Source Code Matching for Reuse of Formal Specifications

Daniela Grijincu

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.

Daniela Grijincu

_____

Acknowledgement

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

Abstract

Although Software Verification technology is rapidly advancing, the process of formally specifying the intended behaviour of a program can still be difficult and time consuming as the program increases in size and complexity. In this project we focus on the source code matching module of *Arís* (Analogical Reasoning for reuse of Implementation & Specification) platform in which we aim to increase the number of verified programs by reducing the effort of writing specifications. Our approach promotes the advantages of code reuse and the possibility of transferring specifications between similar implementations. In order to effectively compare two source code files we represent them using *Conceptual Graphs* that allow us to explore the semantic content of the code while also analysing its structural properties using graph-based techniques. For comparing two conceptual graphs, we propose to use an incremental matching algorithm based on IAM (the *Incremental Analogy Machine* (Keane, et al., 1994)) and find the best mapping between isomorphic (exact matches) or homomorphic (non-identical) sub-graphs. We further develop analogical inferences from the acquired mapping using the CWSG (*Copy With Substitution and Generation*) algorithm for pattern completion and generate new specifications into our target/problem code. Finally, we present our evaluation and show that between structurally similar programs, the formal specifications can be fully transferred and successfully verified. Our overall results are very encouraging and clearly show the potential of reusing formal specifications in creating more dependable software systems.

Contents

1. Introduction

1.1. Formal Software Verification

Nowadays software systems are playing a vital part in all of our day to day activities, as they are embedded in all sorts of systems ranging from entertainment devices like music players, game consoles to more indispensable ones like our telephones, PCs and even into critical safety systems like medical devices, avionics, banking, automotive and many more.

One of today's biggest concerns regarding software is how to demonstrate and guarantee its reliability, since faults in the code can lead to increases in production costs, expose security weaknesses or even cause system failures in critical applications (for example the overflow exception that caused the *Ariane 5* missile to crash (Johnson, 2005)). *Verification* (Hoare, et al., 2009) represents the process of ensuring that such errors will be avoided by formally proving using a rigorous method that the software will behave correctly (within the bounds of its specified properties) and will fulfil its intended purpose.

Formal Software Verification uses mathematical analysis as a rigorous method to construct a formal proof of a program's correctness. The correctness is measured with respect to a *Formal Specification* which describes how the program will behave in certain situations. The past decade brought many improvements for the software verification technology (Woodcock, et al., 2009) and many formal programming languages that implement the *Design-by-Contract* (DbC) approach (Meyer, 1992) have been developed in order to allow the specification of programs written in more popular languages like Java and C#.

1.2. Programming by Contract

Meyer built the Design-by-Contract approach based on two important key concepts: first, it is fundamentally connected with the *Object Oriented* design world and second, each participant (*class*) in the construction process of a program, has a very specific and clear role that it needs to fulfil. Thus, Meyer makes a distinction between a *supplier* role - classes that document and implement the solution, maintain its code and publish just the class interface; and a *client* role

which corresponds to classes that are informed about the supplier's documentation and receive the supplier's interface through which they can access the necessary methods, without knowing anything about their implementation. In this way the client knows exactly in which situations he can use a method and what consequences to expect after running it. The DbC pattern was first applied in the *Eiffel* programming language, but due to Meyer's efforts in popularising the approach, it can now be found in numerous other programming languages such as *Ada* (used in critical safety systems), C++, Java or C# (still active research areas).

The main principles in *Programming by Contract* (applying DbC to a programming language) is being able to specify, using a *formal specification* language, different constraints regarding the public methods in the supplier's interface, some of which are given below:

- *preconditions*: what does the method expect to receive as input in order to function properly. This is the client's obligation to ensure that each method is called with the corresponding preconditions satisfied.
- *postconditions*: what happens when the method executes properly. This is the supplier's responsibility to make sure that the postcondition is true when the method is called with the corresponding preconditions satisfied.
- *class invariants*: statements that are true during the lifetime of the objects for all class instances. This is also the supplier's obligation.

For the C# programming language, *Microsoft Research* has developed the *Spec#* (Microsoft Research, n.d.) formal language which extends C# with the ability to support pre and post conditions, invariants and other specifications using clauses such as *ensures, modifies, requires* or *invariant.* In this project our goal is to transfer and reuse Spec# specifications in order to increase the number of formally verified programs.

1.3. *Arís* – Why reuse of implementations and specifications?

Although we now have the tools for formally specifying the intended behaviour of our programs, the process usually becomes difficult and time consuming as the program increases in size and complexity. Apart from this, users also face another major difficulty in learning how to interact with these tools, how to write good assertions that describe what the program must do and how to develop the appropriate implementations so that the verification goal can be achieved more

easily (Leino & Monahan, 2007). This is one of the main reasons why Design-by-Contract programming is not yet fully adopted in large scale industry projects and the overall set of verified programs still remains very small (see the *Verified Software Repository* in (Woodcock, et al., 2009)).

In the *Arís*[1] (Figure 1) (Pitu, Mihai; Grijincu, Daniela; Li, Peihan; Saleem, Asif; O'Donoghue, Diarmuid; Monahan, Rosemary, 2013) project we address the following questions: *How can we help increase the number of verified programs? How can we aid developers in the process of writing and reusing specifications and/or implementations?* Our framework proposes to reuse existing formal specifications in the same way we reuse code (which is a very common practice among software developers), by transferring specifications and reusing proofs from previous verified programs, thus making software verification more accessible to programmers.

Figure 1. Arís logo

1.4. Source code matching

In the source code matching module of *Arís* we focus on the problem of matching two programs at the implementation level, because as we previously described, if we can find two similar implementations then we can transfer the specification from one to the other - Figure 2).

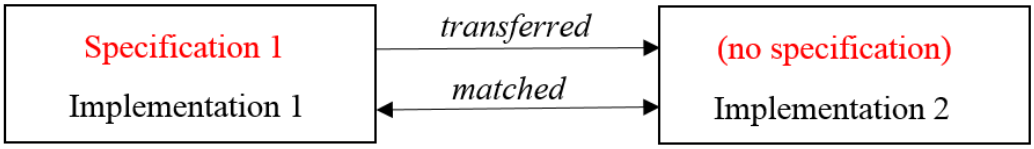

Figure 2. Transferring specifications between two matched implementations

[1] *Arís* – meaning "again" in the Irish Language

Although methods for measuring source code similarity are beneficial in many scenarios as we will see shortly, limited research exists towards a framework for transferring and generating new specifications based on mapping source code level implementations, thus the *Arís* project comes as a novel approach in this area of research.

We've identified a few other fields in which a system for comparing two source code files can prove useful:

- **Code duplication management**. As previous studies show (Chanchal, et al., 2009), duplicated code can represent up to 23% of the total source code in a typical application. This has posed many serious problems for software engineers because "copy-pasted" code is known to be a bad habit that increases the risk of distributing bugs in the system, causing a decrease in productivity (because if something needs to be changed in one place, then the changes have to be made in all the other places where the code was duplicated) and also makes unnecessary use of the system's resources (increasing the program's size and complexity, compilation time, etc.).
- **Plagiarism detection.** Source code plagiarism (copying a piece from someone else's work and presenting it as being your own) has proved a very easy task as source code can be easily modified and copied. It is especially common amongst students programming assignments (Cosma & Joy, 2006), although it can also occur in large, commercial projects.
- **Software evolution**. As software systems develop, their source code implementation is continuously improved and changed to more advance states. A source code similarity measurement that could indicate in which areas was the code changed could help engineers detect trends and patterns in modifications along a software life cycle and better understand how software evolves (Bhattacharya, et al., 2012).

Some of the problems concerning source code modifications that any type of source code matching system needs to be sensible to (i.e. be able to detect as similarity or ignore accordingly) are listed below:

- changes in identifier names, types, comments, whitespaces, layout
- reordered, modified, added or removed statements (that may or may not change the logical structure of the program)

- code fragments that are implemented using different structural (syntactic) constructs and that however perform the same computation

Our system is able to detect all of the above changes and match not only structurally identical programs but also programs that differ in their approach to solving the same problem (which is what we desire in *Arís*, as we want to reuse specifications from previous verified programs that perform the same computational process).

The rest of this paper is organized as follows: Chapter (2) gives an overview of the related work to our problem, more specifically we describe previous solutions to source code representation and source code matching by critically analyzing the methods used and their results. Next, in Chapter (3) we present the theoretical background on which we built our system that is described in detail in Chapters (4) and (5). Finally we present our evaluation results in Chapter (6) and we give our conclusions in Chapter (7) where we also discuss future work directions.

2. Related Work

In this chapter we critically analyse other systems and papers that have researched related problems to our domain or have been especially influential to our solution choice. Section (2.1) talks about previous systems for code similarity detection and their utility, classifying them by the approach they take at comparing the source code files (some are aimed at finding structural matches, others search for certain patterns, identical content, etc.). In Section (2.2) we critically discuss the paper that influenced our choice for the conceptual representation of source code and present their algorithm for comparing two conceptual graphs. We show that their method can be improved and extended and present our proposed solution in Chapter (4) and Chapter (5).

2.1. Detecting source code similarity

Code duplication and plagiarism detection have in the past years become very significant problems in field of software engineering and also very active research areas. Many tools[2] and techniques based on source code comparisons have been proposed in the literature that solve these issues (good reviews on such tools can be found in (Chanchal, et al., 2009) (Rattana, et al., 2013)).

Source code matching algorithms that have been implemented before now can vary depending on the approach they take at modelling the source code or on the degree of similarity they aim to find. Systems such as PMD[3], Simian[4] or CCFinder (Kamiya, 2002) that use patter-matching or tilling algorithms to find pieces of duplicated code in large scale applications, represent the source code as tokens or lexical entities that can be either lines of code or programming language tokens. They are known to be fast, although simple structural changes (e.g. modification of data structures) of the code can affect their accuracy.

Systems that compare the structural properties of the programs, on the other hand, have been shown (Wilkinson, 1994) to be much more effective at measuring similarity. Tools such as MOSS[5],

[2] List of free source code similarity detection tools - http://www.ics.heacademy.ac.uk/resources/assessment/plagiarism/detectiontools_sourcecode.html
[3] PMD: Project Mess Detector. http://pmd.sourceforge.net/.
[4] Simian: Similarity Analyser. http://www.harukizaemon.com/simian/index.html.
[5] MOSS: Measure Of Software Similarity. http://theory.stanford.edu/~aiken/moss/.

YAP3 (Wise, 1996) or JPlag[6] represent programing structures as string tokens that they then compare using string based distance. They are mostly used to detect plagiarism in student's assignments and although they do not provide much written documentation about their internal algorithm, they have been shown to be vulnerable to code reordering (Hage, 2010).

Other systems that explore the structured nature of the source code, parse the programs into different graph-based data structures from which they extract different metrics or perform structural comparisons. For example, in (Yang, 1991) source code similarities are found by comparing the source code *Parse Trees*[7] which express the syntactic structure of the grammar describing the programming language. One of their disadvantage is the fact that the nodes are actual grammar tokens and literals, being a much too verbose representation and providing no abstraction layer. *Abstract Syntax Trees*[8] structures on the other hand (used, for example, in source code evolution analysis by (Neamtiu, et al., 2005)), provide some abstraction compared to the parse trees, but they are still very much detailed, preserving information about whitespaces, punctuations and similar information that we would not need in our representation. Graph-based techniques have been used extensively in the past (a good review of the past 30 years of graph matching algorithms can be found in (Conte, et al., 2004)) and are still a very active research area.

A recent paper (Bhattacharya, et al., 2012) showed how different graph-based metrics extracted from ASTs can be used to detect differences and similarities in structure across programs and can help developers to better understand how software evolves and changes over time. Their study is based on extracting and monitoring different code-based graph metrics across several large open source programs. Of particular interest is their use of the *Node Rank* metric to assign a numerical weight to every node in a graph that represents the relative importance of that node in the program. This metric could be very useful in other graph matching algorithms, where determining which part of the programs are more important to be mapped is essential in order to reduce the algorithm's complexity. Although they prove that a graph-based representation can capture many relevant properties of a software system, their solution is adapted for analysing how software evolves and not for a general purpose code similarity measurement.

---

[6] JPlag: Detecting Software Plagiarism. https://jplag.ipd.kit.edu/
[7] Parse Tree - https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Parse_tree.html
[8] Abstract Syntax Tree - http://www.cse.ohio-state.edu/software/2231/web-sw2/extras/slides/21.Abstract-Syntax-Trees.pdf

2.2. Source code matching on Conceptual Graphs

Sowa's (Sowa, 1984) formalism of Conceptual Graphs (Section (3.1)) has often been used as an abstract layer for representing data in many retrieval and classification systems. Good results have been reported for document retrieval (Montes-y-Gomez, et al., 2000), modelling of complex medical information (Kamsu-Foguem, et al., 2013) or semantic search for structured/multimedia documents (Jiwei, et al., 2000) where WordNet[9] is used as the main concept hierarchy. Although all these document types have much in common with source code files (where structural features can be extracted from the content), very few papers explore the advantages of representing source code as conceptual graphs.

An influential and successful system that implements the conceptual graph formalism to describe source code documents was proposed by Mishne and De Rijke in (Mishne & De Rijke, 2004). Their main contribution is a notion of source code contextual similarity based on conceptual graphs that tries to explore both the structure of the source code and also the content inside the nodes, with reported good results for programs written in the C programming language.

Based on the conceptual graph definition, they create a taxonomy for source code elements (the support of the graph) formed by different concept types that represent actual programming constructs such as *Assign, Function, Variable, Struct* and others. Then they give a set of relation types (*Condition, Contains, Comment, Defines,* etc.) and indicate which concepts they can connect and what referents can each concept type have. In order to construct the graph from a source code file they define an extension of the C programming language parser that allows them to create the conceptual graph in the same time the compiler creates the Abstract Syntax Tree. Although this method proves fast it involves defining graph construction procedures and adding them into certain rules of the grammar language, process which can become very burdensome for some complex programming languages.

For comparing two conceptual graphs they propose a contextual similarity measure that compares the graphs node-by-node using the information stored in each concept. However, they do not actually compare the structure of the graphs – instead they augment each concept node by embedding some structural information regarding the concepts that are adjacent to it, taking into

[9] WordNet: A lexical database for English. http://wordnet.princeton.edu/

account a numerical weight that it is associated with each concept type in the graph (they manually assign weights for each concept and relation type depending on their importance, for example, they can view a *String* concept as more important than an *If* concept – which is a subjective interpretation).

The concept similarity measure is defined as the product of the *concept type similarity* (which is 1 - if the two concepts have the same type or one derives from the other, and a smaller value otherwise) and the *concept referent similarity* (a content similarity measurement) normalized by their *weight*. The content matching algorithm used is in some level rudimentary (the *Levenshtein string-distance*[10]), because this mechanism does not take into account certain aspects of the source code anatomy that our system is sensible to (e.g. type hierarchies such as, if two *Variable* concept nodes are to be matched, say *int i* and *long j*, then the string distance function will yield a 0 similarity score, although both variables are holding integer values).

Finally their work is evaluated by carrying out a number of experiments and comparisons with other baseline retrieval models. Although they leave much room for improvement (in terms of optimization, as their algorithm works in $O(|G|^3)$, the large amount of free parameters used, finding a better content matching algorithm, extending the support defined for conceptual graphs to include more programming constructs), their results strongly suggest that using conceptual graphs to represent source code files and performing a graph matching algorithm that compares both structural and content features, can help in assessing code similarity more accurately.

2.3. Conclusions

In this chapter we have identified and critically analysed related systems that perform source code matching with the purpose of finding similarities between two programs. In Section (2.2) we discussed in detail Mishne and De Rijke's paper on source code retrieval as their approach and results influenced us to represent our source code files as conceptual graphs. In our project we propose to extend their work on representing conceptual graphs from source code and to use a new algorithm for mapping two such representations. We base our new approach on the *Analogical*

[10] Levenshtein distance. http://www.comp.dit.ie/bduggan/Courses/OOP/EditDistance.pdf

Reasoning process (described in the next chapter) that allows us to find detailed correspondences between two domains and create new specifications in the target code. We also mentioned a recent graph metric developed by Bhattacharya, et al. (Section (2.1)) which we will use in determining which parts in a program are the most important and relevant to the mapping process.

Although many systems that compare source code have been proposed in the past, little research exists towards the reuse of formal specifications. A notable example is the work in (Park & Bae, 2011) on *UML* diagram specification matching based on Gentner's *Structure Mapping Theory*, which we also use in our solution and will be described in the following sections.

Thus, in the *Arís* project we come with a novel approach that addresses the problem of source code matching with the purpose of reusing previously verified code and reducing the efforts in writing specifications. We first present the background knowledge of our system in Chapter (3) and then give our proposed solution in Chapters (4) and (5).

3. Background

In this chapter we formally describe different concepts, terminology and algorithms that are used in our proposed solution in Chapter (4) and Chapter (5). Section (3.1) presents the *Conceptual Graph* structure that is a knowledge representation formalism we use in order to model the structural and content features of the source code. Next, in Section (3.2), (3.3) and (3.4) we describe the abstract *Analogical Reasoning* process that will help us identify detailed correspondences between two conceptual graphs and create new specifications for our target (unspecified) code. Finally, Section (3.5) talks about how analogical reasoning can be applied in practice by describing the *Incremental Analogy Machine* algorithm and then we give our conclusions regarding this chapter in Section (3.6).

3.1. Conceptual Graphs

A Conceptual Graph [CG] can be described as a powerful inference system and *knowledge representation*[11] language that was introduced by Sowa in 1984 (Sowa, 1984) with origins from the semantic networks used in Artificial Intelligence and from Charles Sanders Peirce work on existential graphs. They were also devised from linguistic and philosophical grounds (Sowa, 2000), so they reside on a very solid and diverse theoretical background. Although they were introduced decades ago, they can be easily correlated to modern object-oriented and database features. As Sowa said in his book, a conceptual graph "can serve as an intermediate language for translating computer-oriented formalisms to and from natural languages" being humanly readable and at the same time computationally feasible.

A conceptual graph is a directed, finite and bipartite graph in which a node has an associated type (can be either a *concept node* or a *relation node*) and a *referent* value (or marker) that can refer to a generic (usually denoted by a "*" symbol) or particular instance of that node. Concept

[11] Knowledge representation - http://groups.csail.mit.edu/medg/ftp/psz/k-rep.html

nodes usually represent entities, attributes, states and events in the knowledge domain, while relation nodes show how they relate to one another (Figure 3).

Figure 3. Example of a conceptual graph structure. This can be read as: *"The relation of Concept 1 is a Concept 2"*. The arrows indicate the direction of the reading and they also express a hierarchical order.

A conceptual graph needs a *support* or *cannon* that defines different rules, syntactic constraints and background information about the specific knowledge domain that it is built upon. This notion of support contains the following:

- A set of *concept types* that is structured as a finite connected lattice[12] in which nodes are disposed in a "is-a-kind-of" hierarchical order. For example a relation that connects two adjacent concept nodes A and B, denotes the fact that B "is-a-kind-of" A.
- A set of *relation types*.
- A set of "*star graphs*" which indicate for all relation concepts, which other concept types it is allowed to connect.
- A set of *referent sets* for concept nodes that will help to distinguish between generic entities and individual ones. Each set of referents must have at least the generic "*" referent.

A more formal definition of a support is a 4-tuple $S = \langle T_c, T_r, G, R \rangle$ where:

(1) T_c, the concept type set defined as a lattice with $\leq$ as order, 1 as supremum, 0 as infimum

(2) T_r, the finite set of relation types, $T_c \cap T_r = \emptyset$.

(3) G, $\{G_{ri}, r_i \in T_r\}$, the set of "star graphs" every G_{ri} is built like this: every node in G_{ri} is labelled by the corresponding element ri of T_r, thus every such kind of node has an ordered set of neighbors that are pairwise non-adjacent and each one of them is labeled with a concept from T_c.

(4) R is the countable set of individual referents. There also exists a generic referent (*) such that $\forall r_i \in R_t : r_i < *$, where $<$ is the order defined by the lattice.

[12] Lattice - http://mathworld.wolfram.com/Lattice.html

Based on this definition of a support, a conceptual graph can be expressed as a 5-tuple $CG = \langle S_C, S_R, E, ord, lab \rangle$ where:

(1) S_C and S_R refer to the concepts and relations in the graph; $S_C \cap S_R = \emptyset$ and $S_C \neq \emptyset$.

(2) E represents the set of edges in the CG: for every $r \in S_R$, edges connected to the relation r are totally ordered by the defined ord function.

(3) Every vertex in the graph has a label defined by the function lab such that for a given node c, its label $lab(c)$ is a pair $(t, r), t \in T_C \ and \ r \in R$.

An example of a support for conceptual graphs can be seen in Figure 4 (the concepts are represented as simple text and the lines denote the hierarchical order). Figure 5 shows two examples of conceptual graphs that can be built on this support.

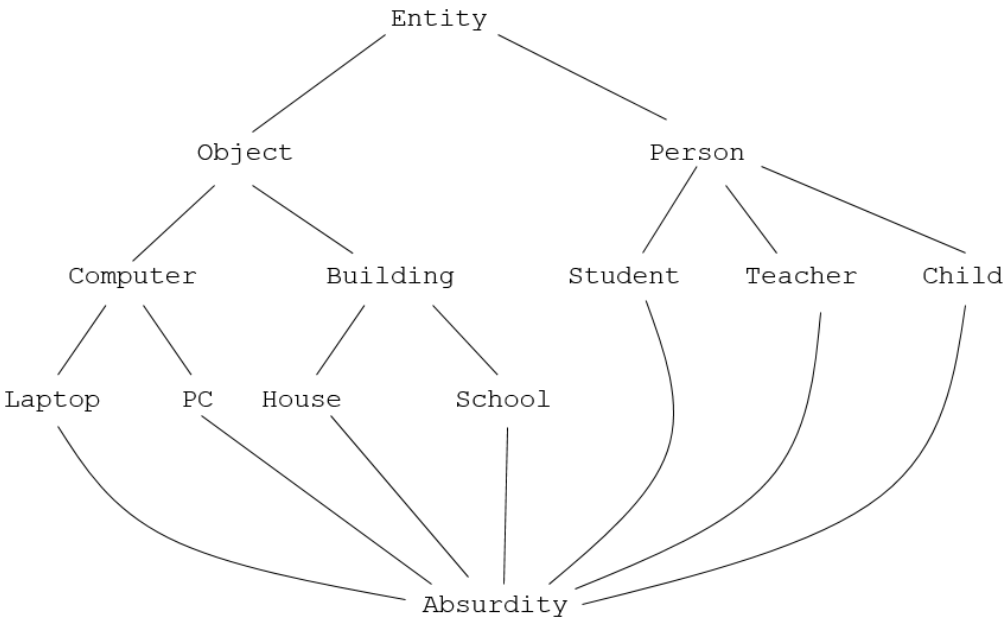

Figure 4. Example of a partial support for a conceptual graph.

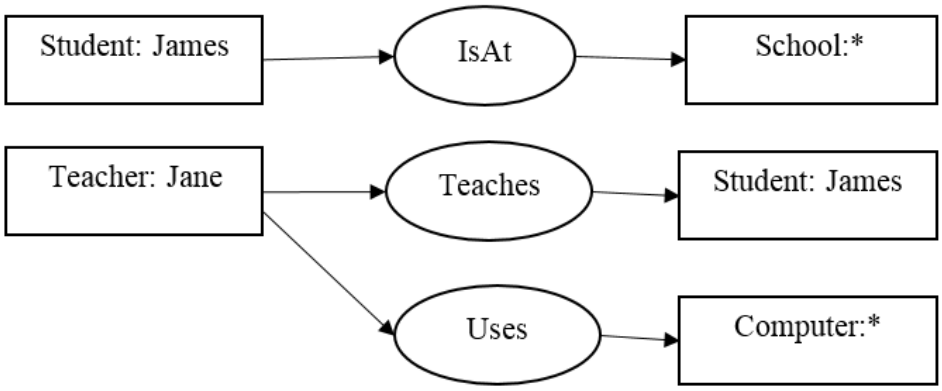

Figure 5. Example of conceptual graphs based on the given support in **Figure 4**. *James* is the individual referent value for the concept *Student* and * denotes the generic referent (unspecified type).

After presenting the general basics about conceptual graphs, in his book, Sowa then describes a set of different operations and extensions that can be performed on them: morphisms, projections, specialization, generalization and others. Because we will not use them later in our system, we will not discuss them further here. More detailed introductions on Sowa's conceptual graphs can be found in (Chein & Mugnier, 1992) and (Polovina, 2007).

In the past decades conceptual graphs have been implemented in a wide range of information retrieval applications (Montes-y-Gomez, et al., 2000) (Chein & Mugnier, 1992), natural language processing, database design and also on source code retrieval (Mishne & De Rijke, 2004) (paper that is influential to us and we discussed in detail in Section (2.2)). The fact that they can be manipulated using graph-based techniques makes them a very attractive and powerful structure[13]. In addition they also provide a rich knowledge representation that facilitates developing inferences, making them fit for representing problems in an analogical reasoning framework that we describe next (Section (3.2)).

3.2. Analogical Reasoning

While CG are used to represent source code, we also need an approach to compare two such graphs. We propose to use Analogical Reasoning (AR) because as we shall see this can be

[13] A set of useful tools for manipulating CGs can be found at - http://conceptualgraphs.org/

effectively reduced down to graph mapping[14]. Analogical reasoning is a very basic but fundamental cognitive ability of human kind. We apply analogical reasoning all the time when we want to understand a new concept or extend our understanding about an old one. As Gentner and Smith (Gentner, 2006) (Gentner & Smith, 2012) describe it, it is the key process in scientific discovery, problem-solving, decision-making and categorization, being a very active research area of Artificial Intelligence and Cognitive Science.

In every analogical process we find a familiar situation (known as the *base* or the *source* domain) and try to match it with less familiar situation (known as the *target* domain) that we are trying to better understand. A classic analogy (Gentner, 2006) example is between the structure of the atom and the solar system[15] – as planets revolve around the sun so do the electrons revolve around the nucleus in the atom domain. Developing an analogical process like this, however, may involve many steps, but the most important (Keane, et al., 1994) ones are described below:

1. **Representation**. In order to find a solution to a problem using analogical reasoning, one has to first represent the problem in a meaningful form. (Novick, 1988) has shown that the form in which a problem is represented affects the later success of the analogical transfers.
2. **Retrieval.** Given a target situation, the retrieval phase of analogy focuses on finding the best candidate to match it with. This usually involves searching through a database of situations and retrieving the most similar one to the target.
3. **Mapping.** This is the core of analogical thinking as it is responsible for discovering which elements of the base domain can be matched to which elements from the target domain. As there can be many different ways of mapping two situations (by object type, attributes, etc.) it can become a very complex and computationally expensive process.
4. **Transfer.** Based on the acquired mapping between the two domains, new knowledge is generated and transferred into the target.
5. **Evaluation.** After the analogical mapping is finished, the transferred knowledge (inferences) need to be validated in order to establish whether the new knowledge can be applied to the target domain.

[14] and the NP hard task of finding the largest common sub-graph (LCS) between two graphs.
[15] A comparison first proposed by the Nobel Physicist Ernest Rutherford around 1914.

In *Arís* we use analogical reasoning for developing inferences and generating new information in our target code (which we want to formally specify using the existing specifications from the base problem). We use the conceptual graph formalism described previously as the representation method for modeling source code files and the *Source Code Retrieval* module (Pitu, 2013) for retrieving the most similar solution to our target problem. In the next section, we describe the theory behind our analogical mapping algorithm for matching two conceptual graph representations.

3.3. Structure Mapping Theory

The most important process and unique to analogical reasoning is *Analogical Mapping*. The process takes as input two structured representations of the base and target domains and finds the detailed collection of correspondences between them (Gentner, 1983): linking particular elements from the base domain with particular elements in the target. It has received a lot of attention from the research community and many computational models have been developed (Gentner & Forbus, 2011) that implement various versions of it. The most influential theory on analogical mapping is Gentner's S*tructure Mapping Theory* (SMT) (Gentner, 1983). It involves aligning the base and target domains by finding a structural similarity between them and developing candidate inferences (from the base to the target) following this alignment.

The SMT theory proposes some constraints (that are also referred to as *informational constraints*) for the analogical mapping process, of which of particular interest are:

- **Structural consistency**: There must to be a one-to-one mapping between the base and the target items. This means that all the ambiguous matches that may occur (one-to-many or many-to-one) have to be discarded. Also, if a correspondence between two objects is found, then the correspondences between their arguments should also be included in the mapping.
- **Systematicity**: Highly connected groups are preferred over independent ones for developing the mapping.

Gentner (Gentner, 1983) demonstrated in her experiments that finding this kind of *structural isomorphism* (a one-to-one matching) between the target and base domains is essential for the

mapping process. Other informational constraints regard the *similarity* of the objects being matched (for example (Gentner, 1983), a constraint my ensure mapping only identical objects) or concern the importance/relevance of the current mapping as people usually prefer matches that are pragmatically more important or more goal relevant than other alternative mappings (these are also called *pragmatic constraints* (Holyoak & Thagard, 1989)).

Other influential factors in the analogical mapping process are the working memory capacity and the background knowledge constraints described in (Keane, et al., 1994) as *behavioural constraints* which have the advantage of better simulating people's performance at the analogical mapping task.

The Structure Mapping Theory was best identified (Gentner, 1983) with graph-matching as the means to efficiently find analogical comparisons between two domains. By representing our base and target source code files as conceptual graphs (which as presented in Section (3.1), are an abstract layer that describe the relational nature of the source code), structure mapping allows us to extract detailed correspondences between them. Previous work has been done by (O'Donoghue, et al., 2006), where graph matching based on Gentner's structure mapping was used to process geographic and spatial data.

3.4. Analogical Inference and Pattern Completion

Although analogical mapping is viewed as the central process in analogical reasoning, the process of generating post-mapping inferences based on the mapping found can be considered in certain situations just as important because it can give a better understanding of the target/problem domain by completing missing information or even generate a solution for it. Gick and Holyoak have shown in their study on analogical problem solving (Gick & Holyoak, 1980) that this process comes naturally to people and that we can easily transfer missing knowledge from the source into the target domain.

After successfully mapping two domains together, we can extract the detailed correspondences between them such that each element in the source has a correspondent element in the target. The process of generating new *analogical inferences* (knowledge) based on this acquired mapping has been referred to by (Holyoak, et al., 1994) as *pattern completion*. In their book they give a simple

pattern completion algorithm *Copy with Substitution and Generation* (CWSG) that has been successfully used in different analogical reasoning computational models (for example ACME-*Analogical Constraint Mapping Engine* (Holyoak & Thagard, 1989) or in the structure mapping algorithm developed by (O'Donoghue, et al., 2006)).

Basically, the CWSG algorithm first finds a statement S and a relation $r(a, b)$ with attribute objects a and b in the source domain that does not have a corresponding mapped statement in the target. Next, based on the fact that both the relation r and its attributes are mapped accordingly with elements in the target ($r \rightarrow r'$, $a \rightarrow a'$ and b $\rightarrow b'$), the algorithm creates and transfers the new statement S': $r'(a', b')$ into the target domain. This process is very useful in our system because it enables us to generate new specification statements in our target code, using the mapping found with the incremental matching algorithm presented in the next section.

3.5. Incremental Analogy Machine (IAM)

The Incremental Analogy Machine (IAM) is a computational model originally developed by Keane and Brayshaw (Keane & Brayshaw, 1988) in 1987 based on Gentner's structure mapping theory and that implements both informational and behavioural constraints using serial constraint satisfaction (Holyoak & Thagard, 1989). Given the base and target domains it constructs a near optimal mapping incrementally, by selecting and matching small portions of the base domain, rather than matching every element in the domain at once.

The IAM algorithm as Keane et al. describe it (Keane, et al., 1994) is given in Table 1 below.

1. **Select the seed group.** Form groups of connected elements in the base domain and order them by some assigned ranking. Take the first such group in the ordered list as the seed group.
2. **Select the seed match**. From the seed group, select an element and try to find a good match for it in the target domain.
3. **Find isomorphic (one-to-one) matches for the group**. For all the elements in the selected group, try and find a mapping with elements from the target domain by applying a set of constraints (structural, similarity, pragmatic).
4. **Find transfers for the group.** Add candidate inferences to the mapping derived from the previous matches found.

5. **Evaluate the group mapping.** If the mapping found is not optimal, backtrack to Step 2 and find an alternative seed match. If there is no better seed match, then backtrack to Step 1 and find another group as the seed group. Otherwise, proceed to the next step.
6. **Continue mapping the other groups.** Try to find successful matches for the remaining unmapped groups and incrementally add them to the inter-domain mapping (Step 1 to Step 5).

Table 1. IAM Algorithm

IAM has been compared (Gentner & Forbus, 2011) (Keane, et al., 1994) with other computational models that implement analogical mapping and has been shown to obtain good performances (and even outperform other models that do not implement behavioural constraints). The previous optimal and greedy search strategies of another analogy model (SME - (Falkenhainer, et al., 1989)) subsequently also adopted Keane's incremental strategy.

The **main advantages** of using the IAM algorithm when comparing two structural representations are:

- Reduced processing complexity (can function in a limited working-memory capacity). Because it is an incremental process that iteratively finds and adds new mappings between the target and base domains it is much faster than trying to map all the elements in a domain.
- Possibility of backtracking. If the acquired mapping is not evaluated as being successful (usually, when less than half of the elements have been mapped) it can go back and find alternative mappings.

The **challenging aspects**, however, of using the IAM algorithm are:

- Selecting a good seed group choice criteria. Choosing an appropriate method of ranking the groups of elements in the base domain is very important for finding a successful seed group to be used in the incremental process.
- Selecting a good seed match choice criteria. After selecting a seed group, the algorithm has to find a seed match (the first *valid* (legal) mapping between an element in the base and an element in the target domain).
- Defining a set of match rules and constraints. In order to determine what it means for a match to be *valid* certain constraints must be satisfied. As IAM produces a *one-to-one* mapping

between the source and target domains, specific rules must be used in order to discard ambiguous matches such as *one-to-many* or *many-to-one*.

In *Arís* we use the IAM algorithm to build an incremental mapping between two conceptual representations of source code files by iteratively selecting and matching sub-graphs. We use a recently developed and promising graph metric as a selection criteria in picking the seed group and the seed match. As match constraints we define different similarity, structural and pragmatic rules that help us achieve an efficient and one-to-one mapping as IAM requires.

3.6. Conclusions

In this chapter we presented different terminology and background concepts that we will refer to in the following sections as we describe our proposed solution. At the beginning we talked about the Conceptual Graph formalism that we use to represent our input source code files (in Chapter (4)) as it has the great advantage of capturing not only the content information from the source code but also its structural properties, in a manner that is inter-leaved. We then described the Analogical Reasoning process with emphasis on its key task of Analogical Mapping. As Gentner's Structure Mapping Theory was best identified with graph-matching as the means to efficiently find analogical comparisons between two domains, we then showed how analogical reasoning can be applied computationally by using the Incremental Analogy Machine model. The following chapters of this paper describe our prototype, evaluation and results.

4. Source Code as Conceptual Graphs

As described earlier in Section (3.1), conceptual graphs are a powerful structure for representing data in which the information is stored not only in the content but also in the structure making them fit to model source code documents. In this chapter we detail (Section (4.1)) the concept and relation types (graph support) that we allow in our conceptual graphs and how they can be constructed from the source code files (Section (4.2)). We also give some examples of graphs built with our system and finally in Section (4.3) we present some limitations of this choice of representation.

4.1. Source code concepts - hierarchy and semantics

In order to build our conceptual graphs, we studied the method described by (Mishne & De Rijke, 2004) in which a partial support for representing C source code files is given. We extended their model so that we could represent more programming features and adapted it to the C# programming language. We start by describing the concepts and relations that are allowed in our graphs and then indicate how they can be connected together and what possible referents each concept type can have.

After examining various C# source code files and based on (Clayton, et al., 1998) and (Mishne & De Rijke, 2004) we constructed a hierarchy of concepts (the support of the graph) which is presented in Figure 6. In Table 2 below we briefly describe the meaning of every concept.

Concept Type	Description
ASSIGNOP	An assignment of a value to a field or variable (including assignments such as "+=", "*=")
BLOCK	A set of concepts that are structurally grouped together (for example, code that is inside curly brackets {...})
CLASS	A declaration or definition of a class
COMPAREOP	A binary comparison operator like "<=", "!=", etc.
ENUM	A declaration of an enumerated set of values
FIELD	A declaration of a variable directly in a class (a class attribute)
IF	A conditional branch statement
LOGICALOP	A binary logical operation, such as "OR", "AND", etc.

LOOP	An iterative process that depends on a condition
MATHOP	A mathematical operation like "+", "*", etc.
METHOD	A declaration or definition of a function inside a class
METHOD-CALL	A method invocation (execution)
NAMESPACE	Defines a scope that can contain one or more classes. Useful for code organization
NULL	A null reference (keyword *null* in C#)
STRING	A textual entity (numbers are also represented as strings)
SWITCH	A conditional statement that has multiple branches
TRY-CATCH STATEMENT	A *try* block followed by one or more *catch* clauses, which specify handlers for different exceptions
VARIABLE	An entity declared in the program that holds values during execution

Table 2. Concept types allowed in the graph and their description

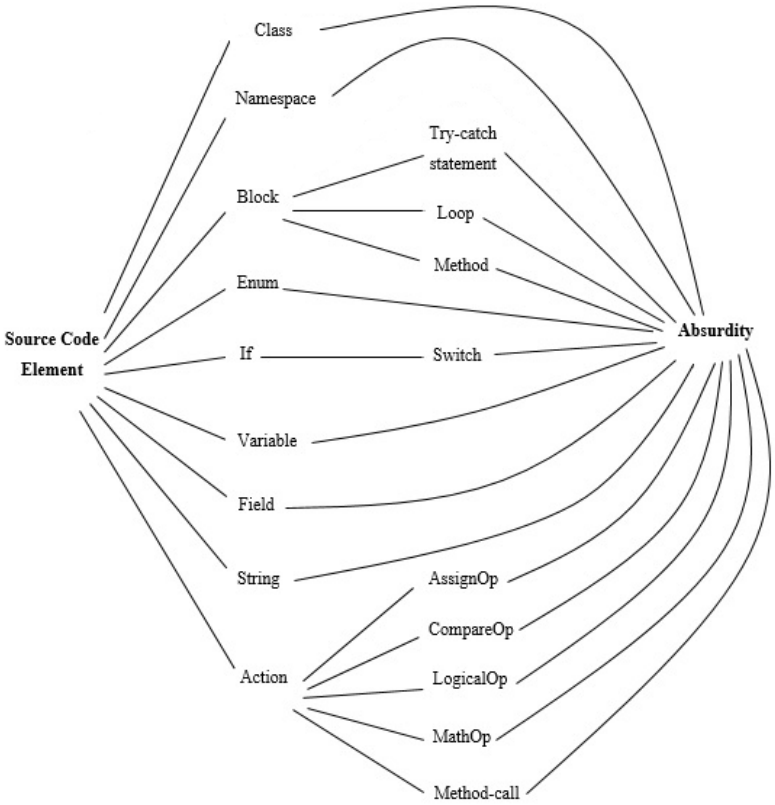

Figure 6. Hierarchy (from left to right) of source code concepts in the conceptual graph cannon.

In order for our conceptual graph support to be complete, we must define a partial order for the concepts, but since we will not use any features of conceptual graphs that would require such an ordering, like (Mishne & De Rijke, 2004), we can define any arbitrary order, for example the lexicographical one.

Next, we present the types of referents (indicating whether the node is an individual entity or a generic one) and also the types of relations that can connect concepts together (Table 3).

- *Variable, Field, Enum, Method, Class, Method-Call* – will always have an individual referent which is a programing language identifier (for C#, identifiers conform to the *Unicode Standard*[16])
- *String* – will always have an individual referent which can be text of any length
- *Block* – can either have a generic ("*") referent or an individual one (an identifier as above)
- All other concepts can have only the generic ("*") referent

Below you can see a list of possible relation types and which concepts they can connect. We used a shortcut notation "*Action*" to imply any of the concepts *Assign*, *CompareOp*, *LogicalOp*, *MathOp*, *Method-call.*

Relation Type	From Concept	To Concept	Description
Condition	If	Action String Variable Field	Describes the conditional statement within the *if* clause.
	Loop	Action String Variable Field	Specifies the conditional statement that determines the iterative process
Contains	Action	Action String Variable Field	The action can use (or depend on) other concepts

[16] Unicode Standard - http://www.unicode.org/standard/standard.html

	Block	Action String Variable If Enum Try-catch	A block can contain this concept
	Class	Field Method	A class definition contains this concept
	Enum	String	The enumeration elements are defined as strings
	Method	Block	The method definition contains a block of concepts
	If	Action Block	The branching statement can contain a block with multiple concepts, or just an action
	Loop	Action Block Variable Field	The loop can contain or initialize other concepts
	Namespace	Class	The namespace definition contains the class
Defines	Block	Namespace	A block (usually the root of the file) gives a definition of the namespace
Depends	Block	Namespace	A block can depend (require) other namespaces
Parameter	Method	String Variable	The method definition contains the concept as parameter
	Method-call	String Variable Field	The method is called with this concept as parameter
Returns	Method	Action Variable Field String	The method returns a value

Table 3. Relation types and how they connect the concepts

So far we described what concept types we allow in our conceptual graphs, in what ways they can be connected together and what information they can store about the source code elements they represent. In order to get a better understanding of how a conceptual graph is created we next

present our graph construction algorithm and also give an example showing how a simple program is transformed into its corresponding conceptual graph representation.

4.2. Graph construction algorithm

In order to analyse C# source code files we require a tool that parses the code into an intermediate representation that contains all information about both the structure and content of the document. As (Mishne & De Rijke, 2004) create an extension of a parser for the C programming language grammar, we first looked at something similar for the C# language. We found that *Coco/R* (Mössenböck, et al., 2011) *compiler* generator tool can take the grammar of a programming language and generate a scanner and a parser it – and similar to the compiler it is able to generate the *Abstract Syntax Tree (AST)* representation of the source code (which reflects the logical structure of the compiled program). However, extracting the AST with *Coco/R* involves manually defining its structure and node classes, which although gives full control over the contents of the AST, can become a difficult process and since we wanted a faster way of obtaining a program's structural representation (that also worked for the last version 4.5 of *.NET framework*[17]) we chose to use the *Microsoft Roslyn Project* (Warren, et al., 2012). Roslyn is a system that exposes the C# compiler's code analysis and can provide us directly with the AST from a C# file without any other requirements.

The AST we obtained is quite similar to the conceptual graph representation in that both give a formal description of the attributes and expressions in the code. However, unlike our source code conceptual graphs, the AST is very detailed (containing a lot of unnecessary information such as whitespaces, punctuation marks, etc.) and it does not provide any level of abstraction (like our conceptual graph, where for example, a *Loop* concept can refer to any of *do-while, while, for* or *foreach* statements). Our conceptual graph construction process takes the AST root (using Roslyn) and traverses all its descendant nodes in a *Depth First Search*[18] manner in order to create the corresponding concepts and relations in the conceptual graph. The key to that is the ability to map expressions (nodes) at the AST level to concept types in the conceptual graph (Figure 7).

[17] .NET Framework - http://www.microsoft.com/net
[18] Depth First Search - http://www.cse.ust.hk/~dekai/271/notes/L06/L06.pdf

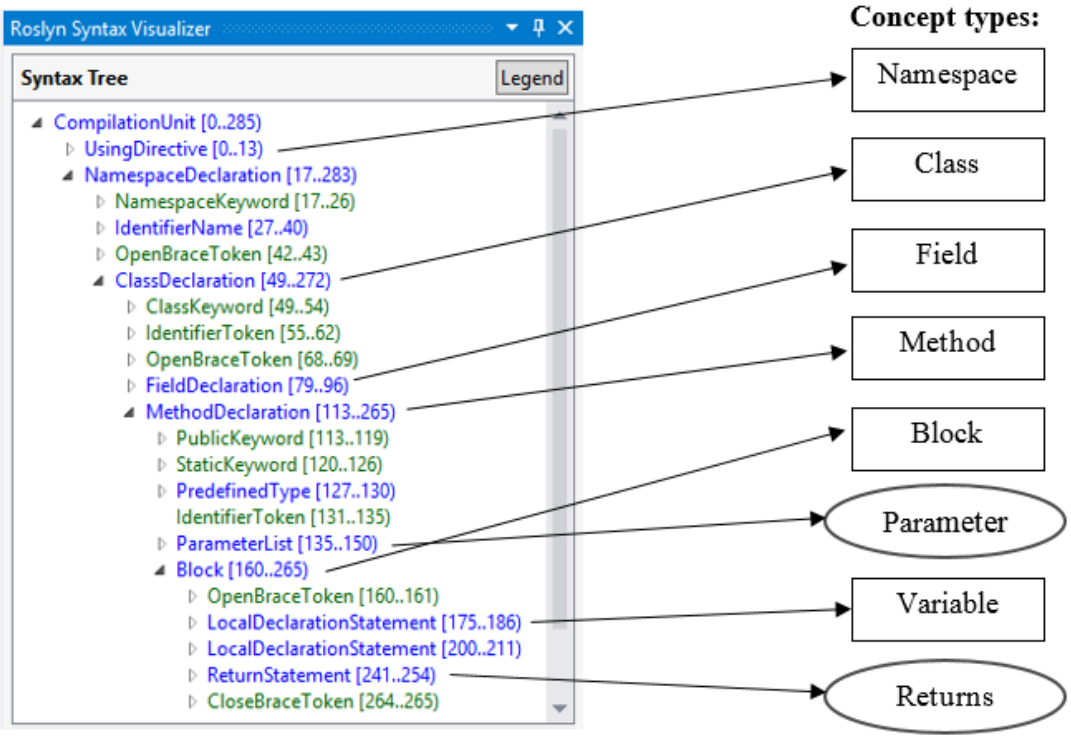

Figure 7. Example of mapping between the Abstract Syntax Tree classes and types of concepts and relations in a conceptual graph (rectangles denote concepts and ovals relations)

Each time we encounter a node of interest in the AST, we create the appropriate nodes in the conceptual graph, which we store using the graph structures in the *QuickGraph*[19] library (another good option would have been to use the *Cogitant*[20] library that provides C++ classes for modelling conceptual graphs, however we did not used it because we opted for .NET as a development and testing grounds platform). Below you can see an example of the conceptual graph construction process: Table 4 describes a simple C# function that sums the first *k* numbers; Figure 8 shows the parsed AST and Figure 9 presents the constructed conceptual graph. A more "real life" conceptual graph, for a more complicated program that contains more than one method and trivial operations, can be seen in Figure 10.

```
public int Sum(int k){
   int s = 0;
   for (int n = 0; n < k; n++)
            s += n;
   return s;}
```

Table 4. `example1.cs`

[19] QuickGraph, Graph Data Structures And Algorithms for .NET - http://quickgraph.codeplex.com/
[20] COGitant - http://cogitant.sourceforge.net/

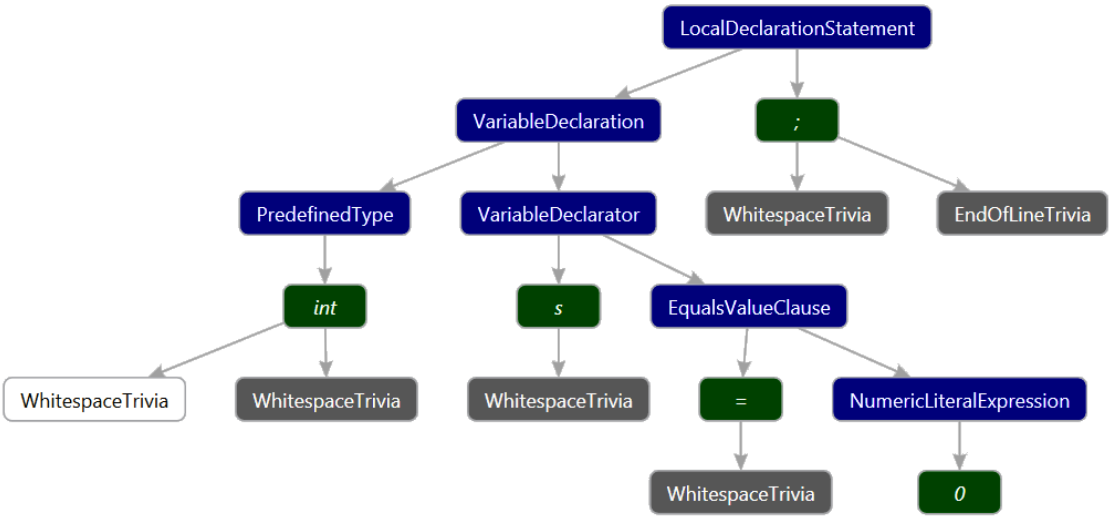

Figure 8. Fragment (describes only the first declaration in the program *`int s = 0;`*) of the AST `example1.cs`.

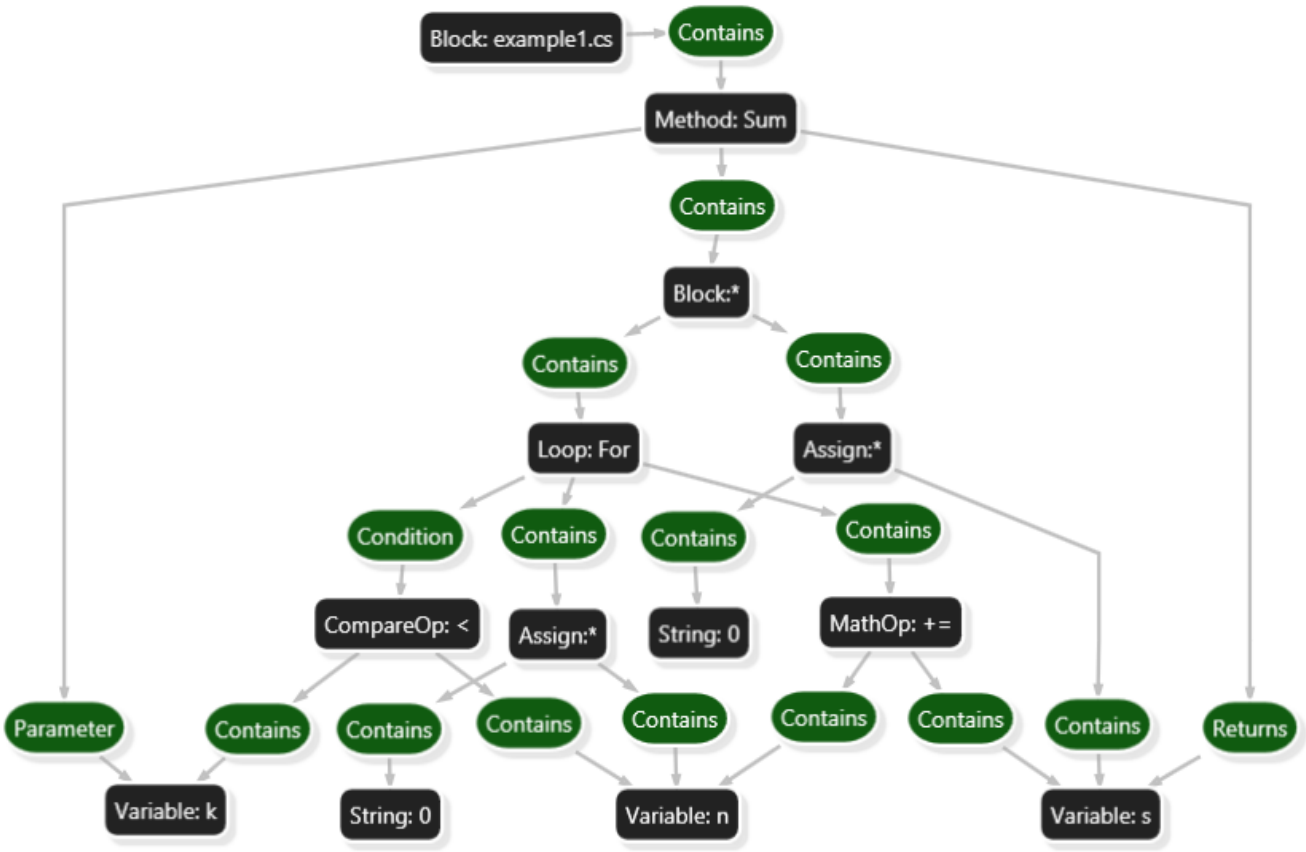

Figure 9. The resulting (full) conceptual graph for `example1.cs` generated with the *Conceptual Graph Visualizer Tool* that we developed in our system using the *GraphViz* .NET library.

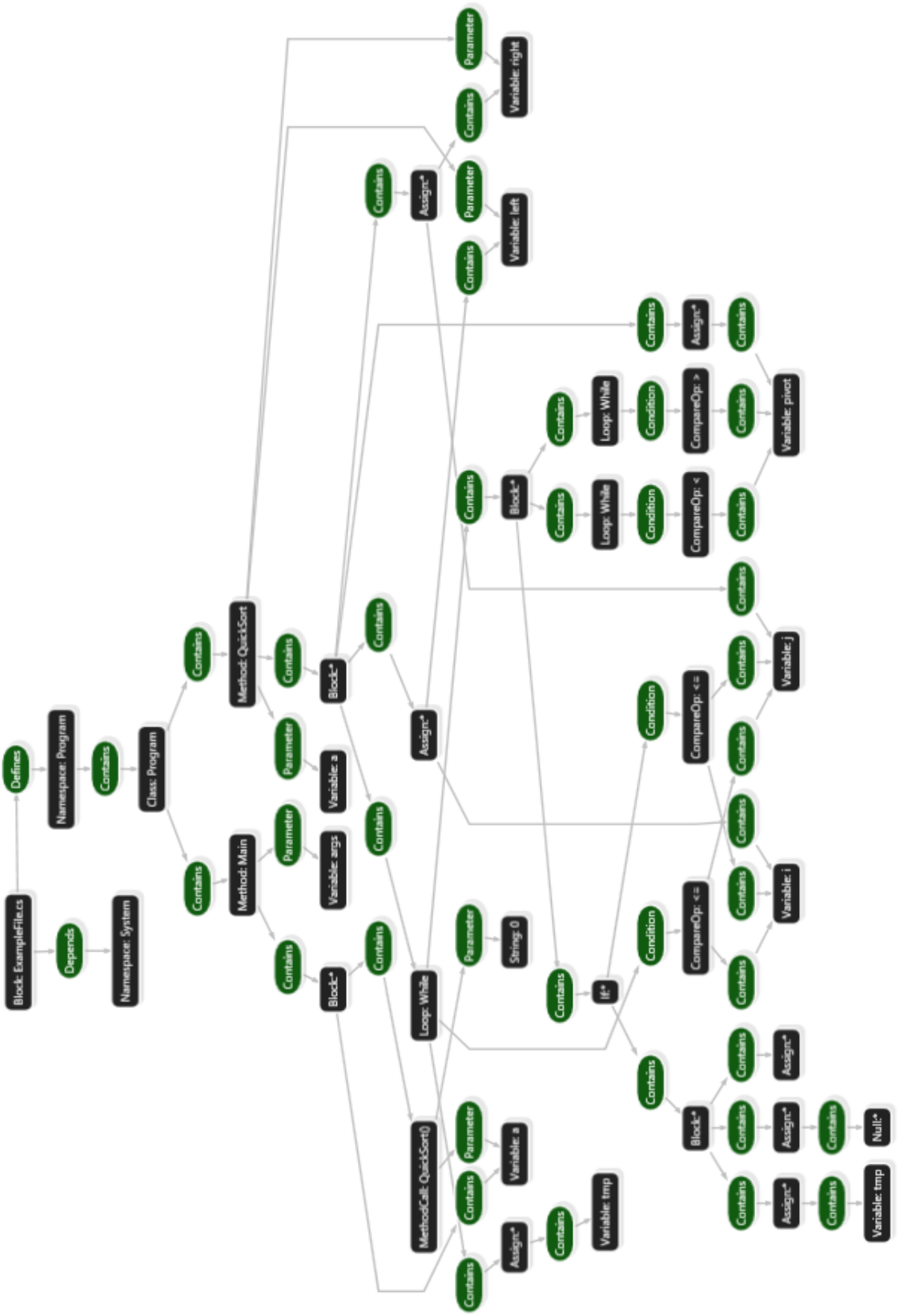

Figure 10. C# class that implements the *QuickSort* algorithm.

4.3. Analysis and conclusions

Although the set of concepts and relations that we defined is not complete (it contains the basic concept types in order to allow us to experiment with our graph matching algorithm, described next in Chapter (5)), it can be easily extended to support more programming features of the C# language such as *properties, delegates, events*, etc. The concepts *Namespace, Class, Try-catch-statement, Field, Null* are an addition to those proposed by (Mishne & De Rijke, 2004) which do not convert some components (for example, pre-processor directives such as "#includes") into concepts at all. However, unlike them, our system does not model comments in the source code (which are natural language descriptions attached to various source code elements). We chose to ignore them as in the *Arís* project, we perform source code retrieval based on extracting semantic and structural information from the programming constructs (*Loop, If, Variable, Block, etc.*). Also, matching code comments is best performed by using *Natural Language Processing*[21] techniques like, for example (Marcus & Maletic, 2001) or (Notkin & Michail, 1999) that implement *Latent Semantic Indexing*[22] or *n-grams*[23]. But such kind of algorithms become less efficient when the source code is poorly documented, which is why most of the tools that match source code ignore the "noise" from the comments.

Another drawback of our conceptual graph representation is that it does not capture any information about the order in which the statements are executed, which in some cases may be useful. For example, if we want to model an "if – else" statement, then an *If* concept is constructed that has a *Condition* relation to an expression and one or more *Contains* relations that represent the statements executed in both the *if* and the *else* blocks. Thus, only by examining the conceptual graph we can't know what statements are executed when the condition is true and when it is false. A solution to this (although in our project we do not need such kind of information) would be to extend the graph formalism to include an *Else* concept that can be mapped to an *ElseClauseSyntax* type in the AST extracted with the Roslyn API. However, extending the graph cannon can also be another possible difficulty, because it requires first to examine the AST classes in order to add another mapping between an AST type and a conceptual graph type that would then be used in the graph construction process described in Section (4.2).

[21] Natural Language Processing - https://en.wikipedia.org/wiki/Natural_language_processing
[22] Latent Semantic Indexing - http://nlp.stanford.edu/IR-book/html/htmledition/latent-semantic-indexing-1.html
[23] N-grams - http://en.wikipedia.org/wiki/N-gram

5. Comparing Conceptual Graphs

5.1. Overview and intuition

As discussed in Section (3.1), conceptual graphs have been used in various fields such as information retrieval, case based reasoning or machine learning. A number of techniques have been proposed for comparing conceptual graphs, most importantly, Sowa's set of projections and morphisms defined in (Sowa, 1984). A notion of *semantic distance* that can measure the distance between two concepts from the support of the graph has also been discussed in (Foo, et al., 1992), although it is not extended to an entire graph. The problem with Sowa's morphisms is that they are too strict (they focus on finding structurally identical graphs or sub-graphs) whereas we wanted our method to be able to match even *homomorphic* graphs, allowing somewhat different structures to be mapped together. Also, as our project is designed to be used in the *Source Code Retrieval* phase of the *Arís* project, this required a more "relaxed" measure of similarity (retrieval models usually permit a certain degree of structural "fuzziness") so that even graphs that don't share the exact same structure (but are related) could be retrieved (see, for example the two graphs in Figure 11). An influential work for us, (Mishne & De Rijke, 2004), that also used conceptual graph comparisons for source code retrieval, defined a contextual similarity measurement with the same goal as us, to combine the structural with the content information stored in the graphs. However, their method does not actually perform a structural comparison (more details about their algorithm are given in Section (2.2)) and their content matching algorithm relies on just a simple string based distance.

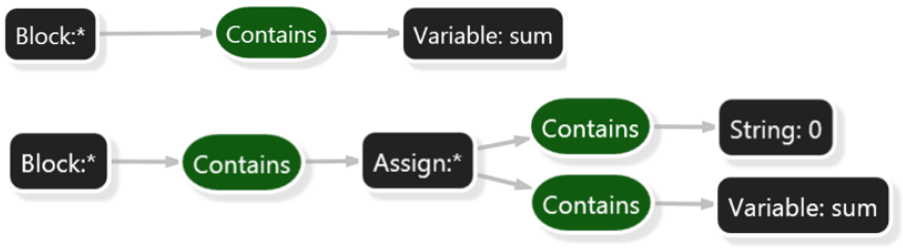

Figure 11. Two related program graphs

Our main motivation in developing a method for mapping source code files is the possibility of transferring specifications from one program to the other, based on the amount of mapped elements found. In this sense, we use the term *source* to refer to each retrieved candidate solution program (with specifications) and the term *target* to refer to our unspecified problem code. The objective is to identify the best mapping between two *isomorphic* graphs (where the two programs can use different identifier names) and also cater for mapping *homomorphic* graphs (with different structural shapes like the ones in Figure 11). But since we opted for the rich representation of a conceptual graph (where each node can have content information), we also wanted to perform a more elaborate content based similarity measurement rather than just the basic string distance (for example, ensure that variables are matched with variables and loops with loops, also, add the ability to indicate that an *int i* and *long j* are two similar statements as they both hold numeric values).

The graph matching problem is known to be difficult (falling into the *NP-complete*[24] class of problems), Bunke (Bunke, 2000) and Conte, et. al. (Conte, et al., 2004) give a good overview over the last 30 years of graph matching applications (including case-based reasoning, machine learning, semantic networks and conceptual graphs), and although polynomial isomorphism algorithms exist for special kinds of graphs, most types of exact graph matching have exponential time complexity in the worst case. (Mishne & De Rijke, 2004) avoid this by doing a node-by-node comparison (in which, based on their defined notion of *maximally similar concept*, they match every concept in a graph *G1* to a concept in another graph *G2* to which it is compared) and manage to bound their algorithm to run in $O(|G|^3)$, but still, considering the sizes of real life source code files, it is not very optimal for a retrieval based system like the *Arís* project).

Our proposal is to use an incremental graph matching algorithm based on the *Incremental Analogy Machine* (IAM) (Keane & Brayshaw, 1988) which is a computational model for analogical reasoning (for more details see Section (3.2)), based on Gentner's structure mapping theory (Section (3.3)). Previous work has been successful in applying analogical reasoning for finding detailed correspondences between two domains, for example (O'Donoghue, et al., 2006) combine Gentner's structure mapping theory with an attribute matching algorithm for mapping geographic and spatial data, or (Park & Bae, 2011) that implement the *Structure Mapping Engine* (SME) for matching UML specifications.

[24] NP-completeness - http://cs.brown.edu/~jes/book/pdfs/ModelsOfComputation_Chapter8.pdf

The main reason for choosing the IAM algorithm is that it allows us to explore the relational nature of the conceptual graphs and also because it is considered (Keane, et al., 1994) to obtain similar results to people's analogical reasoning process: first, because it has the ability to generate complex analogical mappings very quickly and accurately and also, because it can reconsider a mapping and generate new alternative ones. In a nutshell, IAM begins by matching the two largest sub-graphs from the source and target domains. This forms a *seed mapping* from which additional structures from the source and target are added iteratively forming a single mapping between the new code and the previously specified code. The first challenging aspect when applying the IAM algorithm is finding the appropriate "root" concept (the most referenced node in the graph) from which to start the matching process and create the seed match. We will describe our approach concerning this process in the next paragraphs as follows: Section (5.2.1) describes how we extracted and sorted the nodes using the *Node Rank* graph metric, Section (5.2) talks about how we adapted and applied the IAM algorithm for mapping two conceptual graphs, Section (5.2.6) presents the match rules and constrains that must be enforced in order to find *valid* mappings and in Section (5.3) we present the *Copy With Substitution and Generation* algorithm for transferring specification between the source and target programs.

5.2. Incremental matching using the IAM algorithm

In this section we describe, step by step, how we applied Keane & Brayshaw's Incremental Analogy Machine (Section (3.5)) algorithm for mapping two conceptual graphs. Our system receives as input two source code files that have a structure similar to the one in Table 5, for which the graph matching module then finds the detailed mapping and outputs a similarity score based on the extent of the mapping acquired.

```
// using directives, namespace definition
class ClassName{
    // class members
    field_1
    field_2
    …
    // class methods
    method_1(list_of_parameters) {...}
    method_2(list_of_parameters) {...}
    method_3(list_of_parameters) {...}
    … }
```

Table 5. Example a C# program file structure that we receive as input to our system

We first start by constructing the conceptual graph representations for both files (applying the process described in Section (4.2)) and compute the node ranks for each concept in the graphs using the *Node Rank* metric. Next we map the two graphs by selecting parts (sub-graphs) of them and doing a node-by-node comparison in an analogical mapping process (Section (3.5)) composed by the following steps.

5.2.1. Sorting nodes by Node Rank

In this project we experiment using a graph-based metric called Node Rank in order to increase the efficiency of the incremental matching process by mapping nodes based on their relative importance in the graph. In order to do this we sorted the elements in the base and target domains such that the highest ranked nodes represent the most important elements in the program.

Node Rank (NR) is a metric proposed by (Bhattacharya, et al., 2012) and is similar to the *Page Rank*[25] (Brin & Page, 1998) algorithm that represents a probability distribution expressing the likelihood that a person surfing the web will arrive at any particular page. Applied to our problem, the Node Rank algorithm assigns a numerical weight to each node in the conceptual graph, basically measuring the structural importance of that node in the graph (depending on the other nodes that have ongoing or outgoing edges to or from it, see for example Figure 12).

In order to formally describe the recursive calculation of the node ranks in a graph, let u denote a node in the graph, $NR(u)$ its node rank, $IN(u)$ the set containing all the nodes v that have an outgoing edge into u and $OutDegree(u)$ represent the number of edges going out of the node u. Initially, all nodes have an equal node rank (we used $\frac{1}{the\ total\ number\ of\ nodes}$).

An iterative process calculates a new $NR(u)$ in every iteration as the sum over all $v \in IN(u)$:

$$NR(u) = \sum_{v \in IN(u)} \frac{NR(v)}{OutDegree(v)}$$ (Bhattacharya, et al., 2012)

[25] made famous by Google.

The iteration process stops when the values *converge* (when the change in the sum of all the NRs due to one iteration is small enough, e.g. the difference between the old sum and the current one is less than or equal to a $quadratic\ error$ factor, that we set to 0.001, or the iteration limit has been exceed, e.g. > 50). Also, in order to enable convergence, after each iteration we normalize the node ranks so that their sum adds up to one.

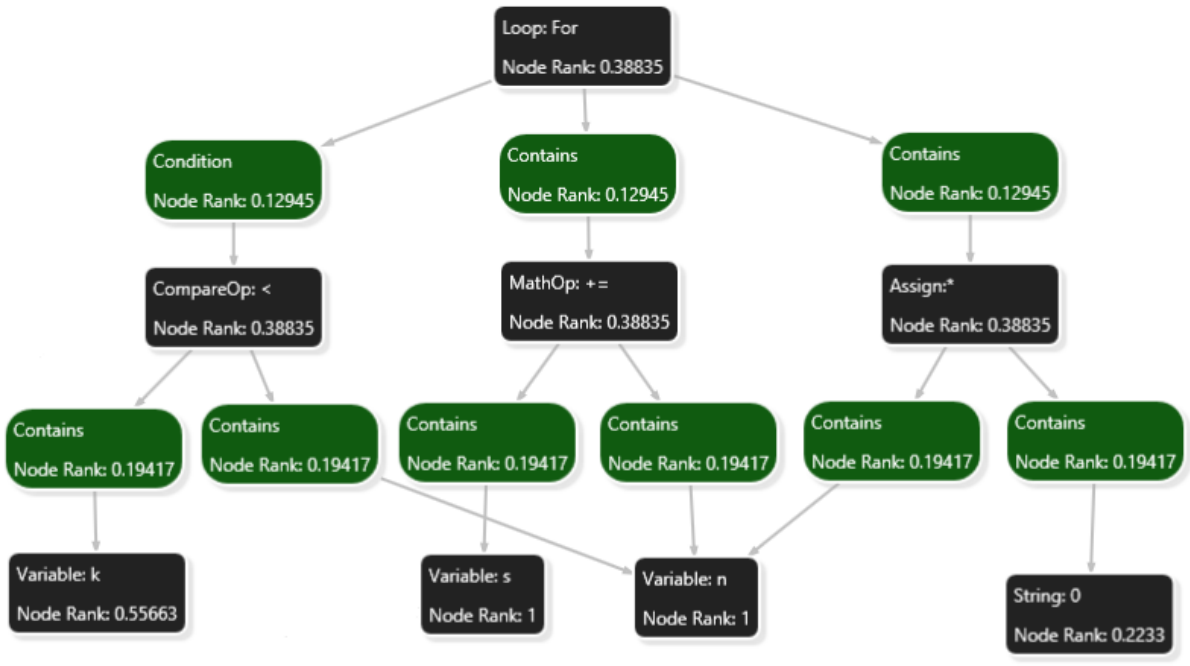

Figure 12. Fragment from the conceptual graph for the `example1.cs` in Table 4 showing the node ranks assigned. The node rank values are normalized to values between 0 and 1 for more clarity. Variables *s* and *n* have the highest NR values because they are important variables in the program, used for the sum calculation and in the return statement.

5.2.2. Selecting sub-graphs

This is the key process that reduces the complexity of IAM's analogical mapping as it incrementally selects portions from the source domain to map with the target, which, instead of mapping all the elements in the base with all the elements in the target in an exhaustive manner, it is much more efficient. The sub-graphs in our case, could be any function in the program, as any function represents an interconnected or systematic group of graph nodes, respecting IAM's initial indication for selecting the *seed group*.

Selecting methods with identical names. Given that two similar source code files that implement related algorithms can possibly also have identical method names, we first preference selecting methods from the base that have identical names with methods from the target domain. This can increase the efficiency of finding fast correspondences between two graphs, although it is more common in cases of source code duplication or plagiarism detection and would not work in a general situation. In cases where the target or the source domain contain multiple instances of the same method (e.g. *overloading*) then we accept the mapping with the highest *similarity score*, which we obtain by evaluating different similarity functions defined in Section (5.2.6).

Selecting methods by their Node Rank order. If no identical function names exist, we then use the Node Rank metric described in Section (5.2.1) and sort all the methods in the graph base domain by their NR values. We then map the methods one-by-one selecting them in a decreasing manner (remember when we described the NR metric as a means of sorting nodes in a graph by their relative importance – thus a method with the highest NR value in a source code file means that it plays an important role in the structure of the algorithm as other functions call/depend on it). This can also be viewed as applying a pragmatic constraint (Holyoak & Thagard, 1989): preferring the elements which are more goal relevant (ore more important) over other alternatives in the analogical mapping process.

The mapping algorithm builds up one single inter-domain mapping, which is used to check and enforce the 1-to-1 mapping constraint required by IAM. In this way, additional sub-graphs are matched and added to the inter-domain mapping only if they are consistent with the previous correspondences found (this implies discarding one-to-many and many-to-one mappings).

Although the IAM algorithm only orders the source domain, in our case this is not enough as it would mean to search through the whole target domain in order to find the most similar functions to the functions in our base domain. Thus, in an analogical way, we also sort by NRs all the methods in the target domain. We then start matching the two domains by comparing the functions one-by-one taken decreasingly by their NRs. The process of finding correspondences between two methods is explained in the following section.

5.2.3. Mapping sub-graphs

After selecting two methods from the source and target domains we begin matching their corresponding sub-graphs (in a conceptual graph every method in the source code is represented by a *Method* concept that is related to other concepts through relation types such as *Contains, Returns* or *Parameter* – thus forming a sub-graph).

Looking at the original IAM algorithm, after finding a seed group in the base domain it continues on to find a *seed match* – a first *valid* match between an element in the seed group and an element in the target domain, before the rest of the seed group is matched. In order to check if two nodes in a CG form a valid match, they must first comply with the mapping constraint that ensures they have the same concept type.

Because we are dealing with a specific type of mapping process that maps two source code methods, we first employ a very simple check between the parameters of the methods. We do this by comparing each parameter of the base domain method with every parameter of the target method (looking at the concepts connected by the *Parameter* relation in the corresponding sub-graphs) and measure a similarity score sim between two concepts (described in Section (5.2.6)) to select the best possible mapping. This ensures that our parameters are matched successfully and that changing their order in the method definition does not influence the matching process. This approach also helps disambiguate later mappings between the method's body implementations.

The rest of the concepts that describe the body of the methods are then sorted by their NRs (similar to the method sorting process) and each concept in the source domain is then mapped to its most similar concept in the target domain by taking each candidate node under decreasing order of its NR value. However, we do not search the whole space of the target domain, instead we use a threshold parameter called $match_{depth}$ that represents the number of concepts in the target to which we compare each concept from the base domain (a *pseudo-code* description of how we compare the concepts can be seen in Table 6).

```
sort BaseDomain decreasingly by NodeRank values
sort TargetDomain decreasingly by NodeRank values
i : = 0; j := 0; match_depth:= 10;
for (i = 0; i < size (BaseDomain); i++)
  validMatch := false; curr_depth := 0; backtrack := false;
  for (; j < size (TargetDomain); j++)
     if (curr_depth >= match_depth)
        if (backtrack = false)
           // if backtrack is possible then backtrack once.backtrack := true
           // otherwise j := i + 1, move to the next elem
        else  j := i + 1; backtrack := false; break;
        end if
     end if
     curr_depth := curr_depth + 1; // count the current comparison
     if (ValidMatch(BaseDomain(i), BaseDomain(j))){
       // resolve many-to-one mappings
       // resolve one-to-many mappings
       // update inter-domain mapping
        …
       validMatch := true;
       break;
      end if
  end for
  if (validMatch == true)
     // find all derived, valid matches
     // update inter-domain mapping
  end if
  j := i + 1; // move to the next corresp target elem of the curr base elem
end for
```

Table 6. Algorithm description of the mapping process between elements in the source method and elements in the target method. Certain parts of the algorithm are left out due to space reasons, however they are explained in the next sections.

In order to ensure that our mappings are consistent we use a Boolean function $valid$ (described in Section (5.2.6)) that determines, based on the similarity score sim, whether a match is accepted as valid and added into our *inter-domain* mapping (consisting of individual elements from the base and their correspondent elements from the target). If a match is not considered valid, we try to match the current base element to the next highest NR valued element from the target and so on, until the $match_{depth}$ threshold is exceeded (in our experiments it was set to 10).

After we've found a valid mapping between two concepts, we then check their immediate context (sub level) in the corresponding graphs for finding other related mappings. We compare each child node of the mapped element in the source to each child node of the mapped element in the target - for example see Figure 13 - and if any comparison is considered a *valid* match then the

match is saved to the current mapping we are building. The algorithm then moves on to match the rest of the elements from base group in an analogical way.

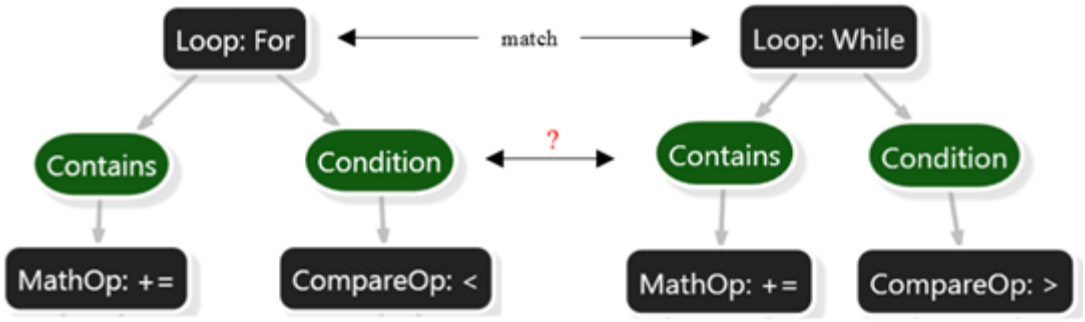

Figure 13. Example of how other mappings are derived after a valid match has been found. At the beginning our inter-domain mapping contains the match {*Loop: For <-> Loop: While*}. We take this mapping and check whether the concepts it relates to (the subsequent, children nodes) can also form valid mappings. In this case, they do, so we also add the pairs {*Contains <-> Contains*} and {*Condition <-> Condition*} to our mapping. We proceed similarly with all the elements in the groups. This process can also be correlated to the *Attribute-mapping* problem (Holyoak & Thagard, 1989), where if two predicates are matched, then their arguments are also mapped accordingly.

Returning to the formal description of IAM, the seed match can be found by trying to map concepts from the source domain (taken in descending NR values) to the first $n < match_{depth}$ elements in the target, using backtracking, until a valid match is found. If no valid mapping can be identified (when none of the elements in the source domain have corresponding similar elements in the target) then this sub-phase of the overall mapping process terminates. A sub-graph mapping is accepted only if at least 50% of the source nodes have been mapped, otherwise it is rejected.

5.2.4. Resolving ambiguities

In order to enforce 1-to-1 correspondences (*isomorphism*) and resolve possible ambiguities we define our inter-domain mapping as a structure holding $< Key, < Key', Value' >>$ triples, where the keys are concepts from the source domain and the values are also $< Key', Value' >$ pairs in which the keys are the corresponding mapped concepts from the target domain and the values are real numbers that represent the sim similarity score obtained. This semantic similarity is used to disambiguate subsequent matches such that if one match in a set of one-to-many mappings has a

greater *sim* score, then it is preferred over other alternatives previously found. Thus an initially poor mapping can potentially be improved at a later stage. For example, if we had an initial mapping *<{CompareOp:<=}, <{CompareOp:>=}, 0.7>>* and later we found another better mapping of the same key *<{CompareOp:<=}, <{CompareOp:<=}, 0.9>>* then the last correspondence would replace the first one. The fact that the algorithm can replace a mapped item by another item can possibly allow non-isomorphic structures to map together, although in practice this happens rarely and can often be attributed to small non-isomorphic details in the code.

Whenever we find a potential valid match between concepts from the source and target domains we check for the following possible ambiguities (you can see the actual code snippet in Table **7**):

1. *Many-to-one mappings*: are when multiple concepts from the source domain are being mapped to the same concept from the target domain.
2. *One-to-many mappings*: are when the same concept from the source domain is being mapped to multiple concepts from the target domain.

In both cases, the algorithm replaces the old mapping with the new one found if it has a higher *sim* score, thus discarding any ambiguities and preserving the 1-to-1 constraint of IAM.

```
// Check content matching
if (ValidMatch(curr_source_elem, curr_target_elem) == true){
  // Save new similarity score found
  new_sim_score = SimilarityScore(curr_source_elem, curr_target_elem);
  // Get the previous base elem that has been mapped to the curr target
elem
  GraphConcept previous_mapp = InterDomainMapping.Keys.FirstOrDefault(k =>
                        InterDomainMapping[k].Key==curr_target_elem);
  if (previous_mapp != null){
  // Resolve many-to-one matchings
     if (InterDomainMapping(previous_mapp).Score < new_sim_score){
       // Delete the previous mapping in order to add the new one
        InterDomainMapping.Remove(previous_mapp);
       // Add the current match to the inter domain mapping
       if (InterDomainMapping.Contains(curr_source_elem) == false){
            InterDomainMapping.Add(curr_source_elem, new
            KeyValuePair(curr_target_elem, new_sim_score);}
       else{
          // Resolve one-to-many matchings
           if (InterDomainMapping(curr_source_elem).Score <
   new_sim_score){
                InterDomainMapping(curr source elem) = new
                KeyValuePair(curr_target_elem, new_similarity_score);}
       } }
   else {// the curr_source_elem has never been matched before
      // Add the current match to the inter domain mapping
```

```
      if (InterDomainMapping.Contains(curr_source_elem) == false){
            InterDomainMapping.Add(curr_source_elem, new
            KeyValuePair(curr_target_elem, new_sim_score);}
      else {
        // Resolve many-to-one matchings
        if (InterDomainMapping(curr_source_elem).Score < new_sim_score){
              InterDomainMapping(curr_source_elem) = new
              KeyValuePair(curr_target_elem, new_sim_score)}
       }
    }
}
```

Table 7. Code example of how our algorithm resolves one-to-many and many-to-one ambiguous matches**.**

5.2.5. Evaluating sub-graph mappings

Just like the IAM algorithm, after finding all the valid matches for the seed group (current method in the source file), we perform a minimal evaluation on it (evaluation that has been shown to work on several other domains of knowledge (Keane, et al., 1994)). If more than half of the elements in the group have been matched successfully, then the mapping is considered as being successful and the algorithm moves on to incrementally map the next groups (the other methods in the class, if we remember the structure of the program that we considered at the beginning in Table **5**). This 50% mapping threshold is known as the IAM mapping constraint. If the mapping found is not successful, the algorithm backtracks and selects an alternative seed match if there is any other (remember how elements are selected under decreasing order of their NR values). If sufficient similarity cannot be found (no successful mapping exists) then the group is abandoned and another two groups are selected (again by their NR order).

After successfully mapping the current group the algorithm proceeds to match the next unmapped groups in the source domain and incrementally adds the corresponding matches found to the inter-domain mapping as long as they respect the 1-to-1 constraint enforced by IAM. We highlight that each of these incremental mapping activities contributes to the one same inter-domain mapping, forming one consistent interpretation of the comparison.

When the mapping process finishes finding all the valid matches between the sub-graphs in the source and target domains, we calculate the *graph similarity* score of the conceptual graphs as

a real number equal to the total number of nodes matched over the total number of actual nodes in the source domain (because we map the source to the target). This can be formally written as:

$$GraphSim(CG_{source}, CG_{target}) = \frac{|InterDomainMapping(CG_{source}, CG_{target})|}{|CG_{source}|}$$

In order for our system to be fit for use in source code retrieval applications such as Pitu's (Pitu, 2013) work in retrieving similar previous verified programs to the target query in *Arís*, our algorithm needs to respect the following constraints:

1. Symmetry: $\forall CG_{source}\ \forall CG_{target}, GraphSim(CG_{source}, CG_{target}) = GraphSim(CG_{target}, CG_{source})$
2. Maximal similarity: $\forall CG_{source}\ \forall CG_{target}, GraphSim(CG_{source}, CG_{source}) \geq GraphSim(CG_{source}, CG_{target})$

Although our graph matching algorithm complies with the second requirement, in order to enable symmetry (which given the analogical reasoning framework of IAM the resulting mapping will not be symmetric if the source and target domains are different, as the process depends on the order and the number of seed groups in each domain, etc.) we calculate the similarities in both ways and take their average score ($\frac{GraphSim(CG_{source}, CG_{target}) + GraphSim(CG_{target}, CG_{source})}{2}$) as the final similarity that we output to the source code retrieval module of *Arís*. We emphasize that when we generate and transfer new specifications into the target, we use the mapping from the source (which has the specifications) to the target domain.

5.2.6. Mapping constraints

In order to establish whether a match between two conceptual nodes is a valid match or not, our algorithm (similar to IAM) enforces some match rules and constraints which we implemented as *similarity functions* that check if certain properties hold in the mapping. Each similarity function receives as input the two concepts being compared (one from the source and one from the target) and outputs a *similarity score* $\in [0,1]$. Below we describe each similarity function in detail.

1. **Type similarity function**

This function checks if the two nodes being matched have the same concept type (i.e. *Variable – Variable, Defines-Defines,* etc.). Ensuring the mapping only between entities of the same type

will reduce the total number of matches that need to be considered (Holyoak & Thagard, 1989) and make the mapping process more efficient. We define the type similarity function as:

$$sim_{type}(node1, node2) = \begin{cases} 1, & concept\ type\ of\ node1 = concept\ type\ of\ node2 \\ 0, & otherwise \end{cases}$$

2. **Structural similarity function**

This function checks that the nodes having edges into or from the input nodes also have the same concept type (for preserving structural consistency, useful in eliminating ambiguous mappings, since the most structurally similar matches will also have the highest similarity score). This means that two concepts are mapped together only if they are found in a similar context within the local graph (where the context refers to all the adjacent nodes, see for example Figure 14).

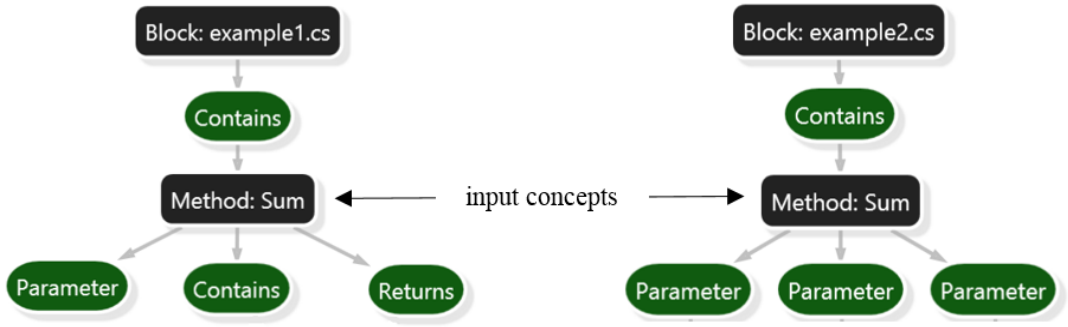

Figure 14. Example of how the structural similarity function ensures structural consistency. On the left hand side we have a part of the method from `example1.cs`. In `example2.cs` we declared a function without implementation.

Given as input two concept nodes, the sim_{struct} function compares the nodes on the immediate upper and lower levels in the two graphs, counts the number of nodes that have the same concept type and divides it over the total number of nodes in the largest context of the input nodes. For the example in Figure 14, where *Method:Sum* are the input nodes, we have 2 pairs of nodes of the same type (*Contains – Contains, Parameter – Parameter*) and an equal number of 4 nodes in both contexts. Thus the structural similarity will always output values between 0 and 1 (in the example given, the $sim_{struct}(Method{:}Sum, Method{:}Sum) = \frac{2}{4} = 0.5$).

3. Content similarity function.

One of the main advantage of our work compared with *Source Code Retrieval using Conceptual Similarity* by (Mishne & De Rijke, 2004) is the fact that we perform a more sophisticated content based comparison in addition to their simple string-based distance algorithm. Our content similarity function applies different comparisons depending on the concepts being mapped. A restriction however, needs to be set, since not all concepts in our conceptual graphs have content information. For example, relation types (*Depends, Contains,* etc.) do not have an individual referent and they only contain information about the concepts they connect. Thus all concepts that do not have an individual referent value are excluded from this type of similarity comparison.

The most important process is how we compare C# variable types, information that is stored in *Variable, Field* and *Method* concept types. Our intuition was that types that do not match exactly (for example, two objects from different classes or two numeric variables, one *int* and the other *double*) should be checked if they have the same super-type or one is the sub-type of the other within a class hierarchy. This makes our algorithm accept more flexible comparisons and find similarities that can be missed by a simple string-distance measure (for example in comparing an *int* to a *double*). In order to achieve this, we add an attribute to *Variable, Field* and *Method* concepts that stores the C# type-specific information extracted from the Roslyn Abstract Syntax Tree. However, with the Roslyn AST, we do not obtain the fully qualified C# type name that we need in order to do a semantic comparison (for example, for an *int* its fully qualified type name is *System.Int32,* since all value types[26] derive from *System.ValueType*). Moreover, for the user defined types (classes in C#) we need to access their compiled *assemblies*[27] in order to establish whether there is a class hierarchy between them. The *Source Code Retrieval using Case Base Reasoning* (Pitu, 2013) module of *Arís* project performs structural and semantic source code retrieval by analysing C# compiled assemblies, thus being able to extract all the types (with fully qualified names) used in a given C# source code file. We use this work to obtain a list of all the types (classes) that are being referenced in the two programs that we are comparing, such that when

[26] Value types in C# include structs, enumerations, numeric types, Booleans - http://msdn.microsoft.com/en-us/library/s1ax56ch(v=vs.80).aspx
[27] C# Assemblies - http://msdn.microsoft.com/en-us/library/ms173099(v=vs.80).aspx

we want to match two concepts that have a *Type* attribute (*Variable, Field* or *Method* concepts) we determine the similarity between their attributes in the following way:

- we retrieve their fully qualified C# types from the list of exported types obtained using the source code retrieval module of *Arís*. This gives us information about the objects and the hierarchy from which they derive.
- we used different C# functions (described in Table 8**)** to determine whether the two types come from the same hierarchy or one is the sub-type of the other or they are both of the same, equivalent type (i.e. numeric type).

bool Type.IsAssignableFrom (Type other)	Checks if an instance of the current Type can be assigned from an instance of the specified Type.
bool Type.IsEquivalentTo (Type other)	Checks if two types have the same identity can be considered as equivalent.
bool Type.IsSubclassOf (Type other)	Checks if the class represented by the current Type derives from the class represented by the specified Type.

Table 8. Example of functions from the C# Type class[28] used to compare types of two C# objects.

If any of the properties described above are true, then we assign a score of 1 to the content matching and 0 otherwise. For the rest of the concepts that have referent values (*CompareOp, LogicalOp, MathOp, Loop*) we use the widely-known *Levenshtein*[29] string distance. Levenshtein's algorithm takes as input two strings *s1, s2* and calculates the minimum number of single edits necessary to transform *s1* into *s2*. The returned score is a value between 0 (no edits necessary) and the maximum length between *s1* and *s2* (when they are totally different). We use this score to compute the similarity between two input strings, however we normalize the Levenshtein output score to the maximum length between s1 and s2, thus our $sim_{content}$ function, obtaining in both cases (C# type comparison and string-based distance) values between 0 and 1, as required.

[28] Documentation for the C# Type class - http://msdn.microsoft.com/en-us/library/system.type.aspx
[29] Levenshtein string distance - http://software-and-algorithms.blogspot.ie/2012/09/damerau-levenshtein-edit-distance.html

For example, if we want to map two variables defined as: `Employee instance1` and `Person instance2`, where `class Employee` derives from `class Person`, then by checking if `typeof (instance1).IsSubclassOf(typeof(instance2))`, which in this case is true, we can infer that `instance1` and `instance2` can be mapped together (*Liskov's Substitution Pronciple*[30]).

As some types of similarities can be more important than others depending on the context and in cases where we want our mapping algorithm to perform a more "relaxed" matching (where, for example, even concepts that don't share the same concept type can be mapped together) we assigned to each similarity function a *weight* representing its contribution to the overall similarity score sim that we defined as a linear combination of the three similarity functions as follows:

$$sim\ (node1, node2) = weight_{type} \times sim_{type}(node1, node2) +$$

$$weight_{struct} \times sim_{struct}(node1, node2) + weight_{content} \times sim_{content}(node1, node2)$$

In our implementation (where we aimed for a more strict matching), we obtained good results using $weight_{type} = 0.5$, $weight_{struct} = 0.3$ and $weight_{content} = 0.2$.

Next we defined, based on the overall similarity score sim, in which cases we accept a mapping as being valid and in which cases we reject it. For this we defined the function $ValidMatch$ that uses another threshold value described below, to choose which mappings are kept and which ones are discarded (basically the mapping is valid only if the function returns true)

$$ValidMatch(node1, node2) = \begin{cases} true, sim\ (node1, node2) > 0.5 \\ false,\ \ otherwise \end{cases}$$

5.3. Using pattern completion to generate target specifications

Although in many cases when we apply analogical reasoning we just want to verify if two given domains can be mapped together in this project the mapping by itself (which contains the detailed correspondences between the source and the target) is insufficient for our goal of reusing formal specifications. In order to achieve this we need an algorithm such that once IAM has found a successful mapping it can generate the analogical inferences and transfer the required

[30] The Liskov Substitution Principle - http://www.objectmentor.com/resources/articles/lsp.pdf

specifications into the given target code. In our system, we generate the analogical inferences by using an algorithm for pattern completion called CWSG - *Copy with Substitution and Generation* (Holyoak, et al., 1994) that has been widely used before in analogical reasoning computational models (see Section (3.4) for references).

CWSG transfers the additional specifications from the retrieved code and adds it to the target code by substituting source code items with their mapped equivalents. This allows our target/problem code to be formally verified using the newly generated specification. In order to increase the number of formally verified programs, in *Arís* we also retain the newly formally verified source code artefacts for further use (for example, in other retrieval queries).

Next we show an example of how the specification is transferred from the base domain `example1.cs` (for which we gave its conceptual representation in Section (4.2)) into the target source code `example2.cs` using CWSG based on the correspondences found after applying IAM. Table 9 shows the two programs received as input by our system and Table 10 presents the detailed correspondences between their conceptual graphs. We omit due to space reasons to show the CGs corresponding to the programs.

```
// base example1.cs
public static int Sum(int k)
requires 0 <= k;
ensures result==sum{int i in (0:k); i};
{
 int s = 0;
 for (int n = 0; n < k; n++)
 invariant n <= k;
 invariant s == sum{int i in (0:n); i};
 {
   s += n;
 }
 return s;}
}
```

```
// target example2.cs
public static int Sum(int x)
{
 int add = 0;
 int k = 0;
 while (k < x)
 {
   add += k;
   k++;
 }
 return add;
}
```

Table 9. Base and target methods received as input by our system. The source is formally verified using Spec#. Both the implementations are highly similar, they calculate the sum of the first *n* numbers, however their structure is slightly different (e.g. one declares a variable inside the loop).

```
{ Parameter }  matched with  { Parameter } (1)
{ Variable: k }  matched with  { Variable: x } (1)
{ Variable: n }  matched with  { Variable: k } (0.96)
{ Variable: s }  matched with  { Variable: add } (0.9429)
{ Method: Sum }  matched with  { Method: Sum } (0.8)
{ Loop: For }  matched with  { Loop: While } (0.725)
{ Condition }  matched with  { Condition } (1)
{ Contains }  matched with  { Contains } (1)
{ Assign:* }  matched with  { Assign:* } (1)
{ Contains }  matched with  { Contains } (1)
{ Contains }  matched with  { Contains } (1)
{ Block:* }  matched with  { Block:* } (0.8)
{ Contains }  matched with  { Contains } (1)
{ Contains }  matched with  { Contains } (1)
{ CompareOp: < }  matched with  { CompareOp: < } (1)
{ Contains }  matched with  { Contains } (1)
{ Contains }  matched with  { Contains } (1)
{ Assign:* }  matched with  { Assign:* } (0.8)
{ Contains }  matched with  { Contains } (1)
{ Contains }  matched with  { Contains } (0.7)
{ Contains }  matched with  { Contains } (1)
{ Block:* }  matched with  { Block:* } (0.8)
{ Contains }  matched with  { Contains } (1)
{ Assign:* }  matched with  { Assign:* } (0.8)
{ String: 0 }  matched with  { String: 0 } (1)
{ String: 0 }  matched with  { String: 0 } (0.8)
{ Contains }  matched with  { Contains } (1)
{ Returns }  matched with  { Returns } (1)
{ Block: Root }  matched with  { Block: Root } (1)
```

Table 10. Output of IAM algorithm containing the correspondences (and their similarity score) derived from the mapping in Table 9.

Based on the correspondences we obtained during the matching process, we can see that the important variables used in the specifications attached to the base domain (written in bold face in Table 10) have been mapped accordingly to the variables in the target code. Thus we can now use this mapping to generate a new specification for the target by replacing the appropriate variables that appear in *requires, ensures* and *invariant* statements with their mapped equivalents. We currently implemented the actual transfer based on string processing. The new specification generated into the `example2.cs` target code can be seen in Table 11.

```
public static int Sum(int x)
requires 0 <= x;
ensures result==sum{int i in (0:x); i};
{
 int add = 0;
 int k = 0;
 while (k < x)
 invariant k <= x;
 invariant add == sum{int i in (0:k); i};
 {
   add += k;
   k++;
 } return add;
```

Table 11. Transferred specification from the source problem *example1.cs* into the target code *example2.cs*. Now our target problem can also be formally verified using the automated verification tool Spec#.

Although in most cases the *requires, ensures* and *modifies* specifications can be correctly generated by our pattern completion CWSG algorithm (because they depend only on the exact parameter matching at which the mapping process obtains good results), generating the corresponding specifications for the *invariant* statements is a much harder problem, since the invariants are very closely related to the structure of the loop. For example, if instead of a loop from $\overline{0..x}$ in the *example2.cs* we used a decreasing loop from $\overline{x..0}$, then our target program could no longer be formally verified because the invariant would be semantically different. Even so our implementation generates both the method contract as well as the invariant clauses that need to be transferred into the target code in order to help the user as much as possible with guidance on how the verify its program.

5.4. Analysis and conclusions

In this chapter we presented our system for comparing two implementations represented as conceptual graphs and finding the detailed correspondences between them. Based on these correspondences we then showed how using a pattern completion algorithm we can generate and transfer missing specifications into the target, relying on the premise that our base problem is formally specified and verified. Our system thus proposes a novel approach for reusing formal specifications and/or implementations and due to the similarity score that we give as output in the

mapping process, it can also be useful in other types of applications like for example in detecting plagiarism or code duplication.

However, our incremental graph matching algorithm has some limitations regarding the extent of the mapping it can perform. A first limitation is that our implementation currently can match only methods in a C# class. This means that any class field members or defined properties that can appear inside a class are not being mapped. We point out that, however, in cases where the fields are being used by the method's code, they will participate in the matching process as the corresponding conceptual sub-graph of the method will contain edges to the respective fields. Similarly, we currently do not match multiple classes in the same file but we propose this as future work in Chapter (7). Other mapping limitations also depend on the capabilities of our conceptual graph construction process and how much of the source code is actually represented in the graph.

Another vulnerable point in our IAM algorithm, is the likelihood of detecting false positive mappings due to the fact all nodes in our conceptual graph representation have equal importance in the matching process. This can sometimes cause the algorithm to find valid mappings between relation nodes such as *Contains, Condition, Depends, Parameter* or *Block:** which are common in every conceptual graph and do not have an impact on the generated specifications, like, for example, a *Variable* or a *Loop* concept would have.

The overall complexity of the algorithm is polynomial due to the fact that we only match methods in decreasing order of their NR values. This means that we compare at most n methods (all the methods from the source domain) and inside the method mapping process we perform 2 sorting operations plus at most $m \times match_{depth}$ (where m denotes the number of nodes in a sub-graph representing a method) comparisons since we do not search the whole space of the target domain but instead we use a threshold parameter $match_{depth}$ to set a limit on the number of concepts in the target to which we compare each node from the source method. Thus in the worst case our algorithm performs in $O(n \times mlogm)$, if we consider a $O(mlogm)$ sorting function.

6. Evaluation

In this chapter we present the experimental setup for testing our system in which we evaluate the impact of the Node Rank metric on the IAM algorithm, the capabilities and boundaries of the conceptual graph matching module in finding similarities between identical, modified and totally different source code inputs and finally we present our promising results in generating and transferring specifications into the target queries.

6.1. Document corpus

In order to evaluate our system's performance at mapping and transferring specifications between a formally verified source domain and an unspecified target, we first need a document collection that is fit for the task at hand. This means that we have to obtain a corpus of source code files that contain only methods (as our current implementation matches only methods) and that each method is formally specified using an appropriate specification language. The main problem with finding formally verified programs is that in the current context of software development there are very few such collections publicly available (which are mostly for research purposes), thus obtaining real world examples of verified source code files is a hard task. Given that, as previously mentioned, our system is built upon .NET framework and analyses C# files (which can be formally specified using the Spec# language) we selected our corpus of verified source code files from the Spec# test suits publicly available on the open-source hosting platform *CodePlex*[31]. We collected a corpus of 102 files which contain 249 formally verified methods and approximately 7470 lines of code. As a sanity check for our system, we also duplicated all the files with their unspecified equivalents to be used in the identical document mapping. For this small set of verified programs, the average ratio between nodes in the conceptual graph and lines of code was 2.63 (with the average document size of 30 lines and the average number of 78 nodes).

To obtain a more thorough evaluation, the source code retrieval module of *Arís* (Pitu, 2013) which uses our system to perform graph matching and to transfer specifications, also collected a large corpus of real world projects (consisting of 2,191 applications with 2,033,623 methods),

[31] CodePlex open-source hosting platform - http://www.codeplex.com

downloaded from open-source repositories publicly available. Their evaluation results also provide us with relevant information regarding the impact of our system in a retrieval task, but more details about this combined evaluation will be presented later, in the end of this chapter.

In order to perform identical and modified document comparison as the evaluation process in (Mishne & De Rijke, 2004), we randomly selected 30 files from our document collection without specifications. We then transformed them by applying different levels of modifications. Some of the most frequent code changes as reported by (Wilkinson, 1994) that we did in order to obtain our modified document corpus from which we selected targets for our system are given below:

1. Lexical changes: renaming variables, methods and parameters. For example changing *finalResult* to *outputValue*.
2. Type changes: changing the type of variables, methods, and parameters to an equivalent type (where possible). For example where a variable was defined as an *int* we changed it to a *long*. User defined types were excluded.
3. Changing code constructs: changing the order of parameters in method declarations or in method calls, rewriting *for* loops as a *while* loops.
4. Adding, removing or modifying comments in the code.
5. Reordering statements: reversing conditional statements, changing the order of statements in a method's body (where it does not affect the program's functionality).
6. Adding extraneous statements: for example declaring unused variables, calling the same method multiple times, etc.
7. Removing certain statements (again, without affecting the functionality of the program).

Therefore we created a corpus of modified documents that we use to test the performance of our system at generating and transferring specifications. We note that for the graph similarity score gave as output by our IAM algorithm, values close to 1 mean that the target is highly similar to the source. We then classified the types of modifications depending on the extent on which they structurally modify the source code as following:

- **Small modifications**: all changes from 1-4 presented above.
- **Medium modifications**: all changes from 5-7 presented above.
- **Large modifications**: changes that alter the functionality of the program so that it does not perform the same computation anymore.

6.2. Experiments

We next present the experiments that we conducted to optimize our system and evaluate its performance at generating and transferring specifications between different programs.

6.2.1. Parameter optimization

We carried out this first experiment as a basic tuning process for the parameters used in the analogical mapping process (match depth, weights for the three similarity functions and the valid match threshold). As presented in Section (5.2.6), we use three types of mapping constraints (functions) when comparing two nodes in a conceptual graph: a type function that ensures type consistency, a structural similarity function that evaluates if the contexts of the two nodes in the graphs are also type consistent and a content similarity function that compares the information inside the nodes (their referent value). All these three functions have associated weight values which represent their contribution to the overall similarity score sim calculated between two given nodes in order to establish whether or not they can be mapped together.

We intuitively set a predefined weight value $weight_{type} = 0.5$ for the type similarity function (that returns 1 if two concepts have the same type and 0 otherwise) and as well for the $match_{depth}$ threshold that helps us identify valid matches based on the similarity score sim. Thus for a match between two nodes to be considered valid it must obtain at least a score of 0.5 (when the two concepts have the same type) in order to enforce type consistent mappings.

For the content and structural similarity function weights we did a limited number of experiments using 30 randomly chosen sources and their corresponding modified targets (with small and medium changes) and recorded the average graph similarity score (computed by the formula we gave in Section (5.2.5) in which we divide the number of mapped concepts over the total number of concepts in the source domain) obtained for each parameter configuration. Based on the results in Table 12 we set the weight values $weight_{struct} = 0.3$ and $weight_{content} = 0.2$ after conducting a *Mann-Whitney*[32] test that showed ($p = 0{,}0922$) the results we obtained using this configuration were significant.

[32] Mann-Whitney Test - http://www.vassarstats.net/

$weight_{struct}$	0.1	0.2	0.3	0.4
$weight_{content}$	0.4	0.3	0.2	0.1
$Avg(GraphSim(source_i, target_i))$	0.720	0.748	**0.8382**	0.8324

Table 12. Parameter optimization for the content and structural weights.

As an observation, we can see that for higher structural weight values the system performs better, meaning that indeed the structural nature of the source code captured by the conceptual graph can help in finding better matches.

For the $match_{depth}$ parameter, we incrementally assigned values between 0 and 100 (see Table 13) and recorded the graph similarity scores obtained. We used the same base document corpus, formed by randomly choosing 30 programs from our verified document collection and taking their corresponding modified versions (with small and medium modifications) as targets. We found out that for $match_{depth} > 10$ the Mann-Whitney test results were no longer significant and reliable (as can be observed from Table 14). Intuitively, the reason for this is that we take the nodes in decreasing order by their NRs, thus the chances of finding a valid match for the current element in the source also decreases as we go further in the target. Thus we set the $match_{depth} = 10$.

Configurations	$\boldsymbol{match_{depth}}$
#C1	0
#C2	10
#C3	20
#C4	30
#C5	40
#C6	50
#C7	60
#C8	70
#C9	80
#C10	90
#C11	100

Table 13. System configurations for testing the $\boldsymbol{match_{depth}}$ parameter

Configurations compared	**Mann-Whitney result**
#C1, #C2	**P = 0.0655**
#C2, #C3	P = 0.2358
#C3, #C4	P = 0.4247
#C4, #C5	P = 0.4404
#C5, #C6	P = 0.5
#C6, #C7	P = 0.4562
#C7, #C8	P = 0.484
#C8, #C9	P = 0.484
#C9, #C10	P = 0.484
#C10, #C11	P = 0.484

Table 14. Mann-Whitney significance test results for the system configurations in Table 13.

6.2.2. Evaluating Node Rank impact on the mapping process

In this experiment we tested the hypotheses that using the Node Rank metric (detailed in Section (5.2.2)) for sorting the elements in the base and target domains can help us find better analogical mappings. The NR metric is closely related to the structure of the program, for example, if a variable is used multiple times in the program, then it will have a high node rank. In our system we used this information to sort the concepts in the base and target methods and to map them decreasingly by their NR order. In order to assess the effect of this metric on our graph matching algorithm, we used as baseline the order gave by a *Breadth-First-Search* algorithm starting from the root of the graph. We selected again 30 random files from our corpus of verified documents to use as source domains and their 30 correspondent modified versions to use as targets. We conducted two experiments, one using the NR metric for ordering and selecting the elements in the matching process and one using the BFS order, and for both runs we stored the mappings obtained and their similarity scores (calculated in terms of the number of valid mappings found relative to the total number of nodes in the source domain).

Our results confirm that using the Node Rank metric as a sorting criteria for the nodes in the source and target, brings visible improvements on the number of valid mappings found by our adapted IAM algorithm (Section (5)). The average similarity score $GraphSim(source, target)$ recorded for the 30 documents we tested using the NR metric was 0.8532 and when using BFS it was 0.786 (where closer to 1 is better). This means that indeed using the NR metric helps in detecting more valid matches, however a Mann-Whitney test revealed that the difference between the two sets of similarity scores obtained was not very statistically significant ($p = 0.1911$) thus, without extended evaluation, we cannot draw a firm conclusion to whether the NR metric is a suitable criteria to use in IAM for selecting the seed groups and finding the seed match.

6.2.3. Evaluating the transferred formal specifications

The most important feature in our system that sets it apart from other source code matching or retrieval systems is the fact we can generate and transfer specifications from a formally verified input into an unspecified target code. The main premise that guided our work in *Arís* is the fact that similar implementations also have similar specifications. In this section we evaluate how much of the transferred specifications can actually be formally verified by an automated verification tool

such as Spec# and also, what are the limits of our system in finding similarities between two different implementations.

Identical Document Setting. As a sanity check evaluation we first tested an identical document setting in which we used as base documents 30 randomly chosen verified methods from our Spec# test suite corpus collection and their mirror version with specifications removed as target documents. For each mapping task we obtained the maximum matching (with a graph similarity score of 1) between the base and target programs, and in every case the specification was fully and successfully transferred and verified by Spec#. This first result emphasizes that our system is capable of detecting identical matches and correctly generating and transferring the specifications based on the mapping obtained.

Modified Document Setting. For the modified document evaluation, we divided our tests depending on the class of modifications (given in Section (6.2)) that were applied to the target documents. We randomly selected a smaller subset of 20 documents from the verified corpus and gradually applied modifications on the code. We thus obtained 40 modified documents, 20 with small modifications and 20 with small and medium modification.

In Table 15 you can see an example of where the target contains small modifications compared to the original verified source and where we also give the similarity score found by our algorithm.

`public int Count_IDD(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 s == count{int j in (0: i); a[j] == x};`
`invariant i <= a.Length;`
`{`
`if (a[i] == x)`
`{`
`s = s + 1;`
`}`
`} Console.Writeline("s");`
`return s;}`
	`public long Number_MOD(long y, int[] array)`
	`{`
	`long add = 0;`
	`int pos = 0;`
	`// iterating the array`
	`while (pos < array.Length)`
	`{`
	`if (array[pos] == y)`
	`{`
	`// updating the sum`
	`add = add + 1;`
	`}`
	`++pos;`
	`}`
	`Console.Write("add");`
	`return add;}`

$GraphSim(Count\_IDD, Number\_MOD) = 0.8648$ (32 mapped / 37 total nodes)	

Table 15. Example of two mapped inputs where the target (right) has small modifications compared to the original source (left).

Result: 0.8648 (32 mapped / 37 total nodes)
1.{Parameter} matched with {Parameter} (1)
2.{Variable:a} matched with {Variable:array} (1)
3.{Parameter} matched with {Parameter} (1)
4.{Variable: x} matched with {Variable: y} (1)
5.{Variable: s} matched with {Variable: add} (1)
6.{Variable: i} matched with {Variable: pos} (0.9)
7.{Contains} matched with {Contains} (1)
8.{CompareOp: ==} matched with {CompareOp: ==} (0.8933)
9.{Contains} matched with {Contains} (1)
10.{Block:*} matched with {Block:*} (1)
11.{Contains} matched with {Contains} (1)
12.{Assign:*} matched with {Assign:*} (1)
13.{Contains} matched with {Contains} (1)
14.{Contains} matched with {Contains} (1)
15.{Assign:*} matched with {Assign:*} (0.8)
16.{Contains} matched with {Contains} (1)
17.{Loop: For} matched with {Loop: While} (1)
18.{Contains} matched with {Contains} (1)
19.{Condition} matched with {Condition} (1)
20.{If:*} matched with {If:*} (1)
21.{Condition} matched with {Condition} (1)
22.{Contains} matched with {Contains} (1)
23.{String: 0} matched with {String: 0} (1)
24.{Block:*} matched with {Block:*} (0.8)
25.{MethodCall: Console.Writeline()} matched with {MethodCall: Console.Write()} (0.8909)
…
	`public long Number_MOD(long y, int[] array)`
	`requires array != null;`
	`ensures result == count{int i in(0: array.Length);(array[i] == y)};`
	`{`
	`int add = 0;`
	`int pos = 0;`
	`while (pos < array.Length)`
	`invariant add == count{int j in (0: pos); array[j] == y};`
	`invariant pos <= array.Length;`
	`{`
	`if (array[pos] == y)`
	`{`
	`add = add + 1;`
	`}`
	`++pos;`
	`}`
	`Console.Write("add");`
	`return add;`
	`}`

Table 16. Part of the mapping of the inputs in Table 15 and the transferred specifications.

In above table we can see the detailed correspondences between the source and target documents and the transferred specifications. In this case the transferred specification was successfully transferred and formally verified by Spec#.

The average graph similarity score we computed for the 20 small modified targets was 0.8581 and 16 out of 20 generated specifications into the targets were successfully verified. This was a great achievement in our project and strengthens the fact that reusing previous verified programs is actually possible. Moreover it shows that even if we insert small modifications into the target, our algorithm is still able to detect the correct mappings between the similar constructs in the programs and successfully transfer the specifications, which is indeed a very promising result. By analyzing the unverified programs, we observed that the problems were regarding the generated loop invariants which had different variable name conflicts (for example if instead of the parameter `long y` we would have used `long j` then the generated invariant condition would have

become `array[j] == j` causing a name conflict) and thus requiring further user input in order to be verified. However, the method contract (*requires, ensures*) was in all cases successfully transferred implying that our mapping algorithm can correctly compute parameter mappings and is not influenced by modifications such as inserting comments, reordering parameters or changing variable names, types or loop constructs as long as they do not change the overall functionality and structure of the program. This encourages us to think that our adapted IAM algorithm coupled with the abstract representation of source code as conceptual graphs are a good combination in a source code mapping system and fit for our goal of generating specifications.

We next give an example of the target program in Table 16 to which apart from the small changes, medium modifications were also applied by inserting extra statements and reordering or removing existing ones (changing the structure of the program, but not its functionality). This represents a more serious challenge to our (retrieval and) code matching and transfer process.

```
public int Count_IDD(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 s == count{int j in (0: i);
                 a[j] == x};
 invariant i <= a.Length;
 {
  if (a[i] == x)
  {
     s = s + 1;
  }
 }
 Console.Writeline("s");
 return s;
}
```

```
public long Number_MOD(bool work, long
y, int[] array)
{
if (work == true){
  int result = 0;
  int add = 0; //long add = 0.0;
  int pos = array.Length - 1;
  if (pos != -1){
     while (pos >= 0){
       if (array[pos] - y == 0) {
         add += 1; // count
         Console.Write("add = "+add);
        }
        pos = pos - 1;
      }
      int has = pos + add;
  }
  return add;
}else return 0; }
```

$GraphSim(Count\_IDD, Number\_MOD) =$ 0.6756 (25 mapped / 37 total nodes)

Table 17. Example of medium modified target and the graph mapping similarity score obtained when compared to the original source

We can observe form the above tables that the mapping similarity decreases as more structural change is applied to the target. In the example in Table 17, the main loop was rewritten in such a way that it does a reverse traversal in the vector, thus when the loop specifications were transferred, the loop invariant condition from the source program did not longer hold. In general, changes like

reversing the logical structure of a loop influence the invariant statements generated, as the invariant is closely related to the loop it describes.

The average graph similarity score we obtained for the 20 medium modified targets was 0.6391 and 9 out of 20 targets were successfully verified by Spec#. The results show that when adding extraneous statements or changing the structure of the target, it affects the outcome of the mapping process and implicitly the transferred specifications. The main reason why inserting unnecessary statements (that, for example, may use other variables defined in the program) affects the number of valid mappings found is because they change the structure of the conceptual graph and due to the fact that the NR metric is strictly computed based on the graph structure, the nodes are going to be mapped in a different order, thus possibly missing important mappings. Extraneous statements also affect the process of verifying the specification, for example, when trying to verify the generated specification for the target in Table 17, Spec# gives a warning about the `variable 'result'` not being used. However, even the fact that we can transfer partially correct specifications gives a great starting point in trying to verify a program. Although in most cases of medium modified targets further used input is required in order to verify them, the results exceeded our expectations and proved that our tool is very useful in guiding the user on writing specifications, which is something that many current verification tools are trying to achieve.

In the cases were the mapping obtained a score lower than 0.5 our algorithm rejected the mapping because transferring specifications between two significantly different programs would be less useful for the user, as the chances for the specification to actually be verified are very small. In Table 18 we give an example of a target that is functionally different than the source and we show that our algorithm is able to reject the mapping and not transfer the specification.

`public int Count_IDD(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 s == count{int j in 0:i);a[j]== x};`
`invariant i <= a.Length{`
`if (a[i] == x){s = s + 1; }`
`} Console.Writeline("s");`
`return s;}`
	`public static void Swap_MOD(int i, int j, int[] Array)`
	`{`
	`int v = Array[i];`
	`int df = i + j;`
	`Array[i] = Array[j];`
	`Array[j] = v;`
	`}`

$GraphSim(Count\_IDD, Swap\_MOD) = 0.3514\ (13/37)$ - Reject	

Table 18. Example showing a rejected mapping where our system does not transfer the specifications.

6.2.4. Combined evaluation in *Arís*

Our system was integrated into *Arís* and evaluated by (Pitu, 2013) in *Source Code Retrieval Using Case Based Reasoning* in which they built a source code retrieval system that uses our detailed structural matching to identify the best mapping for a given query. We receive from their system 2 input files (a verified source and a target program) and based on the similarity score obtained, we also transfer the specifications into the target. Their evaluation was based on a document collection extracted from real world applications as well as on a small set of verified programs (as we discussed in Section (6.1)). The results they obtained show that our graph matching algorithm improves the overall quality of the retrieval. Using our system to check for structural similarity, their results improve by 15%, selecting more accurate candidates for transferring specifications into the target. This means that our system can also be successfully integrated with other systems that compare source code files and need a detailed mapping.

6.3. Discussion

In this chapter we presented the experimental evaluation that we did in order to optimize and test our system. The document collection we used was described in Section (6.1.) and basically consisted of a set of verified C# programs and their manually modified versions.

We first conducted a parameter tuning experiment in which we searched to find the best configuration for the parameters we use in comparing two nodes in a graph. We tested the system with different configurations and, where it was possible, we also applied a Mann-Whitney significance test to check whether our results were indeed statistically significant between consecutive runs of the system. Our conclusion after analyzing the results are that higher weight values for the structural similarity function (given in Section (5.2.6)) increases the number of valid mappings found, as the IAM algorithm depends on the structural nature of the data to efficiently find and derive new mappings.

In Section (6.2.1) we evaluated the Node Rank metric (Bhattacharya, et al., 2012) which we used in our IAM algorithm to order sub-graphs and map the most important ones first. In our experiment using small and medium modified targets we observed that the average graph similarity score improved when we sorted the nodes using the NR metric as compared to the sorting gave by

a BFS traversal. Although more evidence is necessary in order to draw a firm conclusion, given that our system obtained good overall mapping results we can acknowledge the ability of the NR metric in discovering the most relevant parts in a program and its potential as a choice criteria in finding the seed group and seed match in IAM.

The most important part of our project is the ability to generate new specifications into a target program. Our last experiments were conducted in order to try to answer the following questions: *Is specification reuse possible in practice? What degree of similarity is required between two programs in order to successfully verify the transferred specification?* The results we obtained show that reusing specifications is possible in practice and that our system can successfully transfer specifications between two structurally similar programs. We showed that our graph matching algorithm is not influenced by small code modifications like reordering parameters, changing variable type, names, inserting comments or changing loop constructs as long as they don't affect the general structure of the program (which inserting a new statement for example, would do). For 80% of the small modified targets the transferred specifications were successfully verified, result with was very good and exceeded our expectations. The percentage dropped to 45% for the medium modified targets where the code suffered many structural changes like insertion, deletion or reordering of statements, further user input being required in order to verify them.

As an overall conclusion regarding the evaluation, our results firmly show that our tool is capable of successfully transferring specifications between structurally similar versions of the same program and can give at least a partial correct specification to guide the user when trying to verify a more structurally different program compared to the retrieved (verified) source.

7. Conclusions

In the fast growing field of Software Verification technology our aim in this thesis was to explore the possibility of transferring formal specifications between similar programs in order to help increase the number of verified implementations and reduce the effort of writing specifications. Our system was created as a module of a bigger project called *Arís* (Pitu, Mihai; Grijincu, Daniela; Li, Peihan; Saleem, Asif; O'Donoghue, Diarmuid; Monahan, Rosemary, 2013), in which we wanted to provide an interactive user platform for source code retrieval with the purpose of reusing not only specifications but also implementations and proofs.

Our proposed solution focused on the detailed mapping of two source code files and on the process of transferring specifications between them. In order to effectively compare two source code implementations we represented them as *Conceptual Graphs* which have the great advantage of storing not only structural information but also content relevant details. As graph matching was best identified with *Structure Mapping* as the best way to find detailed mappings between two domains, we used an *Analogical Reasoning* computational model called *Incremental Analogy Machine* (Keane, et al., 1994)) to help us find either isomorphic (exact matches) or homomorphic (non-identical) sub-graph mappings. Finally, we used the *Copy with Substitution and Generation* pattern completion algorithm to transfer specifications into the target based on the detailed correspondences found.

In elaborating our solution we used the work in (Mishne & De Rijke, 2004) as the main inspiration and guideline (we critically analysed their system in Section (2.2)). Our work differs from their system in the following ways:

- our conceptual graph construction process can support additional features of the source code such as *namespaces, classes, try-catch-statements* and *fields*;
- in matching two conceptual graphs our approach extensively uses the structural information from the graphs, as opposed to their solution which embeds some structural information inside the nodes but then does a simple string comparison on the content;
- we developed a more specific content matching algorithm to compare the content in the nodes, in addition to the string distance which they propose;
- our incremental graph matching algorithm based on the NR metric assures us that we are mapping the most structurally relevant parts in the program first (taking them in a decreasing

order), whereas they define a notion of *most similar concept* in which they compare (by string distance on the content) all the nodes in the first graph with all the nodes in the second graph in order to find the maximally similar pairs between the two graphs. This exhaustive process is very inefficient compared to our fast and effective IAM algorithm which uses structural inference and pragmatic constraints to find the best matches.

Finally we evaluated our system's performance at finding correspondences between similar and structurally different programs and its capability of generating new specifications into the target document. We obtained very good results at transferring specifications between structurally similar programs where 80% of the total specifications generated were successfully verified using Spec#. The evaluation also showed that our analogical mapping framework is capable of mapping even structurally different programs where, even though in half the cases the specifications were not fully verified, they can be considered valuable user guidance and a starting point in verifying a target program.

Overall, we believe our results are very encouraging and open a promising avenue for future work in this direction as we are convinced by the potential of reusing formal specifications to create more dependable software systems.

7.2. Future work

Although we gave a conceptual graph construction process which can build the most common concept types found in any programming language, it can be easily extended to support more programming features that could also improve the accuracy of the mapping process, because if certain lines of code are not translated into the graph, then our algorithm does not match them. Currently our implementation is capable to match only methods in a class, but the adapted IAM algorithm we gave can be generalized to fields in the class and classes in a file (in the same way we map the methods by taking them in decreasing NR order, we could also map multiple classes in a file).

Because we based our testing on a small set of documents, more experiments could help us firmly establish the impact of the Node Rank metric in the IAM algorithm and also, find better optimized parameter values for our similarity constraint functions. The incremental graph matching algorithm could also be improved and adapted more to our target of generating correct specifications. For example, in a conceptual graph, we have many relation type concepts such as *Contains, Parameter, Condition,* etc. which intuitively do not have the same level of importance as actual programming constructs such as *Loop, Variable, Action* that we actually need to map in order to be able to transfer specifications. However, in our current implementation, they equally affect the similarity score and the mapping obtained. In the future, we plan to assign a weighting scheme based on the importance of each concept (or we could also use the NR metric) and construct a mapping which can filter out any irrelevant nodes that are not used in generating new specifications.

Last, given that our system is capable of finding detailed correspondences between two source code files and give a measure of similarity between them, it makes it a very versatile tool that can be easily integrated with many different source code processing systems where a thorough comparison is needed. In the future we plan to look at code duplication or plagiarism detection integration possibilities as they have many applications in both industry and academia.

References

Bhattacharya, P., Iliofotou, M., Neamtiu, I. & Faloutsos, M., 2012. Graph-Based Analysis and Prediction for Software Evolution. *International Conference on Software Engineering*, June, pp. 419-429.

Brin, S. & Page, L., 1998. The anatomy of a large-scale hypertextual. *Computer Networks and ISDN Systems,* April, 30(1-7), p. 107–117.

Bunke, H., 2000. Recent developments in graph matching. *Pattern Recognition, 2000. Proceedings. 15th International Conference on ,* Volume 2, pp. 117-124.

Chanchal, R. K., Cordya, J. R. & Koschke, R., 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming,* 74(7), p. 470–495.

Chein, M. & Mugnier, M.-L., 1992. Conceptual Graphs: fundamental notions. *Revue d'Intelligence Artificielle,* Volume 6, pp. 365--406.

Clayton, R., Rugaber, S. & Wills, L., 1998. *On the knowledge required to understand a program.* s.l., IEEE Computer Society, pp. 69-78.

Conte, D., Foggia, P., Sansone, C. & Vento, M., 2004. Thirty years of Graph Matching in Pattern Recognition. *International Journal of Pattern Recognition and Artificial Intelligence,* 18(3), pp. 265-298.

Cosma, G. & Joy, M., 2006. *Source-code Plagiarism: a UK Academic Perspective.* s.l., Proceedings of the 7th Annual Conference of the HEA Network for Information and Computer Sciences.

Falkenhainer, B., Forbus, K. . D. & Gentner, D., 1989. The Structure-Mapping Engine: Algorithm and Examples. *Artificial Intelligence,* Volume 41, pp. 1-63.

Foo, N., Garner, B., Rao, A. & Tsui, E., 1992. *Semantic distance in conceptual graphs.* s.l.:s.n.

Gentner, D., 1983. Structure-mapping: A theoretical framework for analogy. *Cognitive Science,* Volume 7, pp. 155-170.

Gentner, D., 2006. *Analogical Reasoning, Psychology of.* s.l.:s.n.

Gentner, D. & Forbus, K. D., 2011. Computational models of analogy. *Cognitive Science,* Volume 2, pp. 266-276.

Gentner, D. & Smith, L., 2012. Analogical Reasoning. In: *Encyclopedia of Human Behavior.* Oxford: UK: Elsevier, pp. 130-136.

Gick, L. M. & Holyoak, J. M., 1980. Analogical problem solving. *Cognitive Psychology,* pp. 306-355.

Hage, J. P. R. N. v. V., 2010. A comparison of plagiarism detection tools. *Utrecht Uni-versity.*

Hoare, T., Misra, J., Leavens, G. T. & Shankar, N., 2009. The Verified Software Initiative: A Manifesto. 41(4), pp. 1-8.

Holyoak, K. J., Novick, L. R. & Melz, E. R., 1994. *Component Processes in Analogical Transfer: Mapping, Pattern Completion and Adaptation.* K. J. Holyoak & J. A. Barden (Eds.), Analogical Connections (Vol. 2, pp. 113-180) ed. s.l.:s.n.

Holyoak, K. J. & Thagard, P., 1989. Analogical Mapping by Constraint Satisfaction. *Cognitive Science,* pp. 295-355.

Jiwei, Z., Haiping, Z., Jianming, L. & Yong, Y., 2000. Conceptual graph matching for semantic search. *Proceedings of the 10th International Conference on Conceptual Structures: Integration and Interfaces,* pp. 92-196.

Johnson, C. W., 2005. The natural history of bugs: using formal methods to analyse software related failures in space missions. *Lecture Notes in Computer Science,* Volume 3582, pp. 9-25.

Kamiya, T. K. S. I. K., 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code.. *IEEE Transactions on Software Engineering,* 28(7), pp. 654-670.

Kamsu-Foguem, B., Diallo, G. & Foguem, C., 2013. Conceptual graph-based knowledge representation for supporting reasoning in African traditional medicine. *Engineering Applications of Artificial Intelligence,* 26(4), pp. 1348-1365.

Keane, M. T. & Brayshaw, M., 1988. *The Incremental Analogy Machine: A computational model of analogy.* s.l.:Third European Working Session on Machine Learning.

Keane, M. T., Ledgeway, T. & Duff, S., 1994. Constraints on analogical mapping: A comparison of three models. *Cognitive Science*, July, pp. 387-438.

Leino, K. R. M. & Monahan, R., 2007. *Automatic verification of textbook programs that use.* Berlin, Germany, ECOOP 2007 Workshop.

Marcus, A. & Maletic, J., 2001. *Identification of High-level Concept Clones in Source Code.* s.l., ASE 2001, pp. 107-114.

Meyer, B., 1992. Applying "Design by Contract". *Institure of Electrical and Electronics Engineers (IEEE),* 25(10), pp. 40-51.

Microsoft Research, n.d. *The Spec# formal language for.* [Online]
Available at: http://research.microsoft.com/en-us/projects/specsharp/
[Accessed 2013].

Mishne, G. & De Rijke, M., 2004. *Source Code Retrieval using Conceptual Similarity.* s.l., s.n., pp. 539-554.

Montes-y-Gomez, M., Lopez, A. & Gelbukh, A. F., 2000. Information retrieval with conceptual graph. *Database and Expert Systems Applications,* p. 312–321.

Mössenböck, H., Löberbauer, . M. & Wöß, A., 2011. *The Compiler Generator Coco/R.* [Online]
Available at: http://www.ssw.uni-linz.ac.at/coco/

Neamtiu, I., Foster, J. S. & Hicks, M., 2005. *Understanding source code evolution using abstract syntax tree matching.* New Tork, MSR '05 Proceedings of the 2005 international workshop on Mining software repositories, pp. 1-5.

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

Novick, L. R., 1988. Analogical transfer, problem similarity, and expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition,* III(14), pp. 510-520.

O'Donoghue, D. P., Bohan, A. J. & Keane, M. T., 2006. Seeing things: Inventive reasoning with geometric analogies and topographic maps. *New Generation Computing,* 24(3), pp. 267-288.

Park, W.-J. & Bae, D.-H., 2011. *A two-stage framework for UML specification matching.* s.l., Elsevier, p. 230–244.

Pitu, Mihai; Grijincu, Daniela; Li, Peihan; Saleem, Asif; O'Donoghue, Diarmuid; Monahan, Rosemary, 2013. Arís: Analogical Reasoning for reuse of Implementation &. *Artificial Intelligence for Formal Methods (AI4FM).*

Pitu, M., 2013. *Source code retrieval using Case Base reasoning,* Dublin: s.n.

Polovina, S., 2007. *An Introduction to Conceptual Graphs.* Berlin, Heidelberg, s.n.

Rattana, D., Bhatiab, R. & Maninder, S., 2013. Software clone detection: A systematic review. *Information and Software Technology,* 55(7), p. 1165–1199.

Sowa, J. F., 1984. Conceptual structures - Information processing in mind and machine.

Sowa, J. F., 2000. *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* s.l.:s.n.

Warren, M., Ng, K., Golde, P. & Hejlsberg, A., 2012. *The Roslyn Project Exposing the C# and VB compiler's code analysis.* [Online]
Available at: http://msdn.microsoft.com/en-us/vstudio/hh500769.aspx

Wilkinson, R., 1994. In: *Effective Retrieval of Structured Documents.* London: Springer, pp. 311-317.

Wilkinson, R., 1994. *Effective retrieval of structured documents.* New York, Springer-Verlag, pp. 311-317.

Wise, M. J., 1996. YAP3: Improved Detection Of Similarities In Computer Program And Other Texts. In: *ACM Special Interest Group on Computer Science Education.* s.l.:s.n., pp. 130--134.

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

Yang, W., 1991. *Identifying Syntactic Differences Between Two Programs,* s.l.: Software - Practice and Experience.