Source Code Retrieval using Case Based Reasoning

Mihai Pitu

Dissertation 2013

Erasmus Mundus MSc in Dependable Software Systems

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Department of Computer Science

National University of Ireland, Maynooth

Co. Kildare, Ireland

A dissertation submitted in partial fulfilment

of the requirements for the

Erasmus Mundus MSc Dependable Software Systems

Head of Department: Dr Adam Winstanley

Supervisors: Dr. Diarmuid O'Donoghue and Dr. Rosemary Monahan

July, 2013

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Science in Dependable Software Systems, is entirely my own work and has not been taken from the work of the others save and to the extent that such work has been cited and acknowledged within the text of my work.

Mihai Pitu

_____

Acknowledgement

I would like to thank Dr. Diarmuid O'Donoghue and Dr. Rosemary Monahan from the National University of Ireland, Maynooth for their support and input on the project.

Contents

Abstract

Formal verification of source code has been extensively used in the past few years in order to create dependable software systems. However, although formal languages like Spec# or JML are getting more and more popular, the set of verified implementations is very small and only growing slowly. Our work aims to automate some of the steps involved in writing specifications and their implementations, by reusing existing verified programs. That is, for a given implementation we seek to retrieve similar verified code and then reapply the missing specification that accompanies that code. In this thesis, I present the retrieval system that is part of the *Arís* (Analogical Reasoning for reuse of Implementation & Specification) project. The overall methodology of the *Arís* project is very similar to Case-Based Reasoning (CBR) and its parent discipline of Analogical Reasoning (AR), centered on the activities of solution retrieval and reuse. CBR's retrieval phase is achieved using semantic and structural characteristics of source code. API calls are used as semantic anchors and characteristics of conceptual graphs are used to express the structure of implementations. Finally, we transfer the knowledge (i.e. formal specification) between the input implementation and the retrieved code artefacts to produce a specification for a given implementation. The evaluation results are promising and our experiments show that the proposed approach has real potential in generating formal specifications using past solutions.

1. Introduction

1.1. Problem statement

Software systems are ubiquitous nowadays in our society. We rely very much on software and we need to make the process of software engineering more rigorous because of economic and safety reasons - software faults are costing a huge amount of money and they expose software systems to risk of catastrophic failures in critical applications (Hoare, et al., 2007). In this context, the Design by Contract paradigm (Meyer, 1992) is a pragmatic design approach that seeks to improve the quality of software and has the potential to truly revolutionize the software development discipline.

Software verification using formal methods is the process of demonstrating software correctness with respect to a formal specification that expresses the desired behaviour. In order to help software engineers to formally specify and verify their implementations, the set of verified software applications, which is currently very small, needs to grow. This will also help Design by Contract and formal methods like Spec# (Rustan, et al., 2010), JML (Leavens & Cheon, 2006), Eiffel (Meyer, 2006) to increase their popularity, because developers will have access to a rich software repository with verified code examples (Verified Software Repository (Woodcock, et al., 2009)). Therefore, a framework capable of automating some of the steps involved in writing specifications and their implementations by reusing existing verified software artefacts, will greatly add to the creation of dependable software systems.

Figure 1: *Arís* logo

This project is part of *Arís*[1] (**Figure 1**) (Pitu M., Grijincu D., Li P., Saleem A., O'Donoghue D. P., Monahan R., 2013), a system that aims to solve the burden of some of these steps, using retrieval techniques i.e. for a given implementation, *Arís* will retrieve similar verified code and then reapply the missing specification that accompanies that code. Similarly, for a given specification, *Arís*

[1] Meaning "again" in the Irish Language

retrieves code with a similar specification and uses its implementation to generate the missing implementation. *Source Code Retrieval using Case Based Reasoning* is responsible in *Arís* for retrieving programs from a large set of samples and to perform knowledge (formal specification) transfer from the retrieved implementations to the query.

1.2. Motivation

The authors of the Verified Software Initiative (VSI) (Hoare, et al., 2007) envisioned a world with fewer software defects due to the use of formal verification. VSI consists of a research program spanning fifteen years with the purpose of turning software verification into a core method for creating dependable software. As part of the VSI, a Verified Software Repository (Bicarregui, et al., 2006) is being created, which is the result of an international effort that will eventually contain hundreds of fully or partially verified programs.

This idea has inspired us to create a system capable of automatic generation and verification of specifications for software components that will help increase the number of formally verified programs and therefore, aid software developers to create reliable software systems. In addition, using an existing set of verified software components, the framework could verify query software artefacts, i.e. one can retrieve similar specified software components and formally verify the query by transferring knowledge (specification) from the retrieved components. This can also mean that the developer does not need to learn the specification language, because the transfer will be done automatically. In addition, the framework aims to assess how well the generated specification performs, because formal methods often require human intervention in order to fully verify an implementation. Thus, the project should facilitate interaction, if automatic generation of new specifications is not successful.

The core concept of this project is software retrieval, which itself can be motivated by a series of used cases. For example, existing source code retrieval systems are used for: detecting plagiarism and code theft, rapid prototyping, knowledge acquisition by comparing different implementations, improving understanding of source code and in programming by analogous examples (PBE) (Repenning & Perrone, 2000).

1.3. Design by Contract

Design by Contract (DbC) (Meyer, 1992) represents an approach used for designing dependable (correct and robust) software systems in the Object Oriented Programming context, while, in the same time assuring that the designed software keeps its OOP qualities: reusability, extendibility and compatibility. One way of achieving such desirable properties is using formal methods, which can verify a software implementation if a formal specification is associated.

Every set of software statements that performs a useful task has an implicit *precondition* and *postcondition* associated. A precondition implies the constraints under which a software component will function correctly, while a postcondition expresses the expected result after the execution. The functionality of that code is expressed by a *specification*, which is implied from the preconditions and postconditions. This specification can also be understood as a contract between two entities: the *client,* who calls a specified routine and expects a functionality without knowing anything about the implementation; the *supplier*, who knows, maintains and assures that the implementation meets the required functionality, expressed by the postcondition.

	Precondition	**Postcondition**
Client	*obligation:* ensures the precondition is true	*benefit:* expects the functionality established by the postcondition
Supplier	*benefit:* assume precondition is true	*obligation:* ensure the postcondition is true

The above obligations and benefits represent the *contract* between the client and the supplier and illustrate how preconditions and postconditions constitute a specification of a given software implementation. These definitions allow us to further understand formal specifications, in order to achieve our main goal of automatic verification of software implementations, through reuse of specifications.

Summary

We have presented a short description of the problem statement and we have motivated this project in the context of Design by Contract paradigm. The rest of this paper is organized as follows: In chapter (2), we present the current state of the art and give a brief critical review of

relevant related work and existing retrieval systems. In chapter (3), we set the theoretical background for our work using definitions. In chapter (4), we give a detailed description of our proposed solution and describe experiments and methods used in the creation of the system. In chapter (5), we present some evaluation experiments for our solution, followed by results and a comparison with existing projects. Finally, in chapter (6) we draw the conclusions from our work and we identify potential areas for future progress.

2. Related Work

In this chapter, we present the current state of the art and systems that were inspirational for our proposed framework. Although source code search engines like Google Code Search (Google Inc., n.d.), CodePlex (Microsoft, n.d.), SourceForge (Slashdot Media, n.d.) and Krugle (Aragon Consulting, n.d.) are helping software developers to reuse software components (Sim, et al., 2011), Source Code Retrieval is still an active area of research (McMillan, et al., 2012) (Gu, et al., 2004) (Kawaguchi, et al., 2006) (Mishne & De Rijke, 2004) (Reiss, 2009). This fact can be explained by the difficulty of the source code retrieval task, which involves understanding the blended nature between structure and content of source code and therefore between syntax and semantics of programming languages.

In the current state of the art, there exists a variety of approaches for source code retrieval, for example: exploring the semantic meaning of the code, taking advantage of the code structure or applying free-text indexing techniques used in conventional information retrieval (Michail & Notkin, 1999). In the area of formal verification, although the Design by Contract paradigm and Formal Methods have gained popularity (Woodcock, et al., 2009), the research activity concerning implementation/specification reuse is still in its inception phase, with few influential papers that reach this area by addressing UML specification reuse (Park & Bae, 2011) (Robles, et al., 2012).

In the related work literature, systems capable of retrieving source code snippets (i.e. methods, classes, etc.) are often called *Component Retrieval* systems. However, in our proposed framework, we refer to methods, classes or collection of classes as *source code artefacts*, because of the possibility of having an associated formal specification with the software component.

2.1. Source Code Retrieval using Conceptual Similarity

The Language & Inference Technology Group from University of Amsterdam has proposed a successful model for source code retrieval (Mishne & De Rijke, 2004). The central representational formalism for source code, is the Conceptual Graph (3.2.1) (Sowa, 1984), which allows the combination of representing the “contents” of documents (source code) with techniques for handling graphs, that capture the structural characteristics of source code.

In order to create a basic taxonomy for source code, the authors of the system defined a set of concept types (e.g. `Assign`, `Func-Call, If, Loop, Variable, Struct`) that are specific to source code documents and a set of relation types (e.g. `Condition, Comment, Defines, Returns, Parameter`) (notions further explained in section (3.2.1)). Using these definitions, the Conceptual Graph is constructed in parallel with the Abstract Syntax Tree (Neamtiu, et al., 2005) (that would be generated by a compiler when generating executable code). The retrieval process consists of ranking all documents from memory according to their similarity score with respect to a given source code snippet. Therefore, the quality of the retrieval results depends on the similarity measure for Conceptual Graphs. The proposed system introduces a new similarity measure that takes advantage of both structure and content, by comparing Conceptual Graphs node-by-node. This process is sensitive to the node type (relation of concept) and to the information contained in the concepts by computing the Levenshtein (3.2.4) string distance between node contents. Because the similarity measure is computationally expensive ($O(|G|^3)$, where G is a Conceptual Graph), reducing the number of graphs compared in the retrieval algorithm with the query was important (only a number of samples were fully compared with the query), by the use of an indexing mechanism for graphs and offline computation of some intermediate results.

The evaluation process is interesting and hard, because of the nature of the system that needs to be evaluated (source code retrieval). The main problem is the lack of a publicly available set of source code artefacts that are grouped by their similarity, since this is an error prone and laborious manual task. Instead, the authors experimented with a set of source code documents that contain with “high probability”, clusters of similar documents. As a primary sanity check, the first test case used random samples from the set of documents in memory and the expected result was that, the retrieved documents had to contain these samples with high similarity score. In the second test case, the random samples from the set of documents in memory were slightly modified and the system was supposed to retrieve these documents as well with high similarity score. The results were compared in terms of precision with other source code retrieval frameworks and the overall conclusion was that the proposed method outperformed other well-established models.

The main contributions of this project are the usage of Conceptual Graphs in the context of Source Code Retrieval and a custom similarity measure developed for CGs. However, the system

has a large amount of free parameters chosen manually and the retrieval algorithm is highly complex from a computational point of view.

2.2. CLAN: Detecting Similar Software Applications

CLAN (Closely reLated ApplicatioNs) (McMillan, et al., 2012) is a novel approach for automatically detecting similar software applications developed in Java. The system is motivated by a practical scenario that relates to the typical lifecycle of a project in the Accenture[2] consulting company, where, at any point in time, the company consultants are working in over 3000 software projects and the company has several thousands of projects contained in repositories. CLAN seeks to improve reusability of these software projects, because this will help save time and resources in building or prototyping software systems.

CLAN is capable of retrieving applications that are relevant to an input text query (short description of requirements) or a query in the form of full software applications. In order to achieve software retrieval, CLAN extends Mizzaro's conceptual framework for relevance (Mizzaro, 1998), where documents are relevant to each other if they share common concepts. Documents (software applications) can be clustered by how relevant they are to these concepts. In Mizzaro's framework, semantic anchors are constructs that accurately define the documents' sematic meaning. In CLAN, API calls (for example, a function call to a cryptographic function) serve as semantic anchors, because of their well-defined semantics.

The retrieval model in CLAN relies on Latent Semantic Indexing (3.2.3) and its use is justified by the fact that CLAN users express textual queries in natural language. Thus the retrieval algorithm must deal with the synonymy problem (different requirements can be described by the same words by different software engineers) and the polysemy problem (different words can describe the same requirements). In addition, CLAN implements LSI because the Java Development Kit[3] exports over 115000 API calls and this may lead to computational infeasibility due to exponential increase of the representational space (Powell, 2007). As a result of LSI, a Similarity Matrix that encodes similarity scores between applications is computed.

[2] http://www.accenture.com
[3] http://www.oracle.com/technetwork/java/javase/downloads/index.html

The authors of CLAN evaluated the system performance by using manual relevance judgments by experts (Manning, et al., 2008) using 33 Java programmers. The participants evaluated CLAN, MUDABlue (Kawaguchi, et al., 2006) (the closest relevant work to CLAN) and a combination of MUDABlue and CLAN. The results have shown that the proposed system finds similar applications with a higher precision than MUDABlue.

In conclusion, the authors of CLAN improved the state of the art by using Mizzaro's framework in context of software retrieval. However, CLAN relies entirely on semantic anchors, without considering the structural characteristics of software implementations. Also, the quality of retrieval is correlated with the number of dimensions, manually set for LSI (in case of CLAN, $r = 300$).

Summary

In this chapter, we discussed the state of the art for influential source code retrieval systems and we identified some tangent work for reuse of software specifications. However, to the best of our current knowledge, we were unable to identify work that has the exact same purpose as *Arís* (i.e. automation of software formal verification by reuse of software implementations/specifications). We will continue our critical analysis of these existing systems by comparing them to the proposed solution.

3. Definitions

We will now define the theoretical foundations that underlie this project, consisting of definitions of algorithms or disciplines used by our system. The concepts are briefly motivated with their practical value in this project, because we detail the overall architecture and design decisions of *Arís* in the following chapter (*Proposed Solution* (4)).

3.1. Artificial Intelligence techniques

This section introduces the theoretical background and methodology for the proposed framework. We discuss Artificial Intelligence disciplines like *Case-Based Reasoning* and *Analogical Reasoning*, which constitute the high-level methodologies for generating new specifications from past retrieved solutions. Moreover, we present the *K-Nearest Neighbours* and *K-Means Clustering* techniques, which are used in *Arís* in order to facilitate efficient retrieval of source code artefacts.

3.1.1. Instance Based Learning

Instance Based Learning (Russell & Norvig, 2003) is a family of machine learning algorithms, often categorized as lazy learning (Mitchell, 1997). Lazy learning algorithms (including K-Nearest Neighbour, Case Based Reasoning below) simply store training data (past problems and their solutions) with only minor or no pre-processing and wait for a query before generalizing. This means that lazy learning methods effectively use a richer hypothesis space than the alternate "eager learning" strategies, because many local linear functions are used to form an *implicit* global approximation of the target (Mitchell, 1997). In contrast, eager learning algorithms (e.g. Decision trees, Naïve Bayes, Support Vector Machines) generalize the hypothesis function before seeing the query, thus creating a single global approximation.

The computational complexity of Instance Based Learning algorithms for the classification step (Mitchell, 1997) is $O(n)$, because the hypothesis function is dependent in the worst case on all n training examples (existing instances in memory). This is one of the main disadvantages associated with the lazy learning methods (computation of the distance metric between the query and all the instances from memory). In addition, finding a good distance metric can be difficult and bad results

may arise if irrelevant attributes of instances are considered – this is known as the "curse of dimensionality" (Powell, 2007).

Among the attractive qualities of Instance-Based techniques for this project are the following qualities:

- Ability to easily incorporate additional solutions (i.e. retain successfully transferred formal specification)
- Represent and learn complex data, containing complex non-numeric data that is full of rich structure (both semantic and structural characteristics can be analysed from source code)

3.1.2. K-Nearest Neighbours

The most common example of an Instance Based Learning is K-Nearest Neighbours (k-NN). As a classification algorithm (**Figure 2**), k-NN works by retrieving the k most similar past cases (with respect to the query case), where each case is a point in $\mathbb{R}^d$ (d-dimensional real vector) and by assigning the most common class amongst the query's k nearest neighbours.

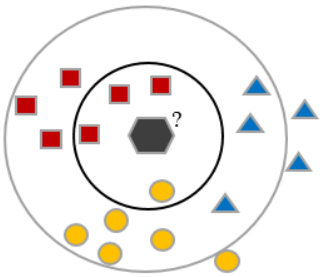


Figure 2: Example of k-NN where $k=4$. The query case (gray hexagon) should be classified as similar to the red squares (3 cases) because these cases represent the majority of samples that are nearest to the query

K-NN relies on a good distance metric because this function is selecting the nearest neighbours. For example, if cases are represented as continuous variables in memory, a common distance function used is the Euclidian distance (Manning, et al., 2008). The "curse of dimensionality" problem (for example, instances are described by 30 attributes, but only five are relevant to the target function) can be avoided by assigning weights to each element of the feature

vectors. In addition, selecting the appropriate *k* parameter can be challenging and it usually depends upon the data or on a heuristic function (Everitt, et al., 2011).

In this project, we use K-NN together with the *K-Means Clustering* (3.1.5) in the *Structural Retrieval* (4.3) model, in order to calculate computationally expensive functions only on the nearest neighbours of a given query, which enables a significant speed-up of the overall process. In chapter (5) we present the advantages and results of this technique.

3.1.3. Case Based Reasoning

Case Based Reasoning (CBR) (Kolodner, 1993) is an Artificial Intelligence technique, being closely related to K-NN. CBR focuses on problem solving and it is a form of Instance Based Learning. In CBR, new problems are solved by searching and retrieving similar previous problems (cases) from memory and reusing old solutions by transferring knowledge to the new problem. CBR frequently uses K-NN as its retrieval mechanism. A key part of CBR concerns the direct relationship between the problem instance and the recorded instances stored within the case base. The CBR process (**Figure 3**) can be divided into four main phases:

1. *Retrieve* most similar cases from previous experience (memory)
2. *Reuse* the information and knowledge learned from past cases and solve the new problem
3. *Revise* by evaluating the generated solution
4. *Retain* the new found solution for future problem solving (optional step)

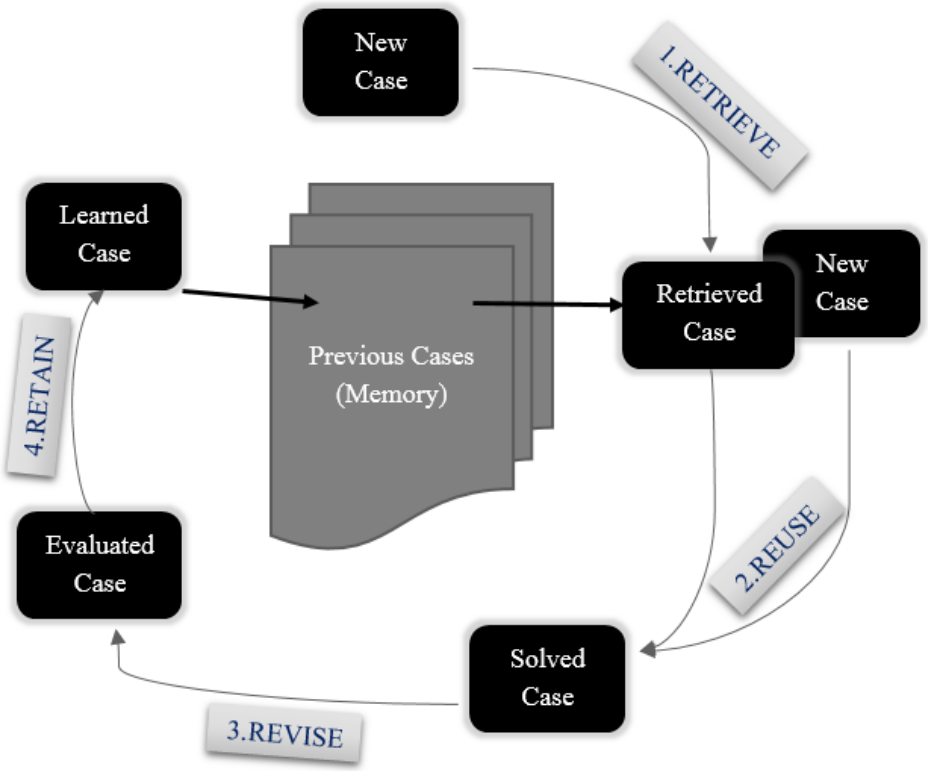


Figure 3: Case Based Reasoning process

Because CBR is a form of Instance Based Learning, cases from memory can be represented as symbolic descriptions (**Table 1**) and this may lead to high storage costs, as each case is often stored with as much information as possible because this might be useful at the *Reuse* step.

Table 1: Example of a symbolic description for software bugs cases in a CBR memory (modified example from (O'Donoghue, 2012))

```
[((bug-type Arithmetic)
(bug-name Division-by-zero)
(bug-severity Normal)
(software-component Database-model)
(author William-Smith)
(stack-trace ???)), …]
```

CBR has foundations in cognitive psychology theories, which have shown empirical evidence in several studies (Aamodt & Plaza, 1994), that previously experienced situations (cases) have an

important role in human problem solving. Using this as a model, the Machine Learning community has developed CBR into a mature and successful discipline that has been applied in many real world situations (e.g. Compaq SMART (Acorn & Walden, 1992), Cool Air system (Watson & Gardingen, 1999), General Electric system (Cheetham & Goebel, 2007) and many personalisation and product recommender systems).

In our proposed system, we use CBR process phases as our primary theoretical methodology, with a strong focus on the retrieval phase, which is achieved using semantic and structural characteristics of source code. The actual problem-solving step is knowledge transfer, where an unspecified implementation is formally verified using information from past cases (implementations with an associated formal specification). The new generated specification is evaluated by use of formal methods and if this previous phase is successful, the new case is retained for potential further use.

3.1.4. Analogical Reasoning

Analogical reasoning (AR) has been defined (Gentner & Smith, 2012) as the ability to perceive and use relational similarity between two conceptual structures (situations, events). The ability to reason using analogy is arguably the most important cognitive mechanism for humans and it implies the use of a familiar domain (the *source*) as a model, which is used to draw inferences about a less familiar domain (*target*) in decision-making or problem-solving situations. Typically, we have one given target problem and a number of available sources – as with CBR. While this project adopts a CBR like approach to creating specifications, our use of a "structure mapping" makes it a form of analogical reasoning rather than traditional CBR.

Analogical Reasoning solves problems not from first principles, but by *re*-using old solutions to solve new problems. Analogical reasoning can be subdivided into the following set of processes (Keane, et al., 2004):

1. *Retrieval:* Given a memory (database) of recorded solutions, a person or a system may use previous situations, similar to a newly arisen one. In the retrieval step, one or more possible sources are retrieved, such that the source and target situations are similar in their relational structure.

2. *Mapping:* Align the source and target representations and transfer the additional knowledge from source to the target. This process is formalized in (Gentner, 1983), by a theory called *structure mapping* which sets the principles used in the mapping process of AR. Structure mapping involves finding the largest "graph mapping" between the source and target descriptions.
3. *Evaluation:* After the mapping process is completed, the inferences are evaluated. The criteria for this step can be grouped into three classes: *factual correctness* (checks if the inferences are actually true), *goal relevance* (the newly formed mapping should be relevant to the problem that needs a solution) and the amount of new knowledge the analogy can provide (Gentner, 1983).

Analogical Reasoning enables creation of formal specifications from previously stored cases, by using the structure mapping theory. This mapping is possible due to *Conceptual Graphs* (3.2.1) representation of source code (Grijincu, 2013) and allows generation of formal specifications by using the knowledge from the source domain (4.6).

3.1.5. K-Means Clustering

Cluster analysis is concerned with the problem of grouping a set of objects in subgroups (clusters) by their similarity. In our framework, we use clustering for grouping source code artefacts representations by their similarity, which will be used at query time. Using the K-NN technique (3.1.2), the "closest" sub-groups of documents are determined, in order to retrieve and efficiently compute the similarity scores only with relevant results relative to the query.

Clustering is computationally difficult (NP-hard) (Dasgupta, 2008) and it is addressed by a number of techniques that are categorized as unsupervised learning (Mitchell, 1997) algorithms. K-Means (MacQueen, 1967) is an example of such a technique, which aims to partition the instances from memory into *k* (defined a priori) clusters where each instance belongs to the closest cluster centre (centroid).

The algorithm's input is a set of n instances $(x_1, x_2, \dots, x_n)$ where each instance is a point in $\mathbb{R}^d$ (d-dimensional real vector) and a number $k \in \mathbb{N}^*$ with $k \leq n$. K-means intends to minimize the squared error function (MacQueen, 1967) (optimization objective):

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \left\| x_i^{(j)} - c_j \right\|^2,$$

where $\left\| x_i^{(j)} - c_j \right\|^2$ is the distance (e.g. Euclidian) between instance $x_i^{(j)}$ and centroid c_j.

Table 2: K-means algorithm

```
Init: Randomly select k centroids (optionally from the set of input instance points)
    Repeat:
            -Assign each instance point to the closest cluster
            -Refresh each centroid position by using the average distance to the
             points in associated cluster
    Until: No change in centroids positions
```

Although the K-means algorithm (**Table 2**) does not necessarily find the most optimal configuration (i.e. the global minimum for the objective function), the algorithm is guaranteed to terminate (Dasgupta, 2008). Therefore, a different random initialization of the centroids positions in the "Init" step, may lead to a different local minimum for the objective function (different cluster sub-groups of the initial set of *n* instances). Then, every instance from the input set is associated with the closest centroid, creating "temporary clusters" and the centroids' positions are updated with respect to the average distance between them and their associated instance points. The process is repeated until no change in any of the centroids positions occurs (**Figure 4**).

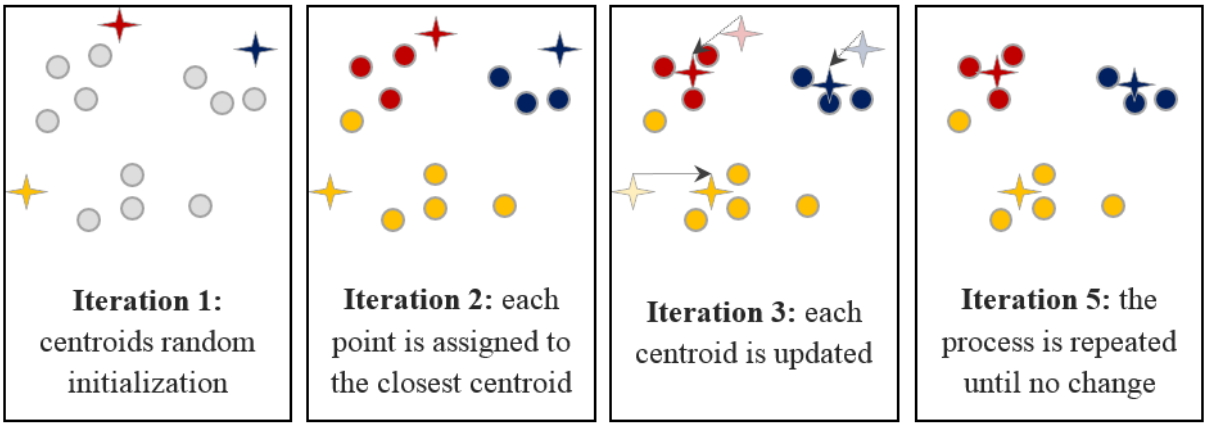


Figure 4: First iterations of K-means algorithm where *k=3* and *n=11*

The number of clusters *k*, may also represent a challenge and it is often dependant on the input dataset or on a later purpose of clustering (why do we need clusters).

3.2. Source code representation and retrieval techniques

This section introduces the techniques used for representing source code artefacts in rich structures, which can be explored for both structural and semantic characteristics. We discuss the power of *Conceptual Graphs* and *Vector Space Model* in expressing source code information and how can we use these concepts to facilitate retrieval. In addition, the *Latent Semantic Indexing* mechanism is presented purely for comparison reasons with the Vector Space Model and we will cover the advantages and disadvantages of each technique later in section (4.2.5). Further, the *Damerau-Levenshtein distance* is an algorithm used in *Arís* in order to overcome some limitations of the Vector Space Model.

3.2.1. Conceptual Graphs

Conceptual Graphs (CG) (Sowa, 1984) are bipartite, directed, finite graphs, where each node in the graph is either a concept node or a relation node (**Figure 5**).

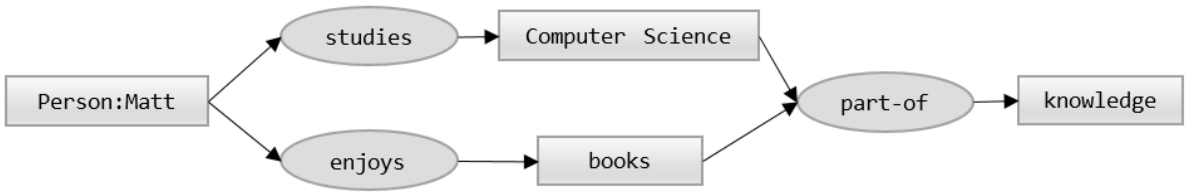


Figure 5: Example of a CG. CGs can be constructed using the standard graphical notation, where concept nodes are represented by rectangles and relation nodes are represented by circles.

Concept nodes in CGs can encode information in various forms, for example states, entities, events, while relation nodes express the interconnectivity between these concept nodes. All nodes in a conceptual graph are either of type *concept* or of *relation*. In addition, each node in a CG is associated with a *referent* value that contains information related to its type (for example, in (**Figure 5**) *Matt* is the referent value for the first concept node). A conceptual graph relies on a so-called *support* (**Figure 6**), which represents the general knowledge base and encodes various rules and syntactic constraints used when constructing a conceptual graph. The support must contain a

set of *concept types* (expressed in a hierarchical structure), a set of *relation types* (with constraints that state which kind of concept types are permitted to connect) and a set of *referent sets* (for each concept type).

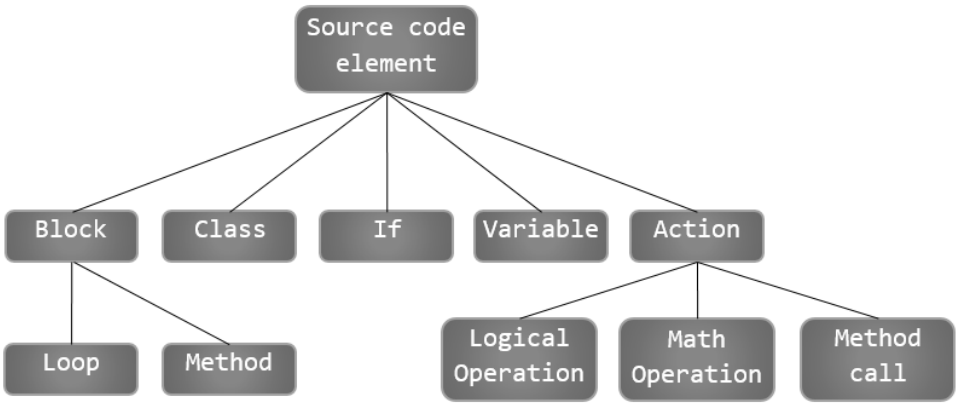


Figure 6: Example of a partial *support* for Conceptual Graphs that express the structure of a piece of source code. The nodes in the *support* represent concept types used in CGs and their inheritance hierarchy is expressed by undirected arrows.

Conceptual Graphs are useful and powerful because of their flexible structure that can be used to represent information in various formats as graphs, which can then be processed and explored using well-known graph-based methods. CG have been successfully used in many retrieval systems (Montes-y-Gómez, et al., 2000) (Mamadolimov, 2012) (Zhong, et al., 2011) as well in the context of source code retrieval (Mishne & De Rijke, 2004). In this project, CGs are used for retrieval of source code artefacts only from a structural point of view, using the *Content vectors* representation (4.3). Using *Analogical Reasoning* structure mapping techniques, the CG matching module in *Arís* (Grijincu, 2013) further explores the information encoded in CGs and the results are integrated with this project as discussed in section (4.4).

3.2.2. Vector Space Model

Vector Space Model (VSM) (Salton & McGill, 1986) is a procedure capable of representing documents as vectors and it is used mainly in Information Retrieval (Manning, et al., 2008). In this model, documents in this context are usually considered textual, but the technique can be applied to various other document types (objects). In this project, we use API calls as words in source code artefacts (documents) as part of the semantic retrieval process (discussed in section (4.2)). VSM treats documents as bags of terms (bag-of-words model) and applies weighting (Ko, 2012) for each

indexed term in order to accomplish document retrieval (with respect to a query, which is also treated as a document).

The main idea in VSM is to represent documents and queries as vectors: $d_1, d_2, \dots, d_n \in D, d_i = (w_{1i}, w_{2i}, \dots, w_{ti})$, where D is the set of all documents (corpus), d_i is a document represented as a vector where each dimension corresponds to a word (term) and *t* is the number of words in the vocabulary (the set of all words). If $w_{ji} \in \mathbb{R}$ is a non-zero value, the *j-th* term appears in the document d_i and this value represents the weight of this term in document d_i (for example, the number of occurrences of a term in the document (**Table 3**)).

Table 3: Example of VSM document indexing using term frequencies

$D = \{$ "Matt likes Computer Science. Andy likes it too.", "Alice likes movies." $\}$

	Matt	*likes*	*Computer*	*Science*	*Andy*	*it*	*too*	*Alice*	*movies*
$d_1 =$	1	2	1	1	1	1	1	0	0
$d_2 =$	0	1	0	0	0	0	0	1	1

VSM has many usage advantages (Manning & Schütze, 1999) including ranked retrieval, term weights by importance, partial matching or simple algebraic model, but also some disadvantages (depending on the context in which VSM is applied) like the assumption that terms are statistically independent, the fact that VSM requires query terms to exactly match document terms (no support for common problems arising in natural languages, for example, synonymy or polysemy (3.2.3)) or the fact that the order in which terms appear in documents is not preserved in the vector representation. Also, VSM's performance is much related with the weighting scheme used for terms in the vector representation (term frequency and TF-IDF are amongst the techniques highly used).

TF-IDF Weighting

TF-IDF (Term Frequency – Inverse Document Frequency) (Salton & McGill, 1986) is a numerical statistic technique that helps assessing the importance of terms in a document with respect to a collection of documents. The model was first used in the classic Vector Space Model and TF-IDF reflects the importance of a term by computing the frequency of the term used in a particular document and, at the same time, how often is the term encountered across all documents.

In the semantic retrieval module of *Aris*, we use TF-IDF weighting scheme for API calls because it provides a good balance between the frequency of an API call in a source code artefact and the occurrence in corpus. Thus, the overall quality of the retrieved source code artefacts will increase because truly important API calls will weigh more when similarity scores are computed (5).

The term frequency (TF) encodes the number of occurrences of the term *t* in document *d*. This simple definition is sometimes augmented in order to prevent problems in VSM, for example, in order to prevent poor representation of longer documents (they might yield low similarity scores when computing a scalar product), term frequency can be divided by the maximum frequency of any term in the document.

The inverse document frequency (IDF) is a measure designed to balance with the term frequency discriminating power (all terms are equally important no matter how frequent a term occurs across the collection of documents) (Manning, et al., 2008). For example, in the English language, the term "the" is very common across all text documents (stop-words), therefore by using only TF weighting, this term will have a high importance in VSM. IDF introduces a mechanism that reduces the importance of such terms by using a factor that grows with the term' frequency across all documents:

$$idf(t, D) = \log \frac{\#D}{1 + \ \#\{d \in D : t \ \in d\}}$$

where *D* is the set of all documents, *#D* is the cardinality of *D, d* is a document and *t* is a term. The formula yields high scores for rare terms across documents and low scores for frequent terms. The TF-IDF is calculated as the product between TF and IDF:

$$tfidf(t, d, D) = tf(t, d) idf(t, D)$$

Cosine Similarity

In VSM, after documents are represented as vectors that encode the importance of terms, similarity scores can be computed between these vectors using measures such as the cosine similarity (Manning, et al., 2008). This is calculated between two input vectors d_1 and d_2, using the dot product (sum of the pairwise multiplied elements) and dividing by the product of the vector Euclidian lengths:

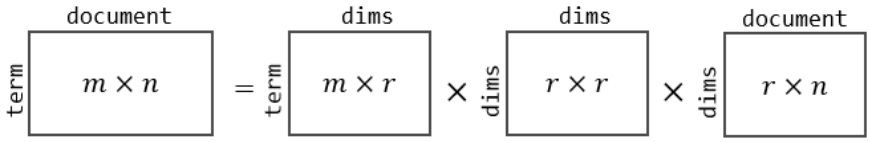
$$simCos(d1, d2) = \cos(\varphi) = \frac{d1 * d2}{||d1|| \, ||d2||} = \frac{\sum_{i=1}^{N} d1_i \, d2_i}{\sqrt{\sum_{i=1}^{N} (d1)^2} \sqrt{\sum_{i=1}^{N} (d2)^2}}$$

The denominator has the effect of normalizing the vectors $d1$ and $d2$ by their length. Because the vectors have non-negative weights (e.g. TF-IDF), the similarity score is always between 0 and 1 (the angle between the vectors is not greater than 90 degrees), with 1 for identical vectors and 0 for absolutely different vectors. In our system, we implemented a modified algorithm of cosine similarity (described later in section (4.2.4), (**Table 5**)) that effectively computes similarity scores between the query and source code artefacts in the case-base. In addition, we will further discuss the advantages and disadvantages of VSM in the context of source code retrieval in section (4.2).

3.2.3. Latent Semantic Indexing

Latent Semantic Indexing (LSI) (Deerwester, et al., 1990) (also known as Latent Semantic Analysis) is a technique for indexing and retrieving documents, capable of overcoming some disadvantages encountered in VSM (3.2.2) (e.g. synonymy and polysemy). Synonymy refers to the problem where two or more different terms that have the same significance. Polysemy refers to the case where a word has multiple meanings in different contexts (Manning, et al., 2008). VSM fails to capture the relationship between terms because it treats each of this term independently (a separate dimension in the vector representation). LSI helps overcome these problems by clustering terms in a number of domains that share the same semantic information. The intuition behind LSI is the fact that terms that are used in the same contexts usually have similar substance; therefore, LSI captures the latent meaning of the terms by taking advantage of the context in which each term is used.

The term-document matrix (TDM) (bag of words model) (Manning, et al., 2008) represents the collection of vector space representations of each document in corpus and has the potential to grow in several hundreds of thousands of rows and columns. LSI uses a matrix decomposition technique called Singular Value Decomposition (SVD) (Gilbert, 2009) in order to reduce the size of TDM by constructing a low-rank approximation of TDM to create domains that share the same conceptual similarity. This decomposition yields three new matrices that have a reduced number of dimensions *r,* generally chosen empirically in the low hundreds:



Computing similarities between documents is done using the Cosine Similarity (i.e. between rows in the approximated matrix). Also, LSI achieves language independence (because of the mathematical abstraction used), which means that the technique can be used to extract semantic content from a variety of structured information.

LSI has some good advantages when contrasted with VSM depending on the context where each technique is used, but it also has some disadvantages (Rosario, 2000): storing the resulting matrix is often more expensive than storing the sparse TDM in VSM; VSM is more robust in terms of efficiency when using techniques like inverted indexing (Luk & Lam, 2007) which allow fast comparison only between the query and a subset of the corpus, as opposed to LSI.

In this section, we discussed the problems that are specifically addressed by LSI, in order to better understand their nature and to conclude whether they apply to API calls in source code documents. We will present the results of our reasoning in section (4.2.5) and we will further justify why we used VSM for semantic retrieval.

3.2.4. Damerau-Levenshtein distance

Because VSM represents source code artefacts as bag-of-words, the order of API calls in documents is lost, but in the case of software implementations, this order is definitely important because a different ordering would yield very different functional results. Damerau–Levenshtein distance can be used to express this information, therefore we can achieve a more accurate retrieval system by applying this technique and our argumentation will be presented in section (4.2.4).

Damerau–Levenshtein distance, also known as the edit distance, represents a technique used to calculate the "distance" between two arbitrary vectors or strings. The distance computes the minimum number of single edits needed to transform one vector into another using four basic operations: insertion, substitution, deletion or transposition of two adjacent characters (Damerau, 1964). The algorithm uses dynamic programming (Vazirani, et al., 2006) by dividing the general problem into sub-problems and by storing the results of these sub-problems in a matrix, C. Each

element $C[i,j]$ represents a sub-problem and the final goal is to compute $C[m,n]$, where m,n the lengths of the two input vectors. Each matrix element is the minimum of all "close" sub-problems:

Table 4: Damerau–Levenshtein distance pseudo-code

```
int DamerauLevenshteinDistance(int[] source, int[] target)
{   for (i = 1; i <= source.Length; i++) C[i,0] = i;
    for (j = 1; j <= target.Length; j++) C[0,j] = j;
    for (k = 1; k <= source.Length; k++)
        for (l = 1; l <= target.Length; l++)
        {
            if (source[k] == target[l]) cost = 0;
            else cost = 1;
            C[k, l] = Min( C[k-1, l] + 1,      //deletion
                           C[k, l-1] + 1,      //insetion
                           C[k-1, l-1] + cost) //substitution
            if (k>1 && l>1 && source[k] == target[l-1] && source[k-1] == target[l])
                 C[k, l] = Min( C[k,l],             //previous
                                C[k-2, l-2] + cost) //transposition
        }
    return C[source.Length, target.Length];
}
```

The algorithm complexity is *O(mn)*. This method for calculating the minimum distance is widely used in Natural Language Processing (Manning & Schütze, 1999), Information Retrieval (Manning, et al., 2008), fraud detection and DNA matching.

Summary

In this chapter, we discussed the theoretical foundations of *Arís* and we briefly outlined some of their advantages and disadvantages. In the following chapters, we will constantly refer to these sections as we describe our proposed solution.

4. Proposed Solution

In this chapter, we describe the architecture of our proposed solution for the retrieval module in *Arís* (1.1). The main purpose of our framework is generation of new specifications using previous knowledge recorded in memory. In order to formally verify correctness of software implementations using formal methods, we propose a Case-Based Reasoning approach that works as follows: for a given unverified implementation as input, similar implementations are retrieved from case-base by exploring structural and semantic similarities between the query and memory source code artefacts. The results have an associated formal specification and the top ranked source code artefacts can potentially be used to generate a new specification for the query. Effectively, the specifications are transferred to the input implementation until a successful knowledge transfer is achieved (the generated specification is verified using existing formal method tools). Finally, if the previous step is successful, the new solution is stored for further use.

In this project, the main focus is on the retrieval process of CBR because this directly affects the rate of success on our proposed overall goal (formal verification of implementations bye ruse of existing specifications). In addition, a good source code retrieval system can be used for a number of other purposes, as described in section (1.2). Therefore, this framework primarily aims to retrieve similar source code artefacts with a given input software implementation and potentially generate a formal specification for the given partial or full implementation. Also, we seek to automate the verification process of the new generated specification and to allow the user to optionally interfere in the process.

In the following sections, we give a detailed description of our system. In section (4.1), we present the case base of source code and formal specifications. Section (4.2) describes the semantic retrieval process using Vector Space Model theory for representing source code artefacts, weighting API calls and computing similarities between these representations. In addition, we critically analyse semantic retrieval by outlining the advantages and disadvantages of the proposed method, in comparison to similar work described in the *Related Work* chapter. Section (4.3) presents the Conceptual Graph representational structure and although this is a powerful method of expressing source code, the structural retrieval process in this project explores only the structural characteristics of source code, by the use of content vectors. Then, we describe a method for comparing content vectors as well a technique that facilitates efficient case comparison and again

we critically analyse the proposed model for structural retrieval. Finally, for the retrieval process, in section (4.4), the structural and semantic results are combined and the top ranked documents are further analysed with the Conceptual Graph-Matching module in *Arís*. Section (4.6) describes how we can practically evaluate the newly generated specifications and section (4.7) briefly presents a user interface for the retrieval system.

On a high-level view, the heart of our framework is the Case Based Reasoning (3.1.3) discipline and our main focus is the retrieval phase (**Figure 7**) of CBR.

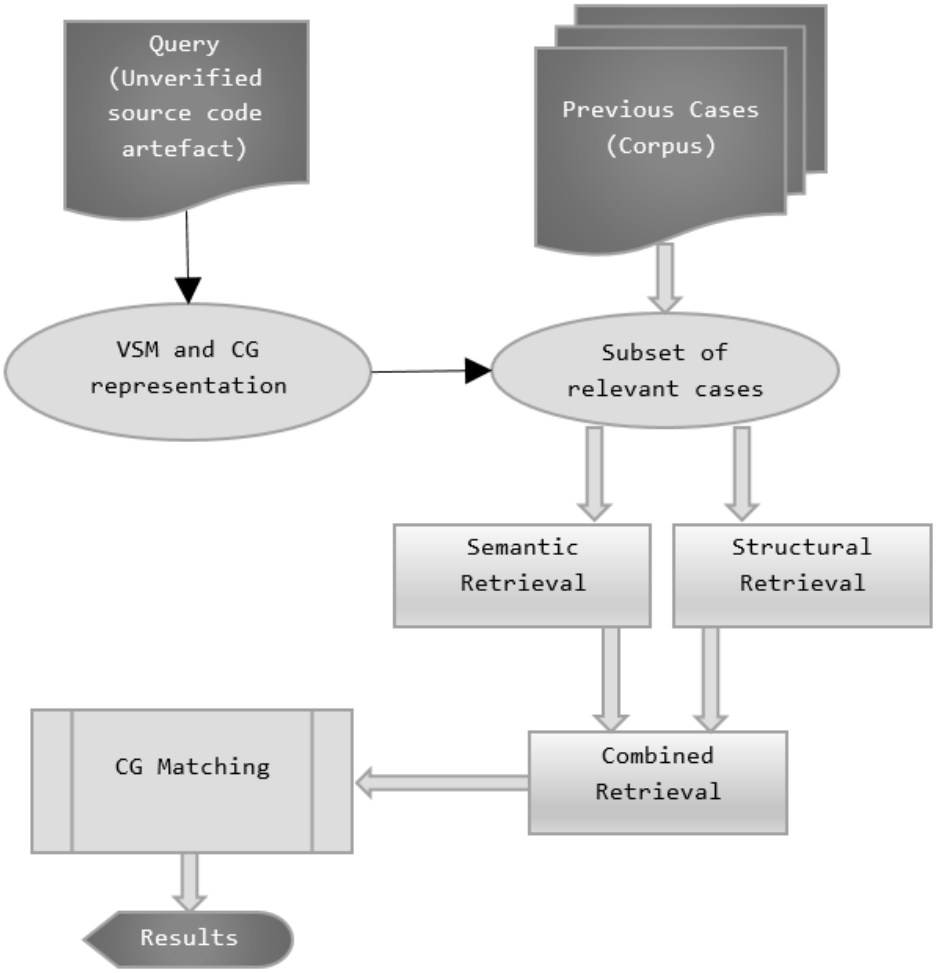


Figure 7: Diagram of the Retrieval process in *Arís*. A query is encoded using Conceptual Graph and Vector Space representations that will be used to compute structural and semantic similarity. These representations allow selection of a subset of relevant cases, which are inputs for structural and semantic retrieval. Because these two processes are designed to be independent and can potentially run in parallel, we combine their results and apply Conceptual Graph-Matching on the top ranked source code artefacts.

Our current model is implemented to perform retrieval for C# source code and corresponding Spec# (Rustan, et al., 2010) specifications. In addition, because we wanted a robust Case-Base (**Figure 8**) model that can handle requests from a retrieval algorithm with respect to a certain type of code artefact, our cases (source code and formal specifications) are organized by their type (either a method, a class or a full application).

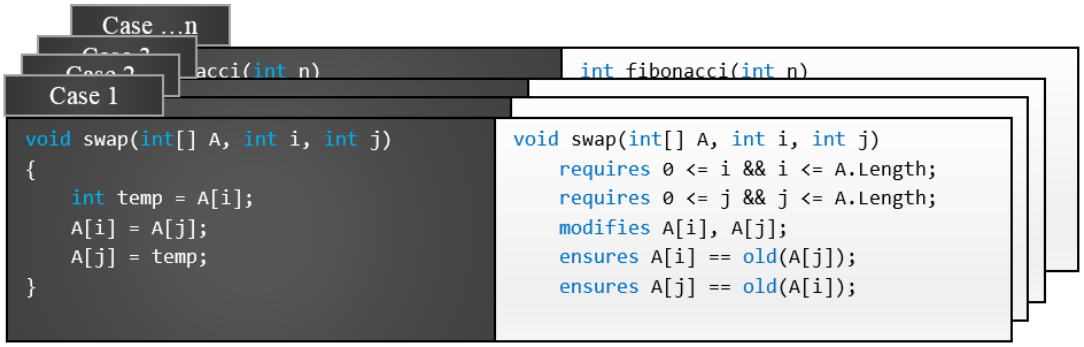


Figure 8: The Case Base contains source code artefacts and corresponding formal specifications

The proposed retrieval algorithm works exclusively on source code artefacts (i.e. for a given query in the form of source code: method, class or a collection of classes) encountered in the Object Oriented Programming paradigm. Therefore, our model is closely related in terms of functionality with other code retrieval systems presented in chapter (2) - Related Work, but differs from them in the sense that the retrieval process is designed to consider both semantic (4.2) and structural (4.3) characteristics of source code. As a comparison note between structural and semantic retrieval processes, they are independent (although their design is complementary) from each other and follow roughly the same pattern: source code representation, selecting a subset of documents from the corpus that are semantically/structurally relevant to the query and defining a similarity function between representations.

In *Arís*, we continue the canonical Case-Based Reasoning process after the Retrieval step is completed, with the Reuse (2), Revise (3) and Retain (4) phases, by transferring our knowledge (formal specification) from the retrieved problem(s) (best ranked source code artefacts) to the current problem (**Figure 9**), evaluating the mapping and optionally retain the new resolved case.

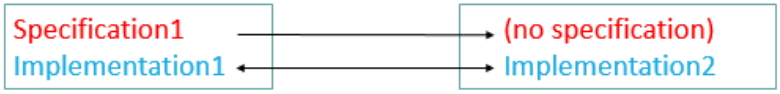


Figure 9: Knowledge transfer for reusability of formal specifications

4.1. The Case Base of Source Code and Specifications

Similar to other Source Code Retrieval systems discussed in the Related Work section, in order to assess the correctness of our retrieval algorithm and to perform knowledge transfer using formal specifications on a larger set of implementations, we needed a repository of software artefacts. From an architectural point of view, the Case Base Memory is a collection of past cases, which aggregates source code artefacts optionally associated with the corresponding formal specification for that implementation.

Because our framework was designed to retrieve C# source code artefacts, we decided to create our corpus of documents from managed compiled assemblies (i.e. only managed assemblies: Dynamic-Link Libraries[4] (DLLs) or Executables[5] (EXEs)). The main advantages of this approach are:

- Compactness of the corpus is achieved by storing only compiled assemblies, without any associated resources and source files that may be needed to compile the software.
- Documents are with high probability finished work (at least free of compilation errors).
- Documents contain Common Intermediate Language (CIL) (Microsoft, 2012) code, which can be converted back into C#, VB, F#, etc. documents using third party libraries (**Figure 10**). This means that the code retrieval algorithm can be used independently for a wide range of programming languages that are translated to CIL.
- Fully qualified API calls can be easily extracted directly from CIL code.

Some disadvantages of this approach are:

[4] http://msdn.microsoft.com/en-us/library/windows/desktop/ms682589.aspx
[5] http://msdn.microsoft.com/en-us/library/windows/desktop/aa368563.aspx

- Reconstruction of the original code is relatively costly from a computational point of view (from our experiments, 0.05 seconds on average to decompile a method from a sample of 30,000 methods).
- All comments in the original source code are lost and code reconstruction may not yield the exact original code (i.e. the decompiled code has the same functionality as the original code, but the coding style may be different).

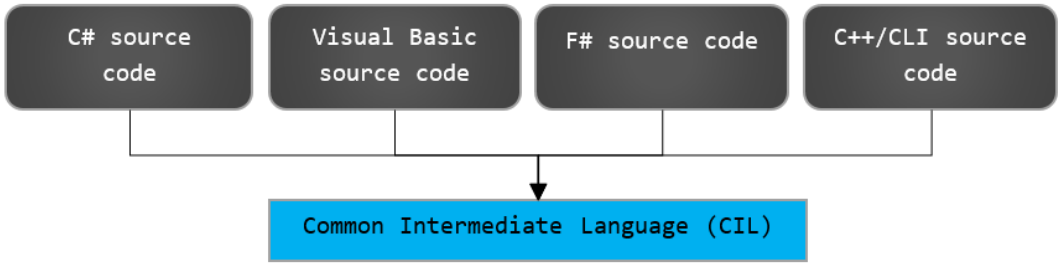


Figure 10: A variety of programming languages are translated to CIL and can be decompiled (reconstructed) into the original code using third party libraries (ICSharpCode, 2012)

One of the primary use cases of *Arís* is knowledge transfer from a small set of verified implementations to a larger set of unverified implementations. However, we were able to find only a small set of verified implementations (102 formally specified classes which contain 249 methods that are associated with a full or partial specification) using Spec# formal method, which consists of various test suits, from the Spec# project source control web-page (Rustan, et al., 2010). While the number of verified software artefacts is growing because of active research in this area, the rate of growth is still relatively modest.

To better test our source-code retrieval process, it was decided to augment the corpus with additional un-verified implementations, because they represent a more realistic test of the potential efficacy of the *Arís* retrieval system. Our corpus of software projects was thus extended with a large number of open source programs downloaded from Codeplex (Microsoft, n.d.), GitHub (GitHub, Inc., n.d.), SourceForge (Slashdot Media, n.d.) and NuGet (Outcurve Foundation, n.d.) and consists of 2,191 software applications, which contain 175,291 classes and 2,033,623 methods. It is worth noting that only 925,014 methods and 147,728 types contain API calls or calls to third party libraries.

An *Abstract Factory* (Gamma, et al., 1994) is responsible of creating cases (**Figure 11**) (`CodeArtefact` objects), by extracting all the information needed (sections 4.2 and 4.3) to accomplish semantic and structural retrieval (we call this step *knowledge acquisition*). The abstract factory mechanism is needed because it provides a way to separate the case objects creation from their usage in the retrieval process.

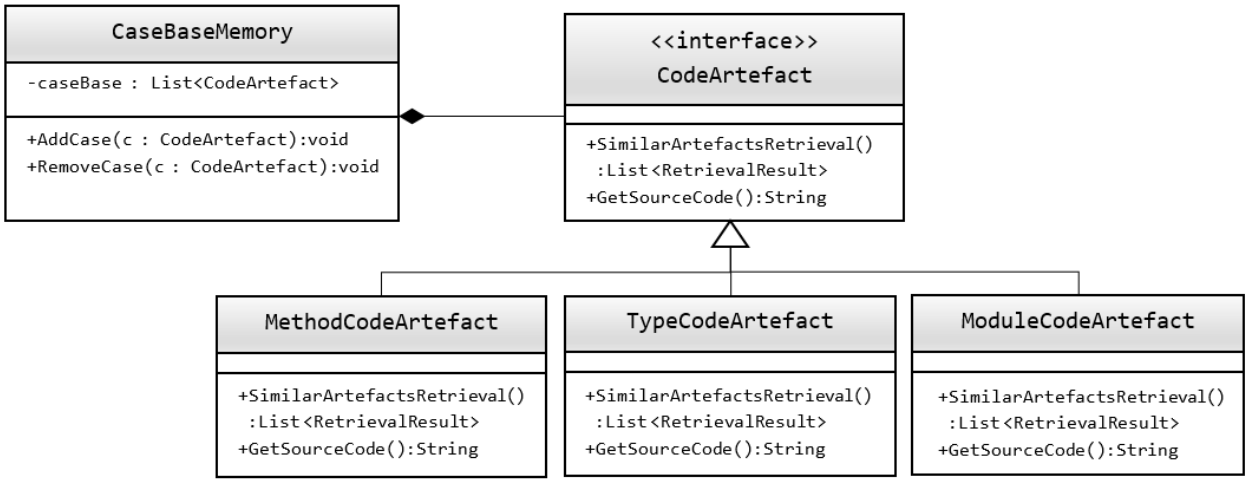


Figure 11: Class diagram of the Case Base model

4.2. Semantic Retrieval

In *Arís,* we desire a retrieval model that is able to detect similarity between cases (source code artefacts) that may have slightly different implementations while conforming to the same high-level specification. This means that these cases must have the same semantic meaning, but with a possible different structure (Biggerstaff, et al., 1993).

Similar to the work of CLAN (2.2) (McMillan, et al., 2012) and other code search engines (Chatterjee, et al., 2009) (Grechanik, et al., 2007), we rely on API calls as semantic anchors, because the meaning of such a construct is precisely defined and the programmer's intent when implementing high-level requirements can be expressed by combinations of such notions. The use of API calls is justified by real world software applications, where existing implementations that solve a given task are often used (for example, if a string replacement is needed, one would most likely use the API call `String.Replace()` method, rather than implementing another solution). These API calls can be considered words in documents (source code) and a similarity score between

documents may be computed based on the number of shared words (Mizzaro, 1998); this fact links our system to techniques used in Information Retrieval.

Further, in *Arís* we extend the usage of API calls as terms in documents, by allowing terms to be function calls to third party libraries, thus not only function calls to the dependent programming framework (e.g. API calls to classes and interfaces exported by the Java Development Kit for Java implementations or to the .NET Framework for C# implementations). This can considerably affect the size of the API calls vocabulary (the set of all possible API calls) (i.e. from our corpus of 2,191 software projects, we extracted 185,303 API calls made exclusively to the .NET framework and 127,398 API calls made to third party libraries). This extension may potentially have a negative impact on the computational time needed in retrieval, but in the same time, it can improve the quality of the retrieval results (with a richer API calls thesaurus, we recognize a larger set of terms in documents).

4.2.1. Extracting API calls

In *Arís,* API calls are indexed as words in source code files; therefore, we need to evaluate the full definition of an API call, with the exact source of the class that exports that call, in order to avoid name conflicts. For example, the use of an API call from a `Timer` class can be ambiguous because both `System.Threading` and `System.Timers` namespaces contain a `Timer` class).

In .NET Framework, namespaces (Gunderloy, 2002) are organized in a tree-like (hierarchical) structure (**Figure 12**), with the System node as root. The absolute root of the namespaces tree is `global` (e.g. `global::System` refers to the `System` namespace). It is worth noting that namespaces can start with something other than the `System` namespace (e.g. `Windows, Microsoft`) and third party libraries as well as client code usually have other defined structure for namespaces (e.g. **Figure 13**).

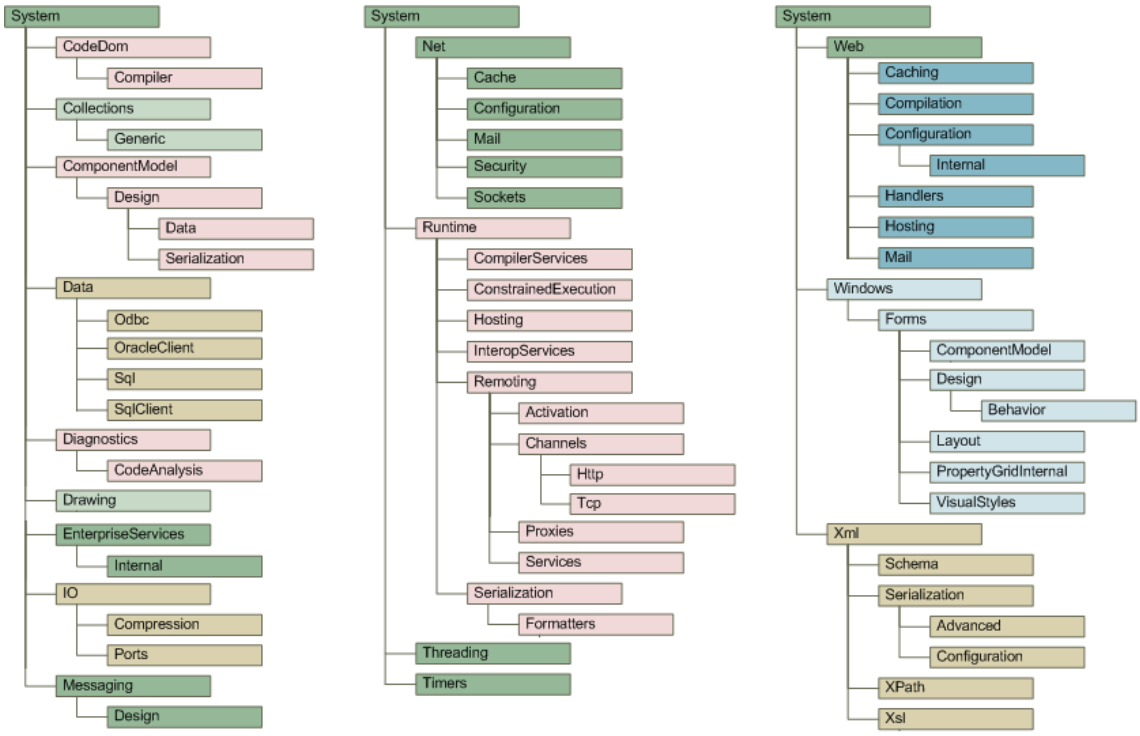


Figure 12: Hierarchical organization (`System` is the parent of all nodes in the tree, split into three sub-trees for space reasons) of a subset of `System` namespace in .NET 2.0 Framework[6]

The task of extracting API calls from source code files can be difficult, mainly because of the anatomy of such a file (**Figure 13**). The C# language allows the use of both fully qualified statements with the corresponding namespace and declaring namespaces with the `using` keyword at the beginning of the source file. The second usage implies finding a correct mapping between the namespace and the API class (e.g. in (**Figure 13**), mapping line 1 to the usage of the String class on line 10). However, in *Arís* we take advantage of the fact that the corpus of documents is an aggregation of managed compiled assemblies, containing CIL code, therefore such a mapping is not needed because the CIL language always encodes fully qualified API calls.

[6] Modified image from: http://grounding.co.za/blogs/brett/archive/2007/07/15/net-2-0-framework.aspx

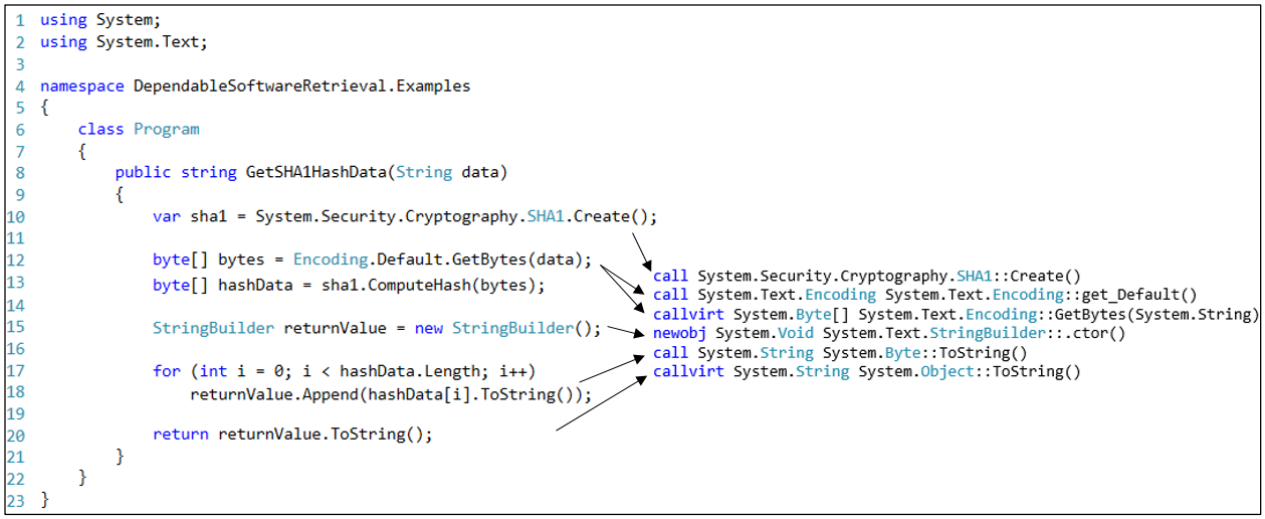


Figure 13: C# source code mapped to CIL code example: The `StringBuilder` class is not fully qualified with the `System.Text` namespace on line 15. On line 10, the `SHA1` class is fully qualified. These C# language constructs are translated in CIL language with their corresponding namespace

From a practical point of view, we loaded CIL instructions from our managed assemblies using the `Mono.Cecil` library (Evain, n.d.) and we inspected a set of `OpCodes` (Microsoft, 2012) representations of the CIL instructions that express API calls:

- `OpCodes.Call:` Calls the method indicated by the passed method descriptor.
- `OpCodes.Calli:` Calls the method indicated on the evaluation stack (as a pointer to an entry point) with arguments described by a calling convention.
- `OpCodes.Callvirt:` Calls a late-bound method on an object, pushing the return value onto the evaluation stack.

4.2.2. Indexing API calls

The previous extraction process produced a very large number of API calls in our thesaurus. Therefore, the retrieval algorithm can easily reach computational infeasibility without applying a proper indexing technique. Because at the core of semantic retrieval is VSM (3.2.2), it is thus imperative to design a method that trims unnecessary computations in our retrieval process, such that we can achieve similar response times to other retrieval systems (curse of dimensionality (Powell, 2007)) (e.g. the use of Latent Semantic Indexing (3.2.3) helps overcome this problem, but with possible trade-offs).

In VSM, documents are represented as vectors of real numbers and the similarity between two documents is computed using the *simCos* distance measure defined in section (3.2.2), which is a division of the dot product of those two vectors and the product of the vector Euclidian lengths. This fact points out that if the dot product of the two vectors is zero (i.e. the vectors share no API call), then the similarity score between the two documents is zero and there is no point in computing the distance measure in the first place. This can be achieved by computing the distance measure only with the set *U* of documents that share common words (API calls) with the query, which is a subset of all documents in the corpus. Therefore, the semantic retrieval results should only contain results that share at least one API call with the query, not only for performance/speed reasons, but also because the results should contain only relevant information that relates to the query.

In order to obtain the set of documents that share words with the query, our first intuition was to organize API calls for indexing in a tree like hierarchy (an extension of the tree in (**Figure 12**)) (this was the model for our first implementation of the indexing mechanism, but in the version where the system was evaluated, the second approach described later in this section was used). For each leaf in this tree associated with an API call, there will be maintained a set of documents (source code artefacts) that use that particular API call. The nodes of the tree would be extracted from the textual representation (e.g. `System.Object::ToString()`, `System.Text.Encoding::getBytes(System.String)`) of API calls, by splitting the definition using the "**.**" and "**::**" operators as delimiters. This structure would then allow us to easily access each document that used every API call. The time needed to access this tree structure (*T*) would be *O(height(T))*, where *height(T)*, the height of the tree *T* is usually a small integer (determined by the longest namespace i.e. with the maximum number of the "**.**" and "**::**" operators in its definition. For example, the namespace `System.Object::ToString()` would yield nodes `System`, `Object` and `ToString` in *T*). The set *U* of documents that share API calls with the query is obtained by reunion of all the sets associated with each API call in the query document (usages). The main advantage of this approach is the fact that a higher level of fuzziness in retrieval can be achieved (a main goal set for the system presented in section (2.1) (Mishne & De Rijke, 2004)), by obtaining the set *U* from union of sets associated with nodes from a lower level (for example, leafs parents in (**Figure 14**)). Furthermore, this model can be augmented with support for class inheritance, by maintaining for each node of type class (e.g. `Encoding`, `Object`, `StringBuilder`), a separate list of pointers to nodes in this tree, which derive from that class.

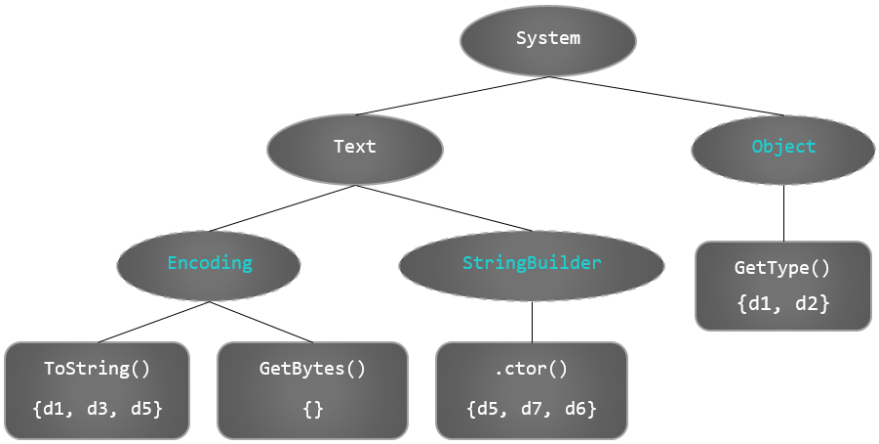


Figure 14: Example of a sub-tree of *T* containing API calls (leafs) with associated documents that use that API call. The set of documents *U* obtained with respect to a query *q* that contains API calls `System.Text.Encoding::ToString()` and `System.Object::GetType()` will be in this case `{d1, d2, d3, d5}`. However, if the query contains only API call `System.Text.Encoding ::GetBytes(),` the set *U* will be empty and a weaker constraint can be imposed, for example, at `Encoding` node level, *U* will be `{d1, d3, d5}` the reunion of all sets associated with child nodes.

Our second approach to obtain the set *U* of documents that share common API calls with respect to the query uses a faster and stricter model, bonded to our primary goal in *Arís*: reuse of formally specified implementations. This goal can be achieved with a lower level of retrieval fuzziness and a faster implementation, by maintaining API calls in a Hash Set data structure using their textual representation as keys (inverted indexing technique (Manning, et al., 2008)). In addition, each API call has an associated set of document usages, similar to our previously described model, which are used to compute the set *U* as the reunion of sets associated with API calls encountered in query.

4.2.3. Weighting API calls

The weighting process of words in documents represents an important task because this will directly affect the quality of the retrieved documents. The authors of CLAN (2.2) (McMillan, et al., 2012) observed that some API calls are likely to be shared between programs that implement different requirements (for example, their experiments have shown that over 60% of 2080 Java programs use `String` objects and 80% contain collection objects). This fact suggests that some API calls are more important than others and their relevance is dependent on the corpus of documents, as well on the document that contains an API call.

We use the TF-IDF (discussed in section (3.2.2)) weighing scheme to overcome these problems and from a practical point of view, we compute IDF scores for each API call after the knowledge acquisition step is finished (i.e. extraction of API calls and the corpus of documents is complete). The TF score of an API call *a* in a document *d* is simply the number of occurrences of *a* in *d* and we encode this information along with the method code artefact (**Figure 11**) using a Hash Map with API calls as keys and TF scores as values.

4.2.4. Computing source code artefacts similarities

In our framework, a source code artefact might be either a method, a class or collection of classes. However, API calls are used in the implementation of methods, which in turn are contained in a class. Therefore, it makes sense to consider only methods as documents in our corpus and to express classes as documents based on the collection of methods that appear in it.

As stated earlier, in VSM documents are represented as vectors of real numbers, where each element in the vector corresponds to the weight of an API call in that document and the size of our API call thesaurus can be very large (311,696 API calls extracted – (4.1)). It is thus impractical to store such a large vector for each of our source code artefacts. Instead, we assign unique integer identifiers to each API call in our thesaurus and we implement a modified version of the classical Cosine Similarity (3.2.2) to calculate the distance between documents (**Table 5**).

Table 5: Modified algorithm (from (Thomee, et al., 2010)) for computation of the Cosine Similarity between two source code artefacts (represented by the `APICalls` class that contains a sorted collection of API calls by their `ApiIndex` i.e. its corresponding unique identifier). The dot product between the two vectors is calculated in the main `while` loop and the result is divided by the Euclidian norms `len1` and `len2` of the two vectors, which are pre-computed in the knowledge acquisition phase.

```
double CosineSimilarity(APICalls calls1, double len1, APICalls calls2, double len2)
{
    double distance = 0.0d; //the final distance between documents calls1 and calls2
    int ac1Index = 0; //an index to the current position in the calls1 vector
    int ac2Index = 0; //an index to the current position in the calls2 vector

    while (ac1Index < calls1.Count)
    {
        if (ac2Index == calls2.Count) break;

        //compare API calls indexes
        if (calls1[ac1Index].ApiIndex < calls2[ac2Index].ApiIndex)
            ac1Index++;
```

```
        else
            if (calls1[ac1Index].ApiIndex > calls2[ac2Index].ApiIndex)
            {
                do
                {
                    ac2Index++;
                    if (ac2Index == calls2.Count) break;
                }
                while (calls1[ac1Index].ApiIndex > calls2[ac2Index].ApiIndex);
            }
            else //equal identifiers i.e. indexes in vectors
            {   //add TFIDF for these API calls in order to compute the dot product
                distance += calls1[ac1Index].TF * calls1[ac1Index].IDF *
                            calls2[ac2Index].TF * calls2[ac2Index].IDF;
                ac1Index++;
                ac2Index++;
            }
    }
    distance /= Math.Sqrt(len1) * Math.Sqrt(len2); //divide by norm: euclidian length
    return distance;
}
```

Using the *Cosine* Similarity algorithm described in (**Table 5**), distance scores between the query and a subset *U* of the entire corpus (obtained as described in section (4.2.2)) are used to rank results. Our design of the Cosine Similarity algorithm is also motivated by the fact that source code artefacts contain a fairly low number of API calls (usually less than a few hundred), thus the dot product is computed efficiently between two documents compared by their vector space representation.

The same algorithm also works for source code artefacts other than methods (i.e. classes and collection of classes), by treating these documents as containers of methods and aggregating every API call in every method used in its definition. For example a source code artefact of type class, we obtain the corresponding set of API calls as the union of all uses in all methods by summing the TF scores: if three method in this class contain the same API call *a* with TF scores 1, 3 and 4, the TF score for *a* in this class will be 8).

Because in VSM (3.2.2) the order of appearance of terms in a document is not preserved, we use *Damerau-Levenshtein distance* to express the differences between the arrangements of API calls in source code artefacts. We employ this technique because in case of implementations, a different order of API calls usages can yield totally different functional results, as the implementation most likely corresponds to a different specification each time this order is changed

(e.g. if the API calls `String.ToString()` and `String.ToUpper()` are interchanged, the corresponding implementations will achieve different functional results). In order to compute the similarity between two documents (source code artefacts) from this point of view, we utilize the Damerau-Levenshtein algorithm presented in section (3.2.4) for two vectors containing unique API call identifiers (in the order they appeared in the source code artefact), as inputs. The Damerau-Levenshtein distance between two vectors of integer identifiers returns the number of minimum single edits needed to transform one vector into another (i.e. Damerau-Levenshtein distance $d \in \mathbb{N}$, with $d \leq Max(m, n)$), therefore, in order to compute the similarity score (i.e. $simDL \in [0,1]$) between the two documents we use:

$$simDL(v1, v2) = 1 - \frac{\text{DamerauLevenshteinDistance(v1, v2)}}{Max(\#v1, \#v2)}$$

where $Max(\#v1, \#v2)$ is the maximum between the lengths of the two input vectors $v1$ and $v2$. The $DLsim$ function will yield scores of one if the vectors are identical and scores of zero if the vectors are completely different.

Finally, the overall semantic similarity score between two source code artefacts is a linear combination of $simCos$ and $simDL$ similarity scores:

$$sim_{semantic}(d1, d2) = w_{cos} * simCos(d1, d2) + w_{dl} * simDL(d1, d2)$$

where w_{cos} is a weight factor for the Cosine similarity and w_{dl} is a weight factor for the Damerau-Levenshtein similarity score, such that $w_{cos} + w_{dl} = 1$. In our implementation of the proposed model, w_{cos} was set to 0.7 and w_{dl} was set to 0.3, because using these values, the best results were achieved in our preliminary experiments, as discussed in chapter (5). Different values for w_{cos} and w_{dl} can be assigned, depending on the type of source code artefact (method, class, collection of classes) used in the retrieval process.

Until this point, we discussed the design of semantic retrieval, which is an independent process (i.e. it is not dependent for example on structural retrieval) in the retrieval module of *Arís*. In addition, an efficient indexing technique for API calls was presented in order to select relevant documents relative to the query, which can then be compared using the cosine similarity technique. Moreover, we analysed the importance of API calls order in source code and we proposed using the Damerau-Levenshtein distance to express this information when comparing implementations.

The following section outlines the differences between our proposed semantic retrieval solution and CLAN (presented in the related work) and argues the validity of our overall architectural design.

4.2.5. Critical analysis of semantic retrieval

In *Arís,* we accomplish semantic retrieval using the Vector Space Model (3.2.2). In contrast, the primary technique used by CLAN system (presented in section (2.2)) for computing similarity scores between applications is *Latent Semantic Indexing* (3.2.3).

LSI's potential and value is still an active area of research (Manning & Schütze, 1999) and has been primarily used in Natural Language Processing and text retrieval. As stated (section (3.2.3)), LSI deals with issues of non-independence of terms in the thesaurus, i.e. synonymy and polysemy. However, in the case of source code retrieval where API calls are considered terms, the problems of synonymy and polysemy slightly change. API calls have precisely defined semantics (i.e. an API call has only one functional role) which means that the polysemy problem of an API call is somewhat non-existent. That is, the API call itself does not change in meaning, though its use within the local context may differ. Also, in a perfect OOP environment where code reuse is a central concept, there will be no two functions that implement exactly the same requirements, yet, software developers still may be implementing functionalities that already exist in some API call, because another (more efficient) implementation is needed or there is simply no knowledge about an existing API call. Therefore, we conclude by stating that the polysemy problem in the case of API calls still exists but is far less prevalent than in natural language, especially when the thesaurus aggregates only API call terms from the programming framework (e.g. Java Development Kit or .NET Framework, where the probability of having two API calls that have the same requirements is low).

The classic VSM is known to be computationally infeasible when applied on a very large collection of documents with large thesaurus and LSI addresses this issue by performing dimensionality reduction using SVD. However, in *Arís* we addressed this problem by indexing API calls and source code artefacts (4.2.2) in an efficient data structure (inverted indexing (Luk & Lam, 2007)) that facilitates fairly good retrieval response times (chapter (5)).

As a last argument that motives our choice in using VSM for semantic retrieval is the main purpose of *Arís* – reuse of formally specified implementations. This can be achieved with a more rigid retrieval mechanism i.e. the best ranked results, share some of the same API calls with the query, which means that there is a higher chance of successfully reapplying formal specifications on unverified implementations in the knowledge transfer phase. This is in contrast to where these results can share API calls that are not exactly the same but are assigned to the same domain (e.g. API calls like `System.Math.Min()` and `System.Math.Max()` could be assigned in the same domain by the LSI algorithm. Therefore, a plausible scenario for such a retrieval model is: for a query that contains the `System.Math.Min()` API call, amongst the best results can be a software artefact that contains the `System.Math.Max()` API call. However, it is clear that these two implementations implement different requirements (because of their use of very different API calls), thus the chance of successfully verifying the query with the formal specification of the retrieved source code artefact, is potentially lower).

4.3. Structural Retrieval

Another technique used in *Arís* as part of its retrieval model is *Structural Retrieval*. This method focuses on structural or topological characteristics of the source code. As with *Semantic Retrieval* (4.2), we seek retrieval at various granularity i.e. methods, classes or collection of classes. In addition, this process can be run independent of the semantic retrieval process because at this stage, we use a different representational structure for source code and because different source code artefacts from the case base are structurally similar to the query. After this process finishes, the results are combined with the results from semantic retrieval (presented later in section (4.4)).

In order to express the source code topology, we rely on the structure derived from *Conceptual Graphs* of source code. This structure was successfully used by (Mishne & De Rijke, 2004) and their work was inspirational for *Arís* both in terms of retrieval and conceptual graph matching (2.1). The main advantage of using Conceptual Graphs is the fact that its versatile structure enables us to explore both the semantic content and structural properties of source code using graph-based techniques. Another advantage of CGs is the fact that its syntax is not as verbose as the alternate Abstract Syntax Trees and Parse Trees (Neamtiu, et al., 2005), which contain much more detail about the original source code document, therefore it enables a more flexible retrieval mechanism.

In *Arís*, the retrieval module focuses on exploring the shallow structural properties of the conceptual graph representation, which enables fast ranking of relevant source code artefacts with respect to a give query. After such a ranking is retrieved, the subsequent *Source Code Matching* module (Grijincu, 2013) further refines this ranking by performing a more expensive but deeper analysis between conceptual graph representations.

4.3.1. Conceptual graph support for source code

In the *Source Code Matching* module of *Arís,* (Grijincu, 2013) defines a taxonomy that allows the construction of conceptual graphs from source code. The CGs formalism was adapted for C# source code documents in order to perform a proof of concept. A CG *support* is composed by a set of concept types (**Table 6**) defined in a hierarchical structure (a partial hierarchy was described in section (3.2.1), (**Figure 6**)).

Table 6: The set of concept types defined in the CG formalism.

Concept Type	*Summary*
AssignOp	An assignment of a value to a field or variable
Block	Set of concepts that are logically grouped together (e.g., code that is inside {…})
Class	Declaration or definition of a class
CompareOp	Binary comparison (e.g. >=, ==, !=, etc.)
Enum	Declaration of an enumerated set of values
Field	Declaration of a variable in a class
If	A conditional branching statement
LogicalOp	Binary logical operation (e.g. &&, ||, etc.)
Loop	Iterative process that depends on a condition
MathOp	Mathematical operation (e.g. *,+,-, etc.)
Method	Declaration or definition of a function part of a class
Method-Call	Method invocation
Namespace	Defines a scope that can contain one or more classes
Null	Null reference
String	Constant textual entity
Switch	Conditional statement that has multiple branches
Try-Catch	*Try* block followed by *catch* clauses

Variable	Entity declared in an implementation that holds values during execution

The set of possible relation types that are part of the defined CG taxonomy in (Grijincu, 2013) are presented in (**Table 7**), without the full description of the specification of which concepts nodes they can connect.

Table 7: The set of relation types defined in the CG formalism.

Relation Type	*Summary*
Condition	Conditional statement within an *If* clause
Contains	A concept that contains or depends on another concept
Defines	Block that gives a definition of a namespace
Depends	Block that depends other namespaces
Parameter	A parameter in a method definition
Returns	Return statement of a method

4.3.2. Content vectors

Content vectors are a type of feature vectors defined in (Gentner & Forbus, 1994) as an encoding mechanism for structured representations, which can be used to entail if the corresponding structural representation are similar by using the dot product between two such vectors. In order to encode the structural properties of source code, we used Conceptual Graph representation extracted by (Grijincu, 2013), together with content vectors extracted by this module, as further described here.

Given a set of *functors* (i.e. relations, connectives, functions, attributes, etc.) that were used to describe the structural content of documents in a memory, a content vector is an *n*-tuple of natural numbers (i.e. for a particular document d from memory, its content vector is: $v_d = (i_1, i_2, \dots, i_n), i_k \in \mathbb{N}$) where each component i_k corresponds to a particular element from the given set of functors. Also, each component i_k is equal to the frequency of the corresponding functor in a document d.

In our framework, we used the union of the concept types and relation types of a Conceptual Graph taxonomy (4.3.1) as the set of functors, which is the necessary basis for the creation of

content vectors. Therefore, for each node type defined in (**Table 6**) and (**Table 7**) there exists a content vector index which corresponds to the number of occurrences of that type of node in the conceptual graph representation of a source code artefact (e.g. (**Figure 15**)). Although content vectors might seem similar to the vector space representations used for semantic retrieval, they are encoding only structural information about the source code (as opposed to semantic information expressed by API calls), therefore a different indexing and retrieval mechanism is needed in order to evaluate structurally similar documents relative to a given query.

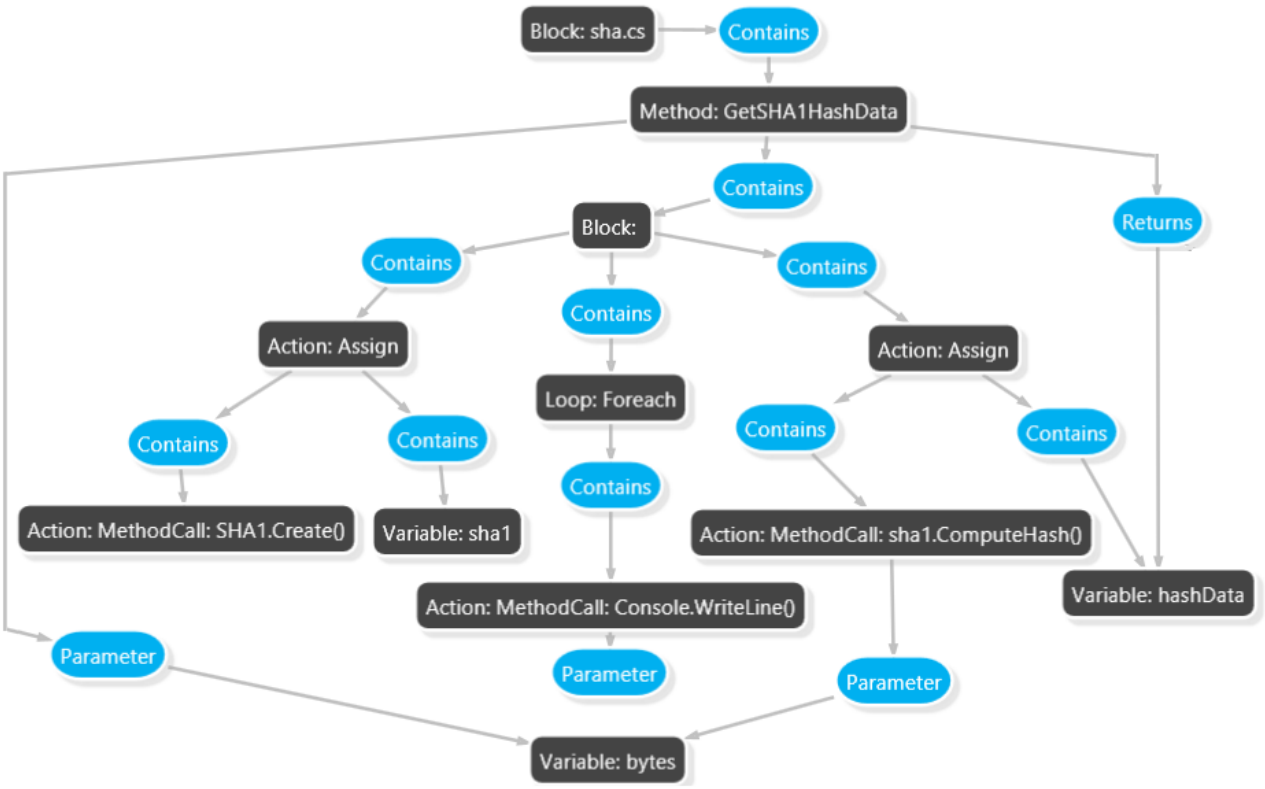


Figure 15: Example of a CG constructed from the source code artefact presented in (**Table 8**), visualized using the Conceptual Graph Visualizer tool from (Grijincu, 2013).

Table 8: Example of method source code artefact

```
public byte[] GetSHA1HashData(byte[] bytes)
{
    var sha1 = SHA1.Create();
    byte[] hashData = sha1.ComputeHash(bytes);
    foreach (byte b in hashData)
        Console.WriteLine(b);
    return hashData;
}
```

In (**Figure 15**), we present an example of a conceptual graph from which a corresponding content vector can be constructed from the source code artefact in (**Table 8**). For example, there exist three concept nodes of type `MethodCall` which means that the value for the `MethodCall` component of its content vector will be 3. Similar values can be obtained for each node type in CG and if such a node type is not present in the graph, the zero value is assigned.

Similar to the work in (O'Donoghue & Crean, 2002) for analogy retrieval, measures and metrics can be extracted from the graph representation in order to express the structure of documents. Because we later describe a versatile mechanism of comparing content vectors, we concatenate some extracted metrics by the Graph-Matching module in *Arís* (Grijincu, 2013), to content vectors as if they were regular feature vectors. For each Conceptual Graph representation of a source code artefact, some of the measures and metrics used are the following:

- *Vertex count (#V):* number of vertices
- *Edge count (#E):* number of edges
- *Average degree:* $\frac{2(\#E)}{\#V}$ two times number of edges divided by number of vertices
- *Graph diameter:* the longest shortest path between any two vertices
- *Maximum out degree:* maximum number of outgoing edges in a node
- *Maximum in degree:* maximum number of incoming edges in a node
- *Average node rank:* average node rank between all nodes (defined in (Grijincu, 2013))

4.3.3. Content vectors similarity

The authors of (Gentner & Forbus, 1994) demonstrated that the dot product between two content vectors $v_{d_1} = (i_1, i_2, \dots, i_n)$ and $v_{d_2} = (j_1, j_2, \dots, j_n)$ is a good estimate for relative similarity, which is a consequence of the fact that each of the product $i_k \times j_k$ is an overestimate of the number of matched hypotheses between functor types. However, in our retrieval model, we use the *Radial Basis Function* (RBF), which is has been successfully used to compute similarity scores between feature vectors in other retrieval systems (Manning, et al., 2008):

$$RBF(x, y) = \exp(-\frac{||x-y||^2}{2\sigma^2}) = \exp(-\gamma||x-y||^2)$$

where x and y are two points in a n-dimensional Euclidian space, ||x - y|| is the Euclidian distance between vectors x and y, σ is a free real parameter and $\gamma = \frac{1}{2\sigma^2}$.

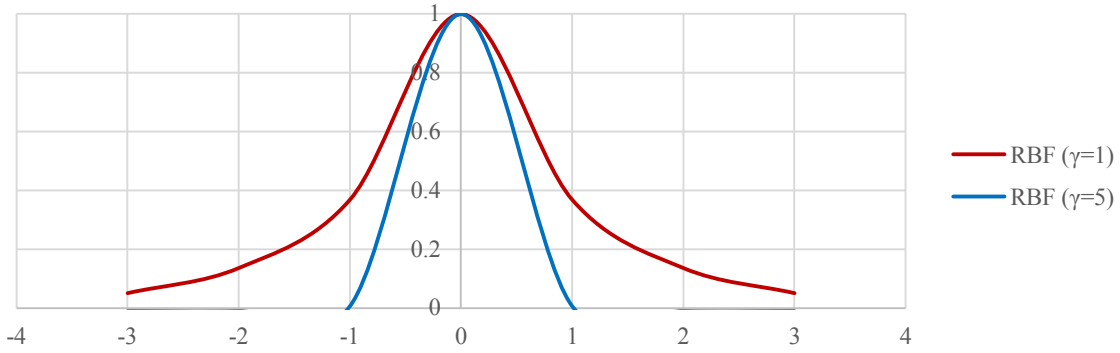


Figure 16: RBF plot with γ=1 and γ=5. For example, if two cases x and y are similar, their Euclidian distance will be close to zero and the similarity score will be close to one.

The main advantage of RBF is the fact that the function is more flexible than dot product when the feature vectors contain other attributes (i.e. different from the functor types established for content vectors) which can express other properties of conceptual graphs (e.g. total number of nodes, number of loops, loop size, connectivity (O'Donoghue & Crean, 2002)). The downside of RBF is the fact that the γ parameter is free, therefore its value must be chosen experimentally. In our implementation, we used custom values for the γ parameter depending on the type of the source code artefact (**Figure 16**) (i.e. $\gamma_{method} = 0.05$, $\gamma_{class} = 0.01$, $\gamma_{classes} = 0.005$ chosen empirically based on the evaluation results) each content vector is describing (i.e. we compute RBF only between content vectors that are both associated with methods, classes, collection of classes).

RBF computes the similarity score applied between the content vectors associated with the query and the cases in memory (associated with source code artefacts) and the result is used in ranking, from a structural point of view (the implementation for Euclidian distance and RBF similarity are presented in (**Table 9**)).

Table 9: Euclidian `Distance` implementation used to compute `RBFSimilarity`

```
double Distance(StructuralFeatureVector v1, StructuralFeatureVector v2)
{
    double d = 0.0d;
    for (int i = 0; i < VectorDimension; i++)
    {
        double f1 = v1.Features[i], f2 = v2.Features[i];
        if (f1 != 0 && f2 != 0)
```

```
                d += Math.Pow(f1 - f2, 2);
            else if (f2 == 0)
                d += f1 * f1;
            else
                d += f2 * f2;
        }
        return d;
}

double RBFSimilarity(StructuralFeatureVector v1, StructuralFeatureVector v2)
{
        if (v1.Artefact is MethodCodeArtefact) // v2.Artefact is also MethodCodeArtefact
            return Math.Exp(-MethodRbfGamma * Distance(v1, v2));
        else if (v1.Artefact is TypeCodeArtefact)
            return Math.Exp(-ClassRbfGamma * Distance(v1, v2));
        else return Math.Exp(-ClassesRbfGamma * Distance(v1, v2));
}
```

So far, we have presented a way of encoding information about the source code topology by using Conceptual Graphs and Content Vectors and we have defined a way of computing similarity scores between these representations. In addition, we argued that this process is analysing different structural characteristics of source code, as opposed to semantic retrieval. Therefore, a different mechanism for selecting the subset of relevant source code artefacts is needed and its designed is described in the following section.

4.3.4. Efficient case comparison

Because our database of source code artefacts will potentially be very large, we need an efficient way of retrieving the most similar cases. We propose using the *K-Means Clustering* algorithm (3.1.5) in order to create sub-groups of content vectors and perform *K-Nearest Neighbours* (3.1.2) only on the "closest" sub-groups to a given input content vector, thus speeding up the structural retrieval process, at the cost of an extra learning step in the knowledge acquisition phase.

Similar to the technique described for semantic retrieval in section (4.2.2) (i.e. inverted indexing used for selecting semantically relevant documents, relative to the query), we need to select the subset *S* of structurally relevant documents from the entire corpus and to compare the query only with this subset *S*. This technique is employed only for content vector (structural) representations, because in the case of semantic retrieval, the proposed inverted indexing method (4.2.2) can easily obtain the set of relevant documents by taking advantage of vector space

representations. However, content vectors are no different than general purpose feature vectors, therefore we cannot expect any special particularities in their topology as in the case of vector space representations.

In order to compute the subset *S* of structurally relevant source code artefacts, we propose prior clustering of the set of content vector representations, which will yield representative cluster centroids. The major challenge when applying K-means clustering algorithm is the choice of K. Currently there exists no well-established method for automatic selection of K (we have experimented with the *elbow method*[7], but with no significant results), therefore we established $K = \sqrt{\frac{n}{2}}$, (where *n* is the total number of content vectors) by analysing the data set of content vectors and evaluating the results. After clustering our source code artefacts in sub-groups by their structural features expressed by content vectors, at retrieval time we compare the query with the established *K* centroids and apply K-nearest neighbours in order to determine the nearest sub-group (all distances between content vectors were computed using the implementation in (**Table 9**)).

An interesting behaviour manifested by the K-means clustering algorithm is the random initialization phase of the centroids, which may cause different local optimal solutions each time we run the algorithm. Empirical evidence has shown that centroids should be placed (in the "Init" step) as far away as possible from each other; thus, centroids should be initialized from randomly chosen points from the input set (MacQueen, 1967). In our implementation, we first select centroids from the data set using the initialization procedure of the K-means++ algorithm (Arthur & Vassilvitskii, 2007). Therefore, as a safety measure, our subset *S* of structurally relevant source code artefacts is initialized from the closest two clusters to query (i.e. 3-nearest neighbours), which will result in a trade-off between slower retrieval times and a longer retrieved list of source code artefacts.

This technique is optional and its purpose is to speed up the retrieval process by comparing the query only with structurally relevant documents, from a potentially large database of source code artefacts. Also, as a consequence, the structural retrieval results will only consist of documents that are related to the query (with high probability). This technique can be used with regular feature

[7] http://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set#The_Elbow_Method

vectors, however the generation of clusters can be computationally expensive, as opposed to the "shortcut" method used with vector space representations in semantic retrieval.

We have discussed the design of the structural retrieval technique, which is an independent process in the retrieval module of *Arís*. In addition, an efficient indexing technique for content vectors (extracted from Conceptual Graphs) was presented in order to select relevant documents relative to the query, which can then be compared using RBF. The following section outlines the advantages, disadvantages and differences between our proposed structural retrieval solution and the system presented in section (2.1).

4.3.5. Critical analysis of structural retrieval

The proposed structural retrieval model in *Arís* relies on conceptual graphs and content vectors. Content vectors are not taking into account the full relational structure of the documents, however their main advantage is the light representational mechanism which allows fast retrieval. Their design role in *Arís* is justified by the fact that a deeper, more computational expensive comparison between the query and the top ranked retrieved source code artefacts is applied later (4.4).

Conceptual graphs (4.3.1) have proven representational power of source code and our primary inspiration for use of this structure was the work presented in section (2.1). However, the retrieval system presented in (Mishne & De Rijke, 2004) is different from the one in *Arís*. Although the authors argued that a lightweight retrieval mechanism was applied in order to rank relevant source code segments, the overall conclusion was that their proposed system is still computationally complex.

As stated earlier, the main threat to validity for our proposed structural retrieval algorithm is the fact that content vectors are not an exhaustive (structural) description mechanism for source code artefacts, but the results from our Evaluation (chapter (5)) show the promising potential of conceptual graphs and content vectors in representing source code.

4.4. Combined Retrieval

Our final step in the retrieval phase in *Arís* is combining the results from semantic (4.2) and structural (4.3) retrieval. These two processes are independent because they analyse different properties of documents in case-base relative to the query, thus their execution can be parallelised and their results can be processed just before the retrieval system responds. In addition, despite their independence, the processes are designed to complement each other by analysing different characteristics found in source code and their symbiosis is reached in combined retrieval, where both structural and semantic features are reflected in the results. Moreover, the design of combined retrieval is modular, which means that other retrieval algorithms can easily be integrated with *Arís*, for example, a technique that explores the documentation of the software in case-base (e.g. comments in the source code).

We obtain the final set of documents relevant to the query by union of the set U of semantically similar documents (4.2.2) and the set S of structurally similar documents (4.3.4). Therefore, each document d from this final set has two similarity scores associated (with respect to the query q) and one combined score between them:

- $sim_{semantic}(d, q)$: Semantic similarity score between document d and query q. If d is not included in U, the score is equal to 0.
- $sim_{structural}(d, q)$: Structural similarity score between document d and query q. If d is not included in S, the score is equal to 0.
- $sim_{comb}(d, q) = w_{semantic} \times sim_{semantic}(d, q) + w_{structural} \times sim_{structural}(d, q)$: Combined similarity score between document d and query q. $w_{semantic}$ and $w_{structural}$ are factor weights for each score, such that: $w_{semantic} + w_{structural} = 1$. In our implementation, the assignment $w_{semantic} = 0.5$ and $w_{structural} = 0.5$ yielded good results.

There are situations when the weights of semantic or structural retrieval need to change. For example, if two identical source code artefacts do not contain any API calls, the semantic similarity score will be zero, although the overall similarity score should be close to one. Instead, when this situation occurs, we alter the weights of semantic and structural similarity in combined retrieval (for example, 0.001 for semantic similarity and 0.999 for structural similarity).

The final set of documents relevant to the query, is then ranked by their corresponding sim_{comb} scores. Further, we seek ranking precision for the top retrieved documents, because their order in the results is crucial (i.e. selecting most similar implementations such that the knowledge transfer (formal specification) step will have a higher chance of success; also, these top documents represent the most interesting subset of corpus, to a possible interactive user of *Arís*). This increased level of attention to ranking of the top retrieved documents is a widely used practice in other retrieval systems (Mishne & De Rijke, 2004) (Gentner & Forbus, 1994).

We rely on the Conceptual Graph-Matching module in *Arís* (Grijincu, 2013) to return a more precise similarity score between query and top retrieved source code artefacts (in our implementation, the top 10% of retrieved source code artefacts) However, if none of the top retrieved documents has a similarity score greater than a threshold i.e. 0.5, the CG matching is no longer applied. This module is using a deeper, more computationally expensive algorithm to match source code artefacts by their conceptual graph (3.2.1) representations. The module is capable of comparing any type of source code artefact (i.e. method, class, collection of classes. In fact, the conceptual graph representation can be extracted from any source code snippet). The returned conceptual graph similarity score $sim_{CGmatch}$ is part of the following linear combination:

$$sim_{overall}(d,q) = w_{comb} \times sim_{comb}(d,q) + w_{CGmatch} \times sim_{CGmatch}(d,q)$$

where w_{comb} and $w_{CGmatch}$ are weights for the combined retrieval score and conceptual graph matching score, such that $w_{comb} + w_{CGmatch} = 1$. Once again, these weights were chosen based on the evaluation results of our implementation: $w_{comb} = 0.35$ and $w_{CGmatch} = 0.65$.

4.5. Overview of all parameters and definitions used in retrieval

As a summary of all parameters, weights, similarities and metrics that affect or tune the retrieval system, in (**Table 10**) we give a description of all such variables.

Table 10: Overview of all definitions used for source code retrieval, in the order presented in this chapter.

	Description
Vector space representation	Vector that contains the importance (weight) of all API calls, used to express the semantics of source code artefacts
U	The subset of semantically relevant source code artefacts relative to the query
w_{cos}	Weight for Cosine similarity between vector space representations
w_{dl}	Weight for Damerau-Levenshtein similarity between sorted lists of API calls
$simCos(d,q)$	Cosine similarity between documents *d* and *q*
$simDL(d,q)$	Damerau-Levenshtein similarity between documents *d* and *q*
$sim_{semantic}(d,q)$	Semantic similarity between documents *d* and *q*
S	The subset of structurally relevant source code artefacts relative to the query
Content vectors	Feature vectors that encodes the number of every node type in Conceptual Graphs (*e.g. Loop, Variable, MethodCall*) and various measures that express CG structure (e.g. *Average degree, Graph diameter, Average node rank*)
K	The number of clusters used for content vector clustering with K-means
γ_{method}	RBF parameter for computing similarity between two methods
γ_{class}	RBF parameter for computing similarity between two classes
$\gamma_{classes}$	RBF parameter for computing similarity between two collections of classes
$w_{CGmatch}$	Weight for Graph-Matching similarity
$w_{semantic}$	Weight for semantic similarity between two vector space representations
$w_{structural}$	Weight for structural similarity between two content vectors
w_{comb}	Weight for combined (semantic and structural) similarity
$sim_{structural}(d,q)$	Structural similarity between documents *d* and *q*
$sim_{comb}(d,q)$	Combined (semantic and structural) similarity between documents *d* and *q*
$sim_{CGmatch}(d,q)$	Graph-Matching similarity between documents *d* and *q*
$sim_{overall}(d,q)$	Overall similarity between documents *d* and *q*

4.6. Evaluation of retrieved specifications

After the combined retrieval process is complete, knowledge transfer (formal specification generation) can now be performed from the top ranked verified implementations to the unverified query.

The top ranked source code artefacts are relevant if their overall similarity score with the query is higher than an established threshold (in our implementation, 0.7). Using the source code artefacts which have an associated formal specification from these top ranked results, we iteratively "feed" each retrieved verified implementation as *source* and the query as *target* to the Specification Generation module in (Grijincu, 2013), which uses a pattern completion algorithm called *Copy with Substitution and Generation* (Holyoak & Thagard, 1999). After the formal specification is generated for the query (*Reuse* phase of CBR), our module is responsible of evaluating this newly formed solution (*Revise* phase of CBR) (**Table 11**).

Table 11: Example of a successful specification generation (right) using the existing Spec# formal specification from the *Source* implementation (left).

Source	***Target***

`string NextWord(string src, int n, StringBuilder sb)`
`  requires 0 <= n && n <= src.Length;`
`  requires sb.Length == n;`
`{`
`  int start = n;`
`  while (n < src.Length)`
`  invariant sb.Length == start;`
`  invariant start <= n && n <= src.Length;`
`  {`
`  if (src[n] == ' ')`
`  {`
`  n++;`
`  break;`
`  }`
`  n++;`
`  }`
`  string s =src.Substring(start,n-start);`
`  sb.Append(s);`
`  return s;`
`}`
	`string NextWord_MOD(MyStringBuilder sb, int i, string source, int ChunkSz)`
	`  requires 0 <= i && i <= source.Length;`
	`  requires sb.Length == i;`
	`{`
	`  int begin = i;`
	`  for (;i < source.Length; i++)`
	`  invariant sb.Length == begin;`
	`  invariant begin <= i && i <= source.Length;`
	`  {`
	`  if (source[i] == ' ')`
	`  {  //extraneous`
	`  i++;`
	`  break;`
	`  }`
	`  }`
	`  string str = source.Substring(begin, i - begin);`
	`  sb.Append(str);`
	`  return str;`
	`}`

In our implementation, we use the Spec# (Rustan, et al., 2010) formal method to verify the implementation. If the generated specification does not formally verify the query, a new source

code artefact from the top ranked results is used as source for specification generation, until the process is successful or there are no more relevant source code artefacts in the retrieval results. In addition, in case of success (a generated specification formally verifies the query), the new generated solution is stored in the case base for further use (*Retain* phase of CBR).

4.7. Interactive interface of Arís

The practical implementation of the proposed system follows closely the specification of the presented model and a useful Graphical User Interface for *Arís* was developed in order to perform proof of concept.

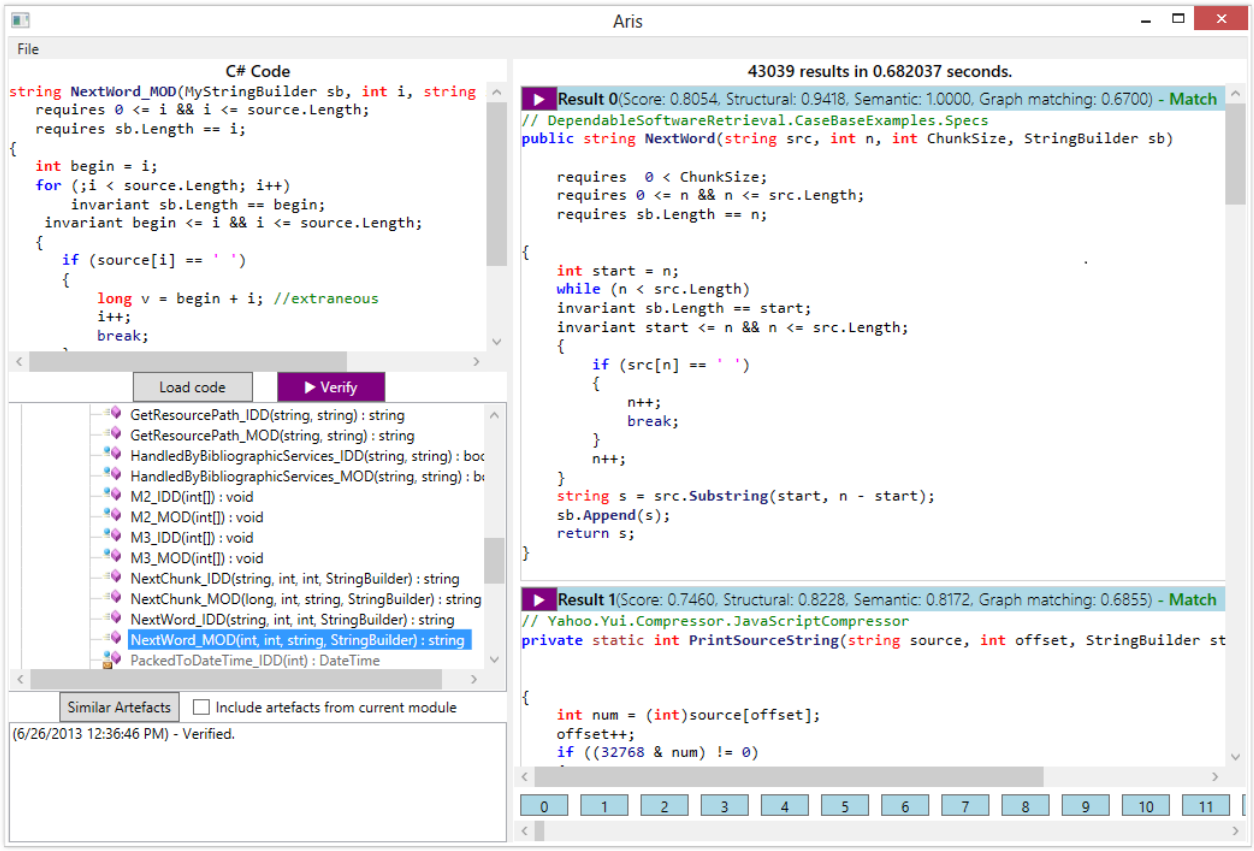


Figure 17: Interactive interface of *Arís*.

The interface (**Figure 17**) was developed using WPF .NET Framework 4.5 in C# and XAML. It consists of a *tree view* from where a particular source code artefact can be selected as query and a *list view* where the retrieval results are loaded and ranked based on their overall similarity score.

In addition, the user can manually transfer knowledge (formal specification) from the retrieved source code artefacts to the query and verify it. This software tool is also justified by use-cases (1.1) where an actual interactive user is searching for similar source code artefacts, relative to a given input. Even in the case of applying knowledge transfer of formal specifications, further assistance or intervention from a human user might be needed for successful verification of implementations (in this case, our framework is similar to *Conversational Case Based Reasoning* systems (Gu, et al., 2004)).

Summary

We presented the proposed solution relative to the original problem statement (1.1) with a strong focus on source code retrieval. Moreover, we analysed our implementation in contrast to existing methods, we justified our design decisions and we described in detail critical fragments of our model by showing practical algorithm implementations or various values assigned to our free parameters. Further, we described how the assessment of formal specifications is performed and we have presented a tool that facilitates interaction with the framework.

5. Evaluation

In this chapter, we will present some of the experiments used for evaluating the proposed system and the associated results. Because this project was focused on the retrieval phase of Case Based Reasoning, the performance of this process will be evaluated separately from the performance of the overall project, which has as a main goal formal specification reuse.

5.1. Source code retrieval evaluation

Although the source code retrieval module can be used independently of our overall purpose, its design was created with the overall goal of *Arís* in mind (as discussed in chapter (4)). Therefore, the experimental setup environment for the source code retrieval process must be aware of the fact that *Arís* doesn't seek the same level of "fuzziness", as desired by other frameworks, presented in the *Related Work* chapter (2). Instead, the system should be able to retrieve source code artefacts that achieve the same functionality, i.e. that implement the same requirements as the query. Also, as opposed to CLAN, which detects similarity between software applications, we seek software retrieval at various OOP granularity: methods, classes or collection of classes, where formal specifications currently define the functionality of implementations.

In the following sections we analyze the computational characteristics of the proposed technique and we discuss the experiments that helped in parameter selection or evaluation of retrieval performance.

5.1.1. Computational characteristics

The source code retrieval process consists of the following two steps: knowledge acquisition and query response. The knowledge acquisition step is responsible of enriching the case-base with source code artefacts and with extraction of various information needed for retrieval.

In the knowledge acquisition step for semantic retrieval, API calls are extracted and indexed from a set of compiled assemblies, as presented in section (4.1). Respectively, for structural retrieval, methods are decompiled into valid C# source code; the result is passed to Conceptual Graph Construction module and content vectors are stored in the case-base. The knowledge

acquisition phase takes approximately 5 hours on a 1.7GHz i7 processor, for the set of 2,191 software applications, which contain 2,033,623 methods. If performed separately, extracting API calls takes 45 minutes and extracting content vectors takes 4 hours and 35 minutes. Optionally, API calls and content vectors computed at this step can be stored offline for further use.

At query time, the source code retrieval system responses vary in time depending on the complexity of the input source code artefact. For example, if the query contains many API calls, the subset of related implementations extracted using the technique described in section (4.2.2) will have a high cardinality, thus many similarity scores must be computed with the query. The complexities for computing the similarity scores for the two retrieval methods are:

- *Semantic similarity:* $O(m^2)$, where m is the maximum number of API calls between the two compared source code artefacts ($O(m)$ for Cosine similarity and $O(m^2)$ for Damerau-Levenshtein distance)
- *Structural similarity:* $O(l)$, where l is the fixed length of content vectors (complexity of RBF similarity function)

The average time needed to retrieve similar source code artefacts using the same number of software applications in the case-base, is 2.89 seconds. This response time is also due to prior use of clustering with K-means of content vectors, which requires another 5 hours in the knowledge acquisition step. If clustering is not used for structural retrieval, the response time increases with 5.5 seconds, on average. We will also show in our experiments, that the use of clustering does not significantly affect the quality of retrieval. As a comparison note, the system presented in section (2.1) reported that the system responds in approximately 5-6 minutes when retrieving from a corpus of 2,932 files, which contain 88363 lines of code.

5.1.2. Evaluation case-base and queries

Evaluation of a source code retrieval system is a difficult task, mainly because of the lack of a publicly available test set, which would contain sets of source code artefacts that implement the same requirements. This setting would allow, for example, selecting one element from such a set as query and evaluating whether the retrieval system is able to retrieve from memory all documents that implement the same requirements. The inexistence of such a test set is due to the fact that its

creation is laborious work and may be subjective or inappropriate for the actual purpose of the retrieval system (i.e. detecting code theft, rapid prototyping, etc.).

The problem was also encountered by work presented in chapter (2) and different solutions were found. The system presented in section (2.1) was evaluated using a small number of queries (i.e. 25 source code files) that were identical or modified versions of documents in the database. The authors of CLAN (section (2.2)) evaluated performance using 33 software engineers that manually evaluated the retrieved software applications and then compared the results with the MudaBlue system (Kawaguchi, et al., 2006).

In order to evaluate source code retrieval in *Arís,* the set described in section (4.1) with 2,191 software applications, which contain 175,291 classes and 2,033,623 methods was selected to aggregate the case-base. The methods have on average 36 lines of source code and the case-base comprises of approximately 74,215,100 lines of code. These documents are real world examples of unverified software implementations and amongst them, we have added test-suite examples of verified implementations from the Spec# version control website (as described in section (4.1)). It is worth noting that at a higher level view on the corpus, the test-suits examples for Spec# contains relatively short implementations of simple tasks (sum of elements in a vector, sorting an array, etc.) with little or no use of API calls (we extracted 30 API calls from all test-suite examples), while the real world examples contain implementations of harder tasks, where the use of API calls is ubiquitous (**Table 12**).

Table 12: Randomly selected methods illustrating the content of each corpus.

Spec# test-suite example	Example from a Codeplex project

`int Count(int[] a, int x)`
`  requires a != null;`
`  ensures result == count{int i in (0: a.Length); (a[i] == x)};`
 	
`{`
`  int s = 0;`
`  for (int i = 0; i < a.Length; i++)`
`  invariant i <= a.Length;`
`  invariant s == count{int j in (0: i); (a[j] == x)};`
`  {`
`  if ((a[i]) == x)`
`  s = s + 1;`
`  }`
`  return s;`
	`string ResolveUrl(string appPath, string relativeUrl)`
	`{`
	`  string result;`
	`  if (relativeUrl == null)`
	`  result = null;`
	`  else`
	`  {`
	`  if (!relativeUrl.StartsWith("~"))`
	`  result = relativeUrl;`
	`  else`
	`  {`
	`  string text = appPath + relativeUrl.Substring(1);`
	`  result = text.Replace("//", "/");`
	`  }`

```
}
```

```
    }
    return result;
}
```

The query set selected to test the source code retrieval technique, contains 50 modified source code artefacts (methods) from documents in our case-base. Amongst these queries, there are some unmodified versions of examples from our case-base (i.e. identical document retrieval) that serve as a sanity test for the retrieval system. We expect that in this setup, the similarity score between the two identical documents will be one, because both semantic and structural similarity is one, for reasons discussed in section (4.2.4) and (4.3.3) respectively. Indeed, in all our experiments, *Arís* was able to retrieve the corresponding identical source code artefact with similarity score of one from the case-base.

For the modified versions of source code artefacts in the query set, we have applied some modifications that do not change the functionality of the original implementation (Wilkinson, 1994):

1. *Lexical modifications:* changing names of variables, methods, parameters
2. *Parameter order:* changing the order of input parameters in the method's definition
3. *Code comments:* adding, removing or modifying comments in source code
4. *Code style changes:* changing the type of loop used (*for, while*)
5. *Order of statements:* changing order of source code statements, where this does not affect the desired functionality
6. *Extraneous statements:* adding source code statements (for example, `int n = i + j;` or unnecessary API calls, `String.IsNullOrEmpty("example")`)
7. *Changing variable types:* changing types of variables where this does not affect functionality (for example, changing type `int` with type `long`, or changing type `Superclass` with type `Subclass`)

Our source code retrieval system is not affected by any of the modifications from 1 to 4 (we do not consider names of variables, parameter order, code comments in our representational structures). However, changing order of statements (5), adding extraneous statements (6) and changing variable types (7) will possibly result in different API calls or a different order of API

calls, which will affect the returned semantic similarity score. Also, this will yield a different Conceptual Graph representation of source code, which affects the structural similarity score.

5.1.3. Retrieval parameters selection experiments

In chapter (4), we defined a number of free parameters that directly affect the quality of retrieval (for example, selection of weights for semantic, structural modules and the Conceptual Graph-Matching similarity score).

In order to assess the performance of retrieval under different parameter settings, we used the case-base and queries defined above. We know that for every query in this set, there exists the original source code artefact in the case-base, from which the query was modified, therefore their similarity score should be close to 1. Thus, the *mean* similarity score between all queries and their corresponding source code artefacts in the case base should be close to 1, in a good retrieval framework. Also, we make use of the Mann-Whitney (Mann & Whitney, 1947) and Kruskal–Wallis (Kruskal & Wallis, 1952) statistical significance tests, in order to determine whether the results of our experiments occurred by chance (our *null-hypothesis* is that the retrieval system tuned with a set of free-parameters is equivalent to the retrieval system tuned with a different set of parameters). In (**Table 13**), we present the results (expressed by the *Mean* column) of a few parameter configurations of the retrieval module configured with different sets of parameters. Because exhaustive selection of values for all parameter combinations is a time consuming and difficult task, we selected a few values by intuition and assessed performance by selecting the configuration with the highest mean. The parameters selected in Config_1 yielded the best results (highest mean).

Table 13: A subset of parameters selected for the source code retrieval system.

Retrieval System	w_{cos}	w_{dl}	$w_{semantic}$	γ_{method}	K	$w_{structural}$	w_{comb}	$w_{CGmatch}$	Mean
Config $_1$	0.7	0.3	0.5	0.05	3	0.5	0.35	0.65	**0.921**
Config $_2$	0.4	0.6	0.5	0.05	3	0.5	0.35	0.65	**0.908**
Config $_3$	0.7	0.3	0.7	0.05	3	0.3	0.35	0.65	**0.893**
Config $_4$	0.7	0.3	0.2	0.05	3	0.8	0.35	0.65	**0.901**
Config $_5$	0.7	0.3	0.5	0.40	1	0.5	0.35	0.65	**0.841**

Config $_6$	0.7	0.3	0.5	0.005	3	0.5	0.35	0.65	**0.899**
Config $_7$	0.7	0.3	0.4	0.99	3	0.7	0.35	0.65	**0.915**
Config $_8$	0.7	0.3	0.7	0.05	3	0.3	0.9	0.1	**0.771**
Config $_9$	0.5	0.5	0.5	0.05	3	0.5	0.5	0.5	**0.873**
Config $_{10}$	0.6	0.4	0.5	0.01	3	0.6	0.2	0.8	**0.901**

In (**Table 14**) we present the results of a few statistical significance tests that test the null-hypothesis between runs from (**Table 13**). Only a few of these tests were computed, depending on the retrieval systems compared. For example, it makes sense to compare $Config_1$ and $Config_2$ in a single test because the only parameters that vary in the two configurations are Cosine similarity and Damerau-Levenshtein similarity in semantic retrieval. The null-hypothesis is rejected if $P \leq 0.1$, which means that there is a significant difference in the results of the configurations compared.

Table 14: Mann-Whitney[8] statistical significance test for testing the null-hypothesis for two retrieval configurations and Kruskal-Wallis generalized test of Mann-Whitney for three or more retrieval configurations.

Mann-Whitney/ Kruskal–Wallis	Systems compared	Null-hypothesis
P=0.482	$Config_1$, $Config_2$	Accept
P=0.241	$Config_1$, $Config_3$, $Config_4$	Accept
P=0.204	$Config_1$, $Config_5$, $Config_6$, $Config_7$	Accept
P=0.046	$Config_1$, $Config_8$	Reject
P=0.354	$Config_1$, $Config_9$, $Config_{10}$	Accept
P=0.191	$Config_7$, $Config_9$, $Config_{10}$	Accept

From table (**Table 14**) we can infer that the null-hypothesis (two or more parameter configurations are equivalent) is accepted in the majority of the tests, which means that changing various values in the system does not significantly change the quality of retrieval. However, this result can be explained by the fact that a small number of queries were used in the experiments and

[8] Computed using: http://www.vassarstats.net/

most of the queries are test-suit examples for Spec# (as discussed above), which explains that assigning a higher weight for semantic retrieval does not make a significant difference in the results.

One important observation that can be inferred from tables (**Table 13**) and (**Table 14**) is the fact that assigning a low value for the Graph-Matching module affects the overall quality of the retrieval system and we can state that the differences in results between $Config_1$ and $Config_8$ are significant and they do not occur by chance (the null hypothesis is rejected in the Mann-Whitney test). In (**Figure 18**) we can observe that by using graph matching, the similarity scores are improved.

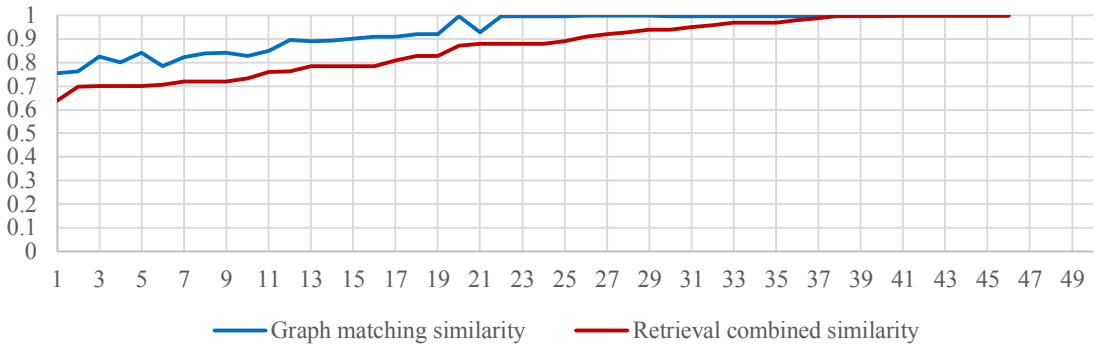


Figure 18: Responses in $Config_1$ for each of the 50 queries sorted by retrieval similarity scores

This fact can be explained by observing that, the combined retrieval similarity score does not take into account some of the modification rules applied when we have created the queries (as discussed in section (5.1.2), order of statements, extraneous statements, changing variable types). For example, in (**Table 15**), the similarity score between the two implementations, assigned by combined retrieval is 0.78 and the score assigned by Conceptual Graph-Matching is 0.88. This is justified by the fact that the Conceptual Graph-Matching module is capable of inferring that `MyStringBuilder` is a subclass of `StringBuilder` and their usage is equivalent (Liskov Substitution Principle (Meyer, 1992)). Although both modules are not sensible at method parameter re-ordering, name changing or loop style change, they are affected by adding or removing statements, mainly because the corresponding Conceptual Graph representations are different (Grijincu, 2013).

Table 15: Example of a query (left) modified from a source code artefact that is part of the case-base (right).

Modified source code artefact	Original source code artefact

`string NextWord_MOD(MyStringBuilder sb, int i, string source, int ChunkSz)`
`{`
`  long begin = i;`
`  for (;i < source.Length; i++)`
`  {`
`  if (source [i] == ' ')`
`  {`
`  long v = begin + i; //extraneous`
`  i++;`
`  break;`
`  }`
`  }`
`  string s = source.Substring((int) begin, i-begin);`
`  sb.Append(s);`
`  return s;`
`}`
	`string NextWord(string src, int n, StringBuilder sb)`
	`{`
	`  int start = n;`
	`  while (n < src.Length)`
	`  {`
	`  if (src[n] == ' ')`
	`  {`
	`  n++;`
	`  break;`
	`  }`
	`  n++;`
	`  }`
	`  string s =src.Substring(start,n-start);`
	`  sb.Append(s);`
	`  return s;`
	`}`

5.1.4. Comparison with other retrieval systems

Another challenging task when evaluating the proposed solution was comparing our results with other source code retrieval frameworks. For a conclusive comparison between retrieval software, the same set of queries must be set as input for each compared system. This task is difficult because an actual copy of the software implemented by the authors of the frameworks described in chapter (2) is hard to obtain (not publicly available). Instead, a solution might be implementing a baseline system using the presented specifications in the original paper (for example, the authors of CLAN (McMillan, et al., 2012) fully implemented MudaBlue (Kawaguchi, et al., 2006) for evaluation purposes). In addition, the goal of our retrieval module is unique (reuse of formal specifications), thus a detailed comparison with other source code retrieval frameworks that seek a different purpose, might be irrelevant. However, even if we use a different set of queries (from the ones used by other systems (2)), we compute some standard measures (Manning, et al., 2008) useful for comparing retrieval systems:

$$Precision = \frac{|Relevant \cap Retrieved|}{|Retrieved|}$$

$$Recall = \frac{|Relevant \cap Retrieved|}{|Relevant|}$$

where $Relevant$ is the set of relevant documents in the collection (relative to a given query) and $Retrieved$ is the set of documents retrieved by the system. Measuring recall means assessing the entire corpus of documents because it requires discovering every source code artefact similar to the query. Precision, on the other hand, requires assessing only the top ranked retrieval results (i.e. above the established threshold in section (4.6)). Both of these measures require human evaluation (by a software engineer) and similar to the experiments in (Mishne & De Rijke, 2004) and (McMillan, et al., 2012), we focus only on measuring precision.

In addition, because ranking is an important aspect of our retrieval system, we measure the Mean Reciprocal Rank (suitable when there is a single relevant item to a given query):

$$MRR(Q) = avg\left\{\frac{1}{rank(d(q_1))}, \dots, \frac{1}{rank(d(q_n))}\right\}$$

where, $rank(d(q_i))$ returns the rank of the associated document in corpus relative to the *i*-th query (e.g. $rank(d(q_i)) = 2$ if the document relative to q_i is returned second).

In (**Table 16**) we present the results of our experiments. These results are purely informative because a statistical significance test that would show if the differences between these measures occurred by chance or not, cannot be applied (the queries used in evaluation of the systems were different).

Table 16: Results for Precision and MRR for Source Code Retrieval system in *Arís*, Source Code Retrieval System using Conceptual Graphs and CLAN.

SC Retrieval Systems	Precision	MRR
SC Retrieval in *Arís* (section 4)	0.442	0.908
SC Retrieval using CG (section 2.1)	0.352	0.813
CLAN (section 2.2)	0.45	(n/a)

When measuring precision, another sanity test was developed for the proposed system: using another set of queries (used only for this experiment) and a smaller corpus of source code artefacts that are very different from the queries, we tested whether these irrelevant queries retrieve any source code artefacts. Again, *Arís* responded as expected and none of the queries retrieved any source code artefacts with a similarity score higher than the threshold established in section (4.6).

We conclude this comparison by stating that the retrieval module in *Arís* performs very well on the set of queries we provided, which means that the system has a good chance in achieving its main goal (formal specification reuse). In addition, our framework outperforms other existing systems, by successfully retrieving relevant source code artefacts from a very large set of documents in a reasonable amount of time.

5.2. Formal specification reuse evaluation

In this section we analyse whether *Arís* is successful in reuse of formal specifications. That is, for a given unverified implementation we aim to retrieve similar verified code and then reapply the missing specification that accompanies that code. In this process, the top retrieved verified source code artefacts are iteratively used to generate formal specifications, as described in section (4.6).

In this experiment, we used the same set of modified queries from the existing verified source code artefacts in our case-base, presented in section (5.1.2) (Spec# test-suite). Thus, our aim is to formally verify 100% of these established queries, because we expect that for a given modified query from a verified implementation existing in the case-base, the retrieval system will give a good ranking of the corresponding verified source code and the Specification Generation module in (Grijincu, 2013) will successfully transfer the associated specification.

From a set of 30 selected queries that have an associated verified implementation correspondent in the case base, we were able to verify 18 (**60% of implementations**), using the above described steps to generate specifications. However, when the output of the Spec# formal method was analysed, we discovered that the generated specifications could not verify most of the queries because of the modifications applied in order to create the set of queries (presented in section (5.1.2)). For example, the Spec# formal method returns the following warning when verifying the modified source code artefact from (**Table 15**):

```
warning CS0219: The variable 'v' is assigned but its value is never used
```

This means that some of the modifications made on the queries generated additional errors when transferring specifications. Therefore, some of the errors are not due to incorrect Specification Generation and this fact justifies again the usefulness of an interactive interface (4.7) when such small errors occur. However, 7 unsuccessful generated specifications were assessed as unverified due to incorrect mappings between source and target (4.6).

In (**Table 17**), we present a typical example of an unsuccessful transfer of specification to the query. We observed that when the similarity score is below 0.7 (the threshold established in section (4.6)), the source code artefacts significantly differ in functionality, and therefore the probability of success in transferring the specification using these sources, is very low.

Table 17: Example of query and retrieved source code artefact, with similarity score of 0.621.

Query	Source for knowledge transfer

`int CountEven(int[] a)`
`{`
`  int s = 0;`
`  int v = s + 1;`
`  int i = 0;`
`  while (i < a.Length)`
`  {`
`  if ((a[i] % 2) == 0)`
`  {`
`  s += 1;`
`  }`
`  i++;`
`  }`
`  return s;`
`}`
	`void CountNonNull(string[] a)`
	`  requires a != null;`
	`{`
	`  int ct = 0;`
	`  for (int i = 0; i < a.Length; i++)`
	`  invariant i <= a.Length;`
	`  invariant 0 <= ct && ct <= i;`
	`  invariant ct == count{int j in (0: i); (a[j] != null)};`
	`  {`
	`  if (a[i] != null)`
	`  ct++;`
	`  }`
	`}`

We are conscious of the fact that the main treat to validity is the limited set of queries used in our experiments, however, we conclude this experiment by stating that the results are encouraging and exceed our initial expectations. By verifying 60% of the set of modified queries provided, we demonstrated that the framework is capable of generating new formal specifications using similar past cases.

Summary

In this chapter we presented the experiments and the results of the proposed solution evaluation process. We decided to separately evaluate the source code retrieval system because this was the main focus in our project and because we can relate the evaluation results to other existing frameworks. We concluded the experiments for parameter selection by stating that using the Conceptual Graph-Matching module generally improves the quality of retrieval and we showed that this result does not occur by chance, by using statistical significance tests. Further, we computed some standard metrics for retrieval systems and informally compared the results with existing frameworks. In terms of formal specification reuse evaluation, the results are encouraging and we confirm that in some of the cases, human assistance might be needed in order to fully verify an implementation. The overall performance of *Arís* is very good not only for generating specifications, but also for fast retrieval of source code artefacts. The experimental setup was discussed in detail and the results were validated when this was possible. Also, we critically analysed our work, by outlining the limitations and strong points of *Arís*.

6. Conclusions

We have presented a novel solution to the problem stated in section (1.1). Our work aimed automation of some of the steps involved in writing specifications and their implementations, by reusing existing verified programs. The results exceeded our initial expectations and demonstrated that this model can successfully generate formal specifications. In addition, *Arís* can retrieve source code artefacts from a large corpus, with significantly faster response times than other existing retrieval frameworks.

This approach is unique, to the best of our current knowledge, in terms of both model design and proposed goal. The overall methodology of the *Arís* project is very similar to Case-Based Reasoning and its parent discipline of Analogical Reasoning, centered on the activities of solution retrieval and reuse. Although using CBR for source code retrieval is not a new approach, in this project we have creatively combined the semantic and structural characteristics of software implementations in a modular framework, which is flexible enough to enable weighting of existing retrieval methods or "plug in" similarity algorithms like Conceptual Graph-Matching. After retrieval, we iteratively attempt to transfer the formal specification from the top ranked verified implementations to the query, using the Specification Generation module in *Arís*, and if the process is successful, the new generated solution is stored in the case-base for further use.

In the evaluation chapter, we discussed the experimental setup that helped in establishing good parameters for *Arís*. The main conclusion of these experiments was that using the Conceptual Graph-Matching module generally improves the overall accuracy of retrieval. Further, some standard measures for retrieval systems were analysed and we constantly compared the proposed solution to the related work systems. We demonstrated that our framework successfully performs software retrieval from a large set of source code artefacts, with very good response times. The overall results of the evaluation process show that the proposed solution performs very well relative to its main goal, therefore, we strongly believe that *Arís* represents promising work towards reuse of formal specifications.

6.1. Future work

The overall architectural design for the proposed framework relies on the assumption that similar implementations have similar formal specifications. In future work, we need to discover whether this is a truly reliable source for generating specifications, by establishing our results with strong statistical significance. Another interesting direction of research for this project can be retrieving similar source code artefacts that are different in implementation. For example, in order to verify a sorting algorithm, the formal specification of a different implementation for sorting can be used to achieve verification.

As discussed in the evaluation chapter, the parameter selection experiments were not exhaustive and further attention to this process may further improve the quality of the system. In addition, a more thorough comparison with existing solutions would more precisely confirm the efficiency of our proposed retrieval system in source code retrieval.

Arís was initially envisioned as a framework for software reuse, thus not only for formal specifications using past verified implementations, but also for generating the "missing" implementations using similar specifications. In addition, a separate *Arís* module focuses on specification reuse using data refinement, which can help software clients to focus only on specifications, rather than on details of software implementations. These tools (Pitu M., Grijincu D., Li P., Saleem A., O'Donoghue D. P., Monahan R., 2013) have evolved into separate projects; however, in the future we seek to combine these approaches into an integrated platform that allows reuse of software implementations and formal specifications.

References

Aamodt, A. & Plaza, E., 1994. Case-Based Reasoning: Foundational Issues, Methodological Variations and System Approaches. *Artificial Intelligence Communications,* Volume 1, pp. 39-59.

Acorn, T. & Walden, S., 1992. *SMART: Support management automated reasoning technology for Compaq customer service.* s.l., MIT Press.

Aragon Consulting, n.d. *Krogle code search.* [Online]
Available at: http://www.krugle.org/

Arthur, D. & Vassilvitskii, S., 2007. K-means++: the advantages of careful seeding. *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms,* p. 1027–1035.

Bicarregui, J., Hoare, J. & Woodcock, J., 2006. The Verified Software Repository: a step towards the verifying compiler. *Formal Asp. Comput.,* 18(2), pp. 143-151.

Biggerstaff, T. J., Mitbander, B. G. & Webster, D., 1993. Concept assignment problem in program understanding. *Proceedings - International Conference on Software Engineering*, January, pp. 482-498.

Chatterjee, S., Juvekar, S. & Sen, K., 2009. *SNIFF: A search engine for java using free-form queries.* s.l., 12th International Conference on Fundamental Approaches to Software Engineering.

Cheetham, W. & Goebel, K., 2007. Appliance call center: A successful mixed- initiative case study. *AI Magazine,* Volume 28, pp. 89-100.

Damerau, F., 1964. A technique for computer detection and correction of spelling errors. *Communications of the ACM,* 7(3), pp. 171-176.

Dasgupta, S., 2008. *The hardness of K-means clustering,* San Diego: University of California.

Dasgupta, S., 2008. *The K-means clustering problem,* San Diego: University of California.

Deerwester, S. et al., 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science,* 41(6), pp. 391-407.

Evain, J., n.d. *Mono Cecil Project.* [Online]
Available at: http://www.mono-project.com/Cecil

Everitt, B., Landau, S., Leese, M. & Stahl, D., 2011. *Miscellaneous Clustering Methods.* 5th ed. s.l.:John Wiley & Sons.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l.:Addison-Wesley.

Gentner, D., 1983. Structure-Mapping: A Theoretical Framework for Analogy. *Cognitive Science,* pp. 155-170.

Gentner, D. & Forbus, K., 1994. *MAC/FAC: A model of similarity-based retrieval,* s.l.: Proc. Cognitive Science Society.

Gentner, D. & Smith, L., 2012. Analogical reasoning. In: *Encyclopedia of Human Behavior (2nd Ed.).*. Oxford: UK: Elsevier, pp. 130-136.

Gilbert, S., 2009. *Introduction to Linear Algebra.* 2nd ed. s.l.: Wellesley-Cambridge Press.

GitHub, Inc., n.d. *GitHub.* [Online]
Available at: http://www.github.com

Google Inc., n.d. *Google Code.* [Online]
Available at: http://code.google.com

Grechanik, M., Conroy, K. M. & Probst, K. A., 2007. *Finding relevant applications for prototyping.* s.l., ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories.

Grijincu, D., 2013. *Source Code Matching for reuse of Formal Specifications,* Dublin: s.n.

Gu, M., Aamodt, A. & Tong, X., 2004. *Component retrieval using conversational case-based reasoning.* s.l., In Proceedings of the International Conference on Intelligent Information Systems.

Gunderloy, M., 2002. *Understanding and Using Assemblies and Namespaces in .NET.* [Online]
Available at: http://msdn.microsoft.com/en-us/library/ms973231.aspx

Hoare, T., Misra, J., Leavens, G. T. & Shankar, N., 2007. The Verified Software Initiative: A Manifesto. *ACM Computing Surveys*, April.

Holyoak, K. & Thagard, P., 1999. Analogical Mapping by Constraint Satisfaction. *Cognitive Science,* 13(3), pp. 295-355.

ICSharpCode, 2012. *ILSpy .NET Decompiler.* [Online]
Available at: http://ilspy.net/

Kawaguchi, S., Garg, P., Matsushita, M. & Inoue, K., 2006. MUDABlue: An automatic categorization system for Open Source repositories. *Journal of Systems and Software,* 79(7), p. 939–953.

Keane, M. T., Forbus, D., Aamodt, A. & Watson, I., 2004. *Retrieval, reuse, revision and retention in case-based reasoning.,* s.l.: The Knowledge Engineering Review.

Kolodner, J., 1993. *Case-Based Reasoning.* San Mateo: Morgan Kaufmann Publishers.

Ko, Y., 2012. *A study of term weighting schemes using class information for text classification.* Portland, Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval.

Kruskal & Wallis, 1952. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association,* 47(260), p. 583–621.

Leavens, G. T. & Cheon, Y., 2006. *Design by Contract with JML,* s.l.: s.n.

Luk, R. & Lam, W., 2007. Efficient in-memory extensible inverted file. *Information Systems,* 32(5), pp. 733-754.

MacQueen, J., 1967. Some Methods for classification and Analysis of Multivariate Observations. *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability,* pp. 281-297.

Mamadolimov, A., 2012. *Search Algorithms for Conceptual Graph Databases,* s.l.: Cornell University Library.

Mann, H. & Whitney, D., 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics,* 18(1), pp. 50-60.

Manning, C. & Schütze, H., 1999. *Foundations of Statistical Natural Language Processing.* Cambridge: Foundations of Statistical Natural Language Processing, MIT Press. .

Manning, C. D., Raghavan, P. & Schütze, H., 2008. *Introduction to Information Retrieval.* s.l.: Cambridge University Press.

McMillan, C., Grechanik, M. & Poshyvanyk, D., 2012. Detecting similar software applications. *Proceedings - International Conference on Software Engineering,* pp. 364-374.

Meyer, B., 1992. Applying "Design by Contract". *Computer IEEE,* 25(10), pp. 40-51.

Meyer, B., 2006. *Basic Eiffel language mechanisms.* s.l.:s.n.

Michail, A. & Notkin, D., 1999. Assessing software libraries by browsing similar classes, functions and relationships. *Proceedings - International Conference on Software Engineering,* pp. 463-472.

Microsoft, 2012. *ECMA C# and Common Language Infrastructure Standards.* [Online]
Available at: http://msdn.microsoft.com/en-us/vstudio/aa569283.aspx

Microsoft, n.d. *CodePlex: Project Hosting for Open Source Software.* [Online]
Available at: http://www.codeplex.com/

Mishne, G. & De Rijke, M., 2004. *Source Code Retrieval using Conceptual Similarity.* s.l., Conf. Computer Assisted Information Retrieval.

Mitchell, T., 1997. *Machine Learning.* s.l.:McGraw Hill.

Mizzaro, S., 1998. How many relevances in information retrieval?. *Interacting with Computers,* 10(3), p. 303–320.

Montes-y-Gómez, M., López-López, A. & Gelbukh, A., 2000. *Information Retrieval with Conceptual Graph Matching.* s.l., In Database and Expert Systems Applications.

Neamtiu, I., Hicks, M. & Jeffrey, S., 2005. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes,* 30(4), pp. 1-5.

O'Donoghue, D., 2012. *K-Nearest Neighbours and Case-Based Reasoning lectures,* s.l.: National University of Ireland Maynooth.

O'Donoghue, D. & Crean, B., 2002. *RADAR: Finding Analogies using Attributes of Structure,* Limerick, Ireland: Proc. AICS.

Outcurve Foundation, n.d. *NuGet Gallery.* [Online]
Available at: http://nuget.org/

Park, W. J. & Bae, D. H., 2011. A two-stage framework for UML specification matching. *Information and Software Technology,* 53(3), pp. 230-244.

Pitu M., Grijincu D., Li P., Saleem A., O'Donoghue D. P., Monahan R., 2013. *Arís: Analogical Reasoning for reuse of Implementation & Specification,* Rennes: Artificial Intelligence for Formal Methods.

Powell, W. B., 2007. *Approximate Dynamic Programming: Solving the curses of dimensionality.* 2nd ed. s.l.:Wiley Series in Probability and Statistics.

Reiss, S. P., 2009. *Semantics-based code search.* Vancouver, Canada, 31st International Conference on Software Engineering.

Repenning, A. & Perrone, C., 2000. Programming by analogous examples. *Communications of the ACM,* 43(3), pp. 90-97.

Robles, K., Fraga, A., Morato, J. & Llorens, J., 2012. Towards an ontology-based retrieval of UML Class Diagrams. *Information and Software Technology,* 54(1), p. 72–86.

Rosario, B., 2000. *Latent Semantic Indexing: An overview,* s.l.: s.n.

Russell, S. & Norvig, P., 2003. *Artificial Intelligence: A Modern Approach.* 2nd ed. s.l.:Prentice Hall.

Rustan, K., Leino, M. & Müller, P., 2010. Using the Spec# language, methodology, and tools to write bug-free programs, Microsoft. *Lecture Notes in Computer Science,* Volume 6029, pp. 91-139.

Salton, G. & McGill, M., 1986. *Introduction to modern information retrieval.* New York: McGraw-Hill, Inc..

Sim, S. E., Umarji, M., Ratanotayanon, S. & Lopes, C. V., 2011. How well do search engines support code retrieval on the web?. *ACM Transactions on Software Engineering and Methodology,* 21(1).

Slashdot Media, n.d. *SourceForge.* [Online]
Available at: htttp://www.sourceforge.net

Sowa, J. F., 1984. *Conceptual structures: information processing in mind and machine.* s.l., Addison-Wesley Longman Publishing Co., Inc..

Thomee, B., Bakker, E. M. & Lew, M. S., 2010. *TOP-SURF: a visual words toolkit.* Firenze, Italy, Proceedings of the 18th ACM International Conference on Multimedia.

Vazirani, U. V., Papadimitriou, C. H. & Dasgupta, S., 2006. *Algorithms.* s.l.:s.n.

Watson, I. & Gardingen, D., 1999. A Case-Based Reasoning System for HVAC Sales Support on the Web. *The Knowledge Based Systems Journal,* Volume 3, pp. 207-214.

Wilkinson, R., 1994. Effective retrieval of structured documents. *Proceeding of the 17th annual internation ACM SIGIR conference on Research and development in information retrieval,* pp. 311-317.

Woodcock, J., Larsen, P. G., Bicarregui, J. & Fitzgerald, J., 2009. Formal Methods: Practice and Experience. *ACM Computing Surveys,* 41(4).

Zhong, M., Duan, J. & Zou, J., 2011. *Indexing conceptual graph for abstracts of books.* Shangha, Proceedings - 2011 8th International Conference on Fuzzy Systems and Knowledge Discovery.