation of patient comprehension. Interactive electronic reports could provide much more information for patients and lead to better healthcare service, and our project also includes this endeavor.

The thesis point provided a method for automatic understanding of spinal MRI reports, including an annotation system, a correction solution for misspellings, a BiLSTM-CRF and BERT-based classification, an identification based on a primitive ontology, and a visualization of the understanding. The results show that the system performs well in understanding and visualizing Hungarian reports.

The author considers the followings as his main contributions:

- The author took a vital part in the creation of the annotation system, contributed to these meetings, and devised plans to deal with exceptions and mistakes in the system.
- The author maintained contact with and provided guidelines to the radiologists working on the annotations during the work.
- The author took part and coordinated the correction of radiologic annotations of the 487 reports, also taking part in the manual resolving of the arising conflicts between annotations of two different radiologists.
- The author devised the groundwork of the rule-based connection-detection system and provided continuous insights during its implementation.
- The author took part in the design of the misspell-correction method, as well as doing manual annotations and classifications of a significant part of the results.
- The author determined the basic rules for handling negations in the radiologic reports.
- The author devised the structure of the final tree-structure display and took part in its creation.
- The author coordinated the implementation of the machine understanding system.
- The author planned the functional evaluation, including the composition of the evaluation guide and providing assistance during the evaluation to the radiologists.
- The author took part in the evaluation work throughout the work.

"Writing means sharing. It's part of the human condition to want to share things – thoughts, ideas, opinions."

— Paulo Coelho

6 Final Conclusions

The thesis discusses three main topics. The first part provides insight into a project conducted with an industrial partner whose 4GL system has undergone software product line adoption. The thesis elaborates on how our work contributed to the project and showcases our solutions on the analysis of the variants of their system. While product line adoption is a well-researched topic, 4GL solutions are still rare, with little literature. Our work contributed with novel solutions on feature extraction based on static analysis and information retrieval and investigated their combinations. This part of the thesis also showcases several new metrics suitable for the analysis of the features of 4GL systems and a community-based method that groups parts of the software according to their mutual connections.

The second part describes the importance of textual methods in test-to-code traceability and how improving them could lead to the improvement of the whole process. The thesis analyzes some of the commonly used techniques, such as naming conventions and LSI, and provides a new alternative in Doc2Vec and several combinations of these, including various source code representations. It has been established that methods for this field work best in combination, and the textual aspect is likely to remain an important part of future solutions.

The third part describes how radiologic reports are made in today's medicinal practice and how their automatic understanding could contribute to better healthcare and future work in the field of machine learning. It establishes our work in the classification of various entities via BiLSTM-CRF and BERT models in the non-structured text of the reports and their connection through linguistic analysis. These elements are connected to a simple ontology as means of proper identification, and a tree-structured representation is constructed of them. Misspellings can hinder this identification, therefore the thesis point also described a solution for their automatic correction.

The future still holds many more interesting directions for these topics. Software reuse is flourishing in the industrial setting and is unlikely to ever lose its importance as it is often more cost-effective to build upon earlier achievements. Even as new solutions for test-to-code traceability tend to produce great results, there is still ample room for improvement. One such improvement can be to upgrade parts of our current methods

as our research has proposed. Our work on radiologic reports is merely the beginning of an effort aiming to provide smarter, more optimised, and less resource-intensive healthcare. The automatic understanding can be extended to different parts of the body or different languages, and the vast amount of knowledge already in the reports can be extracted to enable new methods for more ambitious goals.

Table **??** provides an overview of the author's publications related to each thesis point.

Appendices

A

Summary in English

Humanity's primary channel of information is verbal or written natural language. Throughout history, written words have aided scientific endeavors, contributed to the education of the masses, have made and broken regimes and led to our current society. Even though the information is primarily stored as binary data nowadays, it still has to be transformed to natural language interpretable for human readers. The thesis deals with the extraction of the information within natural language text of various domains. Two of the thesis points are strongly connected to software development, where the source code still has a great amount of natural language with discernable semantic information, while the third thesis point deals with the automatic understanding of radiologic reports. Text is omnipresent in our everyday lives, and its proper automatic processing has immense potential.

I. Feature-Extraction in 4GL Systems

The contributions of this thesis point are detailed in Chapter 3. Software product line adoption over several existing variants can be extremely resource-intensive. The adoption process requires developers and domain experts to work strongly together on the new architecture, as knowledge of the system's features can be invaluable. The features are implemented through parts of the software code. Feature extraction attempts to extract these for each feature for easier modification or merging with the architecture. Working with an industrial partner to aid this extraction on 19 variants of a pharmaceutical logistics system presented a couple of challenges. The software was written in a fourth generation language (4GL), Magic XPA. The code of the variants was assembled through a development environment without actual coding, and the structure of the language is also significantly different from mainstream programming languages that have ready solutions for aiding feature extraction. Magic applications consist of programs and their subtasks calling each other, and also logical and data units. We have introduced several new approaches to the feature extraction topic through our work, aiming to provide more useful information for developers and domain experts alike.

Our feature extraction experiments produced various outputs suitable for different stages of the work. Call graphs through static analysis highlight the structural connections within the software's programs and can produce an output fit for developers. Another method, information retrieval (IR) through latent semantic indexing (LSI), can also highlight program and feature connections by examining the similarities between the parts of the programs written in natural language and the features. This is closer to the view of the domain experts, who typically do not delve deep into the programs but see the conceptual connections. The two extraction methods build on different sources of information. Their outputs could be combined to produce a more strict filtering of programs that contain the most essential programs for each feature.

Domain experts could further benefit from overseeing the connections between features. The call graph contains most of these connections and, while it is hard to comprehend, provides valid structural information. The call graph could benefit from methods commonly used in graph theory, namely community detection in our case. Communities could provide a good mapping of features and their interconnections within the system. Our evaluation has shown that a community mapping can highlight connections between features that may not be apparent to the domain experts, while in general, still providing a picture close to their estimations.

Many aspects of the features could potentially be useful to investigate during the feature extraction process and maintenance. While mainstream languages have the benefit of several well-established metrics, 4GL environments suffer from a lack of these. Our work contributed several new metrics that are adaptations of already established metrics but also some original metrics suitable for the abstraction level of features.

The methods and results of our work had been used during the work of our industrial partner, and the adoption process was concluded successfully.

The Author's Contributions

The author implemented the information retrieval feature-extraction solution and took part in the planning and coordination of the experiments, including the combination possibilities, new metrics, community detection, and evaluation. He took part in the combination effort between static analysis and information retrieval, both in implementation and the analysis of the results. The author implemented a basic feature-extractor with graphical interface based on information retrieval for the initial approach, and later performed the analysis of the variants, and planned the validation procedure.

The publications related to this thesis points are:

Journal publications

- **András Kicsi**, Viktor Csuvik, László Vidács, Ferenc Horváth, Árpád Beszédes, Tibor Gyimóthy, and Ferenc Kocsis. Feature Analysis using Information Retrieval, Community Detection and Structural Analysis Methods in Product Line Adoption. Journal of Systems and Software, 155:70–90, sep 2019.

Full papers in Scopus-indexed conference proceedings

- **András Kicsi**, László Vidács, Arpád Beszédes, Ferenc Kocsis, and István Kovács. Information retrieval based feature analysis for product line adoption in 4gl systems. In Proceedins of the 17th International Conference on Computational Science and Its Applications – ICCSA 2017, pages 1–6. IEEE, 2017.
- **András Kicsi**, Viktor Csuvik, László Vidács, Árpád Beszédes, and Tibor Gyimóthy. Feature level complexity and coupling analysis in 4GL systems. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 10964 LNCS, pages 438–453. Springer Verlag, may 2018.
- **András Kicsi**, László Vidács, Viktor Csuvik, Ferenc Horváth, Arpád Beszédes, and Ferenc Kocsis. Supporting product line adoption by combining syntactic and textual feature extraction. In International Conference on Software Reuse, ICSR 2018. Springer International Publishing, 2018.

Other full paper publications

- **András Kicsi** and Viktor Csuvik. Feature Level Metrics Based on Size and Similarity in Software Product Line Adoption. In 11th Conference of PhD Students in Computer Science (CSCS 2018), pages 25–28, 2018.

II. Textual Methods in Aiding Test-to-Code Traceability

The contributions of this thesis point are detailed in Chapter 4. Test-to-Code traceability is the identification of each test case's focus, finding out which parts of the software they are meant to assess. While their research is less popular than, for instance, requirement traceability, there is substantial literature on the matter, and it is not an easy problem considering that larger systems can have tens of thousands of test cases. Proper test-to-code traceability can facilitate the localization of faults, aid test prioritization, and could even serve as a foundation for good automatic program repair. While the current state-of-the-art solutions tend to employ multiple approaches to the task, our focus was on the lexical techniques. Our research examined eight medium-sized open-source systems with more than 1.25 million lines of code.

Relying on naming conventions (NC) is an exceptionally precise way to identify the tested code elements, but this technique is highly dependent on developer habits and can be complicated in a variety of cases. Our work provided an in-depth investigation about the automatic extraction of naming convention links, the possible combinations of various rules, and their perceived usage in the systems. The results show that on the method level, naming conventions are complicated, and developers rarely strive to uphold perfect traceability. On the class level, they are used quite often, which can greatly contribute to class-level test-to-code traceability. Mirroring the package hierarchy of the production code is also popular, and even if this can only lead to general directions in the code, this can still be extremely useful as filtering information.

Since textual approaches are often used in state-of-the-art solutions, optimizing them could lead to better traceability overall. Thus, our experiments also searched for the best textual method. We introduced several new possible combinations

and examined their usefulness in finding correct traceability links, measured both on a large set of traceability links automatically extracted according to naming conventions and a manual dataset on 220 test cases of four systems. Call information obtained via regular expressions was also factored into our results and found to contribute greatly to good results, reinforcing that a combination of techniques is indeed likely to produce even better results.

Our search for the best code representation provided less conclusive results, the identifier-centric (IDENT) representation that utilizes abstract syntax trees came out on top in the overwhelming majority of the cases during the NC-based evaluation, but the text-centric (SRC) representation proved more precise when compared to the limited amount of manual data.

Out of the main methods of latent semantic indexing, TF-IDF and Doc2Vec, Doc2Vec was found to be most precise both as a singular technique and in combination with call information or naming conventions. The combination of these three techniques could also warrant some attention, but our filtering solution based on a consensus of all three of them provided marginally worse results than Doc2Vec in itself.

Finally, our findings show that the combination of naming conventions and Doc2Vec could lead to a good alloy of the precision of naming conventions and the versatility of other textual techniques.

Our work also contributed a manual dataset called TestRoutes, and traceability extraction experiments on Stack Overflow code snippets featuring LSI an Doc2Vec, in which Doc2Vec performed better.

The Author's Contributions

The author implemented a Latent Semantic Indexing based solution for recovering traceability links, the evaluation code relying on naming conventions and also manual data. He also implemented recovery techniques based on various naming conventions and conducted experiments with them. The author planned and coordinated the manual annotation of the TestRoutes dataset, and also the Stack Overflow experiments. He also took part in the evaluation and explanation of various other results and the planning of all of the published experiments.

Journal publications

- **András Kicsi**, Viktor Csuvik, and László Vidács. Large Scale Evaluation of NLP-based Test-to-Code Traceability Approaches. IEEE Access, 2021.

Full papers in Scopus-indexed conference proceedings

- **András Kicsi**, László Tóth, and László Vidács. Exploring the benefits of utilizing conceptual information in test-to-code traceability. Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, pages 8–14, 2018.
- Viktor Csuvik, **András Kicsi**, and László Vidács. Source code level word embeddings in aiding semantic test-to-code traceability. In 10th International Workshop at the 41st International Conference on Software Engineering (ICSE) – SST 2019. IEEE, 2019.

- Viktor Csuvik, **András Kicsi**, and László Vidács. Evaluation of Textual Similarity Techniques in Code Level Traceability. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 11622 LNCS, pages 529–543. Springer Verlag, 2019.
- **András Kicsi**, Márk Rákóczi, and László Vidács. Exploration and mining of source code level traceability links on stack overflow. In ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies, pages 339–346, 2019.
- **András Kicsi**, László Vidács, and Tibor Gyimothy. Testroutes: A manually curated method level dataset for test-to-code traceability. In Proceedings of the 17th International Conference on Mining Software Repositories, MSR 2020, pages 593–597. IEEE, IEEE, jun 2020.

III. Machine Understanding of Radiologic Reports

The contributions of this thesis point are detailed in Chapter 5. Radiologic examinations produce image data, but their main output that incorporates the expertise of radiologists is still manifested in textual form. Reports and opinions written by the radiologists contain significant information which could be utilized in various services such as quality assurance and automatic report generation.

The thesis point describes a process for the automatic understanding of radiologic reports of the spinal region. This process involves multiple levels, which are elaborated below.

Radiologic reports tend to contain a high number of misspellings. While these are relatively easy to overlook by the human eye, they can inhibit the work of machine learning classifiers, as well as rule-based methods. Correcting these automatically is a hard task not only because of the morphologic richness of the Hungarian language but also because radiologic reports tend to use Latin words according to the rules of other Hungarian terms, attaching Hungarian suffixes. Our work introduced a method based on the HunSpell spelling correction tool as well as its considerable extension with a specialized dictionary and new prioritization rules to accommodate the specific use of the language in the reports. Our results are shown to improve the results of both machine learning based classification (0.35 F-score improvement) and ontology-based identification (more than 20% of the unknown terms correctly identified).

The corrected natural language text goes through automatic labelling via a BiLSTM-CRF, or in the more current version, a BERT classifier. It distinguishes anatomical locations, disorders and properties according to an annotation performed by radiologists on 487 real reports. The classification produced an F1-score value of 96.85, determining locations, disorders and properties with high accuracy.

Negations are determined through linguistic analysis via the Magyarlánc analyzer. Constituent parsing is used to link negating words to specific disorders in the text, detected by our previous step.

Our identification relies on a simple ontology built for the task. The ontology lists identifiers for various locations with a simple hierarchy, providing the vertebra

level of anatomical locations if they are applicable (for example, the disc at the L5 level is denoted as L5_D). The identifiers of disorders also carry additional information as they are grouped in harmful pathologies, describing a normal or intact state and aspects that are needed to specify another accompanying disorder but are meaningless without it.

Our process also maps the elements according to their semantic connections. Disorders are usually connected with one or more locations, and properties usually belong to one or more disorders. Our system uses a rule-based method to determine these, also heavily relying on the clauses of the sentences, extracted via constituents.

The output of our process is visualized in an easy to comprehend tree-structure that showcases the detected elements and their connections.

In a functional evaluation, three radiologists evaluated our machine understanding concerning classifications, the determination of connections, and the identification of locations and disorders and found that it provided correct results in 95.97% to 97.74% of the cases.

While this is still ongoing research, some applications have already arisen. An automatic understanding of the reports could facilitate the generation of training data for automatic analyzer tools aiming to detect disorders on MRI images. As part of a current project, our solution is used for such a task. Another evident use of our work is the facilitation of patient comprehension. Interactive electronic reports could provide much more information for patients and lead to better healthcare service, and our project also includes this endeavor.

The Author's Contributions

The author laid the groundwork and coordinated the manual annotations, and took a big part in the later refinement of the data. He took part in the planning of all aspects of the machine understanding method and coordinated its implementation. The author planned the functional evaluation and its guidelines. He took a big role in the evaluation and explanation of the results and their implications.

The publications related to this thesis point are:

Full papers in Scopus-indexed conference proceedings

- **András Kicsi**, Klaudia Szabó Ledenyi, Péter Pusztai, and László Vidács. Automatic classification and entity relation detection in hungarian spinal MRI reports. In 3rd ICSE Workshop on Software Engineering for Healthcare, 2021.

Other full paper publications

- **András Kicsi**, Péter Pusztai, Klaudia Szabó Ledenyi, Endre Szabó, Gábor Berend, Veronika Vincze, and László Vidács. Információkinyerés magyar nyelvű gerinc mr leletekből. In XV. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2019), page 177–186, Szeged, 2019. (In Hungarian)
- **András Kicsi**, Klaudia Szabó Ledenyi, Péter Pusztai, Péter Németh, and László Vidács. Entitások azonosítása és szemantikai kapcsolatok feltárása radiológiai leletekben. In XVI. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2020), page 15–28, Szeged, 2020. (In Hungarian)

- **András Kicsi**, Klaudia Szabó Ledenyi, Péter Németh, Péter Pusztai, László Vidács, and Tibor Gyimóthy. Elírások automatikus detektálása és javítása radiológiai leletek szövegében. In XVI. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2020), page 191–204, Szeged, 2020. (In Hungarian)
- **András Kicsi**, Péter Pusztai, Endre Szabó, and László Vidács. Szaknyelvi annotációk javításának statisztikai alapú támogatása. In XVI. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2020), page 115–128, Szeged, 2020. (In Hungarian)

Table A.1 summarizes the main publications and how they relate to our thesis points.

№	[77]	[67]	[65]	[78]	[68]	[75]	[31]	[30]	[71]	[76]	[66]	[70]	[73]	[72]	[69]	[74]
I.	♦	♦	♦	♦	♦											
II.						♦	♦	♦	♦	♦	♦					
III.												♦	♦	♦	♦	♦

Table A.1: Thesis contributions and supporting publications

B

Magyar nyelvű összefoglaló

Az emberiség legfőbb információátviteli módja az írott vagy beszélt nyelv. Történelmünk során az írott szó hozzájárult a tudomány fejlődéséhez, az általános iskolázottsághoz, birodalmakat emelt fel vagy buktatott meg, és vezetett a jelenlegi társadalmunkhoz. Habár az információ napjainkban már leggyakrabban bináris adatként kerül tárolásra, ezt mégis az ember számára értelmezhető formába kell önteni. Az értekezés a természetesnyelvű szövegben rejlő információ kinyerésével foglalkozik. Két tézispont szorosan kapcsolódik a szoftverfejlesztés témaköréhez, ahol a programkódban szintén óriási mennyiségű kiaknázható szemantikus információ van, míg a harmadik tézispont radiológiai leletek automatizált értelmezésével foglalkozik. A szöveg mindenhol ott van életünkben, megfelelő feldolgozása pedig hatalmas lehetőségekkel kecsegtet.

I. Feature-kinyerés 4. generációs nyelvű rendszerekben

A tézispont kontribúcióit a 3. fejezet részletezi. Számos variáns felett a szoftvertermékcsaládok bevezetése felettébb erőforrás-igényes feladat lehet. Mivel a rendszer által szolgáltatott funkciók (feature-ök) ismerete elengedhetetlen, a bevezetés folyamata a fejlesztők és területi szakértők szoros együttműködését követeli meg. A feature-öket szolgálja ki a szoftver, és ezek a kódon belül változatosan vannak megvalósítva. A feature-kinyerés ezeket igyekszik kinyerni a kód könnyebb változtatása, és a variánsok megfelelő egyesítése érdekében. Számos kihívással kerültünk szembe egy gyógyszeripari logisztikai rendszer 19 variánsa feletti feature-kinyerés során, amelyet egy ipari partnerrel együtt dolgozva végeztünk. A szoftver egy negyedik generációs nyelven (4GL), a Magic XPA nyelven került implementálásra. A programkódot egy fejlesztői környezeten keresztül készítették valódi programozás nélkül, és a nyelv struktúrája jelentősen különbözik a hagyományos programozási nyelvekétől, amelyekre már vannak kész kinyerési megoldások. A Magic szoftverek ún. programokból és ezek alatti ún. subtask-jaikból állnak, amelyek egymást hívják meg, de gyakoriak a logikai és adat egységek is. Arra törekedve, hogy minél több hasznos információt adjunk a fejlesztőknek és területi szakértőknek egyaránt, számos új megközelítést vezettünk be a feature-kinyerés témakörében.

A feature-kinyerési kísérleteink többféle különböző kimenetet produkáltak, amelyek a munka különböző fázisait hivatottak segíteni. A statikus elemzéssel kinyert hívási gráfok megmutatják a szoftver programjai közti kapcsolatokat, amely rendkívül fontos lehet a fejlesztőknek. Egy másik módszer, az információ kinyerés (IR), amelyet mi a latent semantic indexing (LSI) technika segítségével valósítottunk meg, szintén a programok és feature-ök kapcsolatára mutat rá a szemantikai hasonlóságot felhasználva. Ez a területi szakértők látásmódjához közelebbi nézet, hiszen ők általában nem mélyednek el a programokban, a szemantikai kapcsolatokat viszont ismerik. A két kinyerési módszer különböző információforrásra épít. A két kimenet ezért kombinálható, ezzel egy szigorúbb szűrést biztosítva, kiemelve a legfontosabb programokat minden feature-nél.

A területi szakértőknek további segítség lehet, ha információt kapnak a feature-ök közötti kapcsolatokról. A hívási gráf ezen kapcsolatok majdnem mindegyikét tartalmazza, és bár megérése nehéz, valódi szerkezeti információt nyújt. A gráfelméletben számos megoldás kínálkozik gráfokra, amelyeket akár a hívási gráfokon is alkalmazhatunk. Esetünkben ilyenek a gráf-közösségek kinyerésére készült algoritmusok. A közösségek egy kiváló leképezését nyújtják a feature-öknek és kapcsolataiknak a rendszeren belül. Kiértékelésünk azt állapította meg, hogy a közösségek leképezése olyan kapcsolatokat is felfedhet a területi szakértőknek, amelyekkel egyébként nem lennének tisztában, általánosságban mégis jól közelítve szakértői véleményüket.

A feature-ök számos tulajdonsága hasznos információval szolgálhat a kinyerési folyamat és későbbi karbantartás során is. A hagyományos nyelveken már rengeteg jól megalapozott mérőszám létezik ezek leírására, 4GL környezetben ezek azonban nem feltétlenül alkalmasak. Munkánk meglévő, számos jól megalapozott mérőszám adaptációján túl több új mérőszámot is bevezetett a feature-ök absztrakciós szintjén.

Az általunk kifejlesztett módszerek és kapott eredmények felhasználásra kerültek ipari partnerünk sikeres termékcsalád-bevezetése során.

A szerző kontribúciói

A szerző megvalósította az információ kinyerésen alapuló feature-kinyerési megoldást, és részt vett a kísérletek tervezésében és koordinálásában, beleértve az új mérőszámokat, a közösség-detektálást és az értékelést. Részt vett a statikus elemzés és az információkeresés kombinálásában, mind a megvalósításban, mind az eredmények elemzésében. A szerző egy információ kinyerésen alapuló, grafikus felülettel rendelkező feature-kinyerő alkalmazást valósított meg a kezdeti kísérletekhez, végezte a variánsok feletti elemzést a metrikák alapján, illetve megtervezte a validálási eljárást.

Folyóirat publikációk

- **András Kicsi**, Viktor Csuvik, László Vidács, Ferenc Horváth, Árpád Beszédes, Tibor Gyimóthy, and Ferenc Kocsis. Feature Analysis using Information Retrieval, Community Detection and Structural Analysis Methods in Product Line Adoption. Journal of Systems and Software, 155:70–90, sep 2019.

Scopus által indexelt konferenciakötetben megjelent publikációk

- **András Kicsi**, László Vidács, Arpád Beszédes, Ferenc Kocsis, and István Kovács. Information retrieval based feature analysis for product line adoption in 4gl systems. In Proceedins of the 17th International Conference on Computational Science and Its Applications – ICCSA 2017, pages 1–6. IEEE, 2017.
- **András Kicsi**, Viktor Csuvik, László Vidács, Árpád Beszédes, and Tibor Gyimóthy. Feature level complexity and coupling analysis in 4GL systems. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 10964 LNCS, pages 438–453. Springer Verlag, may 2018.
- **András Kicsi**, László Vidács, Viktor Csuvik, Ferenc Horváth, Arpád Beszédes, and Ferenc Kocsis. Supporting product line adoption by combining syntactic and textual feature extraction. In International Conference on Software Reuse, ICSR 2018. Springer International Publishing, 2018.

Egyéb publikációk

- **András Kicsi** and Viktor Csuvik. Feature Level Metrics Based on Size and Similarity in Software Product Line Adoption. In 11th Conference of PhD Students in Computer Science (CSCS 2018), pages 25–28, 2018.

II. Szöveges módszerek a teszt-kód nyomonkövethetőség elősegítésére

A tézispont kontribúcióit a 4. fejezet részletezi. A teszt-kód nyomonkövethetőség annak azonosítása, hogy mi a tesztesetek fókusza, a szoftver mely kódrészletét igyekeznek tesztelni. Habár ezen nyomonkövezhetőségi probléma kutatása kevésbé népszerű, mint például a követelmény-nyomonkövethetőség területe, mégis jelentős tudományos irodalom áll a téma mögött, és a probléma maga egyáltalán nem könnyű azt tekintve, hogy a nagyobb szoftverrendszerekben több tízezer teszt is lehet. A megfelelő teszt-kód nyomonkövethetőség hozzájárulhat a hibák lokalizációjához, segíthet a tesztek priorizálásában, és jó alapot nyújthat az automatikus program javítás számára is. Bár a jelenlegi legkorszerűbb megoldások általában módszerek kombinációját használják, kutatási célunk a szöveges technikák vizsgálata volt. Nyolc közepes méretű, nyílt forrású rendszeren végeztünk méréseket, amelyek összesen több mint 1,25 millió kódsort öleltek fel.

A névkonvenciók (NC) kivételesen jó nyomot szolgáltatnak a tesztelt kódrészletek megtalálásához, de ez a technika nagyban függ a fejlesztők szokásaitól és sok esetben igen komplikált lehet automatikus használata. Munkánk során mélyreható elemzést végeztünk a névkonvenciók automatikus kinyerésének, más módszerekkel való kombinációjának, és a rendszerekben fellelhető használati szokásainak területein. Eredményeink azt mutatják, hogy a metódusok szintjén a névkonvenciók fenntartása igen komplikált, a fejlesztők ritkán tartják fenn a tökéletes egyezést. Osztályok szintjén a névkonvenciókat már sokkal gyakrabban alkalmazzák, amely nagyban hozzájárulhat egy automatizált nyomonkövetési módszerhez is. A tesztelt kód csomag-hierarchiájának tükrözése szintén népszerű, és bár ez csak általános útbaigazítást nyújt, ezek használata mégis nagyon jól használható lehet az eredmények szűrésében.

Mivel a legjobban teljesítő módszerek is alkalmaznak szöveges hasonlóságot más technikákkal kombinálva, ezen módszerek optimalizálása összességében jobb nyomonkövethetőséget eredményezhet. Ezért kísérleteinkkel a legjobb szöveges módszert is igyekeztünk felkutatni. Bemutattunk több új lehetséges kombinációt a meglévő lexikális módszerek fölött, és megvizsgáltuk ezek pontosságát mind egy nagy, névkonvenciók alapján automatizáltan kinyert, mind egy 220 tesztesetből álló kézi adathalmazon. Hívási információkat szintén felhasználtunk, amelyeket a rendszerekből reguláris kifejezések segítségével nyertünk, ezek nagyban növelték a módszerek pontosságát, alátámasztva azt, hogy a kombinációk valóban jobb eredményeket mutatnak ezen a területen.

A programkód legmegfelelőbb reprezentációja utáni kutatásunk kevésbé egyértelmű eredményeket produkált, az azonosító-központú (IDENT) reprezentáció, amely absztrakt szintaxis fákat (AST) is felhasznál, mutatott legnagyobb pontosságot a névkonvenció-alapú kiértékelésnél, ám a limitált mennyiségű kézi adattal szemben egy másik reprezentáció bizonyult jobbnak, amely egyszerűbb módon szövegként kezeli a forráskódot (SRC).

Az összehasonlított főbb technikák, a latent semantic indexing (LSI), a TF-IDF, és a Doc2Vec közül a Doc2Vec módszer végzett legjobbként mind önálló technikaként, mind hívási információval vagy a névkonvenciókkal kombinálva. A három technika kombinációja szintén elképzelhető lehet, a három módszer konszenzusán alapuló szűrési megoldásunk azonban a Doc2Vec-nél minimálisan rosszabb eredményeket adott.

Kísérleteink alapján a névkonvenciók és a Doc2Vec hasonlóság kombinációja jól ötvözi a névkonvenciók magas pontosságát a szöveges technikák rugalmasságával.

Munkánk során létrejött a TestRoutes nevű manuális adatkészlet, valamint Stack Overflow kódrészletek teszt-kód nyomonkövethetőségével is kísérleteztünk, amelyekben LSI és Doc2Vec összehasonlítása szerepel, itt a Doc2Vec jobban teljesített.

A szerző kontribúciói

A szerző Latent Semantic Indexing technilán alapuló megoldást implementált a teszt-kód nyomonkövethetőség segítésére, illetve kiértékelő kódot névkonvenciókra és manuális adatokra is támaszkodva. Különféle névkonvenciókon alapuló helyreállítási technikákat is megvalósított, és végezte a velük folytatott kísérleteket. A szerző megtervezte és koordinálta a TestRoutes adatbázis manuális annotációját, és a Stack Overflow kísérleteket. Részt vett továbbá a különböző egyéb eredmények értékelésében, magyarázatában és az összes publikált kísérlet tervezésében.

Folyóirat publikációk

- **András Kicsi**, Viktor Csuvik, and László Vidács. Large Scale Evaluation of NLP-based Test-to-Code Traceability Approaches. IEEE Access, 2021.

Scopus által indexelt konferenciakötetben megjelent publikációk

- **András Kicsi**, László Tóth, and László Vidács. Exploring the benefits of utilizing conceptual information in test-to-code traceability. Proceedings of

the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, pages 8–14, 2018.

- Viktor Csuvik, **András Kicsi**, and László Vidács. Source code level word embeddings in aiding semantic test-to-code traceability. In 10th International Workshop at the 41st International Conference on Software Engineering (ICSE) – SST 2019. IEEE, 2019.
- Viktor Csuvik, **András Kicsi**, and László Vidács. Evaluation of Textual Similarity Techniques in Code Level Traceability. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 11622 LNCS, pages 529–543. Springer Verlag, 2019.
- **András Kicsi**, Márk Rákóczi, and László Vidács. Exploration and mining of source code level traceability links on stack overflow. In ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies, pages 339–346, 2019.
- **András Kicsi**, László Vidács, and Tibor Gyimothy. Testroutes: A manually curated method level dataset for test-to-code traceability. In Proceedings of the 17th International Conference on Mining Software Repositories, MSR 2020, pages 593–597. IEEE, IEEE, jun 2020.

III. Radiológiai leletek gépi értelmezése

A tézispont kontribúcióit az 5. fejezet részletezi. A radiológiai vizsgálatokból képi adat készül, de fő kimenetük mégis a radiológusok szakértelmét is tartalmazó szövegben, azaz leletben és véleményben nyilvánul meg. Ezek jelentős mennyiségű releváns információt tartalmaznak, amelyek felhasználhatók lennének számos szolgáltatásban, mint a minőség biztosítása, vagy az automatikus lelet-generálás.

A tézispont a radiológiai gerincleletek automatizált értelmezésére mutat be egy módszert. Folyamatunk több lépésből áll, amelyek a következőkben kerülnek kifejtésre.

A radiológiai leletek gyakran nagy mennyiségű elírást tartalmaznak. Habár ezeket az emberi szem könnyedén figyelmen kívül hagyja, a gépi tanuláson alapuló címkéző, és szabály-alapú megoldások működését jelentősen ronthatják. Automatikus javításuk nem csupán azért nehéz feladat, mert a magyar nyelv morfológiailag igen gazdag, hanem azért is, mert a leletekben igen gyakran előfordulnak latin szavak, amelyeket a magyar helyesírás szabályai szerint kezelnek, magyar ragokat kapva. Munkánk a HunSpell automatikus helyesírás-javító eszköz képességeit egy szakszótárral, valamint számos priorizálási szabállyal kiegészítve igyekezett megfelelő automatizált javításokat adni a gerincleletek szűk területére. Eredményeink alapján mind a gépi tanuláson alapuló címkézésünk (0.35 F-mérték javulás), mind az ontológia-alapú azonosításunk (az ismeretlen kifejezések több mint 20%-a helyesen felismerve) esetében javulás látható.

A kijavított természetesnyelvű szöveg egy automatizált címkézésen esik át, amelyet egy BiLSTM-CRF, illetve fejlesztéseink után már egy BERT címkéző hajt végre, megkülönböztetve testrészeket, elváltozásokat és tulajdonságokat radiológusok által annotált 487 valós leleten tanulva. Az osztályozás 96,85-ös F1-mértéket produkált, nagy pontossággal meghatározva a testrészeket, elváltozásokat és tulajdonságokat.

A tagadásokat nyelvi elemzés segítségével kezeli a rendszer, melynek során a Magyarlánc nyelvi elemzőt használja fel. Ennek során a tagmondatok mentén rendeli hozzá a tagadószavakat elváltozásokhoz, amelyeket az előző címkéző lépés szolgáltatott.

Azonosításunk egy egyszerű ontológián alapul, amelyet a feladathoz készítettünk. Az ontológiában azonosítókat találunk a különböző testészeknek egy egyszerű hierarchiával, amely csigolya-szintet rendel a testrészekhez, amennyiben a testrész erre alkalmas (az L5 szintben lévő porckorong – azaz discus például az L5_D azonosítót kapja). Az elváltozások azonosítói szintén hordoznak releváns információt, az elváltozásokat besorolják a rossz állapotot jelző patológiák, a jó vagy normális állapotot leíró elváltozások, vagy az önmagukban elváltozást nem megadó, de más elváltozásokat pontosító aspektusok csoportjába.

Módszerünk ezen elemeket szemantikai kapcsolataik alapján is feltérképezi. Az elváltozások általában kapcsolódnak egy vagy több testrészhez, míg a tulajdonságok egy vagy több elváltozáshoz. Rendszerünkben egy szabály-alapú megoldás állapítja meg ezeket, amely szintén nagyban épít a mondatok tagmondataira.

Módszerünk kimenete egy könnyen értelmezhető fa-struktúraként kerül vizualizációra, amely bemutatja a detektált és felismert elemeket, és kapcsolataikat.

Egy funkcionális kiértékelés során három radiológus értékelte gépi megértő módszerüket az osztályozás, a kapcsolatok meghatározása, valamint a testrészek és elváltozások azonosítása terén, és úgy értékelték, hogy a módszer az esetek 95,97–97,74%-ában helyes eredményeket adott.

Ugyan ez a kutatás még folyamatban van, máris számos felhasználási lehetőséggel kecsegtet. Az leletek automatizált értelmezése segíthet tanítóadat gyors és pontos generálásában olyan modellek számára, amelyek MRI-felvételeken keresnek elváltozásokat. Egy folyamatban lévő projekt részeként megoldásunkat fel is használjuk ilyen célokra. Munkánk egy másik nyilvánvaló felhasználási lehetősége a páciensek lelet-értelmezésének javítása. Interaktív elektronikus leletek sokkal több információt szolgáltathatnak a páciensek számára, amely javíthat az egészségügyi szolgáltatások minőségén. Ez szintén része jelenlegi projektünknek.

A szerző kontribúciói

A szerző megalapozta és koordinálta a kézi annotációkat, és nagy szerepet vállalt az adatok későbbi finomításában. Részt vett a gépi megértési módszer minden aspektusának tervezésében, koordinálta annak megvalósítását. A szerző megtervezte a funkcionális kiértékelést, és annak irányelveit. Nagy szerepet vállalt az eredmények és következményeik értékelésében, magyarázatában.

Scopus által indexelt konferenciakötetben megjelent publikációk

- **András Kicsi**, Klaudia Szabó Ledenyi, Péter Pusztai, and László Vidács. Automatic classification and entity relation detection in hungarian spinal MRI reports. In 3rd ICSE Workshop on Software Engineering for Healthcare, 2021.

Egyéb publikációk

- **András Kicsi**, Péter Pusztai, Klaudia Szabó Ledenyi, Endre Szabó, Gábor Berend, Veronika Vincze, and László Vidács. Információkinyerés magyar nyelvű gerinc mr leletekből. In XV. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2019), page 177–186, Szeged, 2019. (Magyar nyelven)
- **András Kicsi**, Klaudia Szabó Ledenyi, Péter Pusztai, Péter Németh, and László Vidács. Entitások azonosítása és szemantikai kapcsolatok feltárása radiológiai leletekben. In XVI. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2020), page 15–28, Szeged, 2020. (Magyar nyelven)
- **András Kicsi**, Klaudia Szabó Ledenyi, Péter Németh, Péter Pusztai, László Vidács, and Tibor Gyimóthy. Elírások automatikus detektálása és javítása radiológiai leletek szövegében. In XVI. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2020), page 191–204, Szeged, 2020. (Magyar nyelven)
- **András Kicsi**, Péter Pusztai, Endre Szabó, and László Vidács. Szaknyelvi annotációk javításának statisztikai alapú támogatása. In XVI. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2020), page 115–128, Szeged, 2020. (Magyar nyelven)

A tézispontokat és a kapcsolódó publikációkat a B.1. táblázat összegzi.

№	[77]	[67]	[65]	[78]	[68]	[75]	[31]	[30]	[71]	[76]	[66]	[70]	[73]	[72]	[69]	[74]
I.	♦	♦	♦	♦	♦											
II.						♦	♦	♦	♦	♦	♦					
III.												♦	♦	♦	♦	♦

B.1. táblázat. A tézispontokhoz kapcsolódó publikációk

Bibliography

[1] Gensim's webpage. `https://radimrehurek.com/gensim/`. Accessed: 2019.

[2] The Stack Oveflow platform. https://stackoverflow.com/, Accessed: August 2022.

[3] Leena Al-Hussaini. Experience: Insights into the benchmarking data of hunspell and aspell spell checkers. *Journal on Data and Information Quality*, 8:13:1–13:10, jun 2017.

[4] R Al-msie, a Djamel Seriai, M Huchard, and C Urtado. An approach to recover feature models from object-oriented source code. In *Actes de la Journee Lignes de Produits 2012*, pages 1–12, 2012.

[5] R. Al-msie'deen, A.-D. Seriai, M. Huchard, C. Urtado, and S. Vauttier. Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*, pages 586–593. IEEE, aug 2013.

[6] R. Al-Msie'Deen, A. D. Seriai, M. Huchard, C. Urtado, and S. Vauttier. Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In *Proceedings of the 2013 IEEE 14th International Conference on Information Reuse and Integration, IEEE IRI 2013*, number January, pages 586–593. IEEE, aug 2013.

[7] Ra'Fat Al-Msie'Deen, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. Feature location in a collection of software product variants using formal concept analysis. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7925 LNCS, pages 302–307. Springer, Berlin, Heidelberg, 2013.

[8] 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.

[9] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, oct 2002.

[10] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. Multi-objective reverse engineering of variability-safe feature models based on code dependencies of system variants. *Empirical Software Engineering*, 22(4):1763–1794, aug 2017.

[11] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. Feature location for software product line migration. In *Proceedings of the 18th International Software Product Line Conference on Companion Volume for Workshops, Demonstrations and Tools - SPLC '14*, pages 52–59, New York, New York, USA, 2014. ACM Press.

[12] Ebrahim Bagheri, Faezeh Ensan, and Dragan Gasevic. Decision support for the software product line domain engineering lifecycle. *Automated Software Engineering*, 19(3):335–377, 2012.

[13] Manuel Ballarin, Raúl Lapeña, and Carlos Cetina. Leveraging Feature Location to Extract the Clone-and-Own Relationships of a Family of Software Products. In *Proceedings of the 15th International Conference on Software Reuse: Bridging with Social-Awareness - Volume 9679*, pages 215–230. Springer-Verlag New York, Inc., 2016.

[14] Sebastian Baltes, Christoph Treude, and Stephan Diehl. SOTorrent: Studying the origin, evolution, and usage of stack overflow code snippets. *IEEE International Working Conference on Mining Software Repositories*, pages 191–194, sep 2019.

[15] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[16] N. Bettenburg, B. Adams, A. E. Hassan, and M. Smidt. A lightweight approach to uncover technical artifacts in unstructured data. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 185–188, June 2011.

[17] Marek Blahuš. Morphology-aware spell-checking dictionary for esperanto. In *Proceedings of Recent Advances in Slavonic Natural Language Processing, RASLAN 2009*, pages 3–8, Brno, 2009.

[18] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P1000, 2008.

[19] Markus Borg, Per Runeson, and Anders Ardö. Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. *Empirical Software Engineering*, 19(6):1565–1616, dec 2014.

[20] Philipp Bouillon, Jens Klinke, Nils Meyer, and Friedrich Steimann. EZUNIT: A framework for associating failed unit tests with potential programming errors. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4536 LNCS, pages 101–104. Springer Verlag, 2007.

[21] Keno K. Bressem, Lisa C. Adams, Robert A. Gaudin, Daniel Tröltzsch, Bernd Hamm, Marcus R. Makowski, Chan Yong Schüle, Janis L. Vahldiek, and Stefan M. Niehues. Highly accurate classification of chest radiographic reports using a deep learning natural language model pre-trained on 3.8 million text reports. *Bioinformatics*, 36(21):5255–5261, nov 2020.

[22] Xiaoling Cai, Shoubin Dong, and Jinlong Hu. A deep learning model incorporating part of speech and self-matching attention for named entity recognition of Chinese electronic medical records. *BMC Medical Informatics and Decision Making*, 19(S2), apr 2019.

[23] Cagatay Catal and Cagatay. Barriers to the adoption of software product line engineering. *ACM SIGSOFT Software Engineering Notes*, 34(6):1, dec 2009.

[24] Yao Chen, Changjiang Zhou, Tianxin Li, Hong Wu, Xia Zhao, Kai Ye, and Jun Liao. Named entity recognition from Chinese adverse drug event reports with lexical feature based BiLSTM-CRF and tri-training. *Journal of Biomedical Informatics*, 96, aug 2019.

[25] P. Clements and C. Krueger. Eliminating the adoption barrier. *IEEE Software*, 19:29–31, jul 2002.

[26] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[27] Paul C. Clements, Lawrence G. Jones, John D. McGregor, and Linda M. Northrop. Getting there from here: a roadmap for software product line adoption. *Communications of the ACM*, 49(12):33, dec 2006.

[28] Arman Cohan, Iz Beltagy, Daniel King, Bhavana Dalvi, and Daniel S. Weld. Pretrained language models for sequential sentence classification. In *EMNLP-IJCNLP 2019 - 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing, Proceedings of the Conference*, pages 3693–3699, 2019.

[29] Jonathan Crowell, Qing Zeng-Treitler, Long Ngo, and Eve-Marie Lacroix. A frequency-based technique to improve the spelling suggestion rank in medical queries. *Journal of the American Medical Informatics Association : JAMIA*, 11:179–85, 05 2004.

[30] Viktor Csuvik, András Kicsi, and László Vidács. Evaluation of Textual Similarity Techniques in Code Level Traceability. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11622 LNCS, pages 529–543. Springer Verlag, 2019.

[31] Viktor Csuvik, András Kicsi, and László Vidács. Source code level word embeddings in aiding semantic test-to-code traceability. In *10th International Workshop at the 41st International Conference on Software Engineering (ICSE) – SST 2019*. IEEE, 2019.

[32] Andrew M. Dai, Christopher Olah, and Quoc V. Le. Document Embedding with Paragraph Vectors. jul 2015.

[33] Fred Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7:171–176, 03 1964.

[34] S C Deerwester, S T Dumais, T K Landauer, G W Furnas, and R A Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[35] Ralph A. DeFronzo, Andrew Lewin, Sanjay Patel, Dacheng Liu, Renee Kaste, Hans J. Woerle, and Uli C. Broedl. Combination of empagliflozin and linagliptin as second-line therapy in subjects with type 2 diabetes inadequately controlled on metformin. *Diabetes Care*, 38(3):384–393, jul 2015.

[36] Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, pages 4171–4186, 2019.

[37] Hamzeh Eyal-Salman, Abdelhak Djamel Seriai, and Christophe Dony. Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval. *Proceedings of the 2013 IEEE 14th International Conference on Information Reuse and Integration, IEEE IRI 2013*, pages 209–216, 2013.

[38] Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony, and Ra'fat Al-msie'deen. Recovering traceability links between feature models and source code of product variants. In *Proceedings of the VARiability for You Workshop on Variability Modeling Made Useful for Everyone - VARY '12*, pages 21–25, New York, New York, USA, 2012. ACM Press.

[39] Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony, and Ra'fat Al-msie'deen. Recovering traceability links between feature models and source code of product variants. In *VARiability for You Workshop on Variability Modeling Made Useful for Everyone - VARY '12*, pages 21–25. ACM Press, 2012.

[40] Davide Falessi, Giovanni Cantone, and Gerardo Canfora. A comprehensive characterization of NLP techniques for identifying equivalent requirements. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '10*, page 1, New York, New York, USA, 2010. ACM Press.

[41] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 391–400. IEEE, sep 2014.

[42] J. M. Florez. Automated fine-grained requirements-to-code traceability link recovery. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 222–225, 2019.

[43] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3):75–174, 2010.

[44] Mohammad Ghafari, Carlo Ghezzi, and Konstantin Rubinov. Automatically identifying focal methods under test in unit test cases. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 61–70. IEEE, sep 2015.

[45] J. Guo, J. Cheng, and J. Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 3–14, 2017.

[46] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically Enhanced Software Traceability Using Deep Learning Techniques. In *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, pages 3–14. IEEE, may 2017.

[47] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[48] James Hamilton and Sebastian Danicic. Dependence communities in source code. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 579–582. IEEE, 2012.

[49] John V. Harrison and Wie Ming Lim. Automated Reverse Engineering of Legacy 4GL Information System Applications Using the ITOC Workbench. In *10th International Conference on Advanced Information Systems Engineering*, pages 41–57. Springer-Verlag, 1998.

[50] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. Reverse Engineering Feature Models from Programs' Feature Sets. In *18th Working Conference on Reverse Engineering*, pages 308–312. IEEE, oct 2011.

[51] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Improving after-the-fact tracing and mapping: Supporting software quality predictions. *IEEE Software*, 22(6):30–37, nov 2005.

[52] T. Hey. Indirect: Intent-driven requirements-to-code traceability. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 190–191, 2019.

[53] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*, page 14, New York, New York, USA, 2007. ACM Press.

[54] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[55] Chen Huo and James Clause. Interpreting Coverage Information Using Direct and Indirect Coverage. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 234–243. IEEE, apr 2016.

[56] Iñigo Jauregi Unanue, Ehsan Zare Borzeshi, and Massimo Piccardi. Recurrent neural networks with specialized word embeddings for health-domain named-entity recognition. *Journal of Biomedical Informatics*, 76:102–109, dec 2017.

[57] Bin Ji, Rui Liu, Shasha Li, Jie Yu, Qingbo Wu, Yusong Tan, and Jiaju Wu. A hybrid approach for named entity recognition in Chinese electronic medical record. *BMC Medical Informatics and Decision Making*, 19(S2):64, apr 2019.

[58] Bin Ji, Rui Liu, Wei Sang Xu, Sha Sha Li, Jin Tao Tang, Jie Yu, and Qian Li. A BILSTM-CRF method to Chinese electronic medical record named entity recognition. In *ACM International Conference Proceeding Series*. Association for Computing Machinery, dec 2018.

[59] Azadeh Kamel Ghalibaf, Elham Nazari, Mahdi Gholian-Aval, and Mahmood Tara. Comprehensive overview of computer-based health information tailoring: A systematic scoping review. *BMJ Open*, 9, jan 2019.

[60] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. Variability Mining: Consistent Semi-automatic Detection of Product-Line Features. *IEEE Transactions on Software Engineering*, 40(1):67–82, 2014.

[61] Nilam Kaushik, Ladan Tahvildari, and Mark Moore. Reconstructing Traceability between Bugs and Test Cases: An Experimental Study. In *2011 18th Working Conference on Reverse Engineering*, pages 411–414. IEEE, oct 2011.

[62] Matthew B Kelly, Jason S Alexander, Bram Adams, and Ahmed E Hassan. Recovering a Balanced Overview of Topics in a Software Domain. 2011.

[63] Mark Kernighan, Kenneth Church, and William Gale. A spelling correction program based on a noisy channel model. In *COLING 90: Computational Linguistics in 1990*, pages 205–210, 01 1990.

[64] A. Kicsi, V. Csuvik, L. Vidács, Á. Beszédes, and T. Gyimóthy. Feature level complexity and coupling analysis in 4GL systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10964 LNCS, pages 438–453. Springer, Cham, may 2018.

[65] András Kicsi and Viktor Csuvik. Feature Level Metrics Based on Size and Similarity in Software Product Line Adoption. In *11th Conference of PhD Students in Computer Science (CSCS 2018)*, pages 25–28, 2018.

[66] András Kicsi, Viktor Csuvik, and László Vidács. Large Scale Evaluation of NLP-based Test-to-Code Traceability Approaches. *IEEE Access*, 2021.

[67] András Kicsi, Viktor Csuvik, László Vidács, Árpád Beszédes, and Tibor Gyimóthy. Feature level complexity and coupling analysis in 4GL systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10964 LNCS, pages 438–453. Springer Verlag, may 2018.

[68] András Kicsi, Viktor Csuvik, László Vidács, Ferenc Horváth, Árpád Beszédes, Tibor Gyimóthy, and Ferenc Kocsis. Feature Analysis using Information Retrieval, Community Detection and Structural Analysis Methods in Product Line Adoption. *Journal of Systems and Software*, 155:70–90, sep 2019.

[69] András Kicsi, Péter Pusztai, Endre Szabó, and László Vidács. Szaknyelvi annotációk javításának statisztikai alapú támogatása. In *XVI. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2020)*, page 115–128, Szeged, 2020.

[70] András Kicsi, Péter Pusztai, Klaudia Szabó Ledenyi, Endre Szabó, Gábor Berend, Veronika Vincze, and László Vidács. Információkinyerés magyar nyelvű gerinc mr leletekből. In *XV. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2019)*, page 177–186, Szeged, 2019.

[71] András Kicsi, Márk Rákóczi, and László Vidács. Exploration and mining of source code level traceability links on stack overflow. In *ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies*, pages 339–346, 2019.

[72] András Kicsi, Klaudia Szabó Ledenyi, Péter Németh, Péter Pusztai, László Vidács, and Tibor Gyimóthy. Elírások automatikus detektálása és javítása radiológiai leletek szövegében. In *XVI. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2020)*, page 191–204, Szeged, 2020.

[73] András Kicsi, Klaudia Szabó Ledenyi, Péter Pusztai, Péter Németh, and László Vidács. Entitások azonosítása és szemantikai kapcsolatok feltárása radiológiai leletekben. In *XVI. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2020)*, page 15–28, Szeged, 2020.

[74] András Kicsi, Klaudia Szabó Ledenyi, Péter Pusztai, and László Vidács. Automatic classification and entity relation detection in hungarian spinal mri reports. In *3rd ICSE Workshop on Software Engineering for Healthcare*, 2021.

[75] András Kicsi, László Tóth, and László Vidács. Exploring the benefits of utilizing conceptual information in test-to-code traceability. *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pages 8–14, 2018.

[76] András Kicsi, László Vidács, and Tibor Gyimothy. Testroutes: A manually curated method level dataset for test-to-code traceability. In *Proceedings of the 17th International Conference on Mining Software Repositories, MSR 2020*, pages 593–597. IEEE, IEEE, jun 2020.

[77] András Kicsi, László Vidács, Arpád Beszédes, Ferenc Kocsis, and István Kovács. Information retrieval based feature analysis for product line adoption in 4gl systems. In *Proceedins of the 17th International Conference on Computational Science and Its Applications – ICCSA 2017*, pages 1–6. IEEE, 2017.

[78] András Kicsi, László Vidács, Viktor Csuvik, Ferenc Horváth, Arpád Beszédes, and Ferenc Kocsis. Supporting product line adoption by combining syntactic and textual feature extraction. In *International Conference on Software Reuse, ICSR 2018*. Springer International Publishing, 2018.

[79] Benjamin Klatt and Martin Küster. A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies. In *1st International workshop on Reverse Variability Engineering (REVE'13)*, volume 13, pages 1—-8, 2013.

[80] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, pages 165–176, New York, New York, USA, 2016. ACM Press.

[81] CharlesW. Krueger. *Easing the Transition to Software Mass Customization*, pages 282–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[82] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. Extracting software product lines: a cost estimation perspective. In *Proceedings of the 20th International Systems and Software Product Line Conference on - SPLC '16*, pages 354–361, New York, New York, USA, 2016. ACM Press.

[83] Karen Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24:377–439, 12 1992.

[84] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 282–289, San Francisco, CA, USA, 2001.

[85] Kenneth Lai, Maxim Topaz, Foster Goss, and Li Zhou. Automated misspelling detection and correction in clinical free-text records. *Journal of Biomedical Informatics*, 55, 04 2015.

[86] Scott J. Lee, Brent D. Weinberg, Ashwani Gore, and Imon Banerjee. A Scalable Natural Language Processing for Inferring BT-RADS Categorization from Unstructured Brain Magnetic Resonance Reports. *Journal of Digital Imaging*, pages 1–8, jun 2020.

[87] Luqi Li, Jie Zhao, Li Hou, Yunkai Zhai, Jinming Shi, and Fangfang Cui. An attention-based deep learning model for clinical named entity recognition of Chinese electronic medical records. *BMC Medical Informatics and Decision Making*, 19(5):1–11, dec 2019.

[88] Crescencio Lima, Christina Chavez, and Eduardo Santana de Almeida. Investigating the Recovery of Product Line Architectures: An Approach Proposal. pages 201–207. Springer, Cham, may 2017.

[89] Wang Ling, Tiago Luís, Luís Marujo, Ramón Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W. Black, and Isabel Trancoso. Finding function in form: Compositional character models for open vocabulary word representation. In *Conference Proceedings - EMNLP 2015: Conference on Empirical Methods in Natural Language Processing*, pages 1520–1530, 2015.

[90] Xiaoxuan Liu, Livia Faes, Aditya U. Kale, Siegfried K. Wagner, Dun Jack Fu, Alice Bruynseels, Thushika Mahendiran, Gabriella Moraes, Mohith Shamdas, Christoph Kern, Joseph R. Ledsam, Martin K. Schmid, Konstantinos Balaskas, Eric J. Topol, Lucas M. Bachmann, Pearse A. Keane, and Alastair K. Denniston. A comparison of deep learning performance against health-care professionals in detecting diseases from medical imaging: a systematic review and meta-analysis. *The Lancet Digital Health*, (6):271–297, oct 2019.

[91] Yun Liu, Po-Hsuan Cameron Clen, Jonathan Krause, and Lily Peng. How to read articles that use machine learning: Users' guides to the medical literature. *JAMA - Journal of the American Medical Association*, 322(18):1806–1816, 11 2019.

[92] Zengjian Liu, Ming Yang, Xiaolong Wang, Qingcai Chen, Buzhou Tang, Zhe Wang, and Hua Xu. Entity recognition from clinical texts via recurrent neural network. *BMC Medical Informatics and Decision Making*, 17(S2):67, jul 2017.

[93] Sugandha Lohar, Sorawit Amornborvornwong, Andrea Zisman, and Jane Cleland-Huang. Improving trace accuracy through data-driven configuration and composition of tracing features. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, page 378, New York, New York, USA, 2013. ACM Press.

[94] M2J Software LLC. Homepage of M2J. http://www.magic2java.com, Accessed: May 2017.

[95] Xuezhe Ma and Eduard Hovy. End-to-end sequence labeling via bi-directional LSTM-CNNs-CRF. In *54th Annual Meeting of the Association for Computational Linguistics, ACL 2016 - Long Papers*, volume 2, pages 1064–1074, 2016.

[96] Stephen MacDonell. Metrics for Database Systems: An Empirical Study. *IEEE International Symposium on Software Metrics*, pages 99–107, 1997.

[97] Magic Software Enterprises Ltd. Magic Software Enterprises. http://www.magicsoftware.com, Accessed: May 2017.

[98] M A Maia, Victor Sobreira, K Paixão, S A Amo, and I R Silva. Using a sequence alignment algorithm to identify commonalities and variabilities from execution traces. In *International Workshop on Program Comprehension through Dynamic Analysis*, pages 6–10, 2008.

[99] A. Marcus and J.I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 125–135. IEEE, 2003.

[100] Andrian Marcus, Jonathan I Maletic, and Andrey Sergeyev. Recovery of Traceability Links between Software Documentation and Source Code. *International Journal of Software Engineering and Knowledge Engineering*, pages 811–836, 2005.

[101] Andrian Marcus, Andrey Sergeyev, Václav Rajlieh, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 214–223. IEEE Comput. Soc, 2004.

[102] Nayrolles Mathieu and Abdelwahab Hamou-Lhadj. Word embeddings for the software engineering domain. *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, pages 38–41, 2018.

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

[104] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26 (NIPS 2013)*, 2013.

[105] Tomas Mikolov, Ilya Sutskever, Kan Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *NIPS'13 Proceedings of the 26th International Conference on Neural Information Processing Systems*, 2:3111–3119, dec 2013.

[106] Brian S Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[107] Agnieszka Mykowiecka and Małgorzata Marciniak. Domain–driven automatic spelling correction for mammography reports. In *Proceedings of the International IIS (IIPWM'06)*, volume 35, pages 521–530, 04 2007.

[108] 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 *26th IEEE International Conference on Software Maintenance*, pages 1–6. IEEE Computer Society, September 2010.

[109] Csaba Nagy, László Vidács, Rudolf Ferenc, Tibor Gyimóthy, Ferenc Kocsis, and István Kovács. Complexity measures in 4gl environment. In *Computational Science and Its Applications - ICCSA 2011, Lecture Notes in Computer Science*, volume 6786 of *Lecture Notes in Computer Science*, pages 293–309. Springer Berlin / Heidelberg, 2011.

[110] 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 CSMR 2011 (15th European Conference on Software Maintenance and Reengineering)*, pages 343–346. IEEE Computer Society, March 2011.

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

[112] Dávid Márk Nemeskey. Introducing huBERT. In *XVII. Magyar Számítógépes Nyelvészeti Konferencia*, pages 3–14, 2019.

[113] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. Exploring API embedding for API usages and applications. In *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, pages 438–449. IEEE, may 2017.

[114] Attila Novák and Borbála Siklósi. Automatic diacritics restoration for hungarian. In *Conference Proceedings - EMNLP 2015: Conference on Empirical Methods in Natural Language Processing*, page 2286–2291, 2015.

[115] Attila Novák and Borbála Siklósi. Ékezetek automatikus helyreállítása magyar nyelvű szövegekben. In *XII. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2016)*, page 49–58, 2016.

[116] Ocean Software Solutions. Homepage of Magic Optimizer. http://www.magic-optimizer.com, Accessed: May 2017.

[117] Andrzej Olszak and Bo Nørregaard Jørgensen. Remodularizing Java programs for comprehension of features. *Proceedings of the First International Workshop on Feature Oriented Software Development FOSD 09*, pages 19–26, 2009.

[118] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia. When and How Using Structural Information to Improve IR-Based Traceability Recovery. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 199–208. IEEE, mar 2013.

[119] Reza Meimandi Parizi, Sai Peck Lee, and Mohammad Dabbagh. Achievements and Challenges in State-of-the-Art Software Traceability Between Test and Code Artifacts. *IEEE Transactions on Reliability*, 63:913–926, 2014.

[120] Paulius Paškevičius, Robertas Damaševičius, Eimutis Karčiauskas, and Romas Marcinkevičius. Automatic extraction of features and generation of feature models from java programs. *Information Technology and Control*, 41(4):376–384, 2012.

[121] Jon Patrick, Mojtaba Sabbagh, Suvir Jain, and Haifeng Zheng. Spelling correction in clinical notes with emphasis on first suggestion accuracy. In *2nd Workshop on Building and Evaluating Resources for Biomedical Text Mining*, pages 1–8, 2010.

[122] Xin Peng, Zhenchang Xing, Xi Tan, Yijun Yu, and Wenyun Zhao. The Journal of Systems and Software Improving feature location using structural similarity and iterative graph mapping. *The Journal of Systems & Software*, 86:664–676, 2013.

[123] Denys Poshyvanyk, Yann Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Václav Rajlich. Combining probabilistic ranking and latent semantic indexing for feature identification. In *IEEE International Conference on Program Comprehension*, pages 137–146. IEEE, 2006.

[124] Denys Poshyvanyk and Andrian Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *IEEE International Conference on Program Comprehension*, pages 37–46. IEEE, jun 2007.

[125] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software*, 88:147–168, 2014.

[126] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and David Binkley. SCOTCH: Test-to-code traceability using slicing and conceptual coupling. In *IEEE International Conference on Software Maintenance, ICSM*, pages 63–72. IEEE, 2011.

[127] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and David Binkley. Evaluating test-to-code traceability recovery methods through controlled experiments. *Journal of Software: Evolution and Process*, 25(11):1167–1191, nov 2013.

[128] Martin Robillard, Robert Walker, and Thomas Zimmermann. Recommendation Systems for Software Engineering. *IEEE Software*, 27(4):80–86, jul 2010.

[129] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[130] Alex Roehrs, Cristiano André da Costa, and Rodrigo da Rosa Righi. OmniPHR: A distributed architecture model to integrate personal health records. *Journal of Biomedical Informatics*, 71:70–81, jul 2017.

[131] Javier Rojo, Juan Hernandez, and Juan M. Murillo. A personal health trajectory API: Addressing problems in health institution-oriented systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12128 LNCS, pages 519–524. Springer, jun 2020.

[132] Bart Van Rompaey and Serge Demeyer. Establishing traceability links between unit test cases and units under test. In *European Conference on Software Maintenance and Reengineering, CSMR*, pages 209–218. IEEE, 2009.

[133] Wei Ruan, Naveenkumar Appasani, Katherine Kim, Joseph Vincelli, Hyun Kim, and Won Sook Lee. Pictorial visualization of EMR summary interface and medical information extraction of clinical notes. In *CIVEMSA 2018 - 2018 IEEE International Conference on Computational Intelligence and Virtual Environments for Measurement Systems and Applications, Proceedings*. Institute of Electrical and Electronics Engineers Inc., aug 2018.

[134] Patrick Ruch, Robert Baud, and Antoine Geissbühler. Using lexical disambiguation and named-entity recognition to improve spelling correction in the electronic patient record. *Artificial intelligence in medicine*, 29:169–84, 09 2003.

[135] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 461, New York, New York, USA, 2011. ACM Press.

[136] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3-4):487–517, 2012.

[137] Borbála Siklósi, Attila Novák, and Gábor Prószéky. *Context-Aware Correction of Spelling Errors in Hungarian Medical Documents*, page 248–259. Number Lecture Notes in Computer Science 7978. 2013.

[138] Borbála Siklósi, Attila Novák, and Gábor Prószéky. Helyesírási hibák automatikus javítása orvosi szövegekben a szövegkörnyezet figyelembevételével. In *IX. Magyar Számítógépes Nyelvészeti Konferencia*, page 148–158, Szeged, 2013.

[139] Borbála Siklósi, György Orosz, Attila Novák, and Gábor Prószéky. Automatic structuring and correction suggestion system for hungarian clinical records. In *8th SaLTMiL Workshop on Creation and Use of Basic Lexical Resources for Less-resourced Languages*, page 29–34, 2012.

[140] Livio Baldini Soares, Nicholas FitzGerald, Jeffrey Ling, and Tom Kwiatkowski. Matching the blanks: Distributional similarity for relation learning. In *ACL 2019 - 57th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 2895–2905. Association for Computational Linguistics (ACL), 2020.

[141] SourceMeter webpage. `https://www.sourcemeter.com/`, 2019.

[142] Pontus Stenetorp, Sampo Pyysalo, Goran Topić, Tomoko Ohta, Sophia Ananiadou, and Jun'ichi Tsujii. brat: A Web-based Tool for NLP-Assisted Text Annotation. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 102–107, Avignon, France, April 2012.

[143] Senthil Karthikeyan Sundaram, Jane Huffman Hayes, and Alexander Dekhtyar. Baselines in requirements tracing. In *ACM SIGSOFT Software Engineering Notes*, volume 30, page 1, New York, New York, USA, 2005. ACM Press.

[144] Muzamil Hussain Syed and Sun Tae Chung. Menuner: Domain-adapted bert based ner approach for a domain with limited dataset and its application to food menu domain. *Applied Sciences (Switzerland)*, 11(13):6007, jun 2021.

[145] Ian Tenney, Dipanjan Das, and Ellie Pavlick. BERT rediscovers the classical NLP pipeline. *ACL 2019 - 57th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 4593–4601, may 2020.

[146] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):1–45, jun 2014.

[147] Herman Tolentino, Michael Matters, Wikke Walop, Barbara Law, Wesley Tong, Fang Liu, Paul Fontelo, Katrin Kohl, and Daniel Payne. A umls-based spell checker for natural language processing in vaccine safety. *BMC medical informatics and decision making*, 7, 02 2007.

[148] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, 18:542–553, 2018.

[149] Muhammad Irfan Ullah, Günther Ruhe, and Vahid Garousi. Decision support for moving from a single product to a product portfolio in evolving software systems. *The Journal of Systems & Software*, 83:2496–2512, 2010.

[150] Marco Tulio Valente, Virgilio Borges, and Leonardo Passos. A Semi-Automatic Approach for Extracting Software Product Lines. *IEEE Transactions on Software Engineering*, 38(4):737–754, jul 2012.

[151] Kestutis Valincius, Vytautas Stuikys, and Robertas Damasevicius. Understanding of e-commerce is through feature models and their metrics. *Proceedings of the IADIS International Conference Information Systems 2013, IS 2013*, 8(1):55–62, 2013.

[152] 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.

[153] June Verner and Graham Tate. Estimating Size and Effort in Fourth-Generation Development. *IEEE Software*, 5:15–22, 1988.

[154] Suhang Wang, Jiliang Tang, Charu Aggarwal, and Huan Liu. Linked Document Embedding for Classification. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management - CIKM '16*, pages 115–124, New York, New York, USA, 2016. ACM Press.

[155] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. SNIAFL: towards a static non-interactive approach to feature location. In *Proceedings. 26th International Conference on Software Engineering*, pages 293–303. IEEE Comput. Soc, 2004.

[156] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 87–98, 2016.

[157] Robert White, Jens Krinke, and Raymond Tan. Establishing multilevel test-to-code traceability links. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 861–872, New York, NY, USA, 2020. Association for Computing Machinery.

[158] 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):87–94, 1994.

[159] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings - International Conference on Software Engineering*, volume 11, pages 789–799, may 2018.

[160] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. Understanding feature evolution in a family of product variants. *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 109–118, 2010.

[161] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. Feature location in a collection of product variants. *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 145–154, 2012.

[162] Suresh Yadla, Jane Huffman Hayes, and Alex Dekhtyar. Tracing requirements to defect reports: An application of information retrieval techniques. *Innovations in Systems and Software Engineering*, 1(2):116–124, sep 2005.

[163] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. Combining Word Embedding with Information Retrieval to Recommend Similar Bug Reports. In *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pages 127–137. IEEE, oct 2016.

[164] Azita Yazdani, Marjan Ghazisaeedi, Nasrin Ahmadinejad, Masoumeh Giti, Habibe Amjadi, and Azin Nahvijou. Automated Misspelling Detection and Correction in Persian Clinical Text. *Journal of Digital Imaging*, 33(3):555–562, jun 2020.

[165] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, pages 404–415, New York, New York, USA, 2016. ACM Press.

[166] Mingwang Yin, Chengjie Mou, Kaineng Xiong, and Jiangtao Ren. Chinese clinical named entity recognition with radical-level feature and self-attention mechanism. *Journal of Biomedical Informatics*, 98, oct 2019.

[167] 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.

[168] John Zech, Jessica Forde, Joseph J. Titano, Deepak Kaji, Anthony Costa, and Eric Karl Oermann. Detecting insertion, substitution, and deletion errors in radiology reports using neural sequence-to-sequence models. *Annals of Translational Medicine*, 7(11):233–233, jun 2019.

[169] Taha Zerrouki and Amar Balla. Implementation of infixes and circumfixes in the spellcheckers. In *Proceedings of the Second International Conference on Arabic Language Resources and Tools*, pages 61–65, 2009.

[170] Yu Zhang, Li Guo, Degen Huang, Kaiyu Huang, Jiuyi Li, and Zhang Pan. English Drug Name Entity Recognition Method Based on Attention Mechanism BiLSTM-CRF. In *Proceedings of IEEE 14th International Conference on Intelligent Systems and Knowledge Engineering, ISKE 2019*, pages 831–836. Institute of Electrical and Electronics Engineers Inc., nov 2019.

[171] Shan Zhao, Zhiping Cai, Haiwen Chen, Ye Wang, Fang Liu, and Anfeng Liu. Adversarial training based lattice LSTM for Chinese clinical named entity recognition. *Journal of Biomedical Informatics*, 99, nov 2019.

[172] Teng Zhao, Qinghua Cao, and Qing Sun. An Improved Approach to Traceability Recovery Based on Word Embeddings. In *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, volume 2017-Decem, pages 81–89. IEEE, dec 2018.

[173] Zhaocheng Zhu and Junfeng Hu. Context Aware Document Embedding. jul 2017.

[174] János Zsibrita, Veronika Vincze, and Richárd Farkas. Magyarlanc: A toolkit for morphological and dependency parsing of Hungarian. In *International Conference Recent Advances in Natural Language Processing, RANLP*, pages 763–771, 2013.