Evaluate and Benchmark *Aris*

Felicia Halim

Dissertation 2014

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 fulfillment

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, 2014

European Commission

ERASMUS MUNDUS

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.

Felicia Halim

_____

Acknowledgment

I would like to thank Dr. Diarmuid O'Donoghue and Dr. Rosemary Monahan from the National University of Ireland, Maynooth for their help and support during the project. I also thank Daniela Grijincu, Mihai Pitu, and Fahrurrozi Rahman, who also involved in *Aris* project for their help and input.

Abstract

In this paper, we present evaluation and benchmark of *Aris* (Analogical Reasoning for reuse of Implementation & Specification). *Aris* aims to increase the number of verified programs by promotes the advantages of code reuse and the possibility of transferring specifications between similar implementations. Source code retrieval in *Aris* acts as an enabling technology for the reuse of formal specifications. Although the result of the early version of *Aris* is encouraging and show potential reuse of formal specifications, it still has many rooms for improvements and wide possibility for feature enhancement. By using experimental methodology, we identify the issues and limitation of *Aris 1.0.* We develop *Aris 2.0* that improve the construction of conceptual graph, reduce the occurrences of false variable and loop mapping, enable adjustment for transferring specifications between different iteration process of *for* loop, and support *assert* and *assume* specifications transfer. We also introduce new metric (specification score) to ensure that the top ranked retrieved documents possess good quality specifications for transferring specifications. Finally, we compare the performance of *Aris 1.0* and *Aris 2.0* as retrieval and mapping system and also its ability to create the verified specification. In order to evaluate and benchmark *Aris*, we use 2 million methods of unverified implementations real world example amongst verified implementation. Our overall result shows the improvement in *Aris 2.0* able to produce more successful mapping between similar source code files, increase ranking precision in retrieval phase, and generate more verified specifications.

Contents

1. Introduction

1.1. Problem statement

Nowadays, we depend more with software system to do daily activities such as check bus schedule with mobile application or do online banking transaction. As more people use the software system, it is crucial to ensure that this software is reliable software. However, software faults sometimes are unavoidable and it leads to time and cost inefficiency or even cause system failures in critical applications (e.g. overflow exception that caused the *Arianne 5* missile crash (Johnson, 2005)). Therefore, there are increasing interest to proof the correctness of the system. *Verification* (Hoare, et al., 2009) is able to formally prove that the software will behave correctly (within the bounds of its specification) and fulfill its intended purpose.

Formal Software Verification uses formal methods of mathematics to construct a formal proof of a program's correctness. Proving correctness of a program is measured with respect to a *Formal Specification* which describes how the program will behave in certain situations. (Woodcock, et al., 2009) have discussed various improvements in software verification technology. Formal programming languages that implement *Design-by-Contract* (DbC) (Meyer, 1992) have been developed in order to allow specification of programs written in languages such as Java and C#.

Source code retrieval may act as an enabling technology for the reuse of formal specifications. The basic principle of software reuse is building a new software system based on contents of existing systems. *Aris* (Analogical Reasoning for reuse of Implementation & Specification) project (Monahan. R and O'Donoghue, 2012) presented in this paper as useful for retrieving source code for reuse for creating formal specifications.

1.2. Design by Contract

Object oriented technique is widely used in the software development. There is a particular attention to the *Dependability* in object oriented development. Dependability can be defined as

the combination of correctness and robustness, or in a straightforward term, as the absence of bugs. One way to guarantee the dependability is by relying on the concept of design by contract (Meyer, 1992). A contract protects two entities: the *client,* who calls the specified routines and entitled to receive certain results without knowing about the implementation; the *supplier,* who maintains and assures that the implementation, meets the required functionality. Design by contract (DbC) pattern was first applied in the *Eiffel* programming language and Meyer puts the effort to popularize the approach to different programming languages such as C# and Java.

Applying DbC for a programming language is called *Programming by Contract* (Meyer, 1989). The main principle for Programming by Contract is specifying different constraints using a formal specification language which implied from *precondition*, *postcondition,* and *class invariants*. Precondition implies the constraints under which the software component will function correctly, postcondition implies the constrains of expected results after the execution, and class invariants implies the statements that always true during the lifetime of the objects for all class instances.

Microsoft research has developed Spec# programming language that extends the capability of C# and enables software developer to document software design decision in the code. Based on (Leino, et al., 2004), Spec# programming system has a program verifier to check whether the implementation is correctly corresponds to the specification. While software developer explicitly specifies their assumption in the code, Spec# makes sure the program works under specified assumptions. It supports pre and post condition, invariants, and other specifications using clause such as *ensures, modifies, requires, invariant*, *assert,* or *assume*.

1.3. Introduction to *Aris*

Although there are significant benefits of writing verified software, the practice of writing formal specification is not largely adopted. One reason is because users face major difficulty to learn how to interact with formal verification tool such as how to write good assertions that describe what the program must do, and how to develop appropriate implementation so that the verification goal can be achieved more easily (Leino & Monahan, 2007). Moreover, Formal

verification tool uses many mathematical approach which people sometimes hard to understand in limited amount of time. The cost and time allocated to train people in this field also expensive.

Aris[1] (Analogical Reasoning for reuse of Implementation & Specification) project (Monahan. R and O'Donoghue, 2012) aims to promote the use of Spec# programming language in order to increase the number of verified software. The system reuses and transfers specification from previous verified programs, thus making software verification more accessible to software developers.

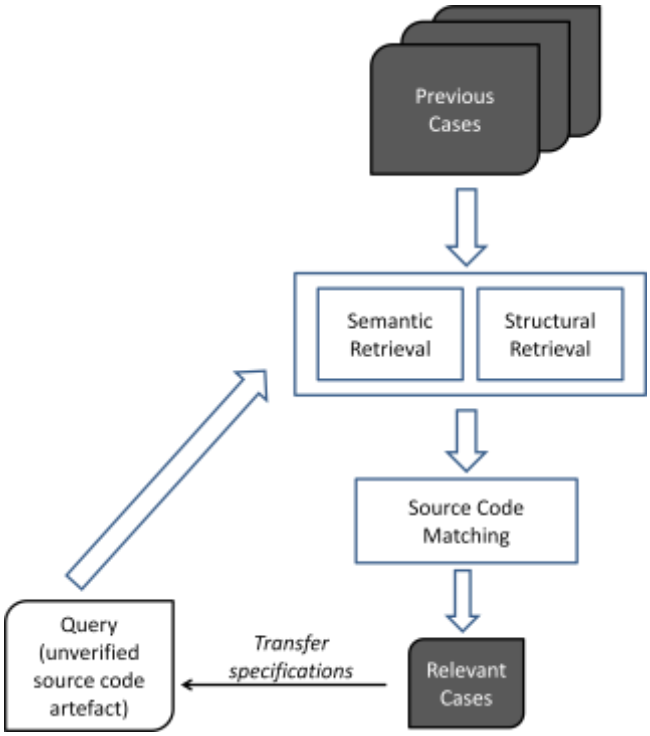

Figure 1.1. Overall architecture of *Aris*

Aris is a form of integration of sub-project of *Source Code Retrieval using Case Based Reasoning* (Pitu, 2013) and *Source Code Matching for Reuse of Formal Specifications* (Grijincu, 2013). *Source Code Retrieval using Case Based Reasoning* is responsible to retrieve similar verified code from past solutions based on semantic and structural characteristics of source code. *Source Code Matching for Reuse of Formal Specifications* (Grijincu, 2013) transfers the

[1] Irish for "again"

specifications from verified program to other program based on the mapping source code level implementations. The overall architecture of *Aris* can be seen in Figure 1.1.

1.4. Source code retrieval

Based on a survey conducted by (B. Dit, etc, 2013), there is various approaches of feature location techniques for software reuse. One specific example is an approach based on information retrieval that uses information from software repositories. Source code retrieval module in *Aris* creates a framework that able to automate the writing of specifications by reusing existing verified software artefacts. *Aris* will retrieve programs from a large set of samples and transfer formal specification from the retrieved implementation to the query. The core concept of this module is software retrieval and a source code retrieval system is also used in:

- **Detecting plagiarism and code theft:** There is a growing concern of plagiarism in University programming class. Therefore, tools using retrieval techniques are built nowadays to improve plagiarism detection.
- **Programming by analogous examples:** *Programming by example* (PBE) helps end-user to build a program without prior formal training in programming. PBE uses analogies mechanism to construct new knowledge based on understood previous knowledge. Instead of creating the solution from the beginning, PBE supports reuse of previous recorded examples. (Repenning & Perrone, 2000).
- **Rapid prototyping:** In the initial phase of software development, project stakeholder and software engineers often use rapid prototype to list the software requirements and discuss possible features. To reduce the cost of creating software prototype, there are tools that can help to find the software component in the open source repositories. (McMillan, et al., 2012).

1.5. Source code matching

Central motivation of our research is the re-use of formal specifications obtained from carefully selected similar methods. Given a repository of verified software, *Aris* matches the

new implementation to the most similar block of source code identified in the repository, reusing that specification so the new implementation can be automatically verified. Although methods for measuring source code similarity are beneficial in various scenarios; there are still little research towards a framework for transferring and generating new specification. Thus, *Aris* project comes as a novel approach in this area of research (Grijincu, 2013).

There are some few other fields in which system compares two source code files in its approach:

- **Source Code Plagiarism Detection:** Plagiarism ranging from copying source code or adopting ideas from other authors without providing adequate acknowledgements (Cosma & Joy, 2006). Plagiarism often occurs in programming class assignment. Therefore, the marker of the assignment uses tool to automatically compare the similarity of programming solution and detect the plagiarism.
- **Software evolution**: To maintain evolving software is an expensive process, as software continues to grow, the complexity of the software also increases gradually. With the increase in complexity, it is usually difficult to fully understand the modifications and also an extension of the software. A source code similarity measurement is used to detect trends and patterns in modification along a software life cycle development. Thus, it would give a better understanding how software evolves (Bhattacharya, et al., 2012).
- **Code duplication management:** Previous research reports (Chanchal, et al., 2009) the proportion of code duplication in software systems may be as low as 5% or even as high as 50%. Duplicated code brings negative impact in maintenance effort. Redundant codes also increase resource requirements. One might remove duplicate code by refactoring, though not all duplicate codes are removable using this way. Therefore, there are needs to detect and manage code duplication efficiently.

1.6. Motivation

Although the result of the early version of *Aris* is encouraging and show potential reuse of formal specifications, it still has many rooms for improvements and wide possibility for feature enhancement. In this project, we address the following question: *How we evaluate and benchmark first version of Aris (Aris 1.0)?* We propose to use experimental methodology (Amaral, et.al, 2007) to address the research question. Experimental methodology can be divided into exploratory phase and evaluation phase. In exploratory phase, we collect a list of questions regarding the performance and competence of *Aris 1.0* as the retrieval system for specifications reuse. Then, we attempt to answer the questions produced in the exploratory phase in the evaluation phase.

Based on the source code matching module evaluation result, we develop *Aris 2.0* to improve graph construction process, enhance IAM algorithm for finding valid mapping, generate more verified specifications e.g. enabling the transfer of *assert* and *assume* specifications, and adapt the specifications in the target context. *Aris 2.0* will also improve the retrieval module of *Aris 1.0* to retrieve and rank the document not only based on the implementation similarity, but also considering the possibility of each document for specification reuse.

In order to characterize the performance of *Aris 2.0,* we will benchmark the system against *Aris 1.0* focusing on the system ability to identify source code similarity, retrieve verified documents in the retrieval module, and generate verified specifications in the problem domain. The benchmarking process assesses the system performance in four different categories that represent different degree of similarity with original unmodified implementation (Wilkinson, 1994):

- **Identical implementations**: There is no source code modification
- **Small modifications**: Transform the source by applying different levels of modifications such as renaming variables¸ data type changes such as *int* and *float*, and change loop constructs with its equivalent such as for and while (without change functionality).

- **Medium Modifications**: The modification introduce redundant statements or removing certain statements (once again without change code functionality)
- **Dissimilar**: Alter the functionality of the program (e.g. from a program that sums the element in the array into program that swap the element in the array)

1.7. Conclusion

We have presented the problem that we have focused on in this project. We also provide the motivation for developing this project and describe its impact. The rest of the paper is organized as follows: Chapter(2) gives an overview of previous version of *Aris* system which later called as *Aris 1.0* and also other related system and research that have influenced the solution choice of *Aris.* In the Chapter (3), we present the methodology we have used to evaluate and benchmark *Aris.* We also will explain our proposed solution to address problems in the earlier version of *Aris* to improve the overall system. Finally, we present our evaluation results in Chapter (4) and give our conclusions with future work discussion in Chapter (5).

2. Related Work and *Aris 1.0*

This chapter will give an overview of the previous version of *Aris* system which later called as *Aris 1.0* and also other related systems and researches that have influenced the solution choice of *Aris.* Section (2.1) explains about code similarity detection techniques and classifies them based on their approaches. Section (2.2) will summarize the concepts, terminology, and algorithm used in the source code matching module in *Aris 1.0*. We will also give an overview of other related researches in the retrieval module in *Aris 1.0* (Section 2.3) and specifically describe the implementation of the retrieval module in *Aris 1.0* (Section 2.4).

2.1 Source Code Similarity Detection Technique

Software reuse has been a common practice in software development field which involves copying and pasting of code blocks. The code clone usually needs slight modifications before it is able to work properly in the new environment. (Bellon, et al., 2007) analyzes and compares various tools and techniques used to detect source code similarity.

System such as CCFinder (Kamiya, 2002), PMD[2], or Simian[3] uses pattern-matching algorithm which found to be effective to detect duplicated code or very similar segment of code that scattered around large-scale enterprise project. The algorithm in pattern matching transforms the source code as tokens or lexical entities and detects the similarity by finding the occurrences of the pattern. This technique is known to be fast, but a very simple structural code change will negatively affect the accuracy of the algorithm.

System such as MOSS[4], YAP, or JPlag[5] compares the structural properties of the programs which is more effective when measuring the code similarity. The algorithm represents source code as string tokens and compares using string based distance. These tools are common to

[2] PMD: Project Mess Detector. http://pmd.sourceforge.net/
[3] Simian: Similarity Analyser. http://www.harukizaemon.com/simian/index.html.
[4] MOSS: Measure Of Software Similarity. http://theory.stanford.edu/~aiken/moss/.
[5] JPlag: Detecting Software Plagiarism. https://jplag.ipd.kit.edu/

detect plagiarism in student's assignments. However, (Hage, 2010) reported MOSS and JPlag are sensitive to insertion of redundant statements such as the insertion of many `Console.Writeline` statements.

Other systems parse programs into graph-based data structure and then extract several metrics and perform structural comparisons of the source code. (Yang, 1991) compares source code *Parse Trees*[6]. One of the weaknesses of this algorithm is each node are the actual grammar of tokens and literals which make this approach too verbose representation and provide no abstraction layer (Grijincu, 2013). On the other hand, *abstract syntax trees* [7] provide some abstraction. However, it still preserves unnecessary information such as whitespaces or punctuation.

Graph-based techniques have been widely used and are very active research area. (Bhattacharya, et al., 2012) showed a different graph-based metrics extracted from AST can help analyze software evolution and subsequently facilitate software development and maintenance. The particular section is the usage of $NodeRank$ to calculate the importance of the certain code component that represented by the node in the overall source code. The $NodeRank$ metric is used to reduce complexity in the graph matching algorithm.

2.2 Source Code Matching on *Aris 1.0*

Aris explores the possibility of transferring the specification between two similar programs in order to reduce the effort of writing specifications. *Aris* measures the similarity of two programs and if the similarity degree has value above the certain threshold, it will transfer the specification. The mechanism of specifications transferring begins by comparing two C# source code files. Each source code is represented as *Conceptual Graphs* (Pitu, *et al.*, 2013) which able to store the structure and content of the source code. It matches the graphs to identify the structure mapping between two domains. The next step is to perform *Analogical Reasoning*

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

computational model called *Incremental Analogy Machine* (Keane & Brayshaw, 1988) to find either isomorphic (exact matches) or homomorphic (non-identical) sub-graph mappings. Finally, it transfers the specifications into target based on the detailed correspondences found by using *Copy with Substitution and Generation* (CWSG) pattern algorithm (Holyoak, et al., 1994).

2.2.1 Conceptual Graphs

A Conceptual Graph [CG] was introduced by Sowa in 1984 with origins from the semantic networks used in Artificial Intelligence and Sanders Peirce work on existential graphs. A conceptual graph is a connected graph that has a finite number of nodes and also a bipartite graph as it models the relation between two kinds of nodes: *concepts* and *relations*. Concept node in CG is able to represent entities, attributes, and events in the knowledge domain. While Relation node describes how concepts relate to one another. Each node in CG associated with *referent value* which is a particular instance of the node.

CG relies on a *support*, which defines the syntactic constraints and provides background information on a specific domain. This notion of support can be grouped by the following:

- A set of *concept types* which structured in a *lattice* and represents "is-a-kind" relationship.
- A set of *relation types*
- A set of star graphs which shows for each relation concepts, which other kind of concept types able to connect.
- A set of *referent sets* for concept vertices: at least a generic "*" marker and individual markers which allow distinguishing and naming distinct entities.

Conceptual graphs have been implemented in various information retrieval applications, natural language processing, database design, and source code retrieval.

In this project, *Aris* uses conceptual graph by starting to define the concepts and relations that are allowed in the graph and indicate how they can connect each other and possible

referents each concept type can have. (Figure 2.1) explains the hierarchy of concepts based on examining C# source code files. The description of each concept and relation can be seen in Table 2.1 and Table 2.2 respectively.

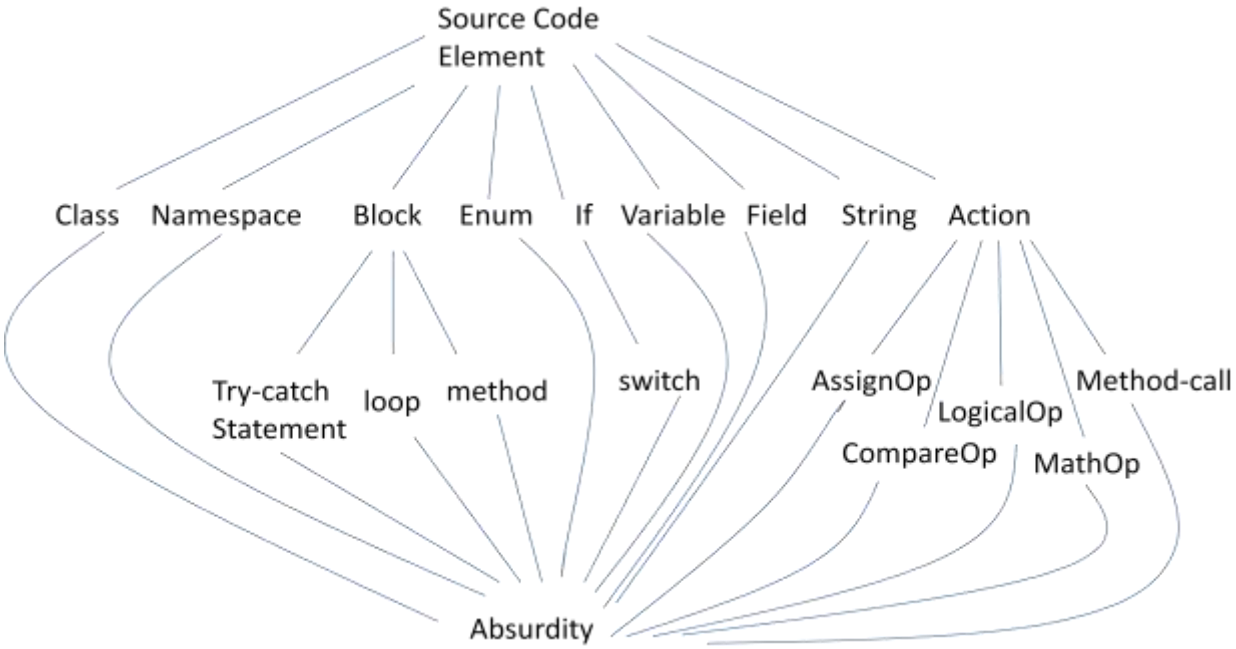

Figure 2.1 Example of Hierarchy of concepts in C# source code files

Concept Type	Description
AssignOP	Assign new value to a field or variable. Assignment also may perform operation such as addition "+=" or subtraction "-="
Block	A set of concepts that are structurally grouped together. For example, code inside curly brackets {..}.
Class	A declaration or definition of a class
CompareOP	A binary comparison operator such as ">=", "!=", etc
ENUM	A declaration of an enumerated set of values
Field	A declaration of variable directly in a class (a class attribute)
IF	A statement that controls conditional branching
LogicalOP	Symbols that perform logical operation such as "OR","AND", etc.
LOOP	Iteration statements that can cause statements to be executed in number of times depend on termination criteria.

MATHOP	Mathematical operations such as “+”, “-“ , etc.
METHOD	Code block that contains series of statements that perform as function.
METHOD-CALL	A method invocation or execution
NAMESPACE	Declare a scope that contains a set of related object such as one or more class, ENUM, etc.
NULL	A null reference
STRING	String represents a sequence of Unicode characters.
TRY-CATCH STATEMENT	Try block followed by one or more *catch* clauses, which specify handlers for specific exception.
VARIABLE	An entity in the program that store specific type of value

Table 2.1 List of concepts and its description

Relation Type	From Concept	To Concept	Description
Condition	If	Action	Describe the conditional statement within the *if* clause.
		String	
		Variable	
		Field	
	Loop	Action	Specifies the conditional statement that determines statements to be executed in number of times depend on termination criteria.
		String	
		Variable	
		Field	
Contains	Action	Action	The action can use or depend on other concepts
		String	
		Variable	
		Field	
	Block	Action	A block can contain this concept
		String	
		Variable	
		If	
		Enum	
		Try-catch	
	Class	Field	A class definition contains this concept
		Method	
	Enum	String	The enumeration elements are defined as strings
	Method	Block	The method definition contains a block of concepts
	If	Action	The branching statement can contain a block with multiple concepts or just an action
		Block	
	Loop	Action	The loop can contain or initialize other concepts
		Block	

		Variable	
		Field	
	Namespace	Class	The namespace definition contains the class
Defines	Block	Namespace	A block gives a definition of the namespace
Depends	Block	Namespace	A block can depend other namespaces
Parameter	Method	String	The method definition contains the concept as parameter
		Variable	
	Method-call	String	The method is called with this concept as parameter
		Variable	
		Field	
Returns	Method	Action	The method returns a value
		Variable	
		Field	
		String	

Table 2.2. List of relation types and explanation for concepts they can connect.

2.2.2 Graph Construction Algorithm

Aris needs an *Abstract Syntax Tree (AST)* representation in order to analyze the C# source code files. One way to generate AST for C# files is using *Microsoft Roslyn Project*. However, the AST generated by Rosalyn does not provide any level of abstraction of conceptual graphs. For example, a Loop concept refers to any of *do-while*, *while*, *for*, or *foreach* statements. Therefore, the conceptual graph construction process takes the AST root and traverses all its descended nodes in a *Depth First Search*[8] manner in order to create the corresponding concepts and relation in the conceptual graph. In Figure 2.2, we can see the mapping between the expressions (nodes) at the AST level to the concept or relation nodes in conceptual graph and in Figure 2.3, there is an example of conceptual graph representation of a simple program.

[8] Depth First Search - http://www.cse.ust.hk/~dekai/271/notes/L06/L06.pdf

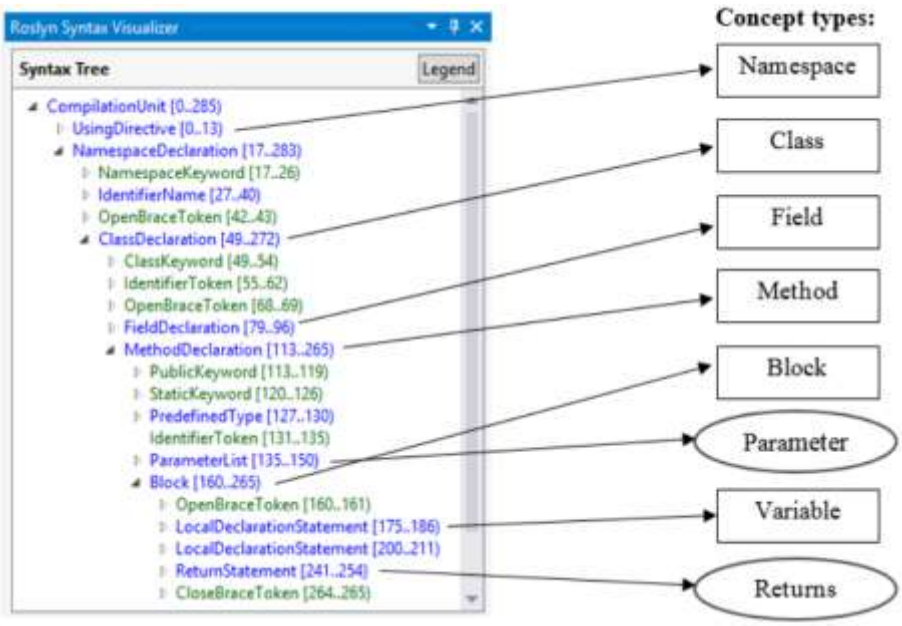

Figure 2.2. Maps of expression (nodes) at the AST level to the concept or relation nodes in conceptual graph (Grijincu, 2013)

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

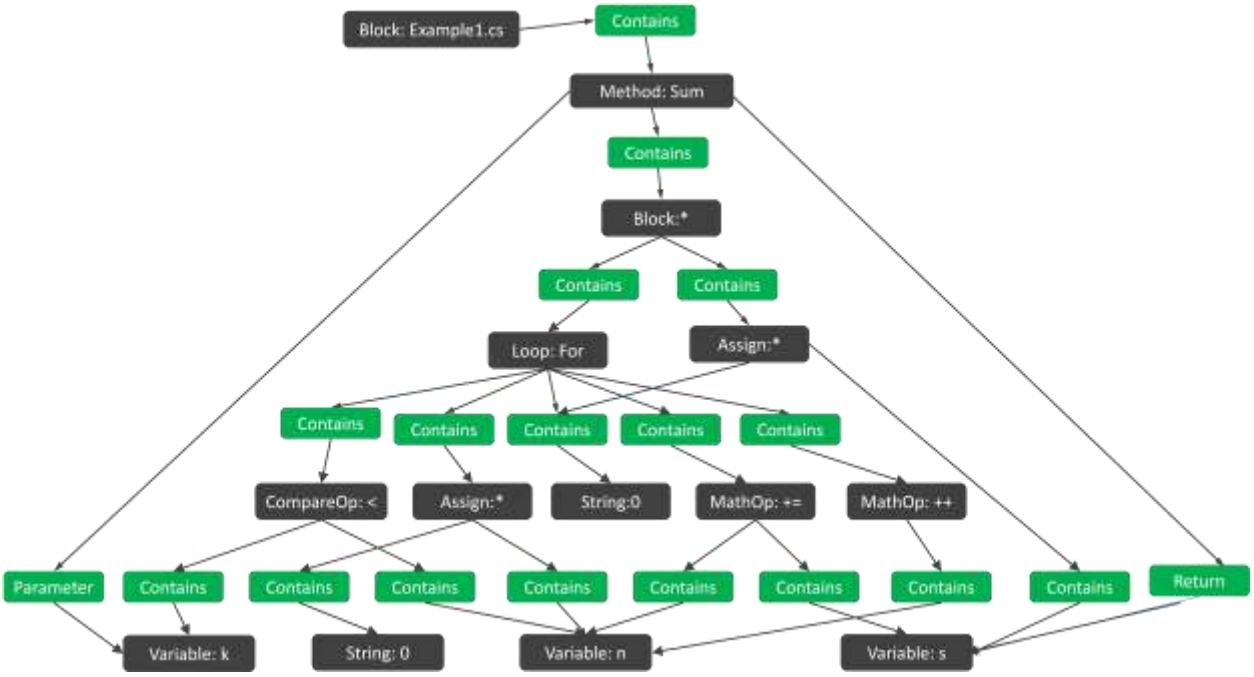

Figure 2.3. The transformation of a program into conceptual graph

2.2.3 Analogical Reasoning

In order reducing the complexity of graph mapping[9], *Aris* uses Analogical Reasoning (AR). In every analogical process, we find similar situation and try to match with less familiar situation. (Gentner & Smith, 2012) describes it as key process in scientific discovery, problem-solving, decision making, and categorization, which is very active research of Artificial Intelligence and Cognitive Science. A classic analogy (Gentner, 2006) example is between the structure of the atom and the solar system – as planets revolve around the sun so do the electrons revolve around the nucleus in the atom domain. (Keane, et al., 1994) point out most important process involved in developing an analogical process:

1. **Representation.** In order to find the solution of a problem, Theproblem needs to be represented in a meaningful form. (Novick, 1988) has shown that how the problem is represented, affects subsequent success of the analogical transferring.
2. **Retrieval.** This step focus on finding the best candidate that matches with the target. This step involves searching through the database and retrieving the most similar one to the target.
3. **Mapping.** This step matches element of the base domain with element from the target domain. It is often very complex and computationally expensive process.
4. **Transfer.** Based on mapping result, new knowledge is generated and transferred into the target domain.
5. **Evaluation.** The transferred knowledge needs to be validated to ensure newly generated knowledge is suitable within the target domain.

In *Aris* we use analogical reasoning of developing inferences and generating new information in target code (which we want to formally specify using the existing specifications from the base problem).

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 domain and finds

[9] Graph Mapping known as NP hard task

the detailed collection of correspondence between them (Gentner, 1983). Correspondences are linking particular elements from the base domain with particular elements in the target. The most influential theory of analogical mapping is Gentner's *Structure Mapping Theory* (SMT) (Gentner, 1983) which involves aligning base and target domains by finding structural similarity between them and developing candidate inferences. The SMT proposes some constraints for the analogical mapping process:

- **Structural consistency:** It enforces one-to-one mapping of element between base and target domain. This means ambiguous matches (one-to-many or many-to-one) have to be replaced with one-to-one mapping.
- **Systematicity:** In order to develop mapping, connected group is preferred over independent ones

Example of analogical mapping in the source code is *for* loop and *while* loop. Based on (Gentner, 1983), the structure mapping theory was best identified with graph-matching because structure mapping help extract detailed correspondences between two conceptual graphs. One of the examples of SMT framework implementation is done by (O'Donoghue, et al., 2006) in the GeoComputation domain.

2.2.4 Incremental Analogical Machine (IAM)

(Keane & Brayshaw, 1988) has developed the Incremental Analogy Machine (IAM) as a computational model based on Gentner's structure mapping theory that implements both informational and behavioral constraints using serial constraint satisfaction (Holyoak & Thagard, 1989). Rather than match every element in the domain at once, IAM constructs the mapping incrementally by selecting small portion of base domain and mapping it before moving to map another portion. The IAM algorithm by (Keane, et al., 1994) is described in the following steps:

1. **Select Seed Group.** Rank each connected elements in the group and reorder them. Take the first such group in ordered list as the seed group.
2. **Select the Seed Match.** Select the element in the first group and find good matching in the target domain.
3. **Find Isomorphic (One-to-One) Matches for the Group.** Find valid matches between elements in the selected group and target domain and enforce constraint satisfaction in order to find one-to-one set of matched that discard ambiguity using pragmatic, similarity, and structural constraints.
4. **Find Transfers for the Group.** Add candidate inferences to mapping based on matches found in step 3.
5. **Evaluate the group mapping.** If the mapping is evaluated as being good, then continue to step 6, otherwise go to step 2 to try with an alternative seed match. If there is no better seed match, go to step 1 and find another group as the seed group.
6. **Find other Group Mappings.** Perform step 1 until step 5 to incrementally map remaining unmapped groups.

IAM as a computational model that implements analogical mapping has shown good performances compare to other computation model (Gentner & Forbus, 2011). IAM algorithm greatly reduce the process of comparing two structural representations as it uses incremental model that iteratively adds new mapping between the base and target domain, rather than try to map all elements in the domain. Moreover, algorithm can do backtrack to find the alternative mappings if current acquired mapping evaluated as less successful. However, there are also challenging aspects of IAM algorithm. Firstly, the difficulty to select the good seed group, algorithm that is used to rank elements in the groups is essential for finding a successful seed group used in incremental process. Second, we have to define set of match rules and constraints in order to determine valid matches and produces *one-to-one* mapping between source and target domain, specific rules must be applied to discard ambiguous matches such as *one-to-many* and *many-to-one* mappings (Griijincu, 2013).

2.2.5 Comparing Conceptual Graph using IAM Algorithm

Earlier research in comparing conceptual graph is Sowa's set of projections and morphisms defined in (Sowa, 1984) by measuring the distance between two concepts from the support of the graph and also known as *semantic distance.* Sowa's approach, however, is too strict and focuses on finding structurally identical graph or sub-graphs, for *Aris* we are interested in finding and matching *homomorphic graphs* (allow different level of structures to be mapped together). Moreover, this algorithm also works together with the *Source Code Retrieval* module in *Aris* where the nature of retrieval model usually allows a certain degree of "fuzziness" (Mishne, 2003).

The graph matching is known to be difficult due to *NP-Completeness*[10] nature of the problem. (Bunke, 2000) has discussed the recent development of graph matching in numerous applications including case-based reasoning, machine learning, conceptual graph, etc. Although there is a standard algorithm for graph and subgraph isomorphism detection that guarantees to find the optimal solution, it requires exponential time and space. Therefore (Mishne & De Rijke, 2004) avoid this by doing node-by-node comparison which also known as *maximally similar concept*. They compare a concept in graph G_1 to all concepts in graph $G_2$and the complexity would be $O(|G_2|.(|G_1|.|G_2|))$.

Aris uses Incremental graph matching algorithm based on Incremental Analogy Machine (IAM) (Keane & Brayshaw, 1988). The main reason for choosing IAM algorithm is because it allows us to explore the nature of conceptual graph and able to obtain similar results to human's analogical reasoning process: it is able to generate complex mapping relatively quick and can reconsider the mapping and generate new alternative ones (Keane, et al., 1994).

Aris compares two source code files that have related structure by first constructs the conceptual graph representation of both files and computes the node ranks for each concept or relation in the graphs using the *Node Rank* metric. We later map the two graphs by selecting

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

their sub-graphs and doing node-by node comparison in an analogical mapping process which composed by following steps:

2.2.5.1 Sorting nodes by Node Rank

(Bhattacharya, et al., 2012) proposed Node Rank which is similar to the *Page Rank* [11](Brin & Page, 1998) that represents a probability score proportional to the number of times a random surfer would visit a particular web page. Node Rank is used to reduce complexity of the incremental matching process by mapping nodes based on their relative importance in the graph. We sort elements in the base and target domains such that the highest ranked nodes means the most important element in the program. This means we measure the structural importance of the nodes based on their ongoing and outgoing edges to or from it.

Node Rank algorithm measures the importance of the certain code component that represented by the node in the overall source code by assigning numerical weights for each node in conceptual graph. We define the formal description of the recursive calculation of the node in a graph as follows: let u denote a node in the graph, $NR(u)$ represents node rank, $IN(u)$ represents the set of nodes v that have an outgoing edge into u and $OutDegree(u)$ represents the number of edges going out the node u. An iterative process calculates a new $NR(u)$ in every iteration as the sum over all$v \in IN(u)$.

$$NR(u) = \sum_{v \in IN(u)} \frac{NR(v)}{OutDegree(v)}$$

The iteration process stops when the values converge. Here the definition of convergence in here is when the difference between old sum and current sum is less or equal to 0.001, or the iteration limit has been exceeded, e.g. > 50. In order to enable convergence, after each iteration, the node ranks are normalized so that their sum adds up to one.

[11] *Page Rank*. http://homes.cs.washington.edu/~pedrod/papers/nips01b.pdf

2.2.5.2 Selecting sub-graphs

Next we map the two graphs by selecting their sub-graphs. This is the key process to reduce the complexity of analogical mapping where it incrementally selects the sub-graph from the source domain to map with target domain instead of mapping all elements from base to target in an exhaustive manner. We uses sorted node rank values and then map the methods one-by-one in a decreasing manner. The mapping algorithm enforce 1-to-1 mapping.

2.2.5.3 Mapping Sub-graphs

Both methods in the source and target domains are represented by conceptual graph. First, we employ parameter subgraphs check between the parameter of the method by comparing each parameter of the base domain method with every parameter of the target method (This can be done by looking at the concepts connected by the *Parameter* relation in the corresponding sub-graphs). Next measure similarity score between two concepts to select the best possible mapping.

The rest of the concepts that describes the body of the methods become the second sub-graph. Each concept in the source domain is mapped to the most similar concept in the target domain by taking each candidate under decreasing order of NR value. 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. To ensure that the mappings are consistent, we use a Boolean function *valid* (Section (2.2.5.6)) that determines whether a match is accepted as valid and added into *inter-domain* mapping (consisting individual elements from the base and their corresponding element from the target). If a match is not considered valid, using backtracking, we try to match the current base element to the next highest NR valued element from the target until $match_{depth}$ threshold is exceeded (it was set to 10).

2.2.5.4 Resolving ambiguities

To enforce 1-to-1 correspondence, *Aris* uses inter-domain mapping as a structure holding $< Key, < Key', Value' \gg$triples where 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 *sim*, the similarity score obtained. Whenever it finds potential valid match between concepts from the source and target domains, *Aris* checks for possible ambiguities:

1. *Many-to-one mappings*: Multiple concepts in the source domain correspond to one concept in the target domain.
2. *One-to-many mappings*: A concept from in the source domain corresponds to multiple concepts in target domain.

In both cases, the algorithm replaces the old mapping with the new found if the new correspondence has higher *sim* score.

2.2.5.5 Evaluating sub-graph mappings

This process corresponds to the evaluation steps in IAM algorithm. *Aris* performs minimal evaluation by setting threshold of mapped elements in the group. *Aris* uses 50% mapping threshold or known as IAM mapping constraint. If the mapping considered as successful, then it incrementally map for the next group (other method) in the class. Although there are separate sub-group mapping, the incremental mapping activities contributes to one same inter-domain mapping, forming one consistent interpretation of the comparison.

When the mapping process finishes finding all the valid matches between sub-graphs in the source and target domain, then the graph similarity score is the number of valid matches between two domains over the total of actual nodes in source domain, which can be formally written as:

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

Aris calculates similarities by considering mapping from source to target and mapping target to source. The mapping result will not be symmetric if the source and target are different as the process rely on the order and the number of seed groups in the domain. The final score for calculate a similarity is average score in both ways or it can be formally written as:

$$\frac{GraphSim(CG_{source}, CG_{target}) + GraphSim(CG_{target}, CG_{source})}{2}$$

However, for generating and transferring the specification *Aris 1.0* only relies on the graph matching result from source to target without considering the target to source graph matching result.

2.2.5.6 Mapping constraints

In order to check the validity match between two conceptual nodes, the algorithm enforces some match rules and constraints which implemented as *similarity functions* that check if certain properties hold in the mapping.

1. **Type Similarity function:** Ensuring the mapping only between the same type entities (i.e *Variable* with *Variable, Defines* with *Defines*) which greatly reduced unnecessary match and make the mapping process more efficient. The formal definition of similarity function is:

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

2. **Structural similarity function:** Ensuring the structural consistency by checking the nodes having edges into or from input nodes also have the same concept type. This function particularly useful to eliminating the ambiguous mappings since the most

structurally similar matches will return the highest similarity score. sim_{struct} function compares the nodes on immediate upper and lower levels in both graphs and then count the number of nodes that have the same concept types and divides it with total number of nodes from the largest context of the input node.

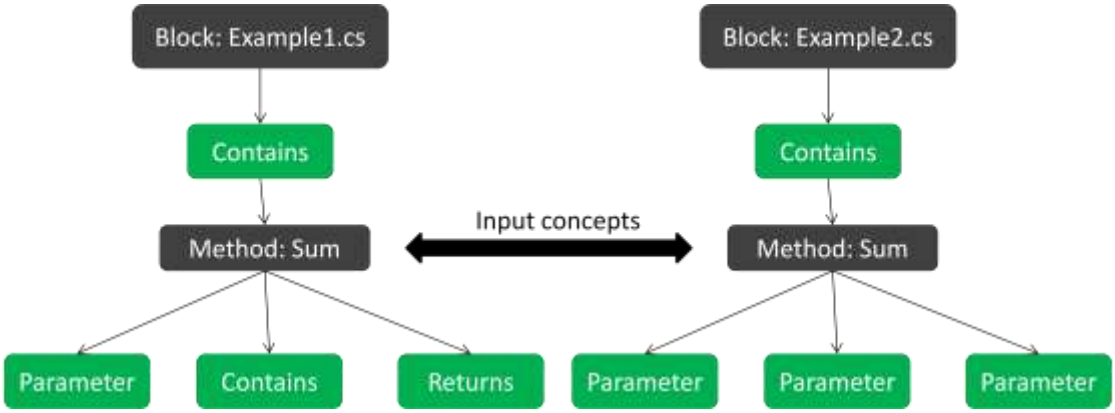

Figure 2.4. Example of graph to calculate structural similarity function where graph in the left hand side is part of the method from `example1.cs` and the graph in the right hand side is the part of the method from `example2.cs` (Grijincu, 2013)

To demonstrate the similarity function, see Figure 2.4 as an example. *Method:Sum* are the input nodes, it has 2 pairs of nodes that have the same type (*Contains-Contains, Parameter-Parameter*) and an equal number of 4 nodes in both contexts. Thus the $sim_{struct}(Method{:}Sum, Method{:}Sum) = \frac{2}{4} = 0.5$

3. **Content Similarity function:** This function compares the similarity of information that is stored in concept types. For *Variable*, *Field*, and *Method* concept types, it checks whether types have the same super-type or one is the sub-type of the other within class hierarcy e.g variable *int* and *double.* If both types come from same hierarchy or one is the sub-type of the other or both have the same type, then content matching similarity score is 1 and 0 otherwise. For the rest of the concepts that have referent value

(CompareOp, LogicalOp, MathOp, and Loop), it use *Levenshtein* [12]string distance that computes the similarity between two input strings. The returned score for content similarity between 0 and 1, where 1 represents the maximum values of similarity.

After computing of each similarity function, then, we need to calculate the overall similarity score *sim*, which can be defined as:

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

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

It uses $weight_{type}$ = 0.5, $weight_{struct}$ =0.3 and $weight_{content}$ = 0.2

From the overall similarity score, it defines the next function $ValidMatch$ that will decide whether to accept mapping as being valid or reject it. The formal definition of $ValidMatch$ can be defined as follows:

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

2.2.6 Analogical Inference and Pattern Completion

Aris uses an algorithm from pattern completion called CWSG – *Copy with Substitution and Generation* (Holyoak, et al., 1994) to generate analogical inferences. CWSG transfers the specification from Base code and adds it to the target code by substituting source code items with their mapped equivalents. Thus, it transfer *modifies, requires, ensures,* and *invariant* statements with mapped equivalents. In Table 2.3, there are examples of program from source and target domain and Table 2.4 presents the detailed correspondences between their conceptual graphs. These examples also presented in (Grijincu, 2013).

[12] Levenshtein string distance - http://software-and-algorithms.blogspot.ie/2012/09/damerau-levenshtein-edit-distance.html

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

Table 2.3. Base and target methods examples. Although their structure slightly different in loop constructs (base uses *for* loop and target uses *while* loop), both implementations are highly similar.

```
{ 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 2.4. Output of IAM algorithm containing element correspondences of base and target domain in Table 2.3

Based on the correspondence found in IAM algorithm, variable in base domain can be mapped with variable in the target domain. We can use these mapping to generate new

specification by replacing the appropriate variables that appear in *requires, ensures,* and *invariant* statement with their mapped equivalents. The solution can be seen in Table 2.5.

```
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 2.5. Example of target code with its specifications transferred

2.3 Source Code Retrieval

Source code retrieval requires understanding of the structure and content of the code (Pitu, 2014). (Michail & Notkin, 1999) presents two matching techniques that used in source code retrieval which are *name matching* and *similarity matching.* The name matching method matches components that have standardized name in the library of specific programming language. Similarity matching uses text in the libraries applying free-text indexing technique that used in conventional information retrieval. (Mishne & De Rijke, 2004) has proposed model for source code retrieval by using conceptual graph as source code representations. Conceptual graph allows extracting the contents and structural characteristics of source code.

2.4 Source Code Retrieval on *Aris 1.0*

Retrieval module in *Aris* generates new specifications using previous knowledge recorded in memory. The overall methodology of retrieval module in *Aris* is similar to Case-Based Reasoning (CBR). Case Based Reasoning (Kolodner, 1993) is an Artificial intelligence technique that focuses

on problem solving and it is a form of Instance Based Learning (Russell & Norvig, 2003). In CBR, new problems are solved by searching and retrieving similar previous problems (cases) from memory and reusing the old solutions by transferring knowledge to the new problem. Case-Based reasoning approach in *Aris* works as follows: for a given unverified implementation as input query, similar implementations are retrieved from case-base by exploring structural and semantic similarities between the query and memory source code artefacts and also combined with the similarity score from the source code matching module. The results will be displayed in decreasing rank manner depends on the similarities between the query and retrieved candidate; the top ranked source code artefacts potentially generate a new specification for the query. The specification is transferred to the input implementation until generated specification is verified using existing formal method tools. If previous step is successful, the new solution is retained for further use. In order to perform knowledge transfer using formal specifications on large set implementations, this module use repository of software artefacts from managed compiled assemblies such as Dynamic-Link Libraries (DLLS) or Executables (EXEs).The repository contains a small set of verified implementations using Spec# formal method and extended with a large number of open source programs.

Semantic retrieval process in the retrieval module of *Aris* uses the API Vector Space Model (VSM) (Salton & McGill, 1986) for representing source codes and computing similarities between these representations. Vector Space Model is a procedure capable of representing documents as vectors where the documents in this context are usually considered textual, but the technique actually can be applied to various other document types (objects). VSM treats documents as bags of terms and applies weighting for each indexed term in order to achieve document retrieval. In this project, the similarity score computed based on number of shared API calls in documents (source code). The used of API calls is justified by real world software applications since they have precisely defined semantics unlike variable names, type, or method names that programmers use. Source code in document is represented as *vector* of real numbers and similarity between two documents by division of dot product of those two vectors and product of the vector Euclidian lengths.

Another technique used in the retrieval phase of *Aris* is *Structural Retrieval.* This technique focuses on structural topological characteristics of the source code. This process can be run independent without semantic retrieval process as it use different representational structure for source code and different source code artefacts from the case base are structurally similar to the query. In this project, Structural retrieval process in the retrieval module of *Aris* computes the similarity between the query and the cases in memory (associated with the source code artefacts) by exploring the structural characteristics of the source code derived from *Conceptual Graphs*. Conceptual Graph is used in this process because it enables us to explore both semantic content and structural properties of the source code using graph-based technique. Conceptual graph also contains much more detail information regarding original source code document rather than other alternate *Abstract Syntax Trees and Parse Trees.* In this project, *Aris* uses *Content Vectors* (Gentner & Forbus, 1994) to represent structures which encode information extracted from the conceptual graph. Content vectors are an encoding mechanism for structured representations. For each of conceptual graph representation of a source code artefact, measures and metrics used are as followed (Pitu, 2014):

- Vertex count(#V) : number of vertices
- Edge count(#E) : number of edges
- Average degree: two times number of edges divided by number of vertices
- Graph diameter: the longest shortest path between any two vertices
- Maximum out degree: maximum number of outgoing edges in a node
- Maximum in degree: maximum number of incoming edges in a node
- Average node rank: average node rank between all nodes

Since the Case-Base size can be very large, therefore the comparison of query is limited to only relevant source code artefacts.

Retrieval in $Aris$ is combining the results from semantic and structural retrieval. These two processes are independent as each of them analyzes different properties of documents relative to the query. Besides working as independent process, they are design to complement

each other and their symbiosis is reached through combined retrieval, where both structural and semantic features are reflected in the results.

In order to assess the correctness of the retrieval algorithm and perform knowledge transfer using formal specification on a larger set of implementations, we use a repository of software artefacts. This repository also supported from an architectural point of view where *Aris* uses Case Base Memory approach. Case Base Memory is a collection of past cases which aggregates source code artefacts optionally associated with the corresponding formal specification for that implementation.

The repository consists of corpus of documents from managed compiled assemblies such as Dynamic-Link Libraries (DLL) or Executables (EXEs). There are some benefits of using this approach which are the compactness of the corpus, documents are with high probability finished work (at least free of compilation errors), and the documents contain Common Intermediate Language (CIL) can be converted back into C#, VB, F#, etc. This means that the code retrieval algorithm can be used independently for a wide range of programming languages that are translated to CIL, and fully qualified API calls can be easily extracted directly from CIL code.

2.5. Conclusion

In this chapter, we have explained related systems that influenced the development of *Aris* and the earlier version of *Aris (Aris 1.0).* We discussed in detail the source code matching module (Section 2.2) (Grijincu, 2013) and Source Code Retrieval (Section 2.4) (Pitu, 2013). The source code matching module represents source code files as conceptual graph and mapping two such representations. Their approach on the *Analogical Reasoning* process finds detailed correspondences between the two domains and creates new specifications in the target code. The source code retrieval system retrieved similar implementations by exploring structural and semantic similarities between the query and memory source code artefacts and also combined with the similarity score obtained from the source code matching module. Semantic retrieval

uses the API vector space model to represent source codes and computing similarities between these representations. The similarity score computed based on number of shared API calls in the documents (source code). The structural retrieval computes the similarity between the query and the cases in memory (associated with the source code artefacts) by exploring the structural characteristics (vertex count, edge count, average degree, etc) of the source code derived from *Conceptual Graphs*. Approaches and results of *Aris 1.0* will be evaluated and benchmarked against *Aris 2.0* in chapter (3) and Chapter (4).

3. *Aris 2.0*

3.1 Overview

In this chapter, we explain in detail the problems we have identified in the previous *Aris* system (*Aris 1.0*). We first present our methodology for improving the first version of *Aris*, leading to version two of *Aris.* The subsequent section explains our proposed solution to address problems in the earlier version of *Aris* and also improve the overall system. The newer version of *Aris* will be mentioned as *Aris 2.0.*

3.2 Methodology

The development process of *Aris 2.0* was subject to regular weekly meetings of *Aris* group, each meeting lasted one or 2 hours. The supervisors of these meetings are Dr. Diarmuid P. O'Donoghue and Dr. Rosemary Monahan and the rest attendees are Felicia Halim, Fahrurrozi Rahman, Donny Hurley, and Abgaz Yalemisew. The supervisors become project leader who sets the meeting agenda and research direction. My role in team meetings was shared progress and issues during the development of *Aris 2.0.* While the supervisor and other attendees gave feedback or suggestion regarding my progress, I carefully document the inputs which are important for the next iterative development process.

Based on the input given during project meeting, I evaluate *Aris 1.0* by using experimental methodology that can be divided into exploratory phase and evaluation phase (Amaral, et.al, 2007). In the exploratory phase, I explore the graph matching result based on similarity score, mapped element, and specifications transferred for different level modification e.g. identical, small modifications, medium modifications, and large modifications (Wilkinson, 1994). Exploratory phase of the source code matching module is driven by a few main questions:

- *Is the similarity score represents the level of modification?* Small modifications should not significantly reduce the similarity score. Meanwhile, medium modification may have

greater affect the similarity score, but we have to make sure that it is still in the acceptable range.

- *Is all the elements in the source can be mapped to the element of the target?*
- *Is the generated specification is transferred into correct position? And is the transferred specifications can be verified?*

Problem code known as a *query* in the retrieval phase will return the subset of documents in a database that is considered relevant to the query. Relevant in this context means its specifications can be reused for problem code. Exploratory phase for source code retrieval can be summarized into questions:

- *Is the retrieved documents are relevant to the input query?*
- *How well the retrieved documents from source code artifacts are ranked with respect to a given query?*

After producing a list of questions from exploratory phase, I begin the evaluation phase that attempts to answer these questions. In the evaluation phase, I can identify problems in the *Aris 1.0.* For each problem found, I attempt to relate it with the algorithm or concept used in *Aris 1.0.* I improve the algorithm and reevaluate the result to measure how it improves the system. These improvements and progresses will be presented in the weekly project meeting to make sure the improvement still in line with research direction and also identify issues that may occur. Algorithm developments that able to improve the overall system are finally used in *Aris 2.0.*

3.3 Issues in *Aris 1.0* and its improvement in *Aris 2.0*

In this section, we focus on identify problems in Aris 1.0 and also further discuss how Aris 2.0 address these problems.

3.3.1 Poor Mapping in Loop Construct

We have identified some of the results of the evaluation of *Aris 1.0* that fail to map equivalent loop statement between base and target code. One of the examples of unsuccessful mapping is bounded search method where the base program has a *for* loop and target program has a *while* loop (see Table 3.1). These 2 loops actually can be matched and shared same specification since they also shared the same functionality in each program domain. However, there is no loop mapping in the matching results in Table 3.2. *Aris 1.0* does not find correspondence for *for* loop in the target context, consequently the loop invariant that embedded in *for* loop cannot be transferred.

```
/* base
Monahan, Rosemary. (2014). Bounded Search.
(Version 1.0) [Computer program] Available
at http://www.rise4fun.com/SpecSharp/SvgB
(Accessed 1 March 2014)*/
public static int BS(int[] a, int key)
requires a != null;
ensures 0 <= result &&
 result < a.Length ==> a[result]==key;
ensures result < 0 ==> forall{int n in
          (0: a.Length); a[n] != key};
{
     int n;
     for (n = 0; n < a.Length && a[n]
                        != key; n++)
     invariant n <= a.Length;
     invariant forall{int i in (0:n);
                        a[i] != key};
     {
     }
     if (n == a.Length)
     return -1;
     else return n;
}
```

```
/* target
Monahan, Rosemary. (2014). Bounded Search.
(Version 1.0) [Computer program] Available
at http://www.rise4fun.com/SpecSharp/SvgB
(Accessed 1 March 2014)*/
public static int BS(int[] a, int k)
{
      int n =0;
      while (n < a.Length && a[n] != k)
      {
         n++;
      }
      if (n == a.Length)
         return -1;
      else return n;
}
```

Table 3.1. Example of *Aris* input system. Base program in the left column use *for loop* and Target program in the right column use *while loop*

G1->G2: 0.6818 (15 mapped / 22 total nodes in the source domain) 1.{Parameter} matched with {Parameter} (1) 2.{Variable: a} matched with {Variable: a} (1) 3.{Parameter} matched with {Parameter} (1) 4.{Variable: key} matched with {Variable: k} (0.9) 5.{Variable: n} matched with {Variable: n} (1) 6.{CompareOp: ==} matched with {CompareOp: ==} (1) 7.{Contains} matched with {Contains} (1) 8.{Condition} matched with {Condition} (1) 9.{Block} matched with {Block} (1) 10.{Contains} matched with {Contains} (0.9) 11.{Contains} matched with {Contains} (1) 12.{Contains} matched with {Contains} (1) 13.{String: 0} matched with {String: 0} (1) 14.{Contains} matched with {Contains} (0.9) 15.{Contains} matched with {Contains} (0.8)	G2->G1: 0.4138 (12 mapped /29 total nodes in the source domain). No specs transferred. 1.{Parameter} matched with {Parameter} (1) 2.{Variable: a} matched with {Variable: a} (1) 3.{Parameter} matched with {Parameter} (1) 4.{Variable: k} matched with {Variable: key} (0.9) 5.{Variable: n} matched with {Variable: n} (1) 6.{Contains} matched with {Contains} (0.9) 7.{If} matched with {If} (1) 8.{Condition} matched with {Condition} (1) 9.{Contains} matched with {Contains} (0.8) 10.{Contains} matched with {Contains} (0.8) 11.{Contains} matched with {Contains} (0.8) 12.{Contains} matched with {Contains} (0.8)

Table 3.2. *Aris 1.0 result* where *for* loop and *while* loop is not successfully match (not found in the correspondences above)

Based on our evaluation, the problem of unsuccessful mapping in the loop statement is because the insufficient range of backtracking process when finding valid mapping in conceptual graph. IAM algorithm map concepts from the base domain to target domain by taking each candidate under decreasing order of node rank value (Section 2.2.5.3). If the current acquired mapping evaluated as not valid, IAM algorithm backtracks to find alternative candidate for mapping. The backtracking processes in *Aris 1.0* only visit node with lower *node rank* from current node to find the alternative candidate. However, there are possibility that valid match can be found in node with higher *node rank*.

Aris 2.0 has improved the condition for mapping the loop statement in conceptual graph by allowing backtracking for higher and lower rank node instead of solely visit node with lower node rank as *Aris 1.0* performed. We expect more valid mapping can be found. *Aris 1.0* also set the depth of backtracking using static values, which may not always relevant for various length of input code to the system. *Aris 2.0* set the depth of backtracking is based on the line of implementation codes in target domain. Hence, the depth of backtracking is become more properly relevant to system input.

Other factor that contributes to unsuccessful mapping is deficient criteria in content similarity function when resolves valid mapping. Content similarity function is one of the mapping constraints that are used in the IAM algorithm to establish a valid mapping (Section 2.2.5.6). When attempt to match *for* loop and *while* loop in the Table 3.1, IAM algorithm in *Aris 1.0* computes the content similarity between the loops by calculating string distance of operator token in the first binary expressions found from both loop statement. For example, *for* loop statement of base program is `for (n = 0; n < a.Length && a[n] != key; n++)`, the first binary expressions found is `n = 0`, and the operator token is "=". The target uses while loop: `while (n < a.Length && a[n] != k)`, the first binary expression found is `n < a.Length && a[n] != k`, and the operator token in the *while* loop graph is "&&". *Aris 1.0* compute string distance between these operator token "=" and "&&" to validate match between *for* loop in base domain and *while* loop in target domain.

Operator token in the first binary expression found is not sufficient to represent the content of the loop statement. Thus, *Aris 2.0* match the loop statement in the base domain and target domain by using operator token in loop condition. The loop condition gives stronger representation in semantically and functionally. Loop condition is a well-formed representation which allows more reliable string distance comparison. It also has a functional role which determines how loop body is executed. In Table 3.1, the loop condition of *for* loop in the base domain is `n < a.Length && a[n] != key` and its operator token is "&&". The loop condition of *while* loop is `n < a.Length && a[n] != k` and its operator token is "&&". *Aris 2.0* compute string distance of operator token in loop condition ("&&" and "&&") to validate match between this loop statement.

The backtracking and content similarity function improvement in *Aris 2.0* effectively mapped *for loop* and *while loop*, therefore it able to transfer the loop invariant where the solution is successfully verified (see Table 3.3).

`// unverified solution of Aris 1.0`
`public static int BS(int[] a, int k)`
`requires a != null;`
`ensures 0 <= result && result < a.Length`
`==> a[result]==k;`
	`// verified solution of Aris 2.0`
	`public static int BS(int[] a, int k)`
	`requires a != null;`
	`ensures 0 <= result && result < a.Length`
	`==> a[result]==k;`

```
ensures result < 0 ==> forall{int n in (0:
a.Length); a[n] != k};
{

      int n =0;
      while (n < a.Length && a[n] != k)
      {
         n++;
      }
      if (n == a.Length)
         return -1;

      else return n;
}
```

```
ensures result < 0 ==> forall{int n in (0:
                    a.Length); a[n] != k};
{
      int n =0;
      while (n < a.Length && a[n] != k)
     invariant n <= a.Length;
     invariant forall{int i in (0:n);
                          a[i] != k};
      {
         n++;
      }
      if (n == a.Length)
         return -1;
      else return n;
}
```

Table 3.3. *Aris 1.0* solution in the left column cannot be verified because loop invariant is not transferred, while *Aris 2.0* solution in the right column can be verified because loop invariant is successfully transferred.

3.3.2 False Variable mapping

Another frequent problem we found is false variable mapping. For example, in the program below, the target is modified from the base code implementation. The modification in target code involves use variables i in the extraneous statement without change the functionality (yellow highlighted). The expected result between the mappings should be variable i in the source with variable i in the target and variable $\_S$ in the source with variable s in the target. However, *Aris 1.0* has mismatch mapping results where variable i at the source with variable s at the target and variable $\_S$ at the source with variable i at the target.

```
//base
/*
O'Donoghue, Diarmuid. (2013). Aris
Online. (Version 1.0) [Computer program]
Available at http://ec2-54-213-12-95.us-
west-2.compute.amazonaws.com/ (Accessed 1
March 2014)
*/
public int CountEven(int[] array)
ensures result == count{int i in (0:
    array.Length); ((array[i] % 2)== 0)};
requires array != null;
{
    int i = 0;
    int _S = 0;

    while (i < array.Length)
    invariant _S == count{int j in (0:
            i); ((array[j] % 2)== 0)};
    invariant i <= array.Length;
    {
        i++;
        i--;
        if ((array[i] % 2) == 0)
        {
            _S += 1;
        }
        i++;
    }
    return _S;
}
```

```
//target
/*
Monahan, Rosemary. (2014). Count Even.
(Version 1.0) [Computer program] Available
at http://rise4fun.com/specsharp/DRkO
(Accessed 1 March 2014)
*/
public int CountEven(int[] a)
{
       int s = 0;
       for (int i = 0; i < a.Length; i++)
       {
              if (a[i] % 2 == 0)
              {
                     s++;
              }
       }
       return s;
}
```

Table 3.4. Example of base and target methods received as input by *Aris 1.0* which produce false variable mapping.

In this section we focus on reducing the occurrence of false variable mapping. *Aris* transfer specification using CWSG - *Copy with substitution and generation* algorithm, the algorithm transfer additional specifications of the base code and add it to the target code by substituting variable in base code items with their mapped equivalents. For example, in the Table 3.4, variable *array* in the base is mapped into variable *a* in the target context. Therefore, variable *array* in *precondition* `requires array != null`, will be replaced and produce `requires a != null` in the target context. The false mapping of variable will form new specifications that cannot be verified.

The problem of frequent occurrence of mismatch variable mapping is because *Aris 1.0* only consider one way graph mapping when generating and transfer specifications into the target. One way graph mapping means it relies on the mapping result from base (which has the

specifications) to the target domain. *Aris 1.0* does not consider mapping result from other way, target to base domains. Mapping result from target to base domains and base to target domains can be different if the base and target domains are not identical, as the process depend various factors such as node order and number of seed groups in each domain, etc (Grijincu, 2013). The default mapping result from base to target in *Aris 1.0* cannot be guaranteed produce more valid mapping than mapping result from target to base domain. Therefore, *Aris 2.0* consider both ways to generate and transfer specifications into target domain.

Aris 2.0 will calculate the total similarity score for variable mapping for both ways graph matching. The CWSG *(Copy with Substitution and Generation)* algorithm will use the graph matching that has the higher total similarity score for variable mapping. Intuitively, graph matching with a higher total similarity score for variable mapping means the graph matching more successful to generate variable mapping. For example, base and target code in Table 3.4 produce different mapping results in both ways (see Table 3.5). Mapping result from base to target was not successful to match 1 variable (variable *_S*) and mapping result from target to base had a higher similarity score for variable mapping as it able to match all the variables in the source domain. *Aris 1.0* always uses mapping result from base to target, although the quality of variable mapping is lower than the other way, while *Aris 2.0* will generate and transfer specification based on the mapping result that gives better quality of variable mapping. In this case, we choose mapping result from target to base as it gives higher similarity score for variable mapping.

Mapping result from base to target	**Mapping result from target to base**
Variable: array} matched with{Variable: a} **(1)** {Variable: i} matched with {Variable: s} **(1)**	{Variable: a}matched with{Variable: array} **(1)** {Variable: s} matched with {Variable: _S} **(1)** {Variable: i} matched with {Variable: i} **(0.92)**
Total similarity score variable mapping = **2.0**	Total similarity score variable mapping = **2.92**

Table 3.5. Mapping result for variable mapping in both ways

3.3.3 Translate Statement in Conceptual Graph

Based on our evaluation, there are certain lines of codes or programming features that are not translated into the graph such as conditional operator[13], `else if` statement, and iterate segment of *for* loop in *Aris 1.0*. In order to improve the accuracy and completeness of mapping process, we translate the conditional operator, `else if` statement, and iterate segment of *for* loop in *Aris 2.0.*

Conditional operator which is one of the programming features in C# is not translated in *Aris 1.0.* For instance, a method with conditional statement in Table 3.6 only generates very few nodes in the conceptual graph as can be seen on the left hand side of (table 3.7). In *Aris 2.0,* this programming feature is translated and produce more nodes compared to nodes produced in *Aris 1.0* (see on the right hand side of table 3.7).

```
/*
Microsoft Developer Network. (2013).
Conditional Operator. (Version 1.0) [Computer
program] Available at
http://msdn.microsoft.com/en-
us/library/ty67wk28.aspx (Accessed 1 March
2014)
*/
static double sinc(double x)
{
 return x != 0.0 ? Math.Sin(x) / x : 1.0;
}
```

Table 3.6. Example of conditional operator statement in the program.

Concept node created in *Aris 1.0*	Concept node created in *Aris 2.0*
1.{*Parameter*} 2.{*Variable*: *x*} 3.{*Block*} 4.{*Contains*}	1.{Parameter} matched with {Parameter} (1) 2.{Variable: x} matched with {Variable: x} (1) 3.{MathOp: /} matched with {MathOp: /} (1) 4.{Contains} matched with {Contains} (1) 5.{Contains} matched with {Contains} (1) 6.{MethodCall: Math.Sin()} matched with {MethodCall: Math.Sin()} (1) 7.{Parameter} matched with {Parameter} (1) 8.{MathOp: !=} matched with {MathOp: !=} (1) 9.{Contains} matched with {Contains} (1) 10.{Contains} matched with {Contains} (1) 11.{Conditional} matched with {Conditional} (1) 12.{Contains} matched with {Contains} (1)

[13] Conditional Operator. http://msdn.microsoft.com/en-us/library/ty67wk28.aspx

	13.{Contains} matched with {Contains} (1) 14.{Contains} matched with {Contains} (1) 15.{Returns} matched with {Returns} (1) 16.{Block} matched with {Block} (1) 17.{Null:*} matched with {Null:*} (1) 18.{Null:*} matched with {Null:*} (1) 19.{String: 0.0} matched with {String: 0.0} (1) 20.{Contains} matched with {Contains} (1) 21.{String: 1.0} matched with {String: 1.0} (1)

Table 3.7. Comparison of concept node created in *Aris 1.0* and *Aris 2.0* for program code that contains conditional operator.

Every *for* loop statement defines initialization, condition, and iteration section (Microsoft, 2013). Part of *for* loop statement that was not translated in the conceptual graph of *Aris 1.0* is the iteration segment (Rahman, 2014). This missing representation is already addressed in the *Aris 2.0.* We also identified `else if` statement was not translated in *Aris 1.0*, therefore the new version of *Aris* have addressed this problem. Our solution is also able to handle multiple `else if` statement.

3.3.4 Lexical Similarity Problem

Aris makes flexible comparison of content similarity between 2 mapped code-graphs. This will include assessing similarities between two variables. If both variables that used identically with different variable type, for example one *int* and one *double*. It will be considered as match variable. However, based on our evaluation in previous *Aris* version, there are some pairs of numeric type that possibly can be matched together, but not included. In *Aris 1.0*, it collects element of one class equivalents that can be considered as convertible, assignable, type such as *int* and *double*. We improve the condition by supporting additional data type in this class, such as numeric type *float* and *decimal*.

Aris 1.0 does not handle variable mapping with *bool* data type. When content similarity function checks between two variables with *bool* data type, it immediately rejected. *Aris* 2.0 address the problem of matching *bool* data type by ensuring the variable with *bool* data type is matched with a variable with *bool* data type.

3.4 Added Features in *Aris 2.0*

In this section, we focus on identifying limitation of available features in Aris 1.0 and we will explain new feature that added in Aris 2.0 to improve the overall system performance.

3.4.1 Specification score metric in the retrieval module

The retrieval module in *Aris* analyzes different properties of documents in the software artefacts relative to the query (see Section 2.4). Each document will be analyzed based on its semantic similarity, structural similarity, and the similarity score from source code matching module in *Aris* (Grijincu, 2013). The semantic similarity score and structural similarity score are combined in the combined similarity score by assigning a particular weight for each score. The overall retrieval score is obtained from the combined similarity score and the similarity score from source code matching module (once again with assigning particular weight for each score).

Retrieval module of *Aris* 1.0 rank the final set of documents to the query according overall retrieval score. Top rank retrieved documents represent a subset of the corpus that has most similar implementations for specification transfer. However, if we look to the purpose of *Aris* is to reuse specifications. But the overall retrieval score does not have components that assess the specifications qualities in each document. Therefore, the documents are ranked only based on the implementation similarity with the query without considering whether a document possesses quality for transferring specifications.

In order to ensure that the top ranked documents possess good quality specifications for transferring specifications, *Aris 2.0* will add the new metric that measure the completeness of specification in each document related to given query. The new metric examines the existence of *preconditions, post-conditions,* loop *invariant* (if applicable), and the relationship between loop *invariants* and *postconditions* in the source code document. Subset of documents that related to the query and have a high degree of specifications completeness should appear in the top ranked of retrieval. Further, we seek ranking precision for the top retrieved documents,

because their order in the results is crucial as these top documents represent the most interesting subset of the corpus, to a possible interactive user of *Aris.*

Aris 2.0 measures the completeness of the specifications in each document by checking the existence of *requires, ensures,* loop *invariants* and also takes into account the relationship between *loop invariant* and *postconditions.* The relationship of *postcondition* and *loop invariant* is important because based on (C. A. Furia and B. Meyer, 2010), loop invariant is a weaken form of postconditions and can be created by modifying postconditions through constant replacement, term dropping, and substitute the variable. Therefore, we measure the relationship between *postcondition* and loop *invariant* based on their string distance between the statements. The relationship of *postcondition* and loop *invariant* is strong if there's minimum edits necessary to transform *postcondition* into loop *invariant*. The existence of *assert* and *assume* currently does not take part in the specification score calculation because it is difficult to predict in which location *assert* and *assume* statements are actually needed.

Aris 2.0 counts the expected and actual number of specifications in each retrieved document. Each document (in this case a method) should have *precondition* and *postcondition* and if the method contains a loop, every loop expected to have *upper bound invariant, lower bound invariant,* and *quantifier invariant.* The loop quantifier invariant that has a relationship with the method's post-condition will have a higher specification score. We will assign 1 for each specification/relationship exists and 0 if it does not exist in the source code. The expected number of specifications is the maximum cumulative value that is expected to appear in the source code. While, actual number of specification score is the actual cumulative value of specifications that appear in the source code. The new metric that will be added in the overall similarity score is obtained from divide the actual number of specification over expected number of specification and multiply the result with a similarity score from the source code matching module.

The formal formula for *specification score* $comp_{spec}(d, q)$ is:

$$\left(\frac{ActualNumberofSpecification}{ExpectedNumberOfSpecification}\right) * \ sim_{CGmatch}(d, q)$$

Aris 2.0 introduce the new specification score metric $comp_{spec}(d, q)$, therefore there is a need to alter the formula for $sim_{overall}(d, q)$, the new formula is:

$$sim_{overall}(d, q) =$$

$$w_{comb}\ x\ sim_{comb}(d, q) + w_{CGmatch}\ x\ sim_{CGmatch}(d, q) + w_{spec}\ x\ comp_{spec}\ (d, q)$$

We also have to alter the weight assigned to the score components to fit with the new formula where w_{comb}, $w_{CGmatch}$, are weights for the combined retrieval score and conceptual graph matching score and w_{spec} are the weights for the specification completeness score, such that $w_{comb} + w_{CGmatch} + w_{spec} = 1$. The weights were chosen based on the experimental results of *Aris 2.0* implementation: w_{comb} = 0.4d, $w_{CGmatch}$ = 0.4d, and $w_{spec} = 0.2$d.

3.4.2 Adapt Specification transfer for Increment/Decrement *for* Loop

Transfer specification between identical implementation can generate a target method with added verified specification without any needs to adjust the newly generated specification. However in most cases source and the target method does not have identical implementation. Therefore there's need to adjust the newly transferred specification in the target code to ensure the successful verification.

Although there are effort to adapt the generated specification in the target context by substitute variables for their mapped equivalents in the base code, however the newly generated specification may not immediately valid within target context. Invalid specification will eventually be rejected by the verification tool. Thus, in some cases, software developer is required to modify generated specifications for formal verification. The modification may involve repositioning variables, change the operator, etc.

Interesting particular case is the transfer specification between methods with increment *for loop* and another related method with decrement for loop and vice versa. This case may share exact *precondition* and *postcondition.* However, boundary of loop *invariant* will be semantically

different as variable for iteration start and end in different value. For example, increment loops from $0..x$ and decrement loop from $x..0$.

The loop invariant for *for* loop can be classified to boundary loop *invariant* and quantifier loop *invariant*. The second type of loop invariant often needs adjustment when the newly generated transfer specification is between increment *for* loop and decrement *for* loop. For example, if we have a summation method in the source domain with increment *for* loop and another summation method in the target domain with decrement *for* loop. *Aris 2.0* will automatically adjust the generated quantifier loop invariant based on target implementation as can be seen in the Table 3.8. The transformation of loop *invariant* of *for loop* is necessary if the iteration section in one domain uses increment, while in other domain use decrement expression. *Aris 2.0* will adapt the specification by swapping position of the variable in loop quantifier and also change its boundary. In Table 3.8, Increment *for* Loop in the source has loop invariant that focuses on what has been summed so far `s == sum{int i in (0: n); a[i]}`.Therefore, when transfer specifications to the target domain , *Aris 2.0* adjust the quantifier loop *invariant* (see the highlighted part in the Table 3.8) that produce verified specification.

```
//base
/*
Monahan, Rosemary. (2010). The Spec#
Programming System. (Version 1.0)
[Computer program] Available at
http://www.loria.fr/~mery/malg/SpecSharpN
ancy2013.pdf (Accessed 1 March 2014)
*/
public static int Sum(int[]! a)
ensures result == sum{int i in
                    (0:a.Length); a[i]};
{
    int s = 0;
    for (int n = 0; n < a.Length; n++)
      invariant 0 <= n;
      invariant n <= a.Length;
      invariant s == sum{int i in (0: n);
                                    a[i]};
    {
      s += a[n];
    }
    return s;
}
```

```
//target
/*
Monahan, Rosemary. (2010). The Spec#
Programming System. (Version 1.0) [Computer
program] Available at
http://www.loria.fr/~mery/malg/SpecSharpNanc
y2013.pdf (Accessed 1 March 2014)
*/
public static int Sum2(int[]! a)
ensures result == sum{int i in
                    (0:a.Length); a[i]};
{
  int s = 0;
  for (int n = a.Length;0 <= --n; )
  invariant 0 <= n;
  invariant n <= a.Length;
  invariant s == sum{int i in (n:a.Length);
                                    a[i]};
  {
      s += a[n];
  }
  return s;
}
```

Table 3.8. Program code example where the right column is the example of transfer and adapt specification of *Aris 2.0.* The solution can be formally verified. Shaded code indicate the transformation of generated specification

3.4.3 *Assert* and *Assume* Transfer Specification

Aris 2.0 enable assert and assume transfer specification which are not being covered by *Aris 1.0*. Assert and assume are interesting because they are part of the proof strategy in the verification. Asserting something in the proof asks the verifier to validate the thing we have asserted is true and if it allows the theorem prover to use assertion in its proof. It also gives the theorem prover a "hint" regarding which proof strategy to take. Assume statement also almost does the similar concept, except the verifier believes everything we have assume, therefore it does not check it before using it.

We find the corresponding location of each of the *assert/assume* statement in target domain based on its location in the source domain. The initial design of *assert* and *assume* specification is $KeyValuePair$ between assert/ assume statement and its location. The location is the last code implementation before the assert/assume statement. However after further analyze the case studies, we conclude $KeyValuePair$ is not suitable to be applied in this solution because $KeyValuePair$ need a key, and neither the assert/assume statement nor the location can be the key. It can be seen from one of the example Table 3.9. Assertion cannot be the key as *assert* statement `assert q == i * x;` can be duplicated and located in different location (after while loop and after `i= i+1` statement). Moreover, the location cannot be the key as well where after while loop there are two consecutive assertions (`assert q == i * x;` and `assert q + x == (i * x) + x`). These two assertions corresponds to the similar location `while (i <y).`

```
/*
Monahan, Rosemary. (2010). The Spec# Programming System. (Version
1.0) [Computer program] Available at
http://www.loria.fr/~mery/malg/SpecSharpNancy2013.pdf (Accessed 1
March 2014)
*/

static int multiply(int x, int y)
requires 0 <= y;
requires 0 <= x;
```

```
ensures result == y * x;
{
        int q = 0;
        int i =0;
        while(i <y )
         invariant 0<= i;
         invariant i<= y;
         invariant q == (i * x);
         {
             assert  q == i * x;
             assert  q + x == (i * x) + x;
            q = q + x;
            assert  q == (i * x) + x;
            assume (i*x) + x == (i+1) * x;
            assert  q == (i+1) * x;
            i= i+1;
             assert  q == i * x;
         }
      return q;
}
```

Table 3.9. Example of program code with asserts and assumes statement

Therefore, the assert statement and the solution are stored in $List < KeyValuePair < value, value >>$ which allow 2 identical assert statement in different location of code implementation and also allow a location corresponds to more than one assertion.

The successful transfer of *assert/assume* statement is rely on valid mapping found in graph matching. The more valid mapping found between base and target domain, the more accurate the transfer location for *assert/assume*. However, In *Aris 1.0*, we found mapping for statement with *postfix unary expression* (`s--` or `s++`) is not successful. There is bug in translating *postfix unary expression* syntax, therefore every statement with that type are not represented in the code. *Aris 2.0* fix the bug of translating *postfix unary expression syntax* statement into conceptual graph. We decide the location of assert/assume in the target domain by taking from the corresponding of location in the source domain. If there is one-to-many mapping, we use string distance measure (*Damerau-levenshtein*) to decide the most similar statement with the current target.

```
/* base
Monahan, Rosemary. (2014).[Computer
program] Available at
http://www.rise4fun.com/SpecSharp/kh6B
(Accessed 5 June 2014)
*/
public static int Summation(int k)
requires 0 <= k;
ensures result==sum{int i in (0:k); i};
{
    int s = 0;
    int n=0;
    while (n < k)
    invariant n <= k;
    invariant s == sum{int i in (0:n); i};
    {
       int variant = k- n;// record value
of variant function
       s = s + n;
       n++;
       assert 0 <= variant; // check
boundedness of variant function
       assert (k- n ) < variant; // check
that variant has decreased
    }

   return s;
 }
```

```
/* target
Monahan, Rosemary. (2014).[Computer
program] Available at
http://www.rise4fun.com/SpecSharp/kh6B
(Accessed 5 June 2014)
*/
public static int Sum(int x)
{
    int add = 0; int k = 0;
    int irrelevantVariable = 0;
    Console.WriteLine(irrelevantVariable);
    while (k < x)
    {
        int variant = x - k;// record value
of variant function
        add += k;
        k++;
    }

    return (int)add;
}
```

Table 3.10. Base and Target code for assertion transfer example

```
public static int Sum(int x)
requires 0 <= x;
ensures result==sum{int i in (0:x); i};
{
    int add = 0; int k = 0;
    int irrelevantVariable = 0;
    Console.WriteLine(irrelevantVariable);
    while (k < x)
    invariant k <= x;
    invariant add == sum{int i in (0:k); i};
    {
        int variant = x - k;// record value of
variant function
        add += k;
        k++;
       assert 0 <= variant; // check boundedness
of variant function
       assert (x- k ) < variant; // check that
variant has decreased

    }
    return (int)add;
}
```

Table 3.11. Verified solution generated from Table 3.10. Shaded statement is generated assertion.

In the Table 3.10, two assertions are located after *n++* statement in the base code. Graph matching result Variable: $n \leftrightarrow k$, therefore the algorithm will transfer the assertion statement after *k++* statement in the target code as can be seen in the shaded statement (Table 3.11). However, *Aris* still has difficulty to transfer the specifications if the statement where specifications related to, is being semantically modified. For example, if the target code changes $k++$ statement into $k = k - 1$ and $k = k + 2$, *Aris* will unable to locate the assertion in the correct position and transfer specification as commented specification in the beginning of the method.

3.5 Conclusion

In this section, we have presented the experimental methodology to evaluate and benchmark *Aris* system. We have found and discussed some issues in *Aris 1.0* and also explain in detail how *Aris 2.0* have addressed such problems. Problems that we have addressed in the new version of the system are poor mapping in the loop constructs, frequent occurrences of false variable mapping, statement translation in the conceptual graph, and lexical similarity problem. We also explained the new features in *Aris 2.0* such as new specification score metric in the retrieval module, adapt transferred specification, and *assert/assume* specification transferred.

The graph matching complexity *Aris 2.0* is polynomial (similar to *Aris 1.0*). The process of matching methods in decreasing order of their NR values means compare at most *n* methods from the source domain and inside method mapping process are 2 sorting operations plus at most $m\ x\ match_{depth}$ (m denotes number of nodes in a subgraph representing a method) comparisons. The algorithm uses threshold parameter $match_{depth}$ to limit the number of concepts in target for comparison. If the sorting function performs $O(m\ log\ m)$, thus the worst case of the algorithm performs $O(n\ x\ mlogm)$ (Grijincu, 2013). In *Aris 2.0,* we also add new metric *specification score* in retrieval module. The complexity of specification score is $O(n)$ (*n* denotes the length target code).

4. Evaluation

In this chapter, we give experimental results for testing $Aris$ 2.0 and compare the performance with *Aris 1.0*. In particular, we evaluate the similarity score, the abilities to transfer specification between identical and functionally similar source code and also evaluate the performance of $Aris$ specifically at the source code retrieval phase. Evaluation is divided into two main sections, the first section will address a recognition problem, and this will focus on the retrieval and mapping phase measuring the capability of *Aris* to identify the similar source code to some given problem. The degree of similarity between pairs of source code artefacts has been carefully controlled, forming four different categories that represent different degree of similarity with original unmodified implementation (Wilkinson, 1994). These results focus on the ability of *Aris* to identify similarities between identical source code as well as code artefacts with gradually reduced similarity.

The second section of the evaluation will address a software generation problem focusing on the ability of *Aris* to generate new specification to the selected method from the open source repositories and also to identical and modified sources specified in (Wilkinson, 1994). An SMT theorem prover is used to verify and validate the generated specifications. Here, the focus is on the ability of *Aris* to produce a verified specification for any given piece of source code.

4.1 Document corpus

In order to evaluate *Aris* performance at retrieval, mapping and transferring specification between the formally verified source code and unspecified target, we collect the corpus of source code files that contains verified methods ($Aris$ matches only using method information). Each method has appropriate *ensures, requires, modifies,* loop *invariants, and assert/assume* annotation depends on each method's need.

Both *Aris 1.0 and Aris 2.0* are built on .NET framework and analyses C# files (which can be formally verified using Spec# language). Therefore, we collected verified source code files

from Spec# test suite publicly available on the open-source hosting platform CodePlex[14]. We also use formally verified sources that using Spec# language from textbook examples[15] and class assignments. 52 methods were collected from these varieties of sources, this collection contains with a total of 54 preconditions statements, 67 *postconditions* statements, 9 *modifies* statement, 110 *loop invariants* statements, 27 *assert/assume* statements, some of which may represent the conjunct of two or more individual specifications. There are 273 Spec# specifications statements as well as associated source code in the database.

We conducted the evaluation by selecting some methods out of 52 methods that have Spec# statements from the database. These selected methods were used to generate four distinct test sets (known as TS1 –TS4) where each test suites represents the increasing level of difference from the original unmodified code. TS1 –TS4 do not contain any Spec# statements which allow us to evaluate *Aris* in generating the specifications.

Test Suite 1 (TS1) Identical Implementations – Isomorphic

The first set of tests was derived directly from original methods from the database, but with their Spec# statements removed. We keep the source code remain unchanged. These problem methods were therefore identical to the original implementations. This represented the first category of challenge to *Aris.*

Test Suite 2 (TS2) Small Modifications – Near Isomorphic

The second set of test was derived by transforming a fresh copy of the source code in TS1 by applying different level of modifications. The changes in this test suite will not affect the functionality of the code, thus the specifications remain usable by TS2.

1. Lexical changes: modify identifier names, parameter, or method name (e.g. `int result` may become `int output`)

[14] Codeplex. http://www.codeplex.com/
[15] Textbooks examples.http://rosemarymonahan.com/specsharp/

2. Type changes: change data type (e.g `double` may become `float`, `int` may become `long`)

3. Changing code constructs: change loop constructs (e.g `for` with `while` and vice versa), changing the order of parameters in method declarations, change order of operands in expression (e.g `a < b` may become `b > a`)

4. Introduces or removing white space or comments

We note that many code obfuscators also change identifiers, so this category should partly challenge *Aris* to retrieve obfuscated code.

Test Suite 3 (TS3) Medium Modifications – Homomorphic

The third test suite applied the following modifications from 5-7 to the methods produced for Test Suite 2. These modifications, once again, do not affect the functionality of the code and all specifications remain unaffected.

5. Change the order of statements: reversing a conditional statement, changing the order of statements, add the redundant conditional statement

6. Introduce dummy field or redundant statements such as declarations or initializations

7. Adding/ remove certain statements that do not change the functionality

TS3 represent the category involving the most serious changes to the original implementation – but which can still use the same specifications as the effective functionality has not been changed.

Test Suite 4 (TS4) Large Modifications – Dissimilar

The modifications in this category alter the functionality of the method. For example, a program that sums the element of an array, was might become modified into storing a sorted array.

Here, we give example of original code in the database in the Table 4.1 and its transformation for every category of test suite in the Table 4.2.

```
public int SumTotal(int[] array)
  ensures result == sum{int i in (0:
                  array.Length); array[i]};
{
    int num = 0;
    for (int i = 0; i < array.Length; i++)
    invariant 0 <= i && i <= array.Length;
    invariant num == sum{int k
                        in(0:i);array[k]};
    {
        num += array[i];
    }
    return num;
}
```

Table 4.1. Example of original methods from database

`public int SumTotal(int[] array)`
`{`
`  int num = 0;`
`  for (int i = 0; i < array.Length;`
`  i++)`
`  {`
`  num += array[i];`
`  }`
`  return num;`
`}`
	`public int SumTotal(int[] arr)`
	`{`
	`  int j =0;`
	`  int sum = 0;`
	`  while(j < arr.Length)`
	`  {`
	`  sum += arr[j];`
	`  j++;`
	`  }`
	`  return sum;`
	`}`

Test suite 1	**Test suite 2**

`public int SumTotal(int[] arr)`
`{`
`  int j =0;`
`  int sum = 0;`
`  int irrelevantVariable = 0;`
 	
`Console.WriteLine(irrelevantVariable);`
`  while(j < arr.Length)`
`  {`
`  sum += arr[j];`
`  j = j - 1;`
`  j = j + 2;`
 	
`  }`
`  return sum;`
`}`
	`public void sortArray(int[] a)`
	`{`
	`  int ka;`
	`  int ta;`
	
	`  if (a.Length > 0){`
	`  ka=1;`
	`  while(ka < a.Length){`
	`  for(ta = ka; ta >0 && a[ta-`
	`  1]>a[ta]; ta--){`
	`  int temps;`
	`  temps = a[ta];`
	`  a[ta] = a[ta-1];`
	`  a[ta-1] = temps;`
	`  }`
	`  ka++;`
	`  }`
	`  }`
	`}`

Test suite 3	**Test Suite 4**

Table 4.2. Examples of source code modification in each test suite.

In order to evaluate the source code retrieval module of *Aris*, we augment the collected verified source code with a large corpus of real world projects from open source programs e.g. CodePlex, NuGet, and SourceForge. This consists of 2309 software applications with 2,315,022 methods. The large corpus downloaded from open-source repositories publicly available. Results focus on main metrics: retrieval and mapping metrics and quantities of information identified. Table 4.3 is the list of entire contacts of the full test suite:

Category	**Number of Methods**
TS1	12
TS2	12
TS3	12
TS4	12
Open source	2,315,022

Table 4.3. List of entire contacts of full test set

4.2 *Aris* result on Mapping and Retrieval (based on Wilkinson 1994)

In this section, we compare the performance of *Aris 2.0* against *Aris 1.0* in mapping phase of the source code matching module and retrieval phase using the problems produced for TS1-TS4. Throughout this section, we use the entire corpus of 2,315,074 methods that contains of small set of verified methods and the rest 2,315, 022 of unverified methods.

Firstly, we compare the performance of *Aris 2.0* against *Aris 1.0* in mapping phase using small set of verified methods that augmented with over 2 millions unverified methods. Detailed results mapping phase for *Aris 1.0* and *Aris 2.0* can be seen in table 4.4. For TS1, both versions produce similar score where mapping scores reach on or are very near the maximum score of 1.0 (blue highlighted in Table 4.4). Thus, we can conclude that the “improvements” discussed in earlier chapters did not significantly change these (already excellent) results.

TS2 involved the next collection of lexical changes to the source code in TS1. TS2 produced the next best set of results in terms of the mapping score. In TS2 category, *Aris* 2.0 have generally higher mapping score, with an average of 0.78 compared to 0.73 for *Aris 1.0* (yellow highlighted in Table 4.4). The increase of mapping scores for *Aris 2.0* also occurs for TS3, from

0.65 in *Aris 1.0* into 0.72 in *Aris 2.0* (orange highlighted in Table 4.4). The increment of mapping scores in *Aris 2.0* for these modified codes in TS2 and TS3 means *Aris 2.0* on average able to produce more successful mapping between similar source code files.

TS4 which contains functionally different to the source code resulted in the weakest mapping score for both *Aris* version, both averaging at 0.36 (green highlighted in Table 4.4). For the most similar categories (TS1), both *Aris 1.0* and *Aris 2.0* show remarkable consistency across their mapping scores. TS2, TS3, and TS4 in both systems showed greater diversity among their mapping results as their increase numbers of difference resulted in difference in the various matching process (see column StDev Mapping Score in Table 4.4). Finally, in *Aris 2.0* we can point out there appears to be the natural threshold occurring around 0.7 for mapping score since TS1-TS3 are above the threshold while TS4 is below this threshold.

Test Suite	Mapping result *Aris 1.0*			Mapping result *Aris 2.0*		
	Average Mapping Score	StDev Mapping Score	#Mapped sources	Average Mapping Score	StDev Mapping Score	#Mapped sources
TS1 - Identical	1.00	0.00	65	1.0	0.00	168
TS2 - Near Isomorphic	0.73	0.05	75	0.78	0.04	156
TS3 - Homomorphic	0.65	0.05	86	0.72	0.05	193
TS4 - Dissimilar	0.36	0.08	78	0.36	0.03	82

Table 4.4. Performance of mapping phase in *Aris 2.0.*

Next, we compare the performance of *Aris 2.0* against *Aris 1.0* in the retrieval phase (once again using 2,315,074 methods that contain a small set of verified methods and 2,315,022 of unverified methods). Retrieval module of *Aris* analyzed structural properties of the conceptual graph representation which enable ranking the relevant source code artefacts with respect to the query (Section 2.4). *Aris 2.0* have done improvements to translate more programming features and fixing bugs that found in the conceptual graph creation process. This improvement affected the number of retrieved sources (see columns #retrieved sources in Table 4.5) that eventually will give better precision (see Table 4.6).

Test Suite	Retrieval result *Aris 1.0*			Retrieval result *Aris 2.0*		
	Average Retrieval Score	**StDev Retrieval Score**	**#Retrieved sources**	**Average Retrieval Score**	**StDev Retrieval Score**	**#Retrieved sources**
TS1 - Identical	0.99	0.00	480,418	0.99	0.00	251,386
TS2 - Near Isomorphic	0.97	0.02	479,082	0.97	0.02	385,381
TS3 - Homomorphic	0.93	0.05	150,395	0.93	0.04	127,211
TS4 - Dissimilar	0.21	0.23	60,041	0.14	0.18	103,095

Table 4.5. Performance of retrieval phase of *Aris* 1.0 and *Aris 2.0.*

Table 4.6 summarizes the precision results for *Aris 1.0* and *Aris 2.0,* as we can see *Aris 2.0* consistently achieve better precision results for similar implementation (TS1-TS3) compare to precision in *Aris 1.0*. The precision increase in *Aris 2.0* is also because we use a new metric of specification score when computing the overall retrieval score. The specification score metric will greatly reduce the value attribute to documents without formal specification. Thus, ensure the top ranked retrieved documents are documents that are relevant for reusing formal specification. As expected, the dissimilar source code (category 4) precision is zero for both *Aris 1.0* and *Aris 2.0.*

Test Suite	Total Retrieved		Precision	
	Aris 1.0	***Aris 2.0***	***Aris 1.0***	***Aris 2.0***
TS1 – Identical	595	20	0.00389	0.40
TS2 - Near Isomorphic	665	11	0.0015	0.15
TS3 - Homomorphic	634	6	0.00161	0.31
TS4 - Dissimilar	508	2	0	0

Table 4.6. Precision result for *Aris 1.0* and *Aris 2.0*

Table 4.7 and table 4.8 gives ranking result for TS1-TS4 categories in *Aris 1.0 and Aris 2.0*. The result shows that for the isomorphic problems in TS1, both systems can return the document as the first rank. As the modification becomes increase in TS2 and TS3, the rank gradually decreases for both versions. However, *Aris 2.0* can rank document better for TS2 where the document on average at 2nd rank, while in *Aris 1.0* the rank dramatically drop into

rank 28th. Ranking result for TS3 is also higher in *Aris 2.0* with on average rank 16th, and fall into rank 86th in *Aris 1.0.* As expected, TS4 that represents dissimilar functionality of the code has the worst rank among all categories, with on average rank 20183rd in *Aris 1.0* and 25000th in *Aris 2.0.*

Method	Avg Retrieval Score	#Retrieved sources	Avg Mapping Score	#Mapped sources	Average Rank	Overall retrieval score
TS1 - Identical	0.99	480,418	1.00	65	1	0.99
TS2 - Near Isomorphic	0.97	479,082	0.73	75	28	0.87
TS3 - Homomorphic	0.93	150,395	0.646	86	86	0.81
TS4 - Dissimilar	0.21	60,041	0.359	78	20,183	0.27

Table 4.7. The rank given by *Aris* to the desired items by presentation of a problem from each category for *Aris 1.0*

Method	Avg Retrieval Score	#Retrieved sources	Avg Mapping Score	#Mapped sources	Average Rank	Overall retrieval score
TS1 - Identical	0.99	251,386	1.0	168	1	0.90
TS2 - Near Isomorphic	0.97	385,381	0.78	156	2	0.746
TS3 - Homomorphic	0.93	127,211	0.72	193	16	0.72
TS4 - Dissimilar	0.14	103,095	0.36	82	25,000	0.20

Table 4.8. The rank given by *Aris* to the desired items by presentation of a problem from each category for *Aris 2.0*

4.3 *Aris* Result on Creating Specifications

To test its usefulness of generating new Spec# code, we explore its ability to generate new Spec# statements for the Open Source repository described above – containing over 2 million methods and over 74 million lines of code. We select methods from the Open Source Repository (methods without specification) and we evaluate how *Aris* generate new Spec#

specifications. Table 4.9 details the number of newly generated specifications. The columns #pre, #post, and #inv list the number of *requires*, *ensures*, and loop *invariant* statements that were generated by *Aris* and that were subsequently verified by an SMT theorem prover – unverified specifications are not listed. For example, a transferred specification from the method Factorial(int n), if the Spec# output does not contain any error about the postcondition, we count the generated postcondition as verified. The 20 specified methods resulted in 82 new and verified specifications being generated by *Aris 1.0* and 103 new and verified specifications being generated by *Aris 2.0.* In the Table 4.9, we also can see that the same specification is suggested by *Aris 1.0* and *Aris 2.0* more than 1 time. Therefore, we would have more confidence in the relevance of suggested specifications than if it was only suggested once. It should be noted that some of the specified methods did not include some of the pre, post, or invariant and thus could not generate specifications of the corresponding categories in the open source collection.

***Aris 1.0* Results**		**#retrieval**	**#mapped**	**verified**			
No	**Method name**			**#pre**	**#post**	**#inv**	**#total**
1	Add(int x, int y)	977220	150			1	1
2	AtLeastSquare(int[] inp)	270857	78	2		2	4
3	BoundedSearch_Spec(int[] a, int key)	165230	19				0
4	CheckArray2(int[] arr)	235432	297				0
5	Count(int[], int)	464061	118	10		3	13
6	CountEven(int[] a)	251766	120	4		4	8
7	CountNonNull(string[] a)	111068	82	6		5	11
8	Factorial(int n)	4480	26			2	2
9	fibonacci(int fib)	7017	12	1	1	2	4
10	max(int x, int y)	1106713	80	2			2
11	multiply3(int x, int y)	150635	52			2	2
12	Product(int limit)	152246	71	4		2	6
13	Search(int[] a, int key)	945382	137	1		4	5
14	SumEvenIndex(int[] a)	138041	68			2	2
15	Square(int n)	5148	20	2		2	4
16	Sum(int[] a)	151448	82		1	2	3
17	Sum_x(int x)	83862	70	1		1	2
18	Swap(int[] A, int i, int j)	83862	1	4	3		7
19	SwapOriginal(int[] A, int i, int j)	205346	546	2	2		4
20	ZeroArrayForLoop(int[] array)	1826751	218	3		6	9

Aris 2.0 Results		#retrieval	#mapped	verified			
No	Method name			#pre	#post	#inv	#total
1	Add(int x, int y)	937050	190	3		1	4
2	AtLeastSquare(int[] inp)	216936	337			2	2
3	BoundedSearch(int[] a, int key)	83824	108	2			2
4	CheckArray2(int[] arr)	38209	67	1			1
5	Count(int[], int)	112747	203	9		5	14
6	CountEven(int[] a)	112042	182	2		6	8
7	CountNonNull(string[] a)	192209	394	5	1	5	11
8	Factorial(int n)	4434	20	2		2	4
9	fibonacci(int fib)	5527	38	1	1	3	5
10	max(int x, int y)	1106278	170		1		1
11	multiply3(int x, int y)	150632	106	3			3
12	Product(int limit)	152150	193	3		2	5
13	Search(int[] a, int key)	83824	108	1	5	2	8
14	SumEvenIndex(int[] a)	114366	199	0		2	2
15	Square(int n)	74858	74	3		2	5
16	Sum(int[] a)	144984	210	0		4	4
17	Sum_x(int x)	224216	632	2		1	3
18	Swap(int[] A, int i, int j)	81004	124	6	2		8
19	SwapOriginal(int[] A, int i, int j)	204893	566	2	2		4
20	ZeroArrayForLoop(int[] array)	1831988	339	5	1	13	19

Table 4.9. *Aris 1.0* and *Aris 2.0* result on creating specifications for selected methods in the 2 million open source repository.

Next, we compare the performance of *Aris 1.0* and *Aris 2.0* at the task of generating and transferring specifications in carefully controlled modified settings (TS1-TS4). The main principle for *Aris* is the fact that similar implementations also have similar specifications. Therefore, we evaluate how many transferred specifications that are formally verified using Spec# automatic verification tool[16]. For identical implementation (TS1), every case of specification was successfully transferred and verified by Spec# for both *Aris 1.0* and *Aris 2.0*.

For Test suite 2, *Aris 2.0 able to perform better by generating 115 verified specifications compared to Aris 1.0 that only generate 73 specifications in total*. We also observed that for the unverified programs in *Aris 1.0*, one of the common problems involved incorrect variable names caused by mismatched variables. Other problems such as poor loop matching in *Aris 1.0*

[16] Automatic verification tool. http://rise4fun.com/SpecSharp/

also affect the smaller number of verified *invariant* statements. Enabling assert or assume statements to be transferred in *Aris 2.0* also contribute higher number of verified statements.

In Test suite 3, the improvements in *Aris 2.0* enable it to perform better by generating 62 verified specifications compared to *Aris 1.0* that only generate 47 specifications in total. In Test suite 3, we add extraneous statement or changing the structure of the target. These modifications affect the mapping process and eventually the transferred specifications. The problems of *Aris 1.0* (mismatch variable mapping and poor loop matching) when generate specifications in test suite 2, also occur in test suite 3.

Transferring specifications between two significantly different programs would be less useful for reusing specifications as the chances for the specification to be verified are very small. Aris 1.0 and Aris 2.0 able to reject the mapping and do not transfer the specification between two functionally different programs.

4.4 *Aris* as Creativity Assistance Tool

The principle of *Aris* is to ease software developer works in creating specifications for their code implementation. This principle also guided our work in the transfer specification. Although, similar implementation may lead to reuse specification, however the transferred specification may not always verified within the target context and thus require human users to adapt the specifications for verification, for example transfer specification between method CountOdd and CountEven (see Table 4.10). This pair of method is not included in the test suite because the functionality of the method is different. The method in source count number of odd elements in an array, while the target method counts number of even elements in an array.

```
public int CountOdd(int[] array)
requires 0 < array.Length;
ensures result == count{int k in
     (0:array.Length); array[k]%2 != 0};
{
   int num = 0;
   for(int i = 0; i < array.Length; i++)
   invariant 0<= i && i <=array.Length;
   invariant num == count{int k in(0:i);
                        array[k]%2 != 0};
   {
          if (array[i] % 2 != 0)
```

```
public int CountEven(int[] array)
requires 0 < array.Length;
ensures result == count{int k in
     (0:array.Length); array[k]%2 != 0};
{
   int i = 0;
   int num = 0;
   while (i < array.Length)
   invariant 0<= i && i <=array.Length;
   invariant num == count{int k in(0:i);
                        array[k]%2 != 0};
   {
```

{
num++;
}
}
return num;
}
	if (array[i] % 2 == 0)
	{
	num++;
	}
	i++;
	}
	return num;
	}

Aris 1.0 sim_score: 0.76 *Aris 2.0 sim_score*: 0.85	

Table 4.10. Example of transfer specification between *CountOdd* method to *CountEven* method.

Although, these two methods in the source and target domain have different functionality, however they are highly related and able to share common specifications. The generated solution at first was not verified as Spec# gives warning `After loop iteration: Loop invariant might not hold: num == count{int k in(0:i); array[k]%2 != 0}.` The transferred specification still holds specification that suits with source method implementation and therefore human users need to modify the generated specifications to make the solution verified. The modification involved human users' knowledge to change the checking odd element process into an even element computation that captured in *postcondition* and loop *invariant* (see Table 4.11).

Generated Specification	Verified Specification
ensures result == count{int k in (0:array.Length); array[k]%2 != 0};	ensures result == count{int k in (0:array.Length); array[k]%2 == 0};
invariant num == count{int k in(0:i); array[k]%2 != 0};	invariant num == count{int k in(0:i); array[k]%2 == 0};

Table 4.11. The modification for generated specifications requires human's user knowledge

Another relevant example is transferring specification between source methods that check whether an array is sorted decreasingly and target method that check whether an array is sorted increasingly (see Table 4.12).

public bool CheckArrayDecr(int [] arr) requires arr.Length >= 0; ensures result == true ==> forall{int k in(0:arr.Length-1);arr[k+1] <= arr[k]};	public bool checkArrayIncr(int[] a) requires a.Length >= 0; ensures result == true ==> forall{int k in(0:a.Length - 1);a[k+1] <= a[k]};

ensures result == false ==>
arr.Length == 0|| exists{int k in
(0:arr.Length-1); arr[k+1] >arr[k]};
{
if(arr.Length > 0)
{
for (int i=0; i<arr.Length-1;i++)
invariant i >=0 && i < arr.Length;
invariant forall{int k in(0:i);
arr[k+1]<= arr[k]};
{
if (arr[i + 1] > arr[i])
return false;
}
return true;
}
else
return false;
}
	ensures result == false ==>
	a.Length == 0||exists{int k in(0:a.Length
	- 1); a[k+1] > a[k]};
	{
	for(int i=0; i<a.Length - 1; i++)
	invariant i >=0 && i < a.Length;
	invariant forall{int k in(0:i);a[k+1]
	<= a[k]};
	{
	if(a[i] > a[i+1])
	return false;
	}
	return true;
	}

Aris 1.0 sim_score: 0.71 *Aris 2.0 sim_score*: 0.74	

Table 4.12. Example of transfer specification between *check array decreasing* method to *check array increasing* method.

Once again, the solution is not verified and gives 3 warning message which are:

1. `unsatisfied postcondition: result == false ==> a.Length == 0 || exists{int k in(0:a.Length - 1);a[k+1] > a[k]}`
2. `Initially: Loop invariant might not hold: i < a.Length`
3. `After loop iteration: Loop invariant might not hold: forall{int k in(0:i);a[k+1] <= a[k]}`

Therefore, software developer needs to modify the generated specification for verification, which include changes in the boundary limit in `requires` and logical operator in the opposite direction in the `postcondition` and the loop invariant (e.g from <= to >= and > to <) as shown in the Table 4.13:

Generated Specification	Verified Specification
requires a.Length >= 0;	requires a.Length >= 1;
ensures result == true ==> forall{int k in(0:a.Length - 1);a[k+1] <= a[k]};	ensures result == true ==> forall{int k in(0:a.Length - 1);a[k+1] >= a[k]};
ensures result == false ==> a.Length == 0 || exists{int k in(0:a.Length - 1);	ensures result == false ==> a.Length == 0 || exists{int k in(0:a.Length - 1);

a[k+1] > a[k]};	a[k+1] < a[k]};
invariant forall{int k in(0:i);a[k+1] <= a[k]};	invariant forall{int k in(0:i);a[k+1] >= a[k]};

Table 4.13. The modification for generated specifications requires human's user knowledge

Aris is able to create the new formal specifications for specific segment of code and also interacts with human users to ensure the quality of generated specifications in the less similar implementation. Aris can yield potentially useful specification between different method functionality. However, human users need to adapt the newly transferred specification for the solution to be verified. (Donoghue et all, 2014) argues the human user's inference and validation will emulate and support the workaday *little-c creativity* of formal software developers within this highly specific domain.

The other example in Table 4.14 is reuse assertion which is one of core feature in *Aris 2.0* since it was not covered by *Aris 1.0*

```
public int factorialforloop(int n)
requires n >= 0;
ensures result == ((n == 0) ? 1
           :product{int j in (1..n); j});{
       if (n == 0)
       {
         return 1;
       }
       Else {
         int f = 1;
         assert f == product{int j in
                          (1..0); j};
         for (int i = 1; i < n+1; i ++)
             invariant 1 <= i;
             invariant i <= n+1;
             invariant f == product{int j
                         in (1..i-1); j};
         {

assert f * i == (product{int j in (1..i-
                              1); j}) * i;
assert f * i == product{int j in (1..i);
                                      j};
               f = f * i;
assert f == product{int j in (1..i); j};
         }
         return f;}
}
```

```
public int factorialwhileloop(int n)
{
   if (n == 0)
   {
       return 1;
   }
   else
   {
       int f = 1;
       int i = 1;
       while (i < n + 1)
       {
           f = f * i;
           i = i + 1;
       }
       return f;
   }
}
```

Table 4.14. Example of transferring assertion in factorial method

And the result is:

```
public int factorialwhileloop(int n)
requires n >= 0;
ensures result == ((n == 0) ? 1 : product{int j in (1..n); j});
{
   if (n == 0)
   {
       return 1;
   }
   else
   {
       int f = 1;
       assert f == product{int j in (1..0); j}; //verified


       int i = 1;
       while (i < n + 1)
       invariant 1 <= i;
       invariant i <= n+1;
       invariant f == product{int j in (1..i-1); j};
       {
           assert f * i == (product{int j in (1..i-1); j}) * i; //verified
           assert f * i == product{int j in (1..i); j}; //verified


           f = f * i;
           i = i + 1;
           assert f == product{int j in (1..i); j}; //not verified


       }
       return f;
   }
}
```

Table 4.15. Verified and unverified example of generated specifications based on program code in the Table 4.14.

In the Table 4.15, we can see that from 4 assertions, there are 3 assertions that can be verified and 1 assertion failed to be verified. To make it verify, we have to modify the variable in assertion from "`assert f == product{int j in (1..i); j};`" into "`assert f == product{int j in (1..i-1); j};`". After modification, the assertion can be verified. This is because in base code use for loop and target code use while loop. While loop in the target code needs to increment the iteration variable through statement "`i=i+1`". Meanwhile, the transferred assertion actually used to check the condition before the iteration variable incremented. Therefore, we need to adapt the assertion by decrements the variable in generated assertion for verification.

4.5 System limitation

The *Aris* system tries to map the equivalent loop construct and transfer the missing specification such as from *for* loop to *while* loop. However, in *Aris 2.0*, we ignore transfer specification from/to *ForEach* loop statement. *For* and *while* loop specifies the loop bounds for its minimum and maximum which enable them to share boundary invariant whereas *ForEach* loop does not specify loop bounds. Spec# does not support verification for *ForEach* loop, neither; Spec# is a research language, therefore all C# features will not be supported.

We acknowledge some Spec# features that are not covered in *Aris* such as object invariant, sub typing and inheritance, and aggregates. The object invariants specify, design of an implementation such as `expose` block; statements under expose blocks indicate that an object's invariant may temporarily be broken. Subtyping and inheritance introduces additive annotation; this annotation needed when an attribute of the subclass is mentioned in the object invariant of the superclass and additive expose. Aggregates object introduces a `Rep` annotation to identify objects that are components of the larger aggregate object. Another abstraction such as modifies and model clause also not covered in current *Aris* version. Some of Spec# features above are not covered because they are case by case basis and difficult to extract as they are embedded in the code.

The source code retrieval process consists of knowledge acquisition and query response. Knowledge acquisition is responsible to enrich case base with source code artefacts. In the knowledge acquisition step for semantic retrieval, API calls are extracted and indexed from a set of compiled assemblies. In the structural retrieval, methods are decompiled into C# source code and used to construct a conceptual graph and content vectors that stored in the case-base (Pitu, 2013). The knowledge acquisition takes approximately 5 hours on 2.4 GHz i5 processor, for the collection that contains over 2 million methods. These 5 hours can be divided into two main tasks which are extracting content vectors and extracting API calls. API calls may take around 45 minutes, and the rest of the time is allocated to extract content vectors. At query time, the source code retrieval system responses vary in time depending on the complexity of the input source code artefact.

4.6 Conclusion

In this chapter we presented the evaluation and benchmark *Aris 2.0* in order to test the significance of improvement in the newer system against *Aris* earlier system version (*Aris 1.0).* Evaluation is divided into two main sections, the first section address a recognition problem, and it focuses on the retrieval and mapping phase measuring the capability of *Aris* to identify the similar source code to some given problem. The degree of similarity between pairs of source code artefacts has been carefully controlled, forming four different categories that represent different degree of similarity with original unmodified implementation (Wilkinson, 1994). The second section of the evaluation will address a software generation problem focusing on the ability of *Aris* to generate new specification for the selected method from the open source repositories and also to identical and modified sources specified in (Wilkinson, 1994). We also explained *Aris* as creativity assistance tool and the system limitation.

5. Conclusions

There is increasing interest in using Formal Software Verification in order to ensure program's correctness and minimize the occurrence of software faults (especially for critical applications). *Aris* project aims to explore the possibility of transferring formal specifications between similar programs in order to help increase the number of verified implementations and reduce effort of writing specifications. The first version of *Aris* was developed by (Grijincu, 2013) and (Pitu, 2013).

Although the result of the early version of *Aris* is encouraging and show potential reuse of formal specifications, it still has many rooms for improvements and wide possibility for feature enhancement. In this project, we address the following question: *How we evaluate and benchmark first version of Aris (Aris 1.0)?* Our solution uses experimental methodology (Amaral, et.al, 2007) to address the research question.

In elaborating our solution, our work differs from the earlier version in the following ways:

- Our incremental graph matching algorithm ensures the backtracking process does not miss the valid match by backtracking to node with higher node rank value and lower node rank value (the earlier system only visits node with lower node rank value, however, in many case valid matches occur with node with higher node rank value). We also consider its efficiency by limit the backtracking into certain threshold that appropriately depends on the length of the source code.

- We reduce the occurrences of false variable mapping by considering two ways graph matching as opposed to one way graph matching in *Aris 1.0*. *Aris 2.0* analyzed both ways graph matching and use the graph matching result that generates a higher variable mapping similarity score.

- We have developed more relevant criteria for matching loop concept. The first version of *Aris* match based on the operator token in the first binary expression found in the loop statement (first binary expression in the loop statement is not sufficient to represent the loop

concept, whereas the loop condition gives stronger representation in semantically and functionally).

- We address the lexical similarity problem when assessing similarities between variables in earlier version by supporting additional data type such as *float,* decimal, and *bool.*

- Our conceptual graph construction process can support additional features of the source code such as conditional operator, else if statement, and the iterated section of *for* loop statement.

- Our solution ensures the top ranked documents possesses good quality of specifications for transferring specifications in the retrieval phase. The new system uses the specification score metric that will measure the completeness of the specification in each document related to the source code query.

- We enable adjustment of specification for transferring specification between different ways of iteration in *for* loop. This adjustment will help produce a verified solution. This adjustment is new in *Aris* because the earlier version of the system only focuses on transfer the specification into the corresponding element of target domain.

- Our solution can support additional specifications by transferring *assert* and *assume* specifications.

Finally, we evaluated *Aris 2.0* and compare its performance over *Aris 1.0* by collecting a small set of methods with specifications and over 2 million methods (without specifications) of real world projects from open source program. We have evaluated retrieval and mapping phase, which measure the ability of both systems to identify the similar source code given to any given problem – forming four different categories that represent different degree of similarity with original unmodified implementation (Wilkinson, 1994). We also evaluate the performance of *Aris* to generate new specification to selected methods from open source repositories and also identical and modified sources specified in (Wilkinson, 1994).

Overall, our improvements in *Aris 2.0* able to generate more verified specifications to the problem code, identify more valid mapping, and increase the precision of retrieved documents that similar to the problem code.

5.2 Future Work

Because our solution of transferring *assert* and *assume* specification based on the succeeding statement in the source code (expecting identical structure of the statement). Therefore, if the succeeding statement in the target domain is not structurally identical, then the algorithm cannot find its corresponding in the target domain. For example, in the base domain the succeeding statement relative to *assert*/*assume* statement is k++ and in the target domain it may become n = n – 1; n = n + 2 (variable k is mapped to variable n). The algorithm expects n++ rather than n = n – 1; n = n + 2, therefore *assert* or *assume* statement cannot be transferred in this case. This challenge for transferring *asserts* and *assumes* statement needs further analysis in the future.

Aris 1.0 and *Aris 2.0* were developed to support C# programming language. (Rahman) has developed *Aris 2.1* to support Java programming language which means *Aris* system is actually can be extended to support more programming feature. In the future, *Aris* can be developed to support other programming language. It is also interesting to enable reuse specification across programming language such as The Java Modeling Language (JML) specifications used in C# programming language.

The development of *Aris* 2.0 depends on Spec# as a research language. Current version of Spec# may not cover all programming feature, for example Spec# is currently does not support specifications for *foreach* loop. Consequently, current system of *Aris* does not support transferring specifications to *foreach* loop. However, Spec# is still an active research field and there are possibilities for *Spec#* to support other C# programming features. In the next development of *Aris,* we should also consider the development of Spec# programming system.

References

Amaral, Jose. 2007. *About Computing Science Research Methodology*. [Online] Available at: http://webdocs.cs.ualberta.ca/~c603/readings/research-methods.pdf [Accessed 2014].

B. Dit, B. Revelle, M. Gethers and M. Poshyvanyk. 2013. Feature Location in Source Code: A Taxonomy and Survey," Journal of Software:Evolution and Process, vol. 25, no. 1, pp. 53-95, 2013.

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.

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.

C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. In Fieldsof Logic and Computation, volume 6300 of Lecture Notes in Computer Science,pages 277–300. Springer, 2010

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

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

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

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

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

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

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

Hage, J. P. R. N. v. V., 2010. A comparison of plagiarism detection tools. Utrecht Uni-versity.
Kamiya, 2002

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

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.

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

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.

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

K. Rustan M. Leino, Peter Müller. 2008. Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs. LASER Summer School: 91-139

Meyer, B. From structured programming to object-oriented design: the road to Eiffel, Structured Programming 10 (1) (1989) 19–39.

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

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

Microsoft. 2013. *For (C# Reference).* [Online] Available at: http://msdn.microsoft.com/en-us/library/ch45axte.aspx [Accessed 2014]

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

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

M. Joy and M. Luck. Plagiarism in programming assignments. IEEE. Transactions on Education, 42(2):129–133, 1999

Monahan. R and O'Donoghue. 2012. D, Case Based Specifications – reusing specifications, programs and proofs. AI meets Formal Software Development. Dagstuhl Report II(7), pp 20-21

Montes-y-Gomez, M., Lopez, A. & Gelbukh, A. F., 2000. Information retrieval with conceptual graph. *Database and Expert Systems Applications,* p. 312–32

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

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

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

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).*

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

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

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

S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, Comparison and Evaluation of Clone Detection Tools, Transactions on Software Engineering, 33(9):577-591 (2007)

S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," Proc. Int'l Conf. Software Maintenance (ICSM '99), 1999.

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

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

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