Arís 2.1: Adapting Arís for Object Oriented Language

Fahrurrozi Rahman

Dissertation 2013
Erasmus Mundus MSc in Dependable Software Systems

NUI MAYNOOTH
Ollscoil na hÉireann Má Nuad

Department of Computer Science
National University of Ireland, Maynooth
Co.Kildare, Ireland

A dissertation submitted in partial fulfillment
of the requirements for the
Erasmus Mundus Msc Dependable Software Systems

Head of Department: Dr. Adam Winstangley
Supervisors: Dr. Diarmuid O'Donoghue and Dr. Rosemary Monahan
July 2014

Declaration

I hereby certify that this material, which I now submit for assessment of 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 other save and to the extent that such work has been cited and acknowledged within the text of my work.

Fahrurrozi Rahman

Acknowledgement

I would like to thank my supervisors Dr. Diamuid O'Donoghue and Dr. Rosemary Monahan from the National University of Ireland, Maynooth for all their help, patience and support on the project. I would also like to thank The Executive Agency Education, Audiovisual and Culture (EACEA) for the funding and all the classmates who support me directly or indirectly while doing this research.

Abstract

In the software development area, software verification is important such that it can guarantee the software fulfills its requirements. Despite its importance, verifying software is difficult to achieve. Additional knowledge and effort are needed to write specification especially if the software is complex and big in size. Nevertheless, there are some software that already have verified specifications. This project will focus on extending Arís (Analogical Reasoning for reuse of Implementation & Specification) which has been developed to increase verified software by reusing and transferring the specification from a similar implementation to a target code. The extension is done to facilitate specification transferring to program written in language other than C#, in this case Java. This extension will add functions to existing Arís that will receive Conceptual Graphs representation of a program and write the specification to a file. Another companion system is also built from Java to generate the Conceptual Graphs in Conceptual Graph Interchange Format (CGIF) and transform the Spec# specification to JML. Finally, this new system is evaluated by running some testing. From the result that we have, we can conclude that the building of conceptual graph and the specification transformation is the most difficult part in our system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We are living in an era that software are involved in every aspect of our life, from the very basic task, such as opening a door, to the complex one, like navigating a plane. In many ways, software can help us in crucial situations including aviation, nuclear power plant, space mission, hospital, stock exchange and banking to name a few.

With lots of involvements of software in our daily life the next important question is how we can make sure that the software we are using is reliable. The reliability of a software is very important especially in a safety-critical environment or when the failures will give expensive consequences. A recent study from Cambridge University shows that the debugging process of software costs $312 billion per year [PRW13]. Some experiences showed that a fault in the software can cause catastrophic impact. One example is the crash of Ariane 5 because of overflow computation [Mar], another one is the failure after updating RBS software system in 2012 that caused a man to stay in prison [BBC12], or stranded in hotel or airport [Mon12], and the latest is Toyota need to recall 466,000 cars because of the braking issues [Onl14].

1.1 Verified Software Overview

Software verification concerns in proving formally a piece of software if it is running correctly according to the specification of its intended behaviour [HMLS07]. By formally here we mean a formal model and the properties of a program are built. Then these formal model and properties, which are written in some formal language based on logic, are verified.

A mathematical model which tallies with the software is constructed and the program correctness is derived from proving the model related to the formal software specification. A formal software specification is expressed in a specific language that is based on mathematical concepts. There are three primary components which commonly compose this language [Pre10]:

1. a syntax which represents the specification with its notation
2. semantics which define the universe of objects used in describing the system, and
3. a set of relations which state the rules indicating the objects that properly satisfy the specification

Formal verification has been applied successfully in some crucial system such as spacecraft system [EL98], railway signaling system [DM95], air traffic control system [Hal96], and medical control system [Hal95].

1.2 Programming Verified Software

Three decades ago, there was a moving toward pragmatism approach of software verification. This was started by Bertrand Meyer who coined the term Design by Contract (DbC) for the Eiffel programming language. The idea of contract can be viewed as the bond between a supplier and its clients in a formal agreement that consists of rights and obligations. Here a supplier is defined as the class that provides the routine and a client is defined as the class that uses the supplier's routine. There are two main formal specification terms related to a routine [Mey97]:

1. precondition, which states the properties that must hold before calling the routine in order to function properly
2. postcondition, which states the properties that must hold after calling the routine when the precondition holds

From the supplier side, it has to guarantee the postcondition (as the obligation) and expect the client calls its routine according to the specification (as the benefit). Meanwhile from the client side, it has to satisfy the precondition (as the obligation) when calling the routine and expect to get the postcondition state (as the benefit).

Another formal specification terms related to the DbC is invariant. An invariant, if defined within a class, means a set of assertions that are satisfied in every life cycle instance of the class. A loop invariant is the property that must be true every time the loop is executed. These terms are the basic specification that a programmer can write using the DbC approach.

Even though Eiffel is the pioneer of implementing formal software verification, now the same approach also used in several programming languages, such as Spec# [BLS04] for C# and Java Modeling Language (JML) for Java [LPC+13]. For these two verification language, more detail information will be discussed in Section 2.1.2 and Section 2.1.3.

1.3 Arís

Recent research at the National University of Ireland Maynooth (NUIM), has developed a system called Arís 1.0 [PGL+13], that if given code (that we will call the target code) the system can find the similar implementation code (that we will call the base code) and transfer the specification from the base to the target. The aim of this system is to increase the verified software by reusing and transferring the specification from a similar implementation.

One motivation for reusing specifications is that it is not easy for the programmer to write a good specification which can depict the intended behaviours of the program. If we want to verify software then we need to prove its formal specification is satisfied by the code. In terms of software development, there is a lot of legacy code and library code without formal specifications that we can reuse in our system. If this is the case then in order to verify the system we need to verify everything including the legacy code. In this scenario we can see the need for a system that can find code with verified specification and transfer it to the legacy code.

Using Arís, which can transfer the specification automatically, the programmers who already have basic understanding of the specific programming language can write software that can be verified using the transferd specification without requiring a deep understanding of the specification language. Even though in some cases where the specification cannot be transferred correctly, Arís will give some hints of the intended specifications in two ways:

1. it will assist the programmer with the concept or semantic of specification writing, and
2. it will assist the programmer with the structure and syntax of specification writing

There are two main modules in Arís

1. Source code retrieval: this module aims to retrieve programs that are similar to the target from a large set of samples
2. Source code matching: this module will match between the target and the retrieved program and transfer the specification to the target

Arís 1.0 deals with specifications written in Spec# for C# programs. These specifications capture the programmer's intentions of how methods and data are to be used and checks consistency between a program and its specifications [BLS04]. The source code file is represented by Conceptual Graphs [Sow84] which are then analyzed to determine the semantic content of the code and the structural properties using graph-based techniques. Comparing two conceptual graphs is done using an incremental matching algorithm which is based on the Incremental Analogy Machine proposed by Keane [KB88] with the help of Node Rank metric [BINF12]. Finally the specification generation and transfer to the target code is achieved using a pattern completion algorithm Copy With Substitution and Generation (CWSG) [KJHM94].

In parallel with this research, Arís 2.0 is also being developed by [Hal14]. The research of Arís 2.0 aims to improve the performance of Arís 1.0. This new version of Arís will be discussed more in Section 2.4.1.

1.4 Problem Statement

Even though Arís 1.0 has performed well to find similar code and transfer the specification, it is also very limited to code written in the C# programming language and specification written in Spec#. In order to assist the programmer when writing the specification and to get more verified software, we need to modify Arís so it can generate specifications that not only restricted to one programming language. We will based our research on Arís 2.0 that already has some improvements in transferring the specification.

We state our problem statement in this research by posing this question: How can we reuse the Arís system with another programming language as the input and output language? This new Arís system will be very suitable for programmers who write software other than in C# programming language. This research aim will still conform to the original aim of Arís that is to reduce the overhead in writing the specifications and to develop more verified software.

To answer that question, we will extend Arís to facilitate the input from the Java programming language. The specifications generated by Arís then will be transferred into JML. By using Java programming language as the input and output of Arís, we also try to address these questions as our objectives in this research: What is the required approaches to build a new extension for another language? What is the overhead of implementing a new extension for Arís? How does Arís for Java perform?

All these questions are very related to a dependable software system. We are not only asking the scalability of the previous system but we also trying to get more verifiable software by implementing it into another programming language. Reusing specification is one benefit that we want to achieve using this system. Another significant aspect of this research is to find the factors that will improve the performance of transforming the specification given by Arís into another specification language. We will discuss these questions in detail in Chapter 5.

1.5 Summary and Report Structure

In this chapter we presented a brief description of the problem statement that we try to address. This problem statement relates to the aim of our project that is to reuse Arís with another programming language as the input and output. To achieve this we will add an extension into Arís so it can accept the input written in Java programming language and convert the specification it produces in Spec# into JML syntax.

The organization of the rest of this paper is as follows: Relevant works that related to specification generation are presented in Chapter 2. In Chapter 3 the theoretical backgrounds on which we built our solution are explained. Then, Chapter 4 gives the detailed solution of the system that we proposed to build with all the components and definitions. And finally we present the evaluation experiments of our system in Chapter 5 with the conclusion and future work in Chapter 6.

Chapter 2

Related Work

In this chapter we will briefly discuss some existing systems which give some influence to our work. It is also important to explain Arís, the system that we will build upon and modify. First we will discuss the basic terms that will construct our understanding of our problem domain, that is to reuse and transfer existing specifications.

2.1 Formal Specification

Recently as the computation machines become better and more powerful, a need to design and implement reliable software becomes more and more important as these software system give big influence to our civilization. One way to achieve this reliability in design software using formal specifications.

We define formal specification as the technique to define the formal model of a program and its properties. These specifications are written in some formal language based on logic which then are used to describe the system and analyse the behaviour. Once the specification is written, it can be evaluated whether the system design is already correct with respect to the specification using formal verification techniques. Formal verification here is the process of proving the correctness of the design of the system according to the specification. Using this formal specification and verification allow us to detect the faults in the system design and correct them. The formal specification can also act as a guide and description to implement the system.

2.1.1 Design by Contract

There are several formal specification languages that are based on the Design by Contract paradigm coined by Bertrand Meyer [Mey97]. These formal specification languages use Hoare calculus style: preconditions, postconditions and invariants which can be defined as follow:

1. preconditions, which state the properties that must hold before calling the routine in order to function properly
2. postconditions, which state the properties that must hold after calling the routine if the precondition holds
3. class or object invariants, which state the properties that must be hold in every life cycle of the instance of the class
4. loop invariants, which define the properties that must be true every time the loop is executed

Another important formal specification component is the frame condition. A frame condition defines which object of the class that can be modified when the routine is being processed.

2.1.2 Java Modeling Language (JML)

Two specification languages that really related to our research are the Java Modeling Language (JML) and Spec#. JML is the formal behavioural specification language designed for code written in the Java programming language. Here we define the *behaviour* of a Java code as what it should do at runtime when the code is called. For a method, its behaviour is defined as the preconditions and the postconditions.

The specifications in JML are written using annotation comments that starts with an '@' sign. The '@' sign must appear right after the comment sign, such as //@ or /*@ ... @*/. The incorrect form, such as // @ or /* @ (which has a single space after the comment sign), will be ignored by JML.

If the specification is a method specification, then it should be written just before the method declaration. If it is a loop invariant, it should be written just before the loop declaration. JML also supports informal specification, which is a specification not written in the formal form instead as a comment in natural language. These specification is usually written in the form of (* `specification written in natural language` *). JML informal specification is considered as an expression that always returns true. This form of specification is really helpful when the programmer finds the difficulty to write a formal and clear specification. An example of a code written in Java and JML can be seen from the Listing 2.1 below.

2.1.3 Spec#

Spec# is a research specification language created by Microsoft. Spec# is used to define the formal specification of programs written in C#. Spec# also comes with the compiler and the static program verifier: Boogie [Lei08], the intermediate verification language that will transform Spec# program into verification conditions which then will be used by Z3 [DMB08], the automatic theorem prover which analyses the program correctness based on these verification conditions.

Similar to JML, Spec# also defines the preconditions and the postconditions for the methods in the code. In terms of object's invariants, Spec# also introduces the `expose` block that can temporarily allow the breaking of the object's invariants. Unlike JML, the specification syntax is written with the reserved specification keywords without the need of annotation. Another difference is the placement of the preconditions, the postconditions or loop invariants is just before the opening curly bracket of the method or the loop. We can see an example of a code written in C# and Spec# in the Listing 2.2.

Below is the example of a method written in Java and JML

Listing 2.1: Java and JML Code

```
1 //@ requires 0 < amount && amount <= balance;
2 //@ assignable balance;
3 //@ ensures balance == \old(balance) - amount;
4 public void debit(int amount)
5 {
6     this.balance -= amount;
7 }
```

And below is the same method written in C# and Spec#

Listing 2.2: C# and Spec#

```
public void Debit(int amount)
requires 0 < amount && amount <= balance;
modifies balance;
ensures balance == old(balance) - amount
{
    balance -= amount;
}
```

As can be seen from the two codes above, the precondition is defined using `requires` keyword, the postcondition is defined using `ensures` keyword, and the frame condition is defined using `assignable` keyword in JML and `modifies` keyword in Spec#. The keyword `old` in Spec# or `\old` in JML states the previous value of the particular variable before the routine is called.

2.1.4 JML vs. Spec#

Even though JML and Spec# look similar, they have several different important features related to how they define the specification. For JML, it has built in data types, called `JMLType`, which are the refinement of Java wrapper class. Some examples for this types are `JMLDouble` and `JMLInteger`. Using these types, we can get more rich specification expression because these types also provide some methods that can be called in the specification definition. For example, when we want to define an approximation of a double value in the specification, we can call the static method `approximatelyEqualTo` from `JMLDouble` class. Another feature that JML has is we can make a distinction between normal and exceptional postconditions. A normal postcondition for a method is defined by the keyword `ensures`, whereas an exceptional one is defined using `signals` or `signals_only`. We use `signals_only` to define the exceptions that can be allowed when the condition is met. Meanwhile, using `signals`, we can rule out what must be true of the program state when the particular exception is thrown.

Ownership hierarchy is one of the most important thing in writing Spec# code. When we define a field from another object of another class in the program, we have to specify explicitly the ownership type of that object. There are two types of object ownership in Spec#: `Peer` and `Rep`. Because an object invariant should always be valid in every life cycle of the object, a field update can only be allowed as long as the invariant is maintained. We call this situation as a valid state. For another situation where the object invariant may be broken, e.g. by updating a field, we call this as a mutable state. A `Peer` object is an object without an owner or, if it has any, the owner should be in a mutable state. Meanwhile, a `Rep` object is an object with an owner and the owner is in a valid state. Related to this object ownership, when calling a field method that may break the invariant, a `Rep` object should be treated specially because this kind of object should always be in a valid state. Spec# allows us to make the specification by putting the field method calling inside an expose block. Within this block, and only in this block, can the invariant be broken.

2.2 Verification Tools

In order to verify the correctness of the specifications, we need some verification tools that can translate the written specifications into some lower level language based on logic and check its truth value. Here the verification tools will generate a collection of mathematical proof obligations. The proof obligations are the truth of which indicate the system in conformity with the specification.

There are two popular approaches for generating proof obligations as discussed by [Mon14]:

1. verification conditions generation, and
2. symbolic execution

For verification conditions generation approach, firstly, the written specifications are translated into an intermediate language. Then the verification conditions are generated from this intermediate language using Hoare's weakest preconditions calculus. The weakest precondition is the minimum requirement needed for the command in the program to run. It is usually derived directly from the postcondition P upon executing statement S. The correctness of the verification conditions are then will be checked by a Satisfiability Modulo Theories (SMT) solver. An SMT solver is a solver based on the satisfiability (SAT) problem solver with some performance improvements by adding some theories (linear arithmetic, bit-vectors, arrays, data types, uninterpreted functions, and quantifiers).

A symbolic execution is an analysis of the program to discover the inputs that can induce the execution of every statement of the program. Every statement of the program is executed and removed in a sequential manner while the strongest postconditions are being calculated from the preconditions. The strongest postcondition is usually derived directly from precondition P upon executing statement S. For every achievable execution branch, a set of constraints portraying the corresponding final program state is generated by a stepwise transformation of the program. Then, using first order reasoning these constraints finally can be evaluated against the indicated properties.

Some verification tools using verification conditions generation approach are the verification tools for Spec# and Dafny. Meanwhile, the verification tools for Java such as ESC/Java2 (will be explained in Subsection 2.2.2 below), KeY, and Verifast, are using the symbolic execution approach.

2.2.1 Spec# Verification

For Spec#, the verification tools have already included in the Spec# programming system (embedded also in the Microsoft Visual Studio tools). The tools consist of two components: Boogie [Lei08] and Z3 [DMB08]. From the specification written in Spec#, it is then converted into Boogie intermediate language. Beside its function as an intermediate verification language, Boogie is also can be used to construct program verifiers for other programming languages.

The process of verifying Spec# starts from translating the bytecode, in Common Intermediate Language (CIL) format, to BoogiePL. From the BoogiePL, the verification conditions which will be used by Z3 theorem prover are generated. The correctness of the specification is then checked by Z3 SMT solver [Hef10].

2.2.2 JML Verification

For specifications written in JML, the verification tool that we will use in this research is the Extended Static Checker for Java version 2 (ESC/Java2) along with the Mobius Program Verification Environment. ESC/Java2 is a program compile tool that tries to verify a Java program annotated in JML using the program code static analysis and its formal annotations.

Firstly, ESC/Java2 will parse and compile the java code into an intermediate form and also uniting the JML specifications into it. Then it will utilise static analysis techniques for symbolic execution of the code in-between JML specifications. A static analysis is the technique of analysing the program code without running it. This kind of analysis is used a lot by compilers in order to detect errors and carry out optimizations. This analysis is very limited to only what is observable during compile time.

The results of these JML specifications and symbolic execution are then used to build the first order logic formulas which are then loaded into a first order theorem prover. The prover will try to find a counter example for the formulas. If none can be found then it indicates the conditions written by the specifications hold [Mis10].

The Mobius that we used is integrated with Eclipse. Basically Mobius is intended to give safety and security guarantee of Java application for mobile phone or PDA. Mobius provides static checking enforcement before the application starts to execute. This static checking enforcement is also adaptable and configurable for security concerns and the real world mix of mobile platforms. ESC/Java2 is the verification tools behind this static checker [INR05].

2.3 Specifications Writing and Specification Generation Tools

As we have discussed earlier, software specifications writing is tedious work because the programmers need to know the knowledge about the specification language and they have to give efforts to construct the specifications. Training of specification writing can be organised in a company but it is usually costly. Meanwhile the company should also train new programmers in the case of the previous trained programmers leave the company. The lack of expertise in this area can bring the downfall of the software quality produced by the company.

Another difficulty arises when dealing with legacy codes or libraries that are used in the software. If these codes do not have the specifications then there will be supplementary work to do by the programmers: write the specifications of the software and the legacy codes as well and then verify them (there is a case where several collaborative European projects have participated in the documentation of legacy systems or reverse engineering [BH06]).

2.3.1 Daikon

To deal with the problem of specification generation, there are several tools that have been developed to generate some part of specifications of a program automatically. One worth to be noticed is Daikon [EPG+07] that can assist programmer in creating the software specification. Daikon is a tool written in Java that dynamically detects promising program invariants (or operational abstraction in Daikon term) using machine learning techniques.

Given a program execution with a lot of test cases, Daikon will examine the values computed by the program to get the patterns and relationships amongst them and report the properties that are true over the execution of the program at every procedures entries (preconditions) and exits (postconditions). These properties can be in many forms including number constants (e.g. x = a), collections (e.g. x in stackArr), pointers (e.g. p = p.left), and conditionals (e.g. if p $\neq$ a then p = p.left).

The key features of Daikon is it can receive input program from several programming languages, for example Java, C, C++, Perl, C# and Eiffel. Another useful feature is the output of the likely invariants can be written in several formats such as: ESC/Java, JML, or Java (even though the output still has a mix expressions between Java and JML). Sample output of Daikon can be seen in Figure 2.1 below.

Due to the nature of invariant analysis based on the test cases provided when running the program, the accuracy of the reported invariants rely on the quality and completeness of the test cases. There are two potential problems that can arise:

1. missing properties (false negatives), which are caused by the limitation of the expression generated by the tool's grammar

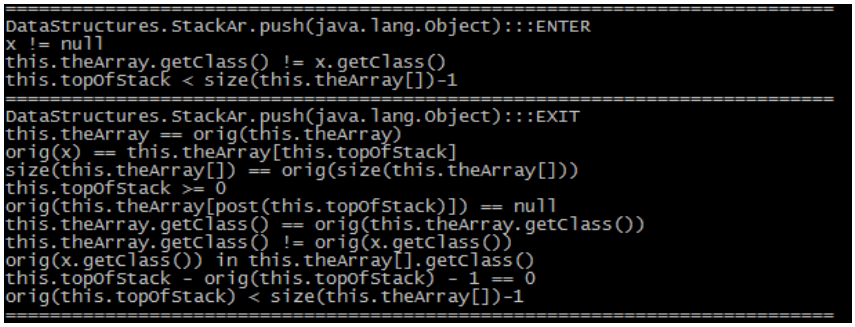

Figure 2.1: Example of Daikon Output

2. spurious properties (false positives), which are caused directly by the test suite.

For false negatives, Daikon provides a way to overcome them by giving extension facility for the user to broaden the grammar. Even though false positive results from the test suit affect the output quality, the static analysis and statistical test in Daikon can cut down the number of their occurrences [ECGN00].

Despite those two weaknesses and the nature of the *educated guess* of the invariants, the output of modest test suites is remarkably accurate. Using Java programs as input and ESC/Java as the verifier, one experiment using data structures classes taken from a textbook and solutions to assignments in programming course shows Daikon's precision (properties found and provable) is 95% and its recall (properties needed and found) is 90% [NE02]. This means that roughly from 100 properties given by Daikon, only 10 or less properties are false and can be discarded and the programmer can add another 10 properties.

2.3.2 gin-pink

Even though Daikon results are really promising, this tool cannot generate the loop invariant automatically. In most case, the loop invariant can be derived from the postconditions, even though more often than not we have to reshape the postconditions in order to get the loop invariant. Study by Furia & Meyer [FM10] shows that indeed the loop invariant mostly can be derived by weakening the postconditions using several approaches:

- constant relaxation: replacing one or more constants in the postcondition by variables
- uncoupling: two occurrences of the same variable in the postcondition are replaced into different variables
- variable aging: a variable in the postcondition is replaced by an expression stating the variable value at the previous iteration. This approach can be quite complex
- term dropping: remove a term from the postcondition, usually if the postcondition is in the form of conjunction

Furia & Meyer built a system called gin-pink [FM10] and did experiments by first manually writing the full preconditions and postconditions of several procedures using Eiffel code and then applying these four approaches above to generate the loop invariants. The result also looked promising even though in some procedures those approaches failed to produce the correct loop invariants, for example, the Dutch National Flag and Stack Reversal procedures.

2.3.3 Abstract Interpretation

As defined by [CC77], an abstract interpretation of a program consists in using the expression to describe the computations into another universe of abstract objects, so that the results of abstract execution give some information on the actual computations. [CC77] gave an illustration by using the mathematical formula written in text as $-1515*17$, it can be understood as the computation on the abstact universe (+), (-), (*). Following the abstract execution of the program, we obtain that $-(+)*(+) \Rightarrow (-)*(+) \Rightarrow (-)$, which has the semantic of saying that $-1515 * 17$ is a negative number. Using this example, [CC77] showed that although it can be incomplete, abstract interpretation can allow programmers or compilers to answer questions which can tolerate an imprecise answer (e.g. by ignoring the termination problems to prove the partial correctness of a program, etc).

In terms of specification writing, abstract interpretations gives the theoretical background for inferring the specifications in the program. This inference problem is, quite similar with the mathematical formula example above, just a matter of abstraction of the trace semantics.

For example, the abstract interpretation to infer the loop invariant is done by computing the abstract element at the loop head using three steps below [Log13]:

- Firstly, a sound abstract domain is set up. The property of interest should be captured in the abstract domain, e.g. array contents or linear equalities/inequalities among the variables.
- Next, the abstract operations and transfer functions are set up. An abstract operation combines two abstract elements, and a transfer function defines how an atomic program statement can modify the abstract states.
- Last, the convergence operators are constructed. This operators guarantee that the loop process will actually terminates.

In his paper, [Log13] also discussed how abstract interpretation is used to infer the specification for postconditions, preconditions, and object invariants. He argued that the inferred specifications are crucial for the success of verification tools as he and his team spent a large amount of time to implement, refine and optimize the algorithm for inferring code specifications. They implement the abstract interpretation into a static checker for Code Contracts.

Code Contracts is "a language-agnostic solution to the problem of software specification and validation." This is another example of a specification language implemented by Microsoft lab. The specification syntax is included in the internal API for .NET programs so the programmers do not need to learn a new specification language [BFdH+09]. For example, we can make preconditions by calling a `Requires` method, just like calling an ordinary method, or we can write the object invariants inside a void method. One advantage of the static checker [Log13] implemented for the Code Contracts is it can provide the hint to repair the specification so that helps the development process. This hint can be used as the recommendation for the programmer when dealing with writing a wrong or an incomplete specification.

As we can see from Daikon and gin-pink, there are difficulties of generating specifications from the very beginning. We have already mentioned that writing specifications for software in the first place is a difficult task, time consuming, tedious and error prone. Understanding the formal specification language is one factor that makes this task is laborious. There is an encouraging result comes from Code Contracts static checker which utilises the abstract interpretation to verify the specification. This static checker can give hint and advice on what to write in case there is a missing specification statement. The Code Contracts itself is a major breakthrough of writing specification in the same language with the software development language. Nevertheless, still in this case the programmers need to do the specification writing and refining.

Based on this observation, we move toward reusing and transferring specifications based on an existing ones. We saw this as an opportunity to build Arís that can find similar implementation with existing specification given a target code and transfer the specification to the target. Using this approach, we minimize of writing the specifications since the beginning by finding the most similar implementation code with the specifications. Even though there will be a case when the transferred specifications are still incorrect, they still can be used as a hint or guideline for the programmer on how to write the specifications.

2.4 Arís – Analogical Reasoning for Reuse of Implementation

[PGL+13] developed Arís 1.0 that aims to increase the number of verified programs and also guide the programmers to write and reuse specifications. This system attempts to find a similar implementation with specifications (the base code) with the given one without any specification(the target code). If there is such code then it will transfer the specifications to the target code. This way, the program developers can write a high reliability software with specifications without the need to have a deep knowledge of the specification language as long as they have a basic understanding of the programming language. Even though if Arís can not transfer the specification correctly, which can happen in some cases, the system will still give benefits for the developers by giving hints on how to write sound and syntactically correct specifications.

2.4.1 Arís 2.0 and Arís 2.1

In spite of the fact that Arís 1.0 has successfully aided programers in writing specifications for C# code, there is still a room for improvement so that Arís can also be utilised to help programmers writing specification in different programming language. Side by side with this research, [Hal14] is developing Arís 2.0 which gives better improvements from Arís 1.0. She improves the conceptual graph construction process, enhances the IAM algorithm for finding the valid mapping, generates more verified specifications, and adapts the specification generated in target code context. For source code retrieval module, she adds a novel metric in terms of specification score to increase the number of candidate code retrieved for specification reuse.

Our research will add new functions into Arís 2.0 so now it can work with code written in the Java programming language as input and generate specifications in JML as output. We choose Java programming language because it is quite similar with C# language and it is also an Object Oriented Programming (OOP) language. This OOP language will correspond with the Design by Contract approach coined by Meyer [Mey97].

The two main modules in Arís are source code retrieval and source code matching that will be explained more in the next subsection.

2.4.2 Source Code Retrieval

Since Arís 1.0, the system runs based on two modules. The first module is source code retrieval which aims to find the similar codes from a given target code [Pit13] that utilises Case-Based Reasoning (CBR) to achieve the objective. Case-based reasoning [Kol93] is a kind of reasoning in Artificial Intelligence which solves new problems by finding the solutions from similar previous problems and transfer the old solutions to the new problem. This type of reasoning, which uses past situations to the current one, is similar and quite important in everyday human problem solving behaviour. For instance, lawyers will use past cases as authoritative example to construct and defend arguments in new cases. Doctors will recall

and look at past similar symptoms and suggest a solution using the old diagnosis for the new problems they are facing. In those examples, recalling the old cases make solution generation can be done easily.

In general, there are four processes that form case-based reasoning [AP94]:

1. *retrieve* from the repository or memory all problems similar to the given target. All problem information such as the solution and how to get the solution must be stored in the memory.

2. *reuse* the solution from the past problem to the target problem. This old solution may need to be modified in order to fit the new target problem

3. *revise* the new solution if needed. Revision is done only after evaluating the mapping of old solution to the new one.

4. *retain* the new problem with its solution in the repository of memory after the solution has been mapped successfully.

Arís uses the CBR approach collaborating with the semantic and structural properties of the source code for retrieving similar codes from a repository of software artifacts. These artifacts are composed by managed compiled assemblies (dynamic-link libraries(DLLs) or executables(EXEs)) of C# programs.

Semantic Property

The semantic properties of the source code is an important part of this module because it gives the ability for the system to find two blocks of code with the same semantic meaning (same high level definition) but with different implementation or structure. API calls are the core component to define the semantic property of the source code. It is because the meaning of API calls are precisely defined by the system and they form the building stacks when the programmers want to build a higher-level code structure. For example, the calling of method Replace for String is reasonably used to do string replacement instead of creating a new solution.

For this module, the core artefact will be consisted of methods because API calls are used when implementing methods which are contained in a class. Based on this examination, only the methods will be regarded as documents in the corpus. API calls from the source code are extracted and then their weight are computed using Term Frequency-Inverse Document Frequency (TF-IDF). TF-IDF is a numerical statistic technique to give weight that points out the importance of a term/word in a document with respect to the whole collection of documents [RU11]. The TF of term `t` in document `d`, $tf_{t,d}$, is computed as the occurrences of that term in the particular document. To compute the IDF of term `t`, first the number of documents containing term `t` is calculated, which denoted as Document Frequency (DF) value or df_t. The IDF of term `t`, idf_t, is the logarithmic value of the ratio between the total number of documents in the collection and the df_t value of that term. The TF-IDF value for term `t` in document `d` is the multiplication between its TF value and its IDF value, or $tf - idf_{t,d} = tf_{t,d} * idf_t$.

The TF score of an API call `a` in a document `d` is computed as the number of occurrences of `a` in `d`. This TF score is then stored in a Hash Map as the value where the API call is the key. The IDF score is computed after all API calls in the collection of documents is completed.

After the TF-IDF scores have been computed, the values will be stored in a Vector State Machine (VSM) [RU11] where each element in the vector correlates to the weight of an API call in the document. The similarity between two documents can be computed using cosine similarity (simCos) distance as the division of the dot product of the two vectors and the product of the vector Euclidian lengths.

Structure Property

Another property that is taken into account for code retrieval is the structure property which handles the topology or structure of the code. Here Arís uses conceptual graphs to represent the structure of the code (this approach is also used in the second module of Arís that will be explained later in the Subsection 2.4.3). This technique of representing source code into conceptual graph has been done successfully by Mishne and De Rijke [MR04] who tried to explore the similarity based on conceptual graphs with a good results for programs written in C programming language.

The conceptual graph from the source code is constructed using the help of Abstract Syntax Tree, a tree which represents the abstract syntactic structure of the source code. The information given by The abstract syntax tree gives a very verbose information about the structure of the code, however not all will be used in constructing the conceptual graph because we do not need them. Thus the resulting conceptual graph will be shallow but gives an advantage for us to get a fast ranking of the source code artifacts which are similar to the given query (target) code. Information about conceptual graph will be discussed in more detail in Section 3.1.

2.4.3 Source Code Matching

The second module of Arís is source code matching [Gri13]. In this module, two source code are compared based on their conceptual graphs representation. The comparison is done to get the best matching between target code and codes retrieved from the retrieval module. As taken from [Gri13], the comparison of two graphs is achieved using Analogical Reasoning approach. This kind of reasoning is important in human cognitive processing. When we find a situation that is less familiar, we first try process it by finding a more familiar situation in our knowledge repository and then match it with the less familiar situation to get a better understanding.

Several important steps when processing information using analogical reasoning can be defined as follow [KLD94]:

1. Representation. To get a solution for a new situation utilising analogical reasoning, first we have to make a meaningful representation of the problem. This representation will influence the performance of the analogical transfer process.

2. Retrieval. This step tries to find the best candidate that matches with the new situation. The main process in this step is searching through a database of known situations and fetch the most similar one to the target.

3. Mapping. This step is the heart of analogical reasoning because it will determine the elements of the retrieved situation that will be mapped to the elements in the target. This process can be very computationally complex and expensive because there can be many ways of mapping two situations.

4. Transfer. After the mapping process has been done, the new knowledge is constructed and transferred to the target.

5. Evaluation. This step will validate the new transferred knowledge in order to determine if the new generated knowledge can be utilised in the target domain.

For this module, the analogical reasoning is used in the process of establishing inferences and constructing new information in the target code that we want to build the formal specification using the existing specifications from the base code.

An incremental matching algorithm is used, adapted from the Incremental Analogy Machine (IAM) [KB88], to find the best mapping between isomorphic (exactly match) or homomorphic (non-identical match) sub-graphs. This algorithm will build a mapping between the target and the source incrementally by choosing and matching a small portions of the source (the seed group and the seed match) rather than matching all element in the source at once. This approach gives several advantages: reducing the complexity of matching process and also enabling backtracking if the matching is not successful. The drawback of this approach are the difficulties of finding the good seed group and seed match criteria. Having found the best match between source code and target code, the new specifications from source code to target code is generated by pattern completion using the Copy With Substitution and Generation (CWSG) [KJHM94].

The CWSG algorithm first discovers a statement `S` and a relation `r`(`a`, `b`) with `a` and `b` are attributes of objects in the base domain that do not have an equivalent mapped statement in the target. From the evidence that after running the incremental matching algorithm, both the relation `r` and its attributes are mapped accordingly with elements in the target ($\texttt{r} \rightarrow \texttt{r}'$, a $\rightarrow$ a′, b $\rightarrow$ b′), CWSG then builds and transfers the new statement $\texttt{S}'$: $\texttt{r}'(\texttt{a}', \texttt{b}')$ to the target domain.

2.5 Arís 2.1

In the previous section we know that Arís 1.0 was developed based on analogical reasoning approach to deal with the source code retrieval and specification matching. Arís 2.0, an improvement of Arís 1.0, is now being developed too by [Hal14]. The aim of this improvement is to get more verified code by enhancing some algorithms related to the creation of the conceptual graphs, the matching process between two conceptual graphs, and by adding the new metric to get more similar codes with the target code.

Because Arís has shown a promising result, we now try to modify it so that it can also process the specification matching and transferring from input written in different programming language and also generate the transferred specification to the particular given programming language.

Our new Arís 2.1 is still intended for program written in object oriented programming language. To achieve this objective, we consider three approaches that we can use to develop this new system:

1. Build Arís 2.1 from the very beginning
2. Convert the source code written in another programming language and feed it to Arís 2.0 and then convert the resulting specification
3. Build extension in Arís 2.0 to receive the conceptual graph of the code and then convert the resulting specification to the particular programming language

2.5.1 Build new Arís from the very beginning

The first option to build Arís 2.1 is to construct the system from the very beginning. This option is very useful if we want to get a destined new system that only deals with the program written in a particular programming language (e.g. Arís only for Java, Eiffel, etc). This approach was the first one that came up to our mind when we try to develop Arís 2.1. However, after spending several weeks we realised that this approach is not a feasible one as we explained its disadvantages below.

The advantage of this approach is the system will have to look only for the code and specification written in the same language with the target code so it does not have to transform the retrieved specifications (as we will see in the next two approaches). Yet, this strategy also has disadvantages. First,

we try to build a new system based on a well performed existing one. This is a software engineering bad approach to build a system because we are trying to reinvent the wheel rather than reuse the existing one. Another disadvantage is this new system only suits for one programming language. The next time we want the system to process a different programming language code, we have to build again another system.

2.5.2 Convert the source code to C# and feed to Arís 2.0

Using this approach, we can reduce the amount of work that needs to be done compared to the first approach. This time we do not have to build a new system from the very beginning. We only have to convert the source code from the target programming language to C# (the programming language supported by Arís since version 1.0) and feed that converted code to Arís. Finally, the specification generated by Aris then is transformed to the specification form in the target language. However, this approach also suffers from disadvantage. We need a way to transform code written in one programming language to C#. Even though there are some existing code converter, mostly they cannot transform correctly and do not support for most programming languages.

2.5.3 Build extension in Arís 2.0 to receive the conceptual graph

This approach will tackle the disadvantage in the second approach, that is the need of a very good code converter, because this time we will not convert the target code to C#. We only need to construct the conceptual graph of the target code and feed it to Arís. Once Arís gets the specification, we will transform it back to the specification form in the target language. Using this approach we do not need to rebuild Arís if we want it to receive a code written in a new programming language. It is because the system will just read and process the conceptual graph of the new target code so we will not reinvent the wheel. The disadvantage of this approach is we need to construct the conceptual graph of the target code as exactly as possible in the same manner with how Arís constructs its conceptual graph.

By considering these three approaches, we design and implement Arís 2.1 that we will discuss more in Chapter 4.

2.6 Summary

In this chapter we discussed the related works in generating specification and also our current running system. We explained about the concept of formal specification and we discussed two examples of formal specification languages that related to our research. Then we continued to discuss the verification tools and the two approaches to get the proof obligations, which will be used to verify the written specification, and the difference between them.

We also explained tools called Daikon and gin-pink, that have been developed to generate specification automatically. We have critically mentioned the advantages and disadvantages using the two specification generation tools. Daikon is a promising tool that can generate likely invariants in a program provided a complete and good quality of the test suites. The challenging task is how to provide some decent test suites. Even though it can generate the likely invariants from several programming language code and produce the output into several formats, this tool cannot generate the loop invariant for the source code. Another tool, gin-pink, is also already developed to generate the loop invariants from the postconditions in a source code. But to use this tools, first we have to provide the full preconditions and postconditions of the source code.

We then discussed the concept of abstract interpretation as the background to be used for inferring the specifications in the program. We explained the static checker developed from this abstract interpretation to verify specification written in Code Contracts. We saw that this static checker can give a hint to repair the wrong or incomplete specification for the programmer. We argued that even though this hint can be very helpful, the programmers themselves still need to write the specifications in the first place.

Based on the observation about these three things, we explained how we move towards reusing the software specification from an existing one by building Arís. We defined the core implementation of Arís 1.0 and its improvement, Arís 2.0, which is still being developed by [Hal14]. We mentioned three approaches that can be used when we develop Arís 2.1 that we will further explained which one we eventually use to build the new system in Chapter 4.

We close this chapter by discussing several approaches that we consider to build Arís 2.1. We can build Arís 2.1 from the very beginning, or we can convert the target code into C#, or we can feed Arís a conceptual graph from the target code. We will explain about the choice we make and the implementation detail in Section 4.

Chapter 3

Concepts and Terminologies

This chapter will discuss the concepts and terminologies that we use in our system. We will describe the Conceptual Graph and how we represent our source code into Conceptual Graphs. Finally we will talk about the Conceptual Graph Interchange Format (CGIF) that will be used as a written representation of our source code Conceptual Graph.

3.1 Conceptual Graph

A conceptual graph (CG) is a graph representation for logic. It is basically developed from the concept of existential graphs coined by Charles Sanders Peirce and also the concept of semantic network in artificial intelligence [Sow08]. John. F. Sowa developed a version of conceptual graphs in 1976. This conceptual graphs were used as a transitional language for mapping between questions and assertion in natural language to a relational database [Sow76].

A conceptual graph is a directed bipartite graph, consisting of two kinds of nodes called a concept node and a relation node.

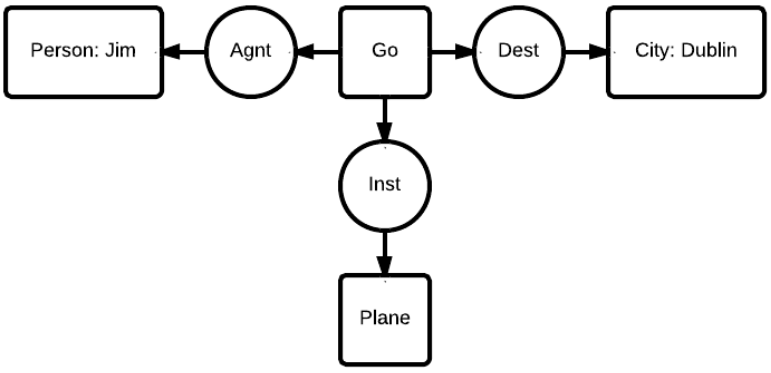

Figure 3.1: CG for sentence *Jim is going to Dublin by plane*

As can be seen from the image above, rectangles or boxes are representing the concept nodes and the circles or ovals representing the relation nodes. An arc with an arrowhead goes approaching a circle denotes the first argument of the relation, and an arrowhead goes away from a circle denotes relation's last argument. In Figure 3.1 above, `Go` is the first argument of relation `Agnt` and `Person` is its last argument.

The conceptual graph in the image above has four concepts, each with a type label that describes the type of entity to which the concept refers: Person, Go, City, and Plane. Jim and Dublin are individuals (or referent) of concept Person and City respectively. The three relations have a type label which denotes the type of the relation: Agnt relation for the agent of going concept, Inst relation for the instrument of going, and Dest relation for the destination. The whole conceptual graph defines that Jim is a person who is an agent of some instance of going with the city of Dublin as the destination and a plane as the instrument of going.

Concept nodes can hold information such as entities, actions, properties in the real world, an abstraction, fantasy or mathematical functions. Meanwhile relation nodes describe the connection between two concept nodes. Every node has a type (either a concept or a relation) and each node is also related with a referent or individual value that stores the data specific for that node. The value of the referent can be a specific one (like Jim in the Person concept or Dublin in the City concept) or a generic one (like can be seen in the Plane and Go concept).

A support in conceptual graph denotes the syntactic constraints, the information about the domain of the conceptual graph, and the rules used when constructing a conceptual graph. The support contains several information such as: the set of concept types (which stated in a hierarchical structure of "is-a" relationship), the set of relation types (which denotes the kind of a concept type that can connect to another concept type), and a set of referent for each concept type (which differentiate between generic and specific ones.)

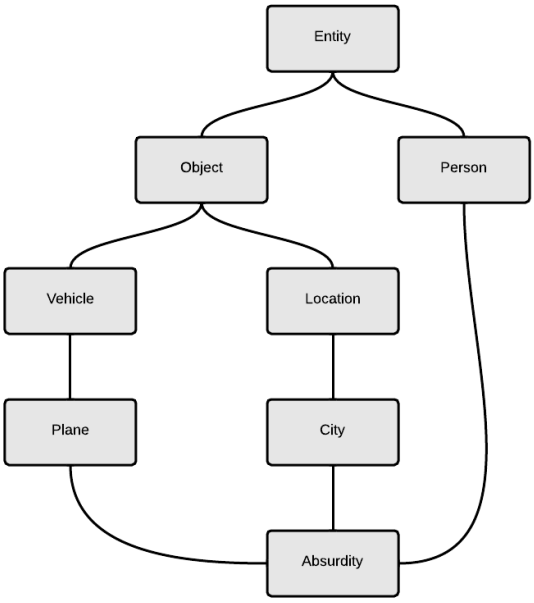

Figure 3.2: Example of Partial Support

Figure 3.2 above shows an example of a partial support for conceptual graph. The top of the hierarchy denotes the entity type for everything while the bottom of the hierarchy denotes the absurdity type. From that partial support, we can make a conceptual graph that we have seen in the Figure 3.1 above.

3.2 Conceptual Graphs for Source Code

As in Arís 1.0, conceptual graphs will still be used to represent the source code. For our project, we have almost similar conceptual graphs building task but this time we build it from source codes written in Java. Here we will explain the concept and relation types that are allowed in Arís 2.1 (which is slightly different with Arís 1.0). The process of constructing the conceptual graphs will be discussed in Section 4.2.

[Gri13] explains a taxonomy that allows the construction of conceptual graphs from source code for the source code matching module of Arís. After studying the method described by [MR04], [Gri13] extended the partial support model given for representing C source code file so it is adapted to the C# programming language. Here we describe the concepts and relation that are allowed in the graphs (the set of concept types) and how they can connect to each other (the set of relation types) and the possible referents for each concept type. Even though the support she made was originally designed for C# programming language, for this project we will still use the same support (this is not so surprising because C# and Java structure can be seen as similar).

Concept Type	Description
AssignOp	An assignment of a value to a field or variable (also including assignments such as "+=", "*=")
Block	A structurally grouped of set of concepts (for example, code that is inside curly brackets ...)
Class	A declaration or definition of a class
CompareOp	A binary comparison operator such as "<=", "!=", etc
Enum	A declaration of an enumeration of values
Field	A declaration of a variable that belongs to a class (a class attribute)
If	A conditional branch statement
LogicalOp	A binary logical operation, for example "||", "&&", etc
Loop	An iterative process that depends on a condition
MathOp	A mathematical operation like "+", "*", etc
Method	A declaration or definition of a procedure inside a class
MethodCall	A method invocation (execution)
NameSpace	Defines a scope that can contain one or more classes. Useful for code organization. We restrict this usage for package definition in Java
Null	A null reference (keyword null in Java)
String	A textual entity (numbers are also represented as strings)
Switch	A conditional statement that has multiple branches
Try-Catch	A block of try statement followed by one or more catch statement, which specify handlers for different exceptions
Variable	An entity declared in the program that holds values during execution

Table 3.1: Set of Concept Types

Table 3.1 shows the set of concept types with the brief meaning of every concept. The hierarchy of the concepts can be seen in the Figure 3.3 below.

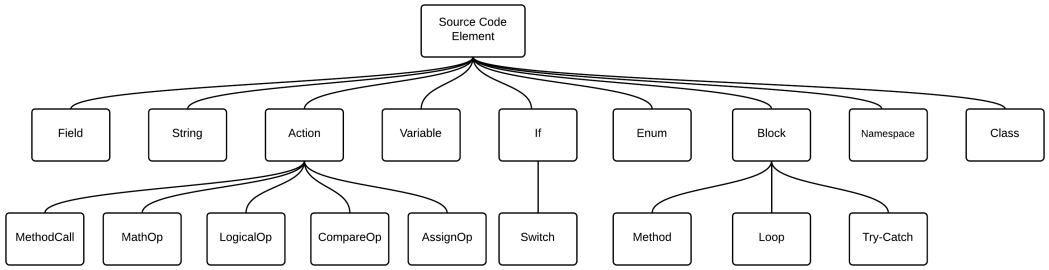

Figure 3.3: Hierarchy of Source Code Concept

The type of referents for each concept type are presented here:

1. Variable, Field, Enum, Method, Class, and MethodCall will always have an individual referent which is a programming language identifier
2. String will always have an individual referent which can be text of any length
3. Block can have a generic ("*") referent or an individual one
4. All other concepts can have only generic ("*") referent

Next, [Gri13] also presented the set of relation types that can be seen in Table 3.2. Basically [Gri13] defines six relation types in her conceptual graph formalism: Condition, Contains, Defines, Depends, Parameter, and Returns. For our research which related to the source code written in Java programming language, we ignore two relation types because they are not applicable in the Java code structure. Those two relation types are:

- *Defines*: a Block concept that gives a definition of a Namespace concept
- *Depends*: a Block concept that depends on another Namespace concept

Relation Type	From Concept	To Concept	Description
Condition	If	Action	Describes the conditional statement within the if clause
		String	
		Variable	
		Field	
	Loop	Action	Specifies the conditional statement that determines the iterative process
		String	
		Variable	
		Field	
Contains	Action	Action	The concepts that can be used by an Action concept
		String	
		Variable	
		Field	
	Block	Action	The concepts that can be used by a Block concept
		String	
		Variable	
		If	
		Enum	
		TryCatch	
	Class	Field	The concepts that can be used by a Class concept
		Method	
	Enum	String	The enumeration concept is defined as string
	Method	Block	The method concept uses Block concept
	If	Action	Branching can contain another Block or a single statement
		Block	
	Loop	Action	The Loop concept can use these concepts
		Block	
		Variable	
		Field	
	Namespace	Class	The Namespace concept utilises Class concept
Parameter	Method	String	Parameter of a method can be a String or a Variable
		Variable	
	MethodCall	String	The method is called with these concepts as parameter
		Variable	
		Field	
Returns	Method	Action	The Method concepts returns these concepts
		Variable	
		Field	
		String	

Table 3.2: Set of Relation Types

3.3 Conceptual Graph Interchange Format (CGIF)

The Common Logic (CL) [sfCL] is a framework for a family of logic based languages with the purpose of standardizing syntax and semantics for information exchange and transmission. As far as possible, this information exchange and transmission will require no translation, and when the translation is needed, CL provides a single common semantic framework. Due to CL supports first order predicate logic, it can be utilised for exchanging first order formulas in a standardized form as well.

There are three dialects that are capable of expressing CL semantics [Sow08]:

1. the Conceptual Graph Interchange Format (CGIF), a linear notation (in text version) for representing conceptual graphs.
2. the Common Logic Interchange Format (CLIF), the representation of CL syntax which has some similarity with the Knowledge Interchange Format (KIF).
3. the XML-based notation for Common Logic (XCL), which is a straightforward mapping of the CL syntax and semantics into an XML form.

For the conceptual graph in Figure 3.1, the CGIF can be defined as follow:

```
[Go *a] [Person: Jim] [City: Dublin] [Plane *b]
(Agnt ?a Jim) (Dest ?a Dublin) (Inst ?a ?b)
```

In CGIF, a concept is characterised by square brackets while a relation is characterised by parentheses. A character string prefixed with an asterisk, such as `*a`, indicates a defining label. A defining label may be referred by the same string prefixed with a question mark, `?a`. These strings represent co-reference labels in CGIF. In CLIF, variables will correspond to the defining labels in CGIF. Unless prefixed with the symbol `@every`, a defining label will be translated to an existential quantifier in CLIF.

The same conceptual graph if written in CLIF will be as follow:

```
(exists ((a Go) (b Plane))
(and (Person Jim) (City Dublin)
(Agnt a Jim) (Dest a Dublin) (Inst a b) ))
```

Using XCL syntax, the same CLIF will be written as follow:

```
<?xml version="1.0"?>
   <text xmlns="http://purl.org/xcl/1.0/"
        dialect="http://purl.org/xcl/1.0/#dialect-clif">
     (exists ((a Go) (b Plane)) (and (Person Jim) (City Dublin)
     (Agnt a Jim) (Dest a Dublin) (Inst a b) ))
   </text>
```

As we can see from the two representation, CGIF and CLIF have a look alike representation. Despite the similarity between CGIF and CLIF, there are several differences between them [Sow08]:

- Because CGIF is the serialized representation of conceptual graph, labels such as `a` or `b` depict connections between nodes in CGIF, meanwhile they denote variables in CLIF or first order predicate logic.
- CGIF uses the prefixes `*` and `?` to distinguish co-reference labels from constants, but CLIF does not use any syntactic convention for differentiating variables and constants.

- Due to there is no inherent ordering in the nodes of a graph, a CGIF sentence is an unordered list of nodes. Reordering the sentence will not affect the semantics.
- Because the nature of conjunction of nodes in CGIF is stated implicitly, there is no need to use additional operator such as `and` as in CLIF.

CGIF, CLIF, and XCL can be used as a representation of a conceptual graph because they are coming from the same CL framework. The advantage of using CGIF is it is the direct dialect of CL that describes a conceptual graph. Even though CGIF has a lot of sophisticated features dealing with representing conceptual graphs that we do not discuss here, we will not need all those features in this research. We will only use basic representation of conceptual graphs which denotes a concept with square brackets and a relation with parentheses to represent our source code conceptual graphs. We will not use CLIF to represent our conceptual graph because in this research the first order logic property of our conceptual graph is not in our main interest. The same with XCL, even though it can represent the same information, however we do not want the representation of our conceptual graphs become more complex (by using tags and DTD). The use of CGIF related to the conceptual graph of a source code in this research will be explained deeper in Section 4.3.

3.4 Summary

In this chapter we discuss the knowledge background and terminologies that we will use to build our system. We discuss the conceptual graph and how we use it to represent the source code files. We also define the set of concepts and relations that we allow in building the code conceptual graph. The same set of concepts for building conceptual graph for C# programming language still can be used for Java programming language.

We also discussed the conceptual graph interchange format that will be useful for our system to represent the source code conceptual graph. This representation that denotes the source code written in another language (in this case Java) will be used by Arís to find the similar implementation and then transferred the specification to the target code.

According to [Gri13], the conceptual graph which is built by Arís 1.0 does not capture any information about the order in which the statements are executed. Using the CGIF that we will explain more in the next section, we will preserve the order of the statements execution or occurrence. Another difference aspect in this project is, related to the order mentioned earlier, we also make a model of "if-else if-else" statement in our conceptual graph (even though we do not define a new conceptual type for it).

Chapter 4

Solution Design and Implementation

In the previous chapter we have discussed the terminologies that we use to build our system such as conceptual graph and conceptual graph interchange format (CGIF). In Section 2.5 we have talked about several approaches that we might choose when we develop Arís 2.1 with their advantages and disadvantages as well. In this chapter we will explain the architecture of Arís 2.1 based on the approach that we choose. This chapter will also explain how the conceptual graphs can be constructed from the source code files. We will also mention some differences we encounter when we build the conceptual graph for Java source code. After that, we explain how the conceptual graphs are written in the CGIF and we give one example of this CGIF. Next, we discuss how we reconstruct the conceptual graph from a CGIF file. Finally, we will explain how we transform the specification written in Spec# syntax to JML.

4.1 Arís 2.1 Solution Design

As we have mentioned earlier that software verification is important to ensure the reliability of the software, especially when the faults in the software can cause catastrophic impact or expensive consequences. However, even though writing a verifiable software is regarded as substantial, when a software is being built the developers seldom write the specificationts because it is difficult and laborious.

To overcome this issue, there are several tools that have been developed to generate specifications automatically. However, these tools still have some disadvantages. For example, one tool can generate the likely specifications only if we provide a modest and abundant test cases. The quality of the resulting specification then really depends on the quality of the test cases. Another tool can generate the loop invariant in a method or function given the preconditions and postconditions beforehand. But back again, writing the specifications (in this case the preconditions and postconditions) in the first place is not an easy task.

We come to the solution of writing specifications by trying to transfer existing specifications found in a similar implementation with our target code. A well performed system called Arís 1.0 that facilitate us to get a similar code written in C# from the given one and transfer the specification from this base code (the retrieved one) to the target code (the given one) has been developed. We will develop Arís 2.1 so Arís can accept input and output written in another programming language, which in this case is Java.

We have defined in Section 2.5 there are three approaches that we can use to develop our new system. These three approaches are:

1. Build Arís 2.1 from the very beginning

2. Convert the source code written in another programming language and feed it to Arís 2.0 and then convert the resulting specification

3. Build extension in Arís 2.0 to receive the conceptual graph of the code and then convert the resulting specification to the particular programming language

From the three approaches, the last one is the most promising one because we do not need to construct the system from the beginning and we do not need to rebuild our new system if we want it to accept input in another new programming language. Another plus point is we are not dependent to some already defined code converter. The construction of the conceptual graph of the target code is actually inevitable because every programming language has their own way to represent their abstract syntax tree. Based on this considerations, we decide to build our new system using the third approach.

Our new system will be implemented on top of Arís 2.0, which is being developed by [Hal14], by adding some new functions. We will use Java programming language as the target code language as we mentioned in Section 3.2. The conceptual graphs of the Java source code will be generated and then will be mapped to the Conceptual Graph Interchange Format (CGIF) as we have discussed in Section 3.3. Then, a new function in Arís 2.1 will read the CGIF and reconstruct it into the conceptual graph recognised by Arís. Using this approach we can feed Arís with every programming language source code because our concern now is only to representing the conceptual graph of the source code. Another new function will be added to write the transferred specification to a file after Arís finds the similar implementation. And the last function will be inside the new system that read the file and transform the specification written in Spec# to JML.

We explain our system design in the subsections follow. We will provide the Use Case Diagram and the Class Diagram for our system.

4.1.1 Use Case Diagram

Figure 4.1 below shows the Use Diagram of Arís 2.1. There are three use cases that user can directly access through the system: Generate CGIF from Java Code, Load the CGIF into Arís, and Get the Transferred Specification from Spec# to JML.

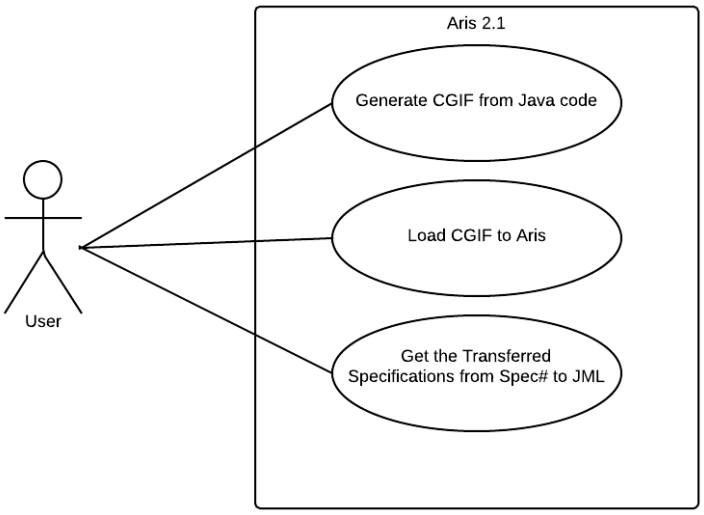

Figure 4.1: Arís 2.1 Use Case Diagram

4.1.2 Class Diagram

Figure 4.2 shows the class diagram of Arís 2.1. These six classes are the major classes that we build for Arís 2.1. There are many other classes that we do not put in the class diagram because we derived it from the Arís 2.0 classes (e.g. class for representing a loop concept, class for representing a field concept, etc). The RebuildFromCGIF class is the new class that we build and put in Arís 2.0, while the rest of the classes are made for the Java part of the system.

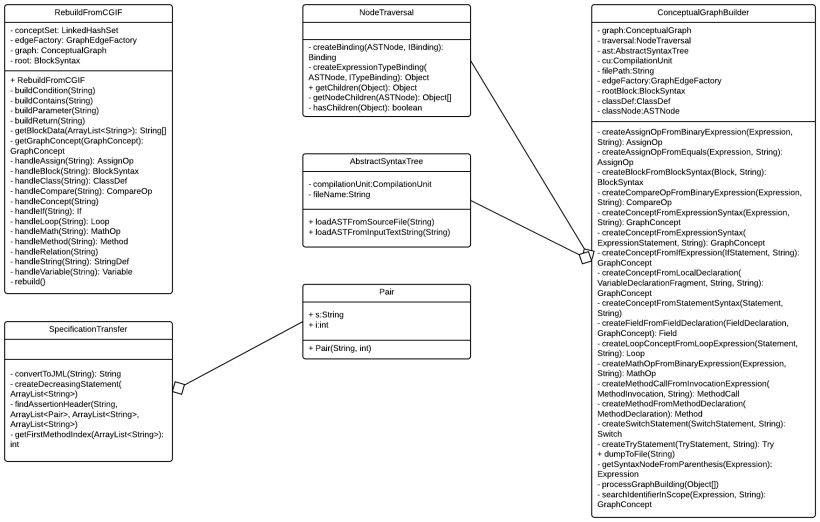

Figure 4.2: Arís 2.1 Class Diagram

4.1.3 Arís 2.1 Architecture

We design the architecture of Arís 2.1 as shown in Figure 4.3 below. The orange color boxes are the new functions that we develop for this research while the gray color boxes are the existing functions in Arís 2.0.

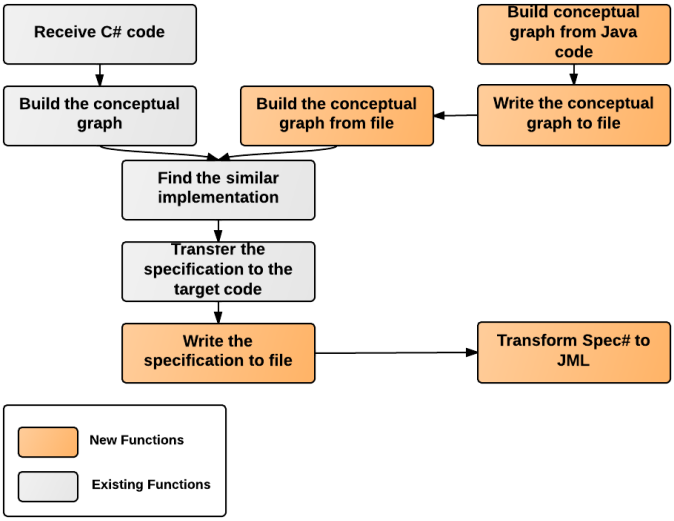

Figure 4.3: Arís 2.1 Architecture Diagram

Departing from the architecture design, the flow diagram below shows the process from the input Java file until the output JML file.

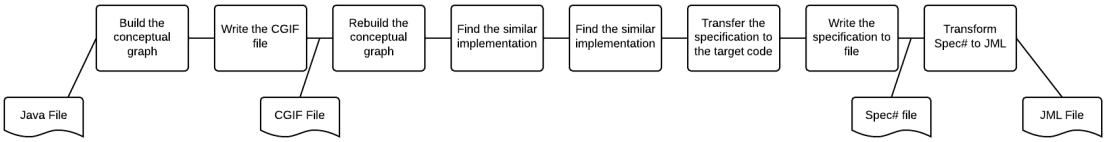

Figure 4.4: Arís 2.1 Input Output

4.2 Representing Source Code as Conceptual Graphs

As Arís 1.0 system requires a tool to parse the source code written in C#, we also need the same tool that can parse the source code written in Java programming language. We found that we can use Eclipse JDT (Java Development Tools) [Fou] to access the Abstract Syntax Tree (AST) representation of the source code. This tool is quite similar to the Microsoft Roslyn Project [Mica] which is used in Arís. This Eclipse JDT tool also has another plug-in called ASTView which resembles with Roslyn Syntax Visualizer. For the Eclipse JDT library, we need to include the Java Archive (jar) file to our project and import all the classes we need in our project. Meanwhile to use ASTView we need to import it to our Eclipse IDE and call the function by clicking this plugin button.

To build the source code conceptual graph, we still use the same approach with Arís 1.0 (explained by [Gri13]) which traversing from the root of AST node through all the descendant nodes in a depth first search manner to construct the concepts and relations in the conceptual graph. Figure 4.5 shows the abstract syntax tree snippet generated by ASTView for class Tes.java given below.

```
1  public class Tes {
2    public static int sum(int x)
3    {
4      int add = 0;
5      int k = 0;
6      while(k < x)
7      {
8        add += k;
9        k++;
10     }
11     return add;
12   }
13 }
```

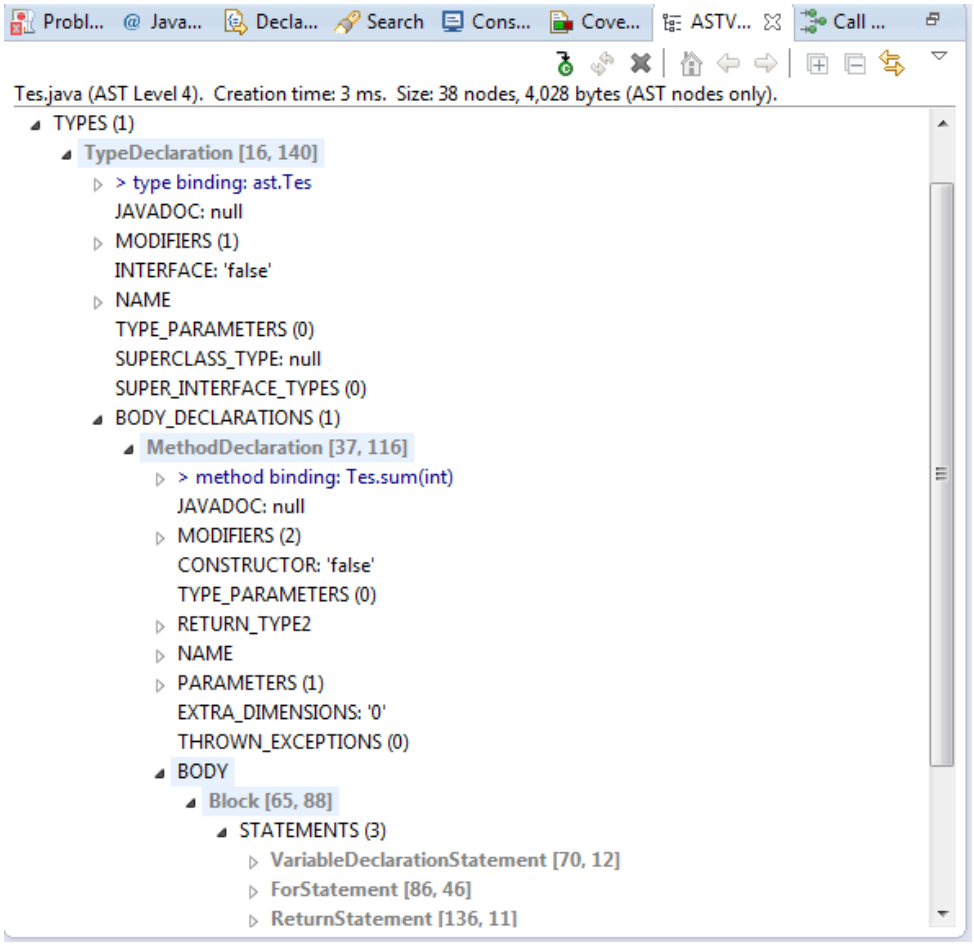

Figure 4.5: ASTView for Class Tes

The ASTView that is shown in the Figure 4.5 gives us an insight that Eclipse JDT gives a very detailed information related to the source code abstract syntax tree such as binding information of a variable, erasure of the class, etc (if we click one node of ASTView it will expand and show a thorough hierarchy under that node). We ignore most of the information because we do not need them due to we only concern with the high level information about each node (a shallow conceptual graph).

We use an open source JGraphT [Nav] library to build the graph structure for each node of interest in the abstract syntax tree. JGraphT is almost similar with QuickGraph [Micb] library for C# which is used by Arís. Similar with Eclipse JDT library, we need to include the jar file to our project and import all the classes we need in our project. The type of the graph that we use to construct the conceptual graph is a directed graph with the graph concepts form the vertex and two graph concepts make an edge. JGraphT also provides graph algorithms such as `BreadthFirstIterator` that we find very useful when we build the CGIF file after we construct the conceptual graph.

4.3 Representing Conceptual Graphs as Conceptual Graph Interchange Format

The next step after building the source code conceptual graphs is transforming those graphs into the conceptual graph interchange format and write the result to a file. This CGIF file then will be read by another function in Arís 2.1 and another conceptual graph will be constructed using Arís conceptual

graph representation to get the similar code in the system.

The CGIF construction is done by writing every node information starting from the root of the graph using a breadth first search manner. It is important to give the details about which node relates to which node because this CGIF will be read by Arís to build another conceptual graph. Different with conceptual graph construction in Arís 1.0 which often ignores the incoming and outcoming concepts for a relation, this time all the information inside a concept graph must be filled, including the leftside and rightside properties of some graph concepts. This way we can preserve the order of the graph concept and do not lose the connection information when we rebuild the conceptual graph.

The partial output of conceptual graph for class Tes in Section 4.2 can be seen below.

```
1  [Block: Tes.java Value: public static int sum( int x){
2      int add=0;
3      int k=0;
4      while (k < x) {
5        add+=k;
6        k++;
7      }
8      return add;
9    }
10 ]
11 (Contains InConcept: Block: Tes.java OutConcept: Class: Tes)
12 [Class: Tes]
13 (Contains InConcept: Class: Tes OutConcept: Method: sum)
14 [Method: sum Modifiers: public static ReturnType: int ParameterList: (int x)]
15 (Parameter InConcept: Method: sum Modifiers: public static ReturnType: int ParameterList: (int x
       ) OutConcept: Variable: x)
16 (Contains InConcept: Method: sum OutConcept: Block: Method: sum)
```

One difference between building abstract syntax tree for Java code using Eclipse JDT and building abstract syntax tree for C# code using Roslyn is Eclipse JDT will not recognize and construct the abstract syntax tree of a program if it is not written inside a class. Meanwhile Roslyn can accept a C# file which contains only a method without a class definition where this method belongs to and build the abstract syntax tree of that file. Because Arís 1.0 takes methods as the basic building block of the source code (as we mentioned in Subsection 2.4.2), this difference should be eliminated when the CGIF file is written.

Due to this difference, we have to eliminate the appearance of class definition in the CGIF so that when Arís reconstructs the conceptual graph the result will be in the same structure (that is the conceptual graph will have a relation connecting the root block to a method, not to a class concept). In the CGIF above, we somehow have to modify the conceptual graphs on line 11, line 12, and line 13 into a single Contains relation. By observing the abstract syntax tree structure of a Java code, we decide to remove the concept information on line 11 and 12 and modify the InConcept property inside the relation on line 13 to hold the root block information. This process will be the same for every Java source code so we do not need to change this CGIF altering procedure for a different Java source code.

Our algorithm to generate the final CGIF can be seen from the code snippet below.

```
1     public void buildCGIF(String fileName){
2       BreadthFirstIterator<GraphConcept, GraphEdge> it = new BreadthFirstIterator<GraphConcept,
            GraphEdge>(graph, rootBlock);
3
```

```
4       try{
5           int i = 0;
6           while(it.hasNext()){
7           GraphConcept g = it.next();
8
9           if(i == 0)
10          //get the root block
11          if(i == 1 || i == 2)
12          //skip the second and the third graph
13          if(i == 3)
14            //modify InConcept to what we have in the root block
15
16          if(!g.isRelation())
17          //write the graph content inside square brackets
18          else
19          //write the graph content inside parenthesis
20          ++i;
21        }
22      //save to file
23    }
```

Using the algorithm in the code snippet above, the final partial CGIF for class Tes can be seen in the listing below.

```
1  [Block: Tes.java Value: public static int sum( int x){
2      int add=0;
3      int k=0;
4      while (k < x) {
5        add+=k;
6        k++;
7      }
8      return add;
9    }
10 ]
11 (Contains InConcept: Block: Tes.java Value: public static int sum( int x){
12     int add=0;
13     int k=0;
14     while (k < x) {
15       add+=k;
16       k++;
17     }
18     return add;
19   }
20   OutConcept: Method: sum)
21 [Method: sum Modifiers: public static ReturnType: int ParameterList: (int x)]
22 (Parameter InConcept: Method: sum Modifiers: public static ReturnType: int ParameterList: (int x
       ) OutConcept: Variable: x)
23 (Contains InConcept: Method: sum OutConcept: Block: Method: sum)
```

As we can see from the two partial CGIFs above, our representation still conform with the CGIF concept we discussed earlier. For every concept, all its attributes are exported in order to get detail

information and to differentiate between two such concept with similar type and name (if there is any). These attributes are sometimes really important to get the right order of execution, for example in [`MathOp: += LeftSide: Variable: add RightSide: Variable: k`] concept, we know that the actual code execution order should be `add += k`. And for every relation, we also export its InConcept and OutConcept attribute values. Due to the construction of the conceptual graphs stores this InConcept and OutConcept value, our relation format in CGIF does not have any co-reference label (and it is also because our Concepts are all have individual so we do not have defining labels anymore).

4.4 Build Conceptual Graph from CGIF File

After the CGIF file has been created, one new function in Arís will read that file and reconstruct the conceptual graph based on that CGIF file. The process of reconstructing the conceptual graph is achieved by reading the CGIF sequentially line by line. Algorithm 1 shows how the reconstruction process is done.

```
read the first concept in CGIF file;
create the root concept and add to the graph;
while not at the end of CGIF file do
    read one line;
    if the line starts with "[" then
        put into conceptSet;
    else
        put into relationList;
    end
end
take the first relation from the relationList;
build the relation concept and add to the graph;
for i ← 2 to Size(relationList) − 1 do
    take the relationList[i];
    build the relation concept and add to the graph;
end
calculate node rank in the graph;
```

Algorithm 1: Conceptual Graph Construction from CGIF

From the algorithm above, there are two steps that quite important so they must be treated separately. The first is the construction of the root. Because the root concept can span to several lines, the process of reading the first concept in the CGIF file is done until we find the first closing square bracket. The process continues to get the name and the value of the root concept (as can be seen from the CGIF in Section 4.3). The second important process is the construction of the first relation. The root and the first relation are important because these two things should have been created and connected in the graph before we can process the rest of the line in the CGIF file.

The core process when constructing the conceptual graphs is in the last `for` loop. Here we try to construct the relation node for every element in the `relationList`. Because a relation connects two concepts, in this same process first we construct the concept nodes connected by this relation and then we construct the relation. The construction of the concept node will refer to the list of concepts in the `conceptSet`. The final step after constructing the whole conceptual graph is calculating the node rank in the graph. This process is done by calling an existing function that has been developed since Arís 1.0.

4.4.1 Computing the Node Rank

Since Arís 1.0, the Node Rank, coined by [BINF12], is the measure of the structural importance of a node in the conceptual graph. The weight of a node is related to the amount of ongoing and outgoing edges to or from it.

The calculation of the node ranks in a graph is as follow [Gri13]: we define u as a node in the graph, *NR(u)* is its node rank, *IN(u)* is the set containing all the nodes that have an outgoing edge into u and *OutDegree(u)* is the number of edges going out of the node u. As initial value, all nodes will be given an equal node rank:

$$\frac{1}{\textit{the total number of nodes}}$$

A process to calculate the new *NR(u)* iteratively is defined as the sum over all $v \in$ *IN(u)*:

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

The process will stop if the value for every node has been convergence, that is when the difference between the old value and the current one is less than or equal to a quadratic error factor, in this case the value is set to 0.001, or the iteration has been exceed the limit, in this case the limit is 50. After each iteration, in order to enable the convergence, the node ranks value will be normalized so that their sum will be equal to one.

4.5 Write the Specification to File

Once Arís has found the similar code and transferred the specification, our next process is to write the code with its specification into a file. For this process, we need to add some notation when we write it so the program that will transform the specification into JML knows when it reads an ordinary line of code or when it reads a specification. We use `//spec:` mark to denote the line contains specification. The process of adding this annotation is done in the same process when Arís is transferring the specification. For each specification line, we modify it by concatenating the header mark in front of it. After all specifications have been transferred, we continue the process by writing all the lines in of code into a file.

4.6 Transforming Specification into JML

After the code with the Spec# specification has been written to a file, the next task is to read it and transform the specification into JML syntax. The algorithm that we use to transfer the specification can be seen in Algoritm 2 below.

```
put source code in sourceList line by line;
put target code in targetList line by line;
while not at the end of sourceList do
    read one line;
    if the line starts with "//spec:" then
        if it is a "requires", "ensures", or "modifies" then
            convert to JML syntax and put in prepostlist;
        else if it is an "invariant" then
            if invariant dictionary does not have this loop yet then
                convert to JML and put it in a list;
                add to the dictionary with the loop as key and the list as value;
            else
                convert to JML;
                get the list from the dictionary and append the specification;
            end
        else if it is an "assert" or "assume" then
            go back until we find an opening curly bracket (the block where it belongs);
            convert to JML and put it in the assert/assume list;
end
go to method declaration in the target code;
fill all preconditions, postconditions, and modifies clause from the prepostlist before the method
declaration;
fill all invariants from the invariant dictionary before the loop declaration;
fill all assertions and assumptions from the assert/assume list to the block where they belong;
create the variant function for the loop;
```

Algorithm 2: Specification Transferring Algorithm

For class Tes that we show in Section 4.2, we find the similar source code with specification shown in the listing below.

```
1  public static int Sum(int k)
2  requires 0 <= k;
3  ensures result == sum{int i in(0:k); i};
4  {
5    int s = 0;
6    for(int n = 0; n < k; n++)
7    invariant n <= k;
8    invariant s == sum{int i in(0:n); i};
9    {
10     s += n;
11   }
12   return s;
13 }
```

The resulting of specification mapping and transferring with annotation can be seen in the listing below.

```
1  public static int sum( int x)
2  //spec:requires 0 <= x;
```

```
3  //spec:ensures result == sum{int i in(0:x); i};
4  {
5      int add=0;
6      int k=0;
7      while (k < x) {
8  //spec: invariant k <= x;
9  //spec: invariant add == sum{int i in(0:k); i};
10       add+=k;
11       k++;
12     }
13     return add;
14 }
```

Following Algorithm 2, we map and transform the specification in the previous listing to the target code with the result shown in the listing below.

```
1  public class Tes{
2
3    //@ requires 0 <= x;
4    //@ ensures \result == (\sum int i; 0 <= i && i < x; i);
5    public static int sum(int x)
6    {
7      int add = 0;
8      int k = 0;
9      //@ maintaining k <= x;
10     //@ maintaining add == (\sum int i; 0 <= i && i < k; i);
11     //@ (* decreasing x - k *);
12     while(k < x)
13     {
14       add += k;
15       k++;
16     }
17     return add;
18   }
19 }
```

As we have seen in Section 2.1, there are differences between Spec# and JML syntax, for example between `old` and `\old`, and between `requires` and `\requires`. The core process that transforming this Spec# syntax into JML is in the `convert to JML` part. In this process, we define the mapping between Spec# keyword and JML (e.g. `requires` and `\requires`, `modifies` and `\assignable`) and also the converting process of quantifier between the two specifications (e.g. `sum{int i in(0:k); i}` and `(\sum int i; 0 <= i && i < k; i)`).

Another different point in JML is it has the variant function that does not exist in Spec#. When writing the specification for the loop, it is expected that we write the variant function in order to define how the loop will terminate. In JML this variant function is defined using the keyword `decreasing`. For our implementation, we try to make the variant function inside JML comment because we only make the variant function appear in the final solution if the condition of the loop is in a simple form i.e. the termination of the loop is marked by two primitive variables separated by the comparison operator such as <=, <, >=, >. If the condition is in the form other than that (e.g. related to array or a combined of several conditions) then our procedure just write nothing as the variant function.

4.7 Implementation Issue

There are several differences that we find between Arís (version 1.0 and 2.0) and Arís 2.1 when we build the conceptual graph:

4.7.1 No Class Source Code

As we have mentioned in Section 4.3 above, building abstract syntax tree for Java program using Eclipse JDT requires the code to be put inside a class. However, for building abstract syntax tree using Roslyn for C# code does not really requires the same thing. In order for our conceptual graph reconstruction will be the same, we have to modify the process of generating CGIF as we also have discussed in Section 4.3.

4.7.2 No Update Part of a `For` Loop

Another difference is in Arís 1.0 the update part of a `for` loop is not captured when the conceptual graphs is built. However, this part is an important one in a loop especially if we try to compare between a for loop and a while loop because this part contains the increment or decrement process that will bring the loop into its termination. This issue has been resolved in Arís 2.0 and Arís 2.1.

4.7.3 No `Else If` Block

A further difference is there is no Else If block in Arís 1.0. So when the system is building the conceptual graph from the source code which contains an `Else If` block, the system will catch an exception but continue to run. In Arís 2.1, because we need to write a complete information from the Java code into CGIF, we have to handle the occurrence of `Else If` block. For the sake of compatibility when comparing the two conceptual graphs, this issue has been solved in Arís 2.0.

4.7.4 Accessing Array Element

When accessing an array element, such as A[i] = A[j], Arís 1.0 and 2.0 capture the information of the assigning of the array element in the Assign object. However, the information of left side and right side of that Assign object is missing (the left side in this example is A[i] and the right side is A[j]). When implementing the same process for Arís 2.1, we decide to weaken the constraint for the new system so we also ignore the left side and the right side information if this Assign object.

4.8 Summary

In this chapter we continued the discussion from Section 2.5 and after considering the advantages and the disadvantages, we choose the approach to feed Arís a conceptual graph and process the specification mapping and transfer from that graph. Using this approach, we do not need to rebuild Arís the next time we want it to find the similar implementation for a source code written in a new programming language.

We also discussed the architecture design of Arís 2.1 and its implementation. The solutions consist of building the conceptual graphs from Java code perspective, writing the conceptual graphs in a standard format that can be read by Arís, rebuilding the conceptual graph from CGIF, writing the specification to a file, and transferring and converting the specification from Spec# to JML syntax.

We also mentioned the issues that we encountered when developing the system and the approach that we took to overcome them.

From this implementation, we will talk and discuss the results that we get and how we evaluate them in the next section.

Chapter 5

Evaluation

In this chapter we present the testing environment that we conduct to evaluate our system. We will follow the approaches of testing defined by [Gri13] that consist of finding similarities between identical, slightly modified and totally different input code.

5.1 The Nature of the Evaluation

We conduct our evalution to measure the performance of the system being developed based on the approach used by [Gri13]. Before we explain more about the document corpus and the result, we need to define the limitation of our system as follow:

1. the program code that will be the target code is written in object oriented language structure, in this case Java. For current Arís, we do not concern about program written in other paradigm, for example functional programming, because programming by contract coined by [Mey97] is intended for program written in object oriented structure.

2. the specifications keyword that we use are related to preconditions (`requires`), postconditions (`ensures`), framing condition (`modifies/assignable`), and invariants (`invariant/maintaining`). We also cover the operators such as `forall`, `exists`, `sum`, `product`, `min`, and `max`.

3. to check the correctness of the transferred specifications, we need a verifier related to the specification language. For Spec#, the verifier can be used directly from the Spec# compiler. Meanwhile, for JML, we need to use a separate verifier where in this project we will use ESC/Java2 integrated with Mobius (as we have mentioned in Subsection 2.2.2.

Even though our base of evaluation is from the work of [Gri13], we will not cover all the tests mentioned there. There are two tests that we will skip because it is not related to our system and have been done for building Arís 1.0:

1. no parameter optimization. This experiment is used to get the best parameters utilised in the analogical mapping process (e.g. match depth, weight for similarity functions and the valid match threshold).

2. no evaluating node rank impact on the mapping process. This experiment is useful to measure the effectiveness of using Node Rank metric, which is used for sorting the nodes in conceptual graphs based on their importance value.

5.2 Testing Corpus

Before we can evaluate the performance of Arís 2.1 in generating the conceptual graph and converting the specifications into JML, we need to build our document corpus that suitable for this task. The corpus should contain a collection of source code files which only have method body because Arís currently only concerns with method matching. These source files should be formally specified in Spec# as they will be the database of specified and verified code.

Our corpus will be based on the corpus used by [Gri13]. She populated the corpus with 102 codes taken from CodePlex Spec# test suits. Inside these 102 files there are 249 formally verified methods with approximately 7,470 lines of code. From these 102 files, 16 are randomly chosen and then used as our testing corpus. These 16 documents are the ones who the conceptual graph can be built either using Arís 2.0 or Arís 2.1. For example, there is one document who does partition. But when we put this document as the base code and we run the system, Arís will catch an exception and stop eventually so the mapping and specification transferring can not be achieved.

After choosing the intended documents from the document corpus, we then strip all specifications inside every files and then converted to Java. Here we convert manually the C# code into Java because mostly the structure and the library used in these documents are not that complex.

We will have four sets of data set according the similarity of the code to the base code: identical ones, the ones with small modification, medium modification, and large modification (completely different from the base code).

To perform the evaluation of modified document, from these 16 files we will transform by adding or removing the part of the code. The types of modification is described as follow [Gri13]:

1. small modification, including the action of renaming variables, methods, parameters; changing the type of the variables; rewriting for loop into while loop; adding/removing comments in the code
2. medium modification, including the action of reordering statements as long as it does not affect the functionality; declaring unused variables; removing certain statements (again as long as it does not affect the functionality)
3. large modification: changing the functionality of the program so it behaves completely different

5.3 Evaluating the Specifications Transferring

In this section we will explain the documents that we use for our testing and the result after performing the test.

5.3.1 Testing for Identical Documents

We perform this testing to check whether the system can recognize two identical code and so transfer the specifications directly. From the 16 files, all the specifications can be transferred to the target code. The highest similarity score we get is 0.8235. From these 16 targets, only three of them can be verified using ESC/Java2. One from these 16 code has partial specification so when it is transferred to the target, the target also suffers from an incomplete specification.

5.3.2 Testing for Documents with Small Modifications

For the modified documents, we applied the rule of modification mentioned in Section 5.2 for the 16 files of code. For the small modification, all code specifications can be transferred even though we get only

two codes from the testing result that can be verified by ESC/Java2. One such code in term of the base and the target file can be seen from the two listing below.

Listing 5.1: Base Code with Specification

```
1  bool LinearSearch(int[] a, int key)
2   requires a != null;
3   ensures result == exists{int i in (0: a.Length); a[i] == key};
4   {
5     for(int i = 0; i < a.Length; i++)
6       invariant 0 <= 1 && i <= a.Length;
7       invariant forall{int j in (0: i); a[j] != key};
8       {
9         if(a[i] == key)
10    {
11          return true;
12    }
13      }
14      return false;
15  }
```

Listing 5.2: Target Code with Specification

```
1  public class LinearSearch_S{
2    //@ requires a != null;
3    //@ ensures \result == (\exists int i; 0 <= i && i < a.length; a[i] == key);
4  boolean LinearSearchWhile( int[] a, int key){
5      int i=0;
6      //@ maintaining 0 <= 1 && i <= a.length;
7      //@ maintaining (\forall int j; 0 <= j && j < i; a[j] != key);
8      //@ (* decreasing a.length - i *);
9      while (i < a.length) {
10       if (a[i] == key) {
11         return true;
12       }
13       i++;
14     }
15     return false;
16   }
17 }
```

5.3.3 Testing for Documents with Medium Modifications

For codes with medium modifications, we run the test and get only one code that can be verified using ESC/Java2, even though for all code files the specification can be transferred successfully. The code can be seen in the listings below.

Listing 5.3: Base Code with Specification

```
1   int ISqrt(int x)
2     requires 0 <= x;
3     ensures result*result <= x && x < (result+1)*(result+1);
```

```
4   {
5     int r = 0;
6     while ((r+1)*(r+1) <= x)
7       invariant r*r <= x;
8     {
9       r++;
10    }
11    return r;
12  }
```

Listing 5.4: Target Code with Specification

```
1  public class ISqrt_M{
2    //@ requires 0 <= x;
3    //@ ensures \result*\result <= x && x < (\result+1)*(\result+1);
4  int ISqrtFor( int x){
5     int mew=0;
6     int r=0;
7     //@ maintaining r*r <= x;
8     //@ (* decreasing x - (r + 1) * (r + 1) *);
9     for (; (r + 1) * (r + 1) <= x; r++) {
10      mew++;
11    }
12    return r;
13  }
14 }
```

5.4 Analysis of the Testing Result

Looking at the result we have got so far, we try to examine the problems in order to find the solution and fix it.

5.4.1 Inconsistency When Building the Conceptual Graph

One thing that can make the code unverified is the mapping between base and target code. As we see from the Listing 5.5 below, variable `r` has two value that must be true inside the loop as stated in the loop invariant. Actually, the second value of `r` should be used by the variable `x`. We have tried to investigate this but we still can not find the source of the problem. Our main suspect for the source of this inconsistency is when we rebuild the conceptual graph from the CGIF file. Even though we already modify the way Arís 2.0 creating the conceptual graph to conform the way Arís 2.1 does, we still can not get the exact equals of two such conceptual graphs. We need to re-investigate this part in the future.

Listing 5.5: Mapping Error in Specification

```
1  public class Square{
2    //@ requires 0 <= n;
3    //@ ensures \result == n*n;
4    public int Square( int n){
5      int r=0;
6      int x=1;
```

```
7       //@ maintaining i <= n;
8       //@ maintaining r == i*i;
9       //@ maintaining r == 2*i + 1;
10      //@ (* decreasing n - i *);
11      for (int i=0; i < n; i++) {
12        r+=x;
13        x+=2;
14      }
15      return r;
16    }
17  }
```

Figure 5.1 shows the representation of two identical code. The one on the left side is built from C# code meanwhile the one on the right is built from the CGIF file. Nevertheless, when the program runs and the conceptual is built, the end result shows the two code do not have the same structure even though the graph construction has been made to be the same way.

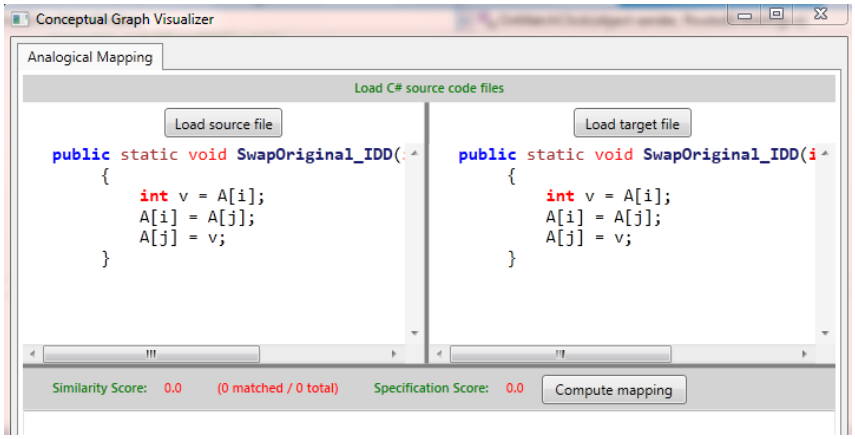

Figure 5.1: Two Identical Code

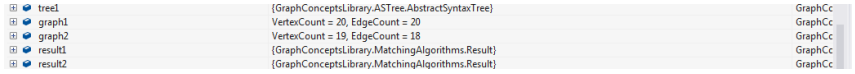

Figure 5.2: The Result of Constructing Conceptual Graphs from Two Identical Code

The Figure 5.2 just shows us that for graph1 (the code who is on the left side above) has 20 vertex and 20 edges meanwhile for graph2 (the code who is on the right side above) has 19 vertex and 18 edges.

5.4.2 Problem With Null Dereference

When dealing with object data as method arguments, Spec# compiler will assume those objects as non null value. Meanwhile, ESC/Java2 does not have that assumption so if we try to compile the transferred specification then we will get an error saying: `Possible null dereference (Null)` in the code. To handle this, we check the type of the argument in the method declaration and if it is an object type then we will add a precondition saying that the particular object is not null inside the method.

5.4.3 Problem With Specific Keyword in Spec#

In three of our test case we have `count` operator inside the Spec# code. For JML, this operator equals to the `\num_of` operator. But when tried to verify our code with ESC/Java2, we will receive error message saying that `\num_of` keyword is not recognized by the verifier. For this matter, we consider this as the verifier version that still can not verify that keyword (as a comparison, current ESC/Java2 with Mobius run in JDK 1.4 environment).

5.5 Discussion

We close this chapter with some discussion related to the result of our evaluation. Firstly, we will revisit our problem statement question: Is Arís can be reused with another programming language as the input and output and how can we reuse it? As we have seen from the evaluation, there are some cases that Arís can transfer the specification into program written in Java language. The specifications also can be verified using ESC/Java2 verifier. Our approach of using CGIF as intermediary representation between Arís and the conceptual graph written in another programming language can be considered as a good approach.

However, as we have discussed in the previous section, not all the transferred specifications can be verified. This is related to the objectives question that we asked in Section 1.4: What is the difficulty of implementing this new extension? and How does this new Arís perform? Because the main process behind Arís to match two source codes, map the similar properties of the two codes, and transfer the specification, lies on the conceptual graph representation of the codes, it is crucial to build the correct conceptual graph in order to be processed by Arís. The difficult part of building the conceptual graph between Arís made by Roslyn and Java made by Eclipse JDT is they always have some differences in representing the abstract syntax tree of the code. In this research, we already tried to make the process to be similar: by revising how Arís building the conceptual graph to conform with how the conceptual graph for Java code is built. However, we still see some difference in the end result (as can be seen from Figure 5.1 and Figure 5.2). This difference between two graph can make a great deal in the transferred specification because the variable mapped can be wrong so the specification becomes incorrect (as we have seen in Listing 5.5).

For current implementation, the correctness of the CGIF file that we create is checked by reconstructing again using the Algorithm 1 in Section 4.4. After we reconstruct the conceptual graph, we compare in a breadth-first search manner every nodes inside the two graphs. For formal correctness of the CGIF file, we consider that as beyond the scode of this research.

In relation with the tools we have discussed in Section 2.3, we think it is a better idea if we can combine Arís and the static checker that implements abstract interpretation. To illustrate this, we can go back to the problem with null dereference mentioned in Section 5.4. We can use Arís to get the specification and after that we use abstract interpretation to get a better information on missing specification (such as an array argument in a method should not be null, etc). Or we can use abstract interpretation with our target code and after we get the result we feed to Arís to get a more rich specification. Or we can run our code with Arís and abstract interpretation tools and combine the result they generate.

We have also mentioned in Section 1.4 another objective question: What approaches do we need to build a new extension for another language? For current Arís 2.1, if we want to add another plugin for a new programming language (that has to be an object oriented programming language), we have to handle the conceptual graph construction and the specification transformation process. As we have seen from the discussion above, there are difficulties in constructing the similar conceptual graphs and in the

specification transformation. Every time another plugin is made for another programming language, we have to face the same task to generate the conceptual graph and transforming the specification. If we see for the future development, we can expect that one day Arís can be as a framework or platform where all object oriented programming language can be put inside and it generates the specification related to the particular programming language.

Chapter 6

Conclusion

Our aim in this thesis project is to investigate the possibility of reusing Arís, a system that can find similar implementations from a program and transfer the formal specifications in order to aid the developer to write the specifications and to get more verified software. Our system is built upon Arís 1.0 under the work by [PGL+13] and its improvement Arís 2.1 by [Hal14].

Our solution still focuses on how to generate the conceptual graph from the new source code file written in Java programming language and transform the specification from Spec# to JML. The basic concepts when constructing the conceptual graphs are taken from [Gri13] and [Pit13] which are almost similar with the concepts for a code written in Java programming language. The algorithm to build the conceptual graph is also taken from their approach yet we will modify theirs eventually. We also use the CGIF to be our intermediary language to give the conceptual graph representation between the two language in our system: C# and Java.

As we build our conceptual graph based on the work of [Gri13] and [Pit13], it is also worth to mention several differences that we have implemented in our thesis:

- our conceptual graph for Java code enforces us to put the method inside a class meanwhile their conceptual graph can be built solely on a single method in a file. We manage to manipulate the conceptual graph representation in order to be the same with their conceptual graph construction.
- our conceptual graph deals with the update part of a for loop meanwhile their work still does not have this implementation. We update Arís 2.0 to handle this to conform with our implementation.
- we provide a way to represent `Else If` block in our conceptual graph. Again, we give an update in Arís 2.0 for this matter.
- although in the beginning we manage to process array element access in our conceptual graph, eventually we decide to weaken this property and ignore it to conform with Arís 1.0 and Arís 2.0.

Finally we evaluate our system's performance by running some test case on identical codes, small modified codes, and medium modified codes. We see the performance of our system is not really excellent. We tried to figure the source of this unexpected result as we define below:

- when we rebuild our target's conceptual graph, there is inconsistency and we still do not know yet where to fix this inconsistency (as the process of building the conceptual graph is a very tedious one).

- the specification we transform from Spec# still need to be fixed when dealing with possibly null dereference. It is because Spec# compiler put the assumption that an object argument inside a method should be not null (which is different when we deal with ESC/Java2 in our case).
- although the basic keywords in between Spec# and JML are quite similar, but in our implementation there is a keyword that we cannot verified. We consider this problem as the verifier versioning, not as our limitation.

Even though our result is not that excellent, our evaluation shows that for every test case the specification can be transferred. However, due to the graph representation can be different which causes the wrong mapping between nodes in the conceptual graph, the specification can be incorrect. Nevertheless, we can still see this as a creativity building for the programmer to think how to repair the specification from the given one.

6.1 Future Work

We have developed the similar approach in Arís 1.0 and Arís 2.0 to build a conceptual graph for a source code written in Java. Nevertheless, when we rebuild that conceptual graph for two identical code, we still do not get the same exact representation. This process is still need some investigation and refinement. We also still need to add another concept if needed to accommodate some code structure that is still not captured in the current implementation (for example how we store an access to element of an array).

Another thing to consider is on how we transforming the specification from Spec# to JML. There are still some room for improvement regarding this issue: assumption and assertion still not handled in this version. Even though we have implemented it in the system, we still do not test it yet so we do not know the performance of our implementation. It is because in this version of Arís we only consider the preconditions, postconditions, framing conditions, and the loop invariant.

Last, we hope that one day Arís can be a framework or platform to accept several (or every) object oriented languages as the input and generate the appropriate specification for the particular programming language. This improvement of Arís will not only be a big advantage in academia but also in industrial area.

Bibliography

[AP94] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Commun.*, 7(1):39–59, March 1994.

[BBC12] BBC. Rbs computer problems kept man in prison. `http://www.bbc.com/news/uk-18589280`, June 2012.

[BFdH+09] M. Barnett, M. Fahndrich, P. de Halleux, F. Logozzo, and N. Tillmann. Exploiting the synergy between automated-test-generation and programming-by-contract. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 401–402, May 2009.

[BH06] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods... ten years later. *Computer*, 39(1):40–48, Jan 2006.

[BINF12] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *International Conference on Software Engineering*, 2012.

[BLS04] M. Barnett, K. R. M. Leino, and W. Schulte. The spec sharp programming system: An overview. In *CASSIS*, 2004.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, ACM Press, 1977.

[DM95] B. Dehboney and F. Mejia. Formal development of safety-critical software systems in railway signalling. In *In Applications of Formal Methods*, 1995.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 449–458, New York, NY, USA, 2000. ACM.

[EL98] S. Easterbrook and R. Lutz. Experiences using lightweight formal methods for requirements modeling. In *IEEE Trans. on Software Eng.*, 1998.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007.

[FM10] Carlo A. Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In *Fields of Logic and Computation*, pages 277–300, 2010.

[Fou] The Eclipse Foundation. Eclipse java development tools (jdt). `http://www.eclipse.org/jdt/`.

[Gri13] Daniela Grijincu. Source code matching for reuse of formal specifications. Master's thesis, Department of Computer Science, National University of Ireland, Maynooth, 2013.

[Hal95] A. Hall. Specifying a safety-critical control system. In *IEEE Trans. on Software Eng.*, 1995.

[Hal96] A. Hall. Using formal methods to develop an atc information system. In *IEEE Software*, 1996.

[Hal14] Felicia Halim. Evaluate and benchmark arís. Master's thesis, Department of Computer Science, National University of Ireland, Maynooth, 2014.

[Hef10] Itsik Hefez. Boogie. a modular reusable verifier for object-oriented programs. University Lecture, 2010.

[HMLS07] T. Hoare, J. Misra, G. T. Leavens, and N. Shankar. The verified software initiative: A manifesto. In *ACM Computing Surveys*, Philadelphia, Pennsylvania, April 2007.

[INR05] INRIA. Mobius: Mobility, ubiquity and security. `http://www-sop.inria.fr/marelle/Mobius/mobius.inria.fr/twiki/bin/view/Summary/WebHome.html`, September 2005.

[KB88] M. T. Keane and M. Brayshaw. The incremental analogy machine: A computational model of analogy. 1988.

[KJHM94] L. R. Novick K. J. Holyoak and E. R. Melz. Component processes in analogical transfer: Mapping pattern completion and adaptation. In *Analogical Connections*, 1994.

[KLD94] Mark T. Keane, Tim Ledgeway, and Stuart Duff. Constraints on analogical mapping: A comparison of three models. *Cognitive Science*, 18(3):387–438, 1994.

[Kol93] J. Kolodner. *Case-Based Reasoning*. Morgan Kaufmann Publishers, 1993.

[KPS95] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security. Private Communication in a Public World*. Prentice Hall, New Jersey, first edition, 1995.

[Lei08] K. Rustan M. Leino. This is boogie 2, 2008.

[Log13] Francesco Logozzo. Technology for inferring contracts from code. In *Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '13, pages 13–14, New York, NY, USA, 2013. ACM.

[LPC+13] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. Jml reference manual. 2013.

[Mar] Stephen Marshall. Software engineering: Ariane 5. `http://www2.vuw.ac.nz/staff/stephen_marshall/SE/Failures/SE_Ariane.html`.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. ISE Inc., second edition, 1997.

[Mica] Microsoft. .net compiler platform ("roslyn"). http://msdn.microsoft.com/en-gb/roslyn.

[Micb] Microsoft. Quickgraph, graph data structures and algorithms for .net. http://quickgraph.codeplex.com/.

[Mis10] Alon Mishne. Jml and esc/java2. University Lecture, 2010.

[Mon12] This Is Money. Can i force natwest or rbs to cover late payment penalties or extra costs i get hit with because of its banking meltdown? http://www.thisismoney.co.uk/money/saving/article-2163238/Do-NatWest--RBS-cover-late-payment-penalties-extra-costs-caused-banking-meltdown.html, June 2012.

[Mon14] Rosemary Monahan. Evaluating software verification systems. University Seminar, 2014.

[MR04] Gilad Mishne and Maarten De Rijke. Source code retrieval using conceptual similarity. In *Proc. 2004 Conf. Computer Assisted Information Retrieval*, pages 539–554, 2004.

[Nav] Barak Naveh. Jgrapht. http://jgrapht.org/.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 229–239, New York, NY, USA, 2002. ACM.

[Onl14] Today Online. Toyota recalls 466,000 vehicles globally for spare tire, braking issues. http://www.todayonline.com/business/toyota-recalls-466000-vehicles-globally-spare-tire-braking-issues, May 2014.

[PGL+13] Mihai Pitu, Daniela Grijincu, Peihan Li, Asif Saleem, Diarmuid O'Donoghue, and Rosemary Monahan. Arís: Analogical reasoning for reuse of implementation. Artificial Intelligence for Formal Methods (AI4FM), 2013.

[Pit13] Mihai Pitu. Source code retrieval using case based reasoning. Master's thesis, Department of Computer Science, National University of Ireland, Maynooth, 2013.

[Pre10] Roger S. Pressman. *Software Engineering A Practitioner's Approach*. 2010.

[PRW13] PRWeb. Cambridge university study states software bugs cost economy $312 billion per year. http://markets.financialcontent.com/stocks/news/read/23147130/Cambridge_University_Study_States_Software_Bugs_Cost_Economy_$312_Billion_Per_Year, January 2013.

[RU11] Anand Rajaraman and Jeffrey D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.

[sfCL] ISO standard for Common Logic. Common logic standard. http://iso-commonlogic.org/.

[Sow76] John F. Sowa. Conceptual graphs for a data base interface. *IBM J. Res. Dev.*, 20(4):336–357, July 1976.

[Sow84] J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[Sow08] J. F. Sowa. *Handbook of Knowledge Representation*, chapter Conceptual Graphs. Foundations of Artificial Intelligence. Elsevier, 2008.