Matching Spec# Specifications

Peihan Li

Dissertation 2013

Erasmus Mundus MSc in Dependable Software Systems

Department of Computer Science

National University of Ireland, Maynooth

Co. Kildare, Ireland

A dissertation submitted in partial fulfilment
of the requirements for the
Erasmus Mundus MSc Dependable Software Systems

Head of Department : Dr Adam Winstanley

Supervisor : Dr. Rosemary Monahan

30th June, 2013

Declaration

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

Signed Date

Acknowledgements

In the last one year, I have had the privilege to study with people who made my time at National University of Ireland, Maynooth enjoyable and memorize. I'd like to thank them all. Without them, my achievement so far would not be possible.

This project was the most challenging work I've ever done since I became a college student; the efforts I paid were worth. Looking back into the last five months, I am deeply grateful to my advisor in National University of Ireland, Maynooth, Dr. Rosemary Monahan. Rosemary is an outstanding computer scientist with great patient, she always be so nice even with my stupide questions. She is also a great mentor with patience and gives me lots of instructions when I was for this project. Under her guidance, I was learned a lot: recording project process, organize thesis and analyze the problem. Her inspiration and warm personality have won my highest respect and trust. If not her, it almost impossible for me to complete this project.

I would like to thank all my lab mates, roommates and classmates in NUIM with whom I have shared hours of discussion, study and laughter. It has always been enjoyable and fruitful to be together with them.

Last but not least, I gratefully acknowledge the support and education from Computer Science Department in National University of Ireland, Maynooth. I have spent almost one year in this lovely campus, all knowledge I've acquired, all things I've been through, the sweets and bitters of life I've shared would never been forgotten. Although I'll go another country in a few months, Ireland will always be remembered.

Abstract

In this project, we develop a tool which compares two Spec# programs (C# code with specification contracts) for signature matching. The tool automatically identifies whether the two specifications are similar, and gives out a new Spec# program which needs to be verified. There are levels of standards to judge how similar these two Spec# programs' specification is. This work contributes to the area of code reuse via match specifications: given a specification we aim to mark it to a similar specification and use its implementation to generate the implementation of the original specification.

In this work we present the process of match specifications in detail for Spec# programs, we discuss how the method may be applied to other languages and indicate future work in this direction. We match specifications according to the work of Amy Moormann Zaremski and Jeannette M. Wing's on "Match specifications of Software Components" [1]. This work proposes a lattice of possible technique for match specifications. Examples include Exact Pre/Post Match, Plug-In Match, Plug-In Post Match, Weak Post Match, Exact Predicate Match, Generalized Match and Specialized Match. We apply these definitions to Spec# programs, provide examples of verification matches and illustrate the level of matching that can be achieved automatically within the Spec# tools.

General Terms

Algorithms, Verification, Code Reuse

Keywords

Match specifications, Spec# programs, Verification, Design by Contract

Contents

Declaration 2
Acknowledgements 3
Abstract 4
Chapter 1 Introduction 7
1.1 Problem Statement 7
1.2 Motivation 8
1.3 Aims and Objectives 8
1.4 Report Structure 9
Chapter 2 Related Work 10
2.1 Code Reuse 10
2.2 Current approaches to code reuse 10
2.2.1 Structure Matching 11
2.2.2 Semantic Matching 12
2.3 Program Verification 13
2.3.1 What is Program Verification 13
2.3.2 Program Verification Tools 13
2.3.3 How We Can Use Program Verification Tools for Specification Matching 13
2.4 Specification Matches 14
2.4.1 Specification Matching by Wing and Moormann 14
2.5 Conclusion 16
Chapter 3 Tool Design 17
3.1 Overviews of Tool 17
3.1.1 Spec# Programming System 17
3.1.2 Design of the Solution 18
3.2 Why Do Subtype Match 20
3.3 Conclusion 20
Chapter 4 Implementation 21
4.1 Signature Matching 21
4.2 Rename Process 22
4.3 Print Process 23
4.3.1 Print of Specification Part 23
4.3.2 Final Print Effect 24
4.4 Text Operations VS Print Process 25
4.5 How to Achieve Subtype Match 25
4.6 Conclusion 27
Chapter 5 Case Study 28
5.1 Generic Pre/Post Match 28
5.1.1 Examples for Exact Pre/Post Match and Plug-In Match 28
5.1.2 Examples for Plug-In Post Match 29
5.1.3 Examples for Weak Post Match 31
5.2 Generic Predicate Match 32
5.2.1 Examples for Exact Predicate Match 33

5.2.2 Examples for Generic Predicate Match 34
5.2.3 Example of Specialized Match 35
5.3 Conclusion 36
Chapter 6 Evaluation 37
6.1 Performance of the Solution 37
6.2 Comparing to Solutions Already Have 38
6.3 Application 39
6.3.1 Preparing for C# Code Match 39
6.3.2 Application in Other Areas 40
6.4 Conclusion 40
Chapter 7 Conclusions...41
7.1 Summary 41
7.2 Future Work 41
7.3 Conclusions 42
Reference 43

Chapter 1 Introduction

1.1 Problem Statement

Spec# is an object-oriented language, a superset of C# v2.0 (released in 2005), compiling to the Microsoft Intermediate Language byte code (MSIL) and running on the .NET virtual machine and integrated into Visual Studio's IDE, which provides language services [2]. Formal methods and formal verification of source code has been used extensively in the past few years to create dependable software systems. However, although formal languages like Spec# or JML are quite popular, the set of verified implementations remains small. What's we are doing in this project is helping to promote reuse. To do this we determine automatically whether two specifications of Spec# programs are similar.

Specification retrieval could be performed using distinct semantic or structural characteristics or using the combination of the both. In the structural retrieval process, we focus on these conceptual graphs and by analyzing metrics of these representations (for example, number of nodes, number of loops, loop size, connectivity (O'Donoghue & Crean, 2002)). Based on conceptual graphs, we extract content vectors (Gentner &Forbus, 1991) which express the structure of the specification and the number of concepts (for example, statements, variable declarations, etc.). Because our database of specification artifacts will potentially be very large, we need an efficient way of retrieving the most similar cases.

In this project, we match Spec# specifications based on their semantics. We also have considered about using structure matching. For structural retrieval process, some breakthroughs have already been done as well as the contributed in the relevant fields, for example, Wei-Jin Park and Doo-Hwan Bae public the article "A two-stage framework for UML match specifications" in 2010 which presents a two-stage framework for matching two UML specifications [3]. However, we finally choose semantic matching as our approach. Semantic retrieval process is more suitable for Spec# match specifications as well as more precise, so in this thesis, we focus on the match Spec# specification using the semantic retrieval process.

We use the following picture (Figure 1.1) to describe the problem. The input of our tool is two Spec# programs, and the output of our tool is a new Spec# program which needs to be verified on line. The new Spec# program is composed by the specification part and the code part as well. The code part is exactly the same as one of the input Spec# program's code (in this project, we take Spec# program 1 as a template), and the specification part is a new one which generated by our tool. The specification part of the output Spec# program only include the ensures part. While the ensures part is generated from the rule proposed in the article "Specification Matching of Software Components" [1]. What's we need to verify actually is the ensures part, we still need the code part

because the on line verify tool can only verify a complete program.

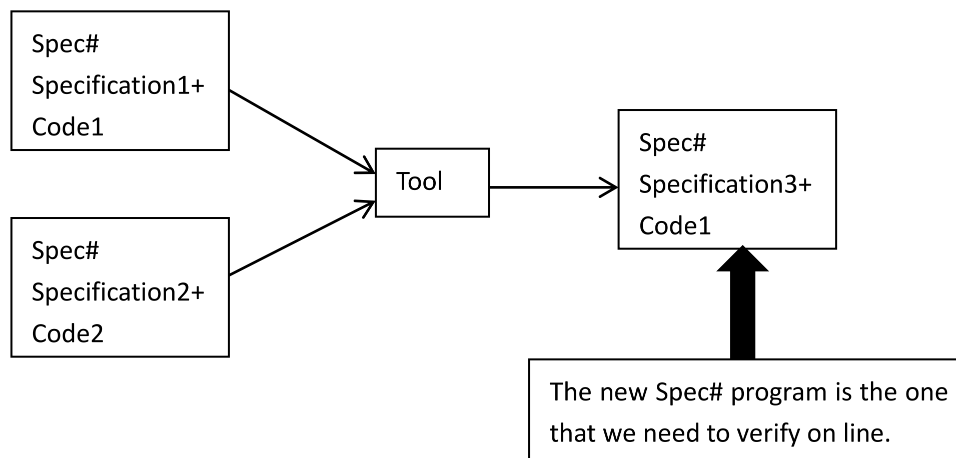

Fig 1.1: Description of the Problem

1.2 Motivation

Our main target to do the Spec# specification matching is for software reuse. A basic principle in software reuse is the notion that most software systems are not new and cab thus be developed using existing components [4].

Software reuse includes many areas, such as code reuse, specification reuse, and so on. It has many advantages, such as increasing software productivity, decreasing software development time, and improving software system interoperability. It's save both time and money if we can reuse software, especially when the software project is huge.

We are going to do Spec# specification matching in this project. If we can successfully find the similar Spec# specification, it can be used in many areas. At least it can be used for retrieve and prepared for C# code match.

1.3 Aims and Objectives

We focus on Spec# specification match in this thesis; it can be prepared for C# code match or as a complement for C# code match, also can be used for retrieve. What's we expected is that, for two Spec# programs as input, the output will tell us automatically whether the specification of these two Spec# programs are similar, if they are similar, on which level they two matches.

For two similar specifications of Spec# programs, we can also use this for retrieve code, as Spec# language could be the specification of C# code, so we expect it could be used for C# code match.

Our work aims to automate some of the steps involved in writing specifications and their implementations, by reusing existing verified programs i.e. for a given implementation, we aim to retrieve similar verified code and then reapply the missing specification that accompanies that code. Similarly, for a

given specification, we aim to retrieve code with a similar specification and use its implementation to generate the missing implementation.

In this thesis we focus on match specifications, more exactly, we will show how to match two specifications in written the Spec# programming system. The method could as well as be used to other areas, for example, the JML matching. This thesis implements the algorithm that Amy Moormann Zaremski and Jeannette M. Wing proposed in the article "Specification Matching of Software Components"—Lattice of function specification matches.

The finally object is to design such a tool that would apply the proposed algorithm [1] to Spec# specification matching, and automatically recognize whether two Spec# specifications match. Input of this tool is two Spec# programs, our final target for this tool is to detect whether their specification part match, if they are, how similar they are, according to Lattice of function specification matches, the output will show us automatically in which level they match. We also will do some case study after the tool can work automatically.

1.4 Report Structure

In this thesis, we will show some related work first in chapter 2, such as current approaches to code reuse (e.g. Artificial intelligence, case based reasoning, graph matching, analogical reasoning, structural matching, and semantic matching) and refer to papers on reuse of OCL / class diagrams which we supposed to use at the very beginning.

In chapter 3, we are going to show our overview solution of this project, and our expectation for the project.

In chapter 4, we show the details of what we have done and how we have done it. Explain how much we realize our expectation.

In chapter 5, we will give an example for each level of matching, and analyze my results. Based on these examples, we will analyze different levels of matching and their relationships.

In chapter 6, we'll explain what was evaluated or validated. Present my results as well as explain your results. In the final part of this thesis——chapter 7, we will summarize my results. Provide my conclusions (limitations &recommendations) based on the results obtained. And assess how well we have met my project goals as well as what we could have improved upon. And possible future work

Chapter 2 Related Work

In this chapter we'll going to have an overview of some related work and background knowledge, comparing the method we used in this thesis with other methods which used on specification matching area.

2.1 Code Reuse

"Code reuse, also called software reuse, is the use of existing software, or software knowledge, to build new software [17]". Companies are constantly looking for ways to get products to customers faster, provide higher quality at the mean time reduce product costs. Software reuse plays a very important role in achieving these results.

"Code reuse is a form of knowledge reuse in software development that is fundamental to innovation in many fields [5]". When doing code reuse, a lot of things should take into consideration. We should consider how to deal with the renaming of variables and functions as well as dealing with variable type changes, dealing with function or code reordering and so on.

We pay a lot of effort on code reuse for the benefit code reuse brings to us. We can reuse the generic functions that created before, especially when the project is big, it's more important, for saving time and money. Also, writing fewer lines of code (through reuse) leads to fewer bugs. This benefits developer in the debugging phase; if there is a bug in the system, there's a much greater probability it came from a new piece of code than from a code module that has been used in several previous projects without fail.

Code reuse also benefits the end user. For example, all the Microsoft Office applications share the same code for a number of tasks, such as the toolbar functionality and the menu system. Such code reuse provides the end user with a consistent look and feel, flattening the learning curve associated with the various Office applications.

2.2 Current approaches to code reuse

The approaches to code reuse are rich and colorful. For example, Artificial intelligence, case based reasoning, graph matching, analogical reasoning, structural matching, and semantic matching and so on. In our project, we used semantic matching for Spec# specification, comparing to structural matching, it has a lot of advantages, and we will discuss this later in chapter 3.

2.2.1 Structure Matching

There are many areas which structure matching has been used. For example, Yang [6] developed a system which is for AST match. However, it used on code match, what's we study here is specification matching.

At the very beginning of the project, we also considered of using structure and the method has already been used very successfully in the area of retrieval of UML Class Diagrams [7]. This article [7] gives us an idea of how to do Spec# specification matching at the very beginning. Now let's has an overview of how authors do it in the article [7].

At first, authors of article "Towards an ontology-based retrieval of UML Class Diagrams" calculate the semantic distance based on Wordnet. Finally, they choose to use another very complex algorithm to calculate. The semantic similarity manifestations of semantic distance in Figure 2.1, the similar the semantic distance is, the similar the two UML class diagram is.

We have consider using the method proposed by Karina Robles, Anabel Fraga*, Jorge Morato, Juan Llorens in article "Towards an ontology-based retrieval of UML Class Diagrams". In their article, they proposed using a Structure-Mapping Engine algorithm, the input are two class diagrams——the base and the target, through some transform rules, transform a UML class diagram to Structure-Mapping Engine and then execute the SME to generate two important categories of information, analyze the Gmap, finally identify the set of retrieved components.

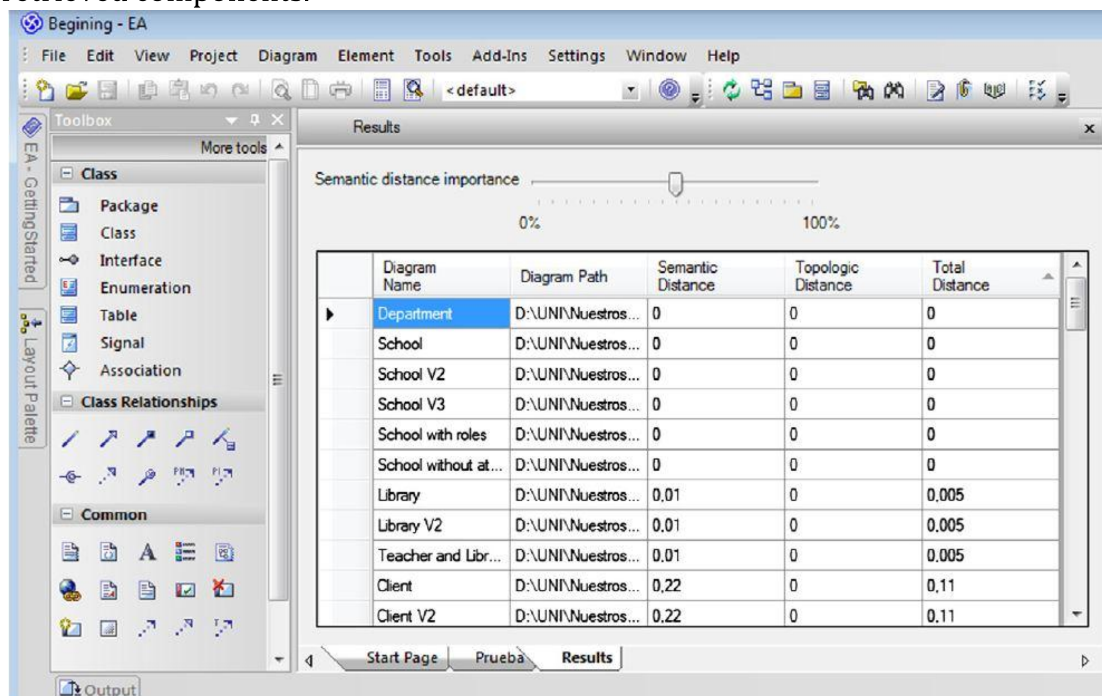

Figure 2.1 shows their finally work

To use their method, we first need to find a way to translate our Spec# specification to UML class diagram and then using the method proposed in the article to match them. We try to translate Spec# specification into the following

UML class diagrams like that which follows and with the OCL part, we get the Spec# contract: variables, methods and prototype, methods and requires, ensures.

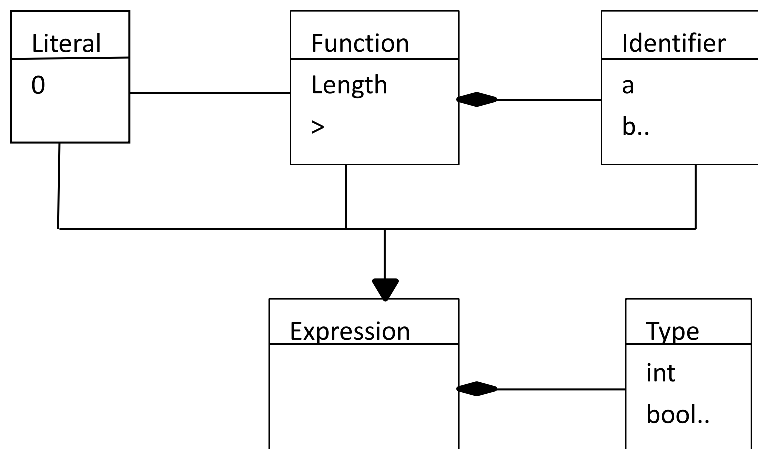

Fig 2.2: UML Class Diagram for Spec# Contract

We finally give up this idea for three reasons: the first reason is that we still can't find a way to transfer the Spec# contract to UML class diagram automatically. The second reason is that it is hard for us to define the semantic similarity. The third reason is comparing to semantic matching, semantic matching is more accurate.

2.2.2 Semantic Matching

"Semantic matching is based on two ideas: (i) we discover mappings by computing semantic relations (e.g., equivalence, more general); (ii) we determine semantic relations by analyzing the meaning (concepts, not labels) which is codified in the elements and the structures of schemas." [8] The algorithm we used in this project is proposed in article [1]. Comparing to structure, semantic matching is more suitable for this project. For the problems we worry before in section 2.2.1 are not exist now and semantic matching is more accurate.

Our matching Spec# specifications method is based on semantic matching rather structure matching. After structure matching, it still can't promise the semantic matching. For structure matching, for example, a line segment connecting point a and point b, we can match a similar line segment connecting point c and point d, but we have no idea the context in a, b, c and d. So comparing to structure matching, semantic matching is more precise.

2.3 Program Verification

2.3.1 What is Program Verification

As a working process, the program verifier layer extracts proof obligations from the specified program and passes them to the theorem prover [9]. Formally, Verification is used to prove whether a program satisfies a formal specification or use to test to whether a program works as specified. On the website rise4fun.com there is verification on line tool for different kinds of languages, such as Spec#, Boogie, Z3 and so on. Programming systems that include mechanical verification as part of the compilation process will make programs more efficient, reliable, and flexible—if they can ever be made practical. Software Verification means, for example, what it means that my program is correct, that means we need models and tools to reason about them. Software complexity is increasing.

2.3.2 Program Verification Tools

Verification/Analysis tools need some form of symbolic reasoning. There a lot of verification tools help us to do such jobs——SAT/SMT Solvers, interactive /automatic theorem provers, extended static checkers and so on. As we all know, software errors are expensive, so verification tools are really necessary.

The verification tool is wide range of applications; let's take SMT Solver for example here. An SMT-based program verifier is automatic in that it requires no user interaction with the solver [10]. Everything in SMT2 (SMT-Lib Version 2.0) is a function; Z3 as a kind of SMT Solver; we can use Z3 to show whether the formula is valid. For example, the formula: $(j+h > h+n) \wedge (n = 2j)$. We can express this formula in the following way and then try this in http://rise4fun.com/z3.

```
(declare-fun j () Int)
(declare-fun h () Int)
(declare-fun n() Int)
(assert (> (+ j h) (+ h n)))
(assert (= n (* 2 j)))
(check-sat)
(get-model)
```

2.3.3 How We Can Use Program Verification Tools for Specification Matching

Specification is a description of what a program should do, usually in English. "The Spec# specification is usually consists of three parts, the requires clause, ensures clauses, and the modify clauses [1]".

Spec# is written using first order logic so it is easy to compare specification.

Since the Spec# project started, the Verified Software Initiative [11] has organized the verification community to work towards larger projects, larger risks, and a long-term view of program verification. We can also use SMT Solvers to verify whether the code is correct with respect to these specifications extended static checkers such as Spec#. As we know, C# + annotations = Spec#; While the Annotations here means pre/post conditions, invariants, and other annotations. We can translate Spec# to Boggie, and then using SMT Solver Z3 to verify it. For Z3, the input is the verification conditions and the output is correct or a set of errors.

The definition of C# specification is that it could be used to understand the behavior of C# in details under all circumstances. Also, it can be used to create a new implementation of C#. In this thesis we focus on the Spec# match specifications. For which we can see the Spec# match specifications as a filter of C# code match for Spec# language could be used as the specification part of the C# code. Also as retrieve function. Combine with code match; it could achieve the goal of reuse of program and specification. It could benefit each other, as well as complementary for each other.

In figure 2.3, it explains the relationship between Spec# specification match and C# code match and how they benefit each other. This means if two programs' specifications part matches, we could reuse the code. In reverse, if two programs' code part matches, we could reuse the specification part.

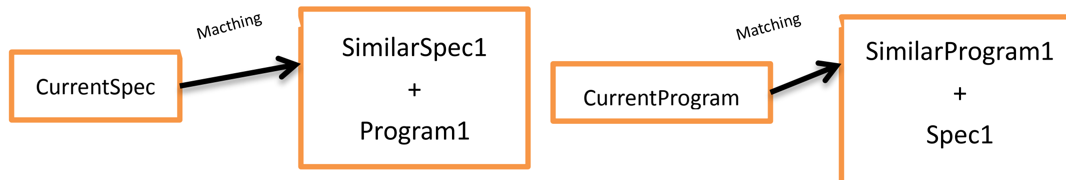

Fig 2.3: Reuse of Specification and Programs

2.4 Specification Matches

2.4.1 Specification Matching by Wing and Moormann

A lot of work has been done about match specifications, for different languages' specification matching and different methods, as we discussed in section 2.2, some use a structure retrieve process and some by semantic retrieved process. In this thesis, we focus on the semantic retrieved process and using the algorithm proposed by AMY MOORMANN ZAREMSKI and JEANNETTE M. WING in the article "Specification Matching of Software Components" --- Lattice of function specification matches. They use Larch/ML [Wing et al. 1993]. Then the rule Lattice of function specification matches can also be applied on the Spec# specification. It gives us a general idea of different levels of match specifications, form the most strict condition Exact Pre/Post Match to the most

relax condition match Guarded Post Match. Fig 2.4 has presented us the overview of the matching level. Every match method has an algorithm and exact pre/post is the one with the strongest condition on the top most.

In Fig 2.4 we marked the arrow and word with different colors, the blue arrows with the red words forming a path used the rule of Generic Pre/Post Match. Black arrows with the green words forming a path used the rule of Generic Predicate Match. We will discuss this later in chapter 5 in detail.

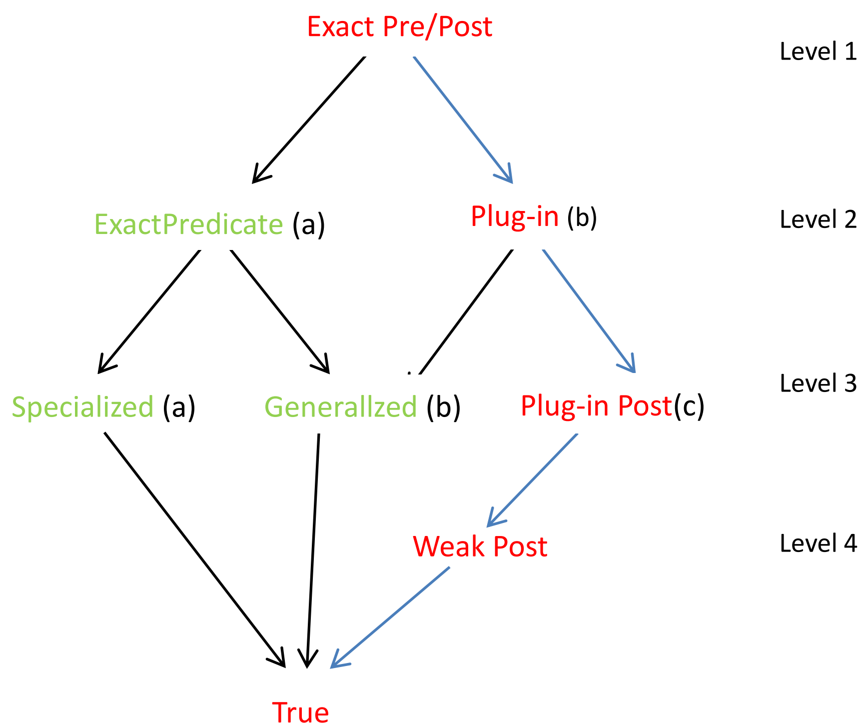

Fig 2.4: Lattice of function specification matches.

The article “Specification Matching of Software Components” also gives a definition of each level of match. Taking the right most path for example. Suppose we have two specifications named as `spe1` and `spe2`, while `pre1` and `pre2` represent the precondition of the two specifications, and `post1` and `post2` represent the postcondition the two specifications. The main algorithm (the algorithm we mean here is proposed by Amy Moormann Zaremski and Jeannette M. Wing’s on "Specification Matching of Software Components") for the right most path of the figure 2 is the following [1]:

`(Pre2 R1 pre1)&& (post2 R2 post2)`

R1 and *R2* here could represent⇔, =>, or nothing in different level of match. The article “Specification Matching of Software Components” has given a clearly definition of this:

Level 1 should hold the strongest condition which is

`(Pre2 ⇔ pre1)&& (post1 ⇔post2)`

While level 2(b) applies for the following rule, it is obvious that comparing to the level 1, the condition of level 2(b) is more relaxed, in level 1; it requires `pre2`➔`pre1` as well as `pre1`➔`pre2`, the same for postcondition. While for

level 2(b), it only requires for one direction imply.
`(Pre2 ➔ pre1)&& (post1 ➔post2)`

Level 3(c) apply for the following rule, it is more relax than level 2(b), it just ignore the precondition: `post1 ➔post2`

For level 4:
`(pre1&&post1) ➔post2`

In this article "Specification Matching of Software Components", it applies these algorithms on Two Larch/ML Specifications to show their difference. In our work, we will apply these rules to Spec# match specifications.

2.5 Conclusion

In this chapter, we've show lots of related work about our project——Spec# specification matching. We also have a basic idea of how to deal with this kind of problem and through comparing different methods to our project, finally deciding our approach about this project.

Chapter 3 Tool Design

In this chapter we'll show our overview of our tool and explains why we are going to use the specification matching approach by Moormann and Wing rather than graph matching. Actually we can get the output program which needs to verify from the input programs (the two spec# programs whose specification part we are going to match). However, we didn't realize the last step——paste the output program on online verify website and run it automatically.

3.1 Overviews of Tool

3.1.1 Spec# Programming System

The Spec# language is a superset of C#, an object-oriented language targeted for the.NET Platform. Spec# adds to C# type support for distinguishing non-null object references from possibly-null object references, method specifications, a discipline for managing exceptions, and support for constraining the data fields of objects [18]. Spec# shows how contracts and verifiers can be integrated seamlessly into the software development process [12].

The Spec# programming system is high-quality software which is also very costly. Programming system that includes mechanical verification as part of the compilation process will make programs more efficient, reliable, and flexible [13].The Spec# static program verifier (SscBoogie): generates logical verification conditions from a Spec# program uses an automatic reasoning engine (Z3) to analyze the verification conditions proving the correctness of the program or finding errors in it. Figure 3.1 gives us an overview of the process of Spec# Programing System.

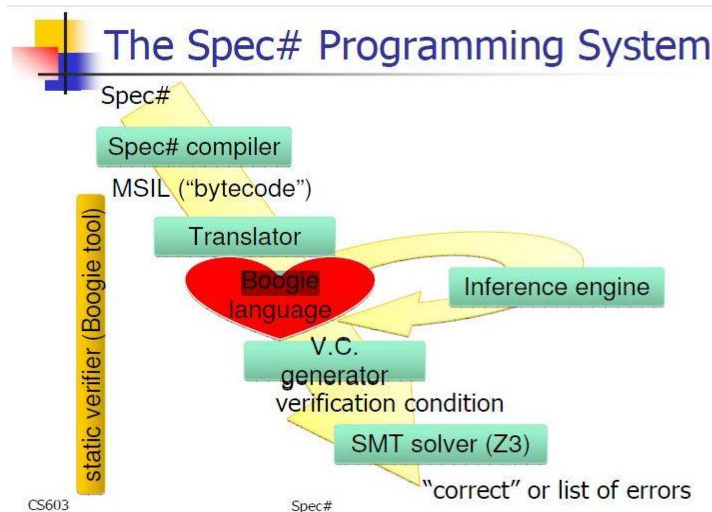

Fig 3.1: Overview of Spec# Programing System

Spec# programming system includes a language and compiler, the name of that compiler is Boogie, which checks specifications statically. That is the reason why we use Spec# Programming System here, as it provides an automated verification environment. It is a very important characteristic of Spec# Programming System, based on this characteristic, we can check Spec# programs automatically, getting the result whether the program is correct, if the program is not correct, there should be a list of errors or warnings, according to these errors or warnings, we have a general idea of which part is out of control in the Spec# programs. In this project, the output of our tool is Spec# program, and then using Spec# Programming System, we can easily check whether the program is correct, more exactly, whether the rule we used is applied for the programs.

Before doing this project, we should install Spec# on Visual Studio; here we use the vision of VS2010. Download the Spec# package on Microsoft Research Spec# website; generally we should make sure whether the Spec# compatible with VS, if not, we should do something before run Spec# on VS, the method to deal with compatible problem can search on Google. After all this preparation work has been done, we could do the following work.

3.1.2 Design of the Solution

Figure 3.2 provides an overview of our tool. At the beginning we parse the two Spec# programs and then we get two abstract syntax trees (ASTs) (in the following part of this thesis, we will use AST represent for abstract syntax trees), we are supposed to collect variable types and variable names. After collecting these information, we get into the signature matching and renaming process, and then get a new Spec# program which will be verified. In chapter 5, we will describe matching algorithm in detail and in chapter 4, we will show the implementation part in detail, illustrate how exactly signature matching are processed, and under what's condition it goes into the next step, what's its output was supposed to look like, and how different levels of matches are defined and the tool's performance.

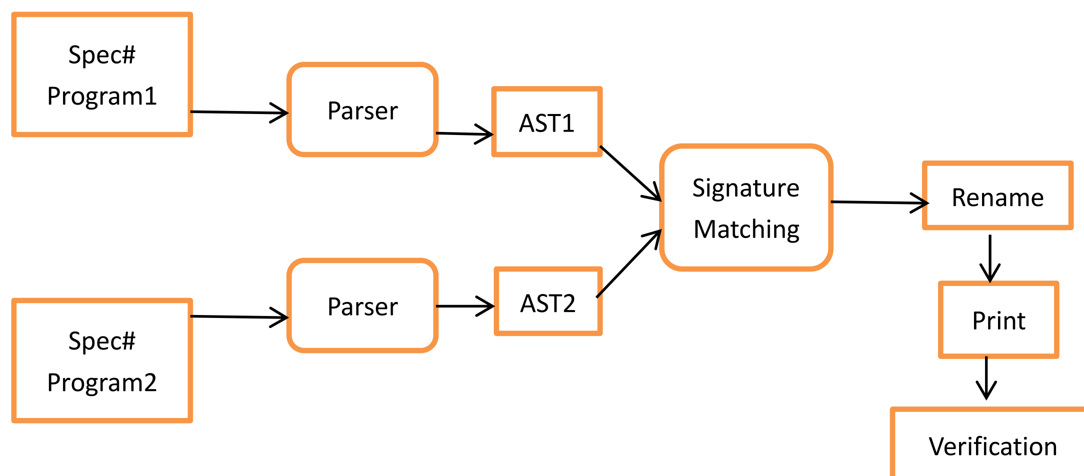

Fig 3.2: Our Solution Overview

There is a problem here (if we do as figure 3.2) that the type we collected stored in the AST is in the form of string. Actually, it doesn't make any influence when we compare whether these two types is the same. But what really matters is that when one class is inherited form another class, or one type is the subtype of another type, for example, binaryExpression is the subtype of the type Expression, it could not tell the relationship between these two types, for the types we get there are all stored in the AST in the form of string. We will talk about this problem later in section 4.5.

So the whole process is to get the AST (without the type information which cloud recognized by the compiler) first, and then doing the signature matching, as we use the exact signature matching here, so successfully signature here means signature return type, parameter number and parameter type should be the same, if the signature matched successfully, then we go into the rename process, the rename process is operated on the AST. In this project, we use the program1 as the template, so changing parameter name in parogrm2 into parameter name in program1. Finally we go into the print process, to print the finally program which we want to verify. The print process can be finished by using two methods, one is the formal method, using the printer to print, another method is to operating on text file directly, and both ways have their own advantage. This part we will discuss in the following chapter. Finally, we'll do case study, analyzing the examples, these examples we list here are very representative which should clearly show the different between different levels of match.

Our finally tool interface would be some kind like figure 3.3. The input would be two Spec# programs, the output will show on the right blank whether their specification part is similar, if they do, show out their similar level.

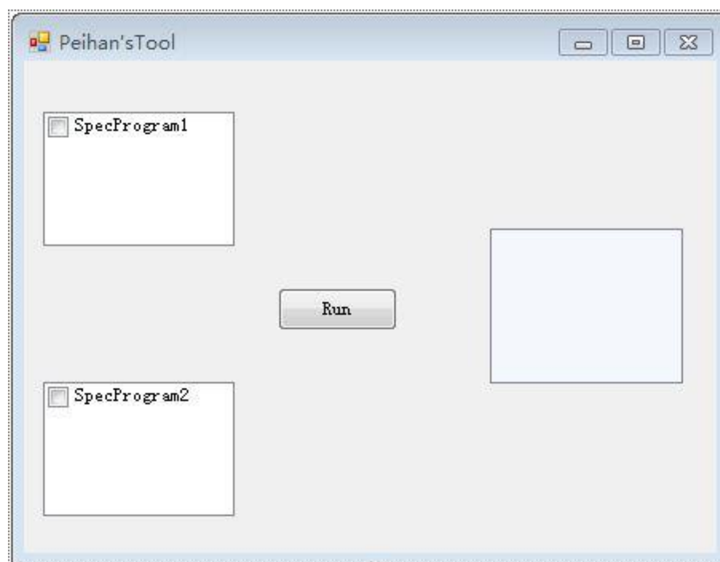

Fig 3.3: Supposed Interface of Our Tool

3.2 Why Do Subtype Match

The subtype we mean here is that, for example, public class BinExpression inherited form public class Expression, the object type of BinExpression is the subtype of the object type of Expression.

The subtype here has a relationship with inheritance and inheritance could achieve re-usability of code. As we know, inheritance involves reuse of implementations; we could have an inheritance relationship between classes that are incomparable in the subtype relationship. If two object types have the relationship of subtype that they must have some similar part. So even the signature return type is not exactly the same, it is still necessary for us to check whether they have the relationship of sub type.

3.3 Conclusion

In this chapter, we just have an overall idea of this project and wishes of how it works and what's its interface would look like. It likes a dream we has for our project and the following chapter is how we realize it.

Chapter 4 Implementation

We will use the following example to describe each process. Class Test1 and Class Test2 are source code which we are going to match their specification part. We choose this as example is to ensure the match can go through the whole process, form signature match to rename, print finally verification.

Example<1>

```
public class Test1
{
  int ISqrt (int y)
  requires y>=5;
  ensures   result*result   <=   y   &&   y   <
            (result+1)*(result+1);
    {
    int r = 0;
    while ((r+1)*(r+1) <= y)
    invariant r*r <= y;
         {
          r++;
         }
    return r;
    }
}
```

```
public class Test2
{
      int ISqrt(int x)
      requires 3 <= x;
      ensures   result*result   <=   x   &&   x   <
result+1)*(result+1);
      {
       int r = 0;
       while ((r+1)*(r+1) <= x)
       invariant r*r <= x;
           {
            r++;
           }
return r;
      }
}
```

Fig 4.1: Example used to analyze the Implement Process (source code based on [15])

4.1 Signature Matching

The overall matching idea is to combine the signature matching and the match specifications together, relationship between signature matching and match specifications is conjunction, if we say two Spec# specifications is matched, that means it satisfy both the signature matching and the Spec# match specifications. Their relationship is more than conjunction; we can think of signature match as a “filter” [1], so it can eliminate the programs which are obvious non-matches, for trying the specification match is expensive, that helps save money. In this paper, for signature match, we use module of functions type equivalence variable renaming (“exact match” in [14]), with exact signature Match. This method has both advantage and disadvantage. The advantage is that after the signature matching process, the match specifications part followed will be much more precise, and we don’t need to worry about the problem of parameter number and parameter type. While the disadvantage is that it may miss some part, that even two Spec# specifications are exactly the same, our tool could not detect it. The example<2> followed gives us a good explain about its short advantage, as well as example<3>. But we still using exact signature match for our final goal is to retrieve a similar specification and use its implementation to generate the implementation of the original specification, in the C# implementation match part, this problem has already been solved.

Example<2>

(Parameter types do not match: The specification part is exactly the same, but using this exact signature matching as a filter here, the output is that the specification of these two Spec# programs is not similar).

Program 1:	Program 2:

<pre>Public class Test1
{
Public int add (int x, int y)
Requires x>0 && y>0;
Ensures result >0;
{
Return x+y;
}
}</pre>
	<pre>Public class Test2
	{
	    Public double add (double x, double y)
	    Requires x>0 && y>0;
	    Ensures result >0;
	    {
	       Return x+y;
	    }
	}</pre>

Fig 4.2 Example which can't detected by our tool

These two programs should be the similar, but they can't go through the signature matching. For the rule we use here is exact signature match. The exact signature match here means the signature return type should be the same or has the relationship of inheritance, the parameter number and parameter type should be exactly the same. There is no requirement about the parameter name, the job of the rename process is to deal with this problem. The Spec# match specifications is to prepare for the C# code match, so even the parameter number and parameter type do not exactly the same the C# code could also be similar, and this part is considered in the part of C# code match, so we do not take this part into consideration in the Spec# match specifications.

After the parser process, we get two ASTs, and there is a class whose name is StandardVisitor, this class is used for traversal the ASTs we get, to make the code more clearly, we define a class whose name is MyVisitor to inherit the class StandardVisitor, and override the method we need. We use the ArrayList to store the parameter type and parameter name information. For signature return type and parameter number, there is no need to use ArrayList.

4.2 Rename Process

If the signature is totally matched, then we go into the rename process. Rename process is very necessary, when two programs are exactly the same expect the parameter names, we need to rename and then to compare them.

When we compare whether two programs are similar, we judge this by analyze whether they two do the similar things, the parameter name is just a name or just symbol, it doesn't means much in analyzing the action of the program, so if we want two programs' action could be compared automatically, we must rename first. Rename here means if we have two programs A and B, we can use the parameter name in A as template, and change the parameter name in

B into A's according to some rule.

When doing the rename process, we obey to the rule of renaming in parallel in this design, that is rename the parameter name which at the same position. Trying to rename, we need another sub class of StandVisitor for rename process, in the program we just name the class as VisitorForRename, here we just rename the Spec# program2's parameter name to Spec# program1's parameter name in AST, the source code do not change.

Taking example<1> as an example, the parameter named "x" in public class Test2 changed into "y" which is the same as public class Test1's parameter name. After the rename process, the Spec# program2's AST will be exactly the same as the AST of the Spec# program1. This would a preparation for the print process.

4.3 Print Process

4.3.1 Print of Specification Part

After the signature matching, rename process, we are going to print out the final Spec# program which is going to verified on Spec# on line test tool. First we need to consider how to pick out the specification part of the Spec# program.

There is a class with the name of CodePrinter in the SPEC# bag, which is used to print out. But the function is very limited and it just offer us a structure, we have to rewrite some part as well as add some new part in the class of the CodePrinter, for convenient and make the code more readable, we create a new class with the name of OutputVisitor which is inherited from the StandardVisitor, we modify the class CodePrinter and then invoke the print method in the class OutputVisitor. The class MyVisitor (which to get the parameter list), VisitorForRename (which used to rename the parameter) and the class of OutputVisitor (to realize the print part) and the main function should under the same namespace.

First we meet the problem that it printed out `expression2str` instead of variable name and printed out `op2str` instead of the real operators, we have no idea what exactly it printed out at the very beginning. It does not print the way we expected, it just print out the general concept instead of specific one. We should convert `expression2str, operand` and many other signs to exactly what they are in the program.

And then we meet a similar problem, it just print out the word "`identifier`", we should change the `identifier` to the parameter name that what they exactly stand for. This CodePrinter here do not offer this function in this part, so we have to write by ourselves as we needed. Here we must get familiar with the Spec# language grammar first, and write every part for them. Luckily, the Spec# language grammar is not too difficult, that we could write one by one. However, if the Spec# language grammar is difficult, things will get very troublesome. Then write the printer will be a huge job, so I think of another way

to deal with this problem. That method we will introduce in section 4.4.

Third, we should format it, so Spec# on line test tool can recognize it. For example it prints out the “LogicalAnd” instead of “&&”, “Imply” instead of “➔”, which Spec# on line test tool could not recognize it. As they are operators, so we can try to search `opr` to see whether that works. Sometimes, programing is just guess and tries.

Finally, applying the Lattice of function specification matches on the specification part we get, let’s see what we get here. We will take the Plug-In Match for example: take example<1> to do the analyze, after the signature match, it goes into the rename process successfully, rename parameter name “y” to “x” in the AST. Then we print its specification part, it should look like the following:

```
ensures ((3<=y)==>(y>=5)) &&((result*result <= y && y <
(result+1)*(result+1))==>(result*result <= y && y < (result+1)*(result+1)));
```

4.3.2 Final Print Effect

Another very important part is that we must delete the original specification part (preconditions, postcoditions, invariant) in the source code. If we do not delete them, it would inference the verification process, say the precondition is y>=5, then y is always above 5, that 3<=y will be meaningless, another reason we must delete the original specification is that the only part we need to verify is the new added ensures clause, in this example, that is :

To verify it we need to combine it with the code, for the variable “y” should be defined and the result must have some meanings, figure 4.3 shows our final result of print. After all these work, we can paste it directly to the on line Spec# verification tool to test it whether it is verified. We can use web knowledge to verify the program one by one until the one which is satisfied, and then it can automatically recognize which level of match it belongs to. (For this part is not our main job, I didn’t do it in this project).

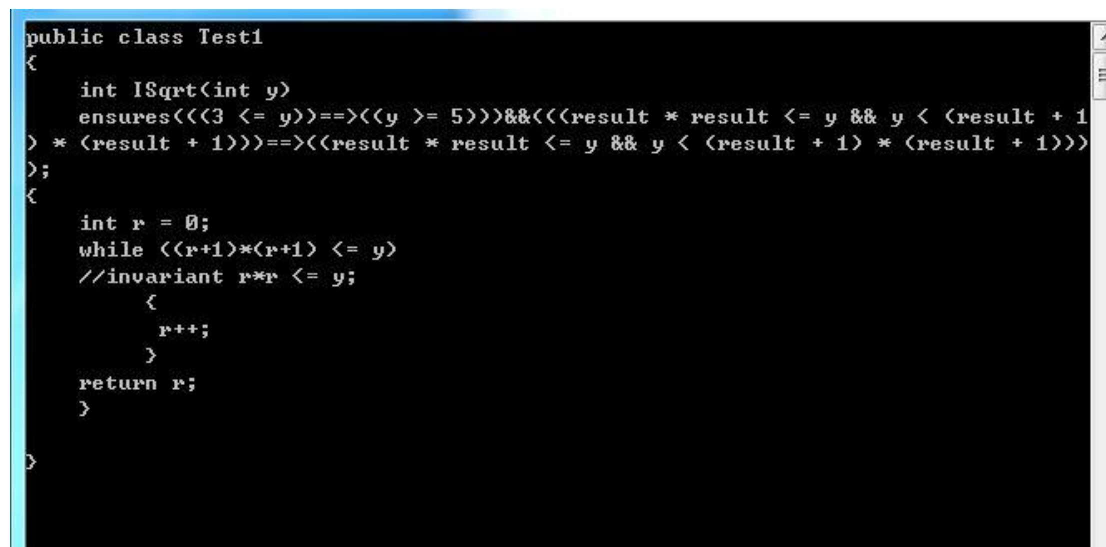

Fig 4.3: Output of our tool

4.4 Text Operations VS Print Process

We could use another way the get the same result. In the AST, we c find that the HelpText is just the specification part, there is no need for us to find the contract part, every single variable name and where these variables stored and considering the way to invoke them and print it out in a right form which on line Spec# verify tool could recognize.

As we mentioned before, the Spec# grammar is not too difficult, so rewriting the printer is not a huge job. When the grammar is very difficult, rewrite the printer is not an easy job. So what we consider while doing this project is——if we can use the HelpText directly then the print process could be omission.

But the problem is that the HelpText is pointed to the source code directly which means the rename process doesn't work on the HelpText, if the rename doesn't work on the HelpText, even we can get it we can't use it. So we can't use the HelpText directly. Before we use it we should try to do the rename process on it too, this rename process here is not done on the AST, we do this process by change the parameter name of the HelpText directly, actually we just operate text file directly. We operate on the Spec# program2 text file by using the Regular Expression, for example in the program2 HelpText, when we meet the variable named "x" then we all change it into "y".

Using the Regular Expression is a very important step in text operation, if we just write program, for example, said when meet with "a" and then change it into "b", unexpected error may happen, "forall" as a key word which also include "a", it may change into "forbll". While using the Regular Expression this kind of problem will not happen, for we use it to detect whether it is a variable name or not, we only change the variable name. After the rename process, we can invoke the HelpText directly, it has the same effect as the printer, and the following steps are exactly the same.

In this project, I also use this method, to make a compare to the printer method, it is much easier to realize and it is accurate too. What's more this idea can also apply to other programs, say boogie, where the printer is needed. For in most situations, the printer does not perform as we supposed to, and it is not an easy job to rewrite the printer. However, there may have some short advantages of this method, for the formal way to deal with this kind of problem is using the printer to print the program out, but at least now I didn't find any short advantage or bad inference of the method in this project. If the reader found it, please do not hesitate to contact me.

4.5 How to Achieve Subtype Match

To solve this problem, we've tried many methods, at beginning we try to use the reflect skill in C# but it doesn't work. Finally, we try to find the type information in the complier, in some step of the process of compilation, it must

produce an attribute list, this attribute list will record all the information, such as class attribute, the superclass type, using this attribute list, we could judge the relationship between two types, even the type is defined by user. So the main problem now is to find the attribute list.

To find the attribute list in the compiler is not an easy job, for the compiler is too complex, if we follow it step by step, it's both time consuming and unrealistic. As we all know, when the source code is correct, nothing is output for the compiler, while the source code is wrong, there must be some output of the compiler, so at last we choose the way to search how the compiler judging a variable's type is wrong, there may be exist the attribute list we need.

Then we write a simple source code with error then there is output showing the error, and also there is a class that dealing with the error, and then we found the source code in the compiler with a word "error" in the class name, setting a breakpoint at all of its functions, with debugging many times, we can find the error message output function, and then when the compiler runs into to this function, view all the stack which call the function, then we find all the function which are invoked by the compiler when something is wrong with the variable, and then analyze these function and parameter through debugging, finally we found how the compiler determines the variable type(the function that determines the variable type) and the data structures.

Before doing these things, some basic knowledge is required. Having a general understanding of compiler theory is quite necessary, also having a knowledge of the compiler first lexical, syntax analysis, and generate syntax tree. The information we need is in the syntax tree, but only after going through semantic analysis that the information of the variable type will be added to the syntax tree, without the process of semantic analysis, as we showed before, in the syntax tree the variable type is existed in the form of string. So as long as we find the semantic analysis function that we can get the type information, that explain why at the beginning the AST we get do not conclude to information of variable type, but no type information. But there is no need to get the variable type if we don't want to deal with the subtype, just comparing two type whether same of not, even the type information on the AST is stored in the form of string, we still can compare them using the function of equalTo.

Now we defined the following relationship:

```
namespace MyObject
{
public  class Node{
}
public class Expression : Node{
    public Node test(Node node, Expression exp)
     {
         return node;
       }
  }
```

```
 public class BinExpression : Expression{
  }
}
```

And the input is like below:

```
namespace MyObject
{
class Rev1
{
      public Expression Reverse1(int[] !a, int[]! b)
      requires a.Length == b.Length;
      modifies b[*];
   ensures forall{int i in (0: a.Length); b[i] ==
a[a.Length-1-i]};
      {
         int low = 0;
         int high = a.Length;
         while (low < high)
         invariant high + low == a.Length;
  invariant forall{int i in (0: a.Length), i < low || high
<= i;b[i] == a[a.Length-1-i]};
               {
                        high--;
                        b[low] = a[high];
                        b[high] = a[low];
                        low++;
                }
             return new Expression();
      }
}
}
```

```
namespace MyObject
{
class Rev1
{
      public BinExpression Reverse1(int[] !x, int[]! y)
      requires x.Length == y.Length;
      modifies y[*];
   ensures forall{int i in (0: x.Length); y[i] ==
x[x.Length-1-i]};
      {
         int low = 0;
         int high = x.Length;
         while (low < high)
         invariant high + low == x.Length;
  invariant forall{int i in (0: x.Length), i < low || high
<= i;y[i] == x[x.Length-1-i]};
               {
                        high--;
                        y[low] = x[high];
                        y[high] = x[low];
                        low++;
                }
             return new BinExpression();
      }
}
}
```

Fig 4.4: Example for Subtype Match (source code based on [15])

As a result, they two are Exact Pre/Post Match, even the signature return type is not the same, and they do similar. For further application, we could study the reuse function of inheritance.

4.6 Conclusion

In this chapter, we have showed how we realize each step in detail, what's we have done for our dream in chapter 3 and what's we haven't done and the reason.

Chapter 5 Case Study

In this chapter we'll going to analyze each level of matching in Fig 2.4 with a Spec# program example, and find out what is reasonable for our design and what is not reasonable. As a result, we've found an example with each level of matching for Fig 2.4, and use these examples to explain the relationships between different levels of matching.

5.1 Generic Pre/Post Match

The Generic Pre/Post Match here we mean the four matching algorithm [1]:

```
--Exact match: pre1<=>pre2 && pos1<=>pos2
--Plug-In Match: pre2==>pre1 && pos1==>pos2
--Plug-In Post Match: pos1==>post2
--Weak Post Match: pre1 ==> (pos1==>post2)
```

It is proposed by Amy Moormann Zaremski and Jeannette M. Wing's on "Specification Matching of Software Components". `Pre1` and `post1` here represent for the preconditions and postconditions of the first Spec# specification, and `pre2` and `post2` represent for the preconditions and postconditions of the second Spec# specification. The first Spec# specification and second Spec# specification are the two specifications which we are going to match. These four matching level are just the red words on the blue path in Fig 2.4. From top to root of the tree of Fig 2.4, the condition is looser and looser. That means if the match applies to level 1, that it must also applies to level 2, level 3 and level 4. Similarly, if the match applies to level 2, it must also apply to level 3 and level 4.

5.1.1 Examples for Exact Pre/Post Match and Plug-In Match

```
public class Test1
{
        public static int SeqSum(int[] a, int i, int j)
        requires 0 <= i && i <= j && j <= a.Length;
        ensures result == sum{int k in (i:j); a[k]};
        {
        int s = 0;
        for (int n = i; n < j; n++)
        invariant i<=n && n<=j;
        invariant s==sum{int k in (i:n); a[k]};
                {
                 s += a[n];
                }
return s;
        }
}
```

```
public class Test2
{
        public static int SeqSum(int[] b, int x, int y)
        requires 0 <= x && x <= y && y <= b.Length;
        ensures result == sum{int k in (x:y); b[k]};
        {
        int s = 0;
        for (int n = x; n < y; n++)
        invariant x<=n && n<=y;
        invariant s==sum{int k in (x:n); b[k]};
                {
                 s += b[n];
                }
return s;
        }
}
```

Fig 5.1: Input to tool for Exact Pre/Post Match example

As we can see in Class Test1 and Class Test2, they are doing the same thing and the only difference is the parameter name. We put these two test class into our tool the output will be (source code based on [19]):

```
public class Test1
{
      public static int SeqSum(int[] a, int i, int j)
      ensures ((0 <= i && i <= j && j <= a.Length) <==> (0 <= i && i <= j && j
<= a.Length)) && ((result ==
sum{int k in (i:j); a[k]})<==> (result == sum{int k in (i:j); a[k]}));
      {
      int s = 0;
      for (int n = i; n < j; n++)
      //invariant i<=n && n<=j;
      //invariant s==sum{int k in (i:n); a[k]};
          {
           s += a[n];
          }
return s;
      }
}
```

As we mentioned before, the previous specification should be deleted and add the new one which need to be verified into the program, we get our final program. When we past the code above on http://rise4fun.com/SpecSharp and run it, if there is no errors and no warnings, it means it apply to this matching algorithm and the matching level is defined.

So the new specification part we add here is:

```
ensures ((0 <= i && i <= j && j <= a.Length) <==> (0 <= i && i <= j && j <= a.Length))
&& ((result == sum{int k in (i:j); a[k]})<==> (result == sum{int k in (i:j); a[k]}));
```

If we change it into the rule of Plug-In Match, it can go through the verification too.

```
ensures ((0 <= i && i <= j && j <= a.Length) ==> (0 <= i && i <= j && j <= a.Length))
&& ((result == sum{int k in (i:j); a[k]}) ==> (result == sum{int k in (i:j); a[k]}));
```

Examples for Plug-In Match is just the Example<1>, we have already used the example to explain the process of our solution. To avoid duplication, we no longer list it here.

5.1.2 Examples for Plug-In Post Match

For Class Test1 and Class Test2 in example<1>, we just reverse them, and then we get Plug-In Post Match. For the difference of rules of Plug-In Match and Plug-In Post Match is in Plug-In Match we have the part: `pre2==>pre1`, we reverse class Test1 and class Test2, and then `pre2==>pre1` is no more set up. While I am doing this job, I have a wondering that these two examples similar level is Plug-In Match but after I do the reverse, their similar level is lower,

changing into Plug-In Post Match. But after reversing, they are two different things, and another new example.

```
public class Test1
        {
                int ISqrt(int y)
                requires y>=5;
                ensures result*result <= y && y <
(result+1)*(result+1);
                {
                int r = 0;
                while ((r+1)*(r+1) <= y)
                invariant r*r <= y;
                        {
                          r++;
                        }
                return r;
                }
        }
```

```
public class Test2
        {
                int ISqrt(int x)
                requires 3 <= x;
                ensures result*result <= x && x <
(result+1)*(result+1);
                {
                 int r = 0;
                 while ((r+1)*(r+1) <= x)
                 invariant r*r <= x;
                        {
                          r++;
                        }
        return r;
                }
        }
```

Fig 5.2: Input to tool for Plug-In Post Match example (source code based on [15])

We put these two test class into our tool the output will be (source code based on [15]):

```
public class Test1
{
   int ISqrt(int y)
   ensures ((result*result <= y && y < (result+1)*(result+1)) ==> ( result*result
<= y && y < (result+1)*(result+1)));
    {
    int r = 0;
    while ((r+1)*(r+1) <= y)
    // invariant r*r <= y;
         {
          r++;
         }
      return r;
}
}
```

As always do, the previous specification is deleted and add the new postcondition. We get our final program and then we past the code above on http://rise4fun.com/SpecSharp and run it.

So when we use rule of level 1 or the rule of level 2(b) which higher than level 3(c), it both can't go through verification. It has a warning said: `Method Test1.ISqrt(int y), unsatisfied postcondition: ((y>=5)<==>(3<=y))` or `Method Test1.ISqrt(int y), unsatisfied postcondition: (((3<=y)==>y>=5))`.

We do the match in the sequence of strong condition to weak condition, which means if two Spec# specifications do not apply to the level 1, then we try level 2(b) and so on. For this example, it does not apply to level 1and level 2(b), while it applies to level 3(c).

5.1.3 Examples for Weak Post Match

As we do before, the example here should be with representatives and could explain Weak Post Match quite well. So the first thing we need to do is to analyze the difference between Plug-In Post Match and Weak Post Match. For the rule of level 3(c) is `pos1==>post2` while the rule of level 4 is `pre1 ==> (pos1==>post2)`. We must find an example which could apply to the rule of Weak Post Match but not apply to the rule of Plug-In Post Match. If we see `pre1` as `p` and `(pos1==>post2)` as `q`, the problem changes into the following truth table:

```
p   q   |   p==>q
-------------------------
T   T   |    T
T   F   |    F
F   T   |    T
F   F   |    T
```

For the example should apply to the rule of Weak Post Match but not apply to the rule of Plug-In Post Match. So both `p` and `q` should be false while p➔q should be true. From the true table, that only `F➔F==T` applies. However, if `p` is false, `pre1` is false, which means the precondition of the program should be false. There is a basic knowledge that before program runs if the precondition is false, the specification is always valid. Based on this logical, even we can find the example, it is meaningless, the precondition should be false, and no matter how different the two Spec# specification's postconditions are, the specification is always valid.

So in this part we will show the Anti-cited examples to prove this algorithm doesn't apply to matching Spec# specifications.

```
public class Test1
{
    int ISqrt(int y)
    requires false;
    ensures result*result <= y && y <
(result+1)*(result+1);
    {
    int r = 0;
    while ((r+1)*(r+1) <= y)
    invariant r*r <= y;
        {
         r++;
        }
    return r;
    }
}
```

```
public class Test2
{
        int factorial(int m)
        requires false;
        ensures result == product{int j in (1..m); j};
        {

          int f=1,i=1;
              while(i<m+1)
              invariant 1 <= i && i <= m+1;
              invariant f == product{int j in (1..i-1); j};
                        {
                         f=f*i;
                         i++;
                        }
           return f;
        }
}
```

Fig 5.3: Input to tool for Weak Post Match example

As we can see these two programs above are not similar at all, they are doing two different things. They can't go through level 1, level 2(b) and level 3(c) (the

blue path), which as we expected. However, they can go through Weak Post Match! For we set the precondition false. And the public class Test1 and public class Test2 can both go through the verification part, so the source code is ok, the examples above can go through the signature matching as well, the rename process, and then we get the following the output of the program which can be verified on the website (source code based on [15]).

```
public class Test1
{
   int ISqrt(int y)
ensures( false==>((result*result <= y && y < (result+1)*(result+1))==>(result
== product{int j in (1..y); j})));
    {
    int r = 0;
    while ((r+1)*(r+1) <= y)
   // invariant r*r <= y;
         {
          r++;
         }
    return r;
    }
}
```

So even source code like this which not similar at all. It can still apply to Weak Post Match, of course, as the root of the tree——Weak Post Match, the condition is very relax, so we decide not use this matching method here for precise reasons.

5.2 Generic Predicate Match

There are many rules of Generic Predicate Match [1]. For example:

```
(pre1 R1 post1) R (pre2 R2 post2)
```

Where the relation *R* is either equivalence < == > , implication➔, or reverse implication 🡐. *R1* and *R2* Here Could be ➔ or &&, in this project we use the symbol of → which is weaker than &&.

Then the Generic Predicate Match here is these three matching algorithms:

Exact Predicate Match:

```
(pre1 ==> post1) <==> (pre2 ==> post2)
```

Generalized Match:

```
(pre1 ==> post1) ==> (pre2 ==> post2)
```

Specialized Match:

```
(pre2 ==> post2) ==> (pre1 ==> post1)
```

Comparing to Generic Pre/Post Match, from the algorithm part which is the most obvious part, we can see that for Generic Pre/Post Match always using the preconditions imply preconditions and postconditions imply postconditions,

while for Generic Predicate Match, it always using preconditions to imply postconditions.

5.2.1 Examples for Exact Predicate Match

The example we show here must apply to rule of Exact Predicate Match, at the meantime, it should not apply to Exact Pre/Post Match. As in Fig 2.4: Lattice of function specification matches. The condition of Exact Pre/Post Match is stronger than Exact Predicate Match, so this example here should show their difference. Also if the example could apply to the rule of Exact Predicate Match, it could apply to the rule of Generic Predicate Match and Specialized Match.

<table>
<tr>
<td>

```
public class Test1
{
bool LinearSearch(int[] a, int key)
ensures result == exists{int i in (0: a.Length);
a[i] == key};
    {
      int n = a.Length;
      do
invariant -1<=n && n<=a.Length;
invariant forall{int i in (n: a.Length); a[i] != key};
        {
          n--;
        if (n < 0) {
                break;
        }
          } while (a[n] != key);
return 0 <= n;
    }
}
```

</td>
<td>

```
public class Test2 {
bool LinearSearch(int[] b, int key)
requires false;
ensures result == exists{int i in (0: b.Length);
b[i] == key};
    {
      int n = b.Length;
      do
invariant -1<=n && n<=b.Length;
invariant forall{int i in (n: a.Length); b[i] != key};
        {
          n--;
        if (n < 0) {
                break;
        }
          } while (b[n] != key);
return 0 <= n;
    }
}
```

</td>
</tr>
</table>

Fig 5.4: Input to tool for Exact Predicate Match example (source code based on [15])

In the above example, the difference between class test1 and class test2 is their precondition part, in class test1, it doesn't have precondition, so it should set into true by default, while in class test2, it has precondition part and it set as false.

Now we use the rule of Exact Predicate Match and Exact Pre/Post Match on the program separately to see the result. The following program figure 5.5 is the output which uses these two algorithms. The left program is the output using the rule of Exact Predicate Match while the right one is the output using the rule of Exact Pre/Post Match. Paste the program on Spec# online verify tool, as we expect, when running the left one, it's correct, while running the right one, it has a warning said: `Method C.LinearSearch(int[]! a, int key), unsatisfied postcondition: ((true<==>false)&&((result == exists{int i in (0: a.Length); a[i] == key})<==>(result == exists{int i in (0: a.Length); a[i] == key})))`

public class Test1	public class Test1
{	{
bool LinearSearch(int[] a, int key)	bool LinearSearch(int[] a, int key)
ensures ((true==>(result == exists{int i in (0: a.Length); a[i] == key}))<==>(false==>(result == exists{int i in (0: a.Length); a[i] == key})));	ensures ((true<==>false)&&((result == exists{int i in (0: a.Length); a[i] == key})<==>(result == exists{int i in (0: a.Length); a[i] == key})));
{	{
int n = a.Length;	int n = a.Length;
do	do
invariant -1<=n && n<=a.Length;	invariant -1<=n && n<=a.Length;
invariant forall{int i in (n: a.Length); a[i] != key};	invariant forall{int i in (n: a.Length); a[i] != key};
{	{
n--;	n--;
if (n < 0) {	if (n < 0) {
break;	break;
}	}
} while (a[n] != key);	} while (a[n] != key);
return 0 <= n;	return 0 <= n;
}	}
}	}

Fig 5.5: output for Fig 5.4 using different algorithms (source code based on [15])

5.2.2 Examples for Generic Predicate Match

```
public class Test1
{ int factorial(int n)
   requires 0 <= n;
   ensures result == ((n == 0) ? 1 : product{int j in (1..n); j});
      {
        if (n == 0)
              return 1;
        else
              {
               int f=1,i=1;
               while(i<n+1)
               invariant 1 <= i && i <= n+1;
               invariant f == product{int j in (1..i-1); j};
                          {
                           f=f*i;
                           i++;
                          }
                  return f;
               }
        }
}
```

```
public class Test2
{
          int factorial(int m)
          requires 3 <= m;
          ensures result == product{int j in (1..m); j};
          {
                int f=1,i=1;
                while(i<m+1)
                invariant 1 <= i && i <= m+1;
                invariant f == product{int j in (1..i-1); j};
                          {
                           f=f*i;
                           i++;
                          }
               return f;
          }
}
```

Fig 5.6: Input to tool for Generic Predicate Match example

The function of the example above is doing the factorial, the only difference between class Test1 and class Test2 is that the ending number. So they are doing the similar job. If we using the rule of Generalized Match then we will get the following output, like doing before, the previous specification should be deleted and adding the new one using the rule as the postconditions part.

```
public class Test1
{
        int factorial(int n)
ensures (((0 <= n)==>(result == ((n == 0) ? 1 : product{int j in (1..n); j})))==>((3 <= n)==>(result == product{int j in (1..n); j})));
        {
            if (n == 0)
```

```
            return 1;
          else
              {
                int f=1,i=1;
                    while(i<n+1)
                  // invariant 1 <= i && i <= n+1;
                  // invariant f == product {int j in (1..i-1); j};
                    {
                     f=f*i;
                     i++;
                    }
          return f;
              }
      }
}
```

As the output, the program with the specification part which uses the rule of Generalized Match is the following:

```
ensures (((0 <= n)==>(result == ((n == 0) ? 1 : product{int j in (1..n); j})))==>((3 <= n)==>(result == product{int j in (1..n); j})));
```

If we use the rule of Exact Predicate Match on it, the postcondition part will be:

```
ensures (((0 <= n)==>(result == ((n == 0) ? 1 : product{int j in (1..n); j})))<==>((3 <= n)==>(result == product{int j in (1..n); j})));
```

And when we run it on the online verify tool. There is a warning said: `Method Test1.factorial(int n), unsatisfied postcondition: (((0 <= n)==>(result == ((n == 0) ? 1 : product{int j in (1..n); j})))<==>((3 <= n)==>(result == product{int j in (1..n); j})))`. As we expected, the condition of Exact Predicate Match is stronger than that of Generalized Match.

5.2.3 Example of Specialized Match

Comparing the rule of Generalized Match and Specialized Match, we can find that the difference is just a reverse. So we just reverse the public class Test1 and public class Test2 then we can get the example for Specialized Match. When doing the examples for Plug-In Match and Plug-In Post Match, we do the similar things, finding the difference of the rule first and then do the reverse! While doing the example analyze, we find not only the difference between different levels of matching but also the similarity between them.
Then the output will be:

```
public class Test1
{
      int factorial(int m)
      ensures (((0 <= m)==>(result == ((m == 0) ? 1 : product{int j in (1..m);
```

```
j}))) ==>((3 <= m)==>(result == product{int j in (1..m); j})));
        {
                    int f=1,i=1;
                     while(i<m+1)
                    // invariant 1 <= i && i <= m+1;
                    // invariant f == product{int j in (1..i-1); j};
                      {
                       f=f*i;
                       i++;
                      }
            return f;
        }
}
```

As showed in Fig 2.4: Lattice of function specification matches, Generalized Match, Specialized Match and Plug-In Post Match are in the same level——level 3. So the example we list here may also apply to Plug-In Post Match. Then we change the postcondition part into the following one using the rule of Plug-In Post Match. And run it on the Spec# online verify tool, there is no warnings, no errors, it's correct, just as we expected.

```
ensures ( ((result == ((n == 0) ? 1 : product{int j in (1..n); j})==>(result ==
product{int j in (1..n); j})));
```

If we use the rule of Plug-In Match, for the example of Generalized Match, the postcondition part is as the following, yes, it works, it is correct.

```
ensures (((3 <= n)==>(0 <= n))==>((result == ((n == 0) ? 1 : product{int j in
(1..n); j})==>(result == product{int j in (1..n); j}))));
```

But if we use the rule of Plug-In Match, using the example of Specialized Match, the postcondition part is wrong now, for n>=0 would never implies n>=3. So these examples explain well why in Fig 2.4: Lattice of function specification matches, Generalized Match, Specialized Match and Plug-In Post Match are in the same level and their level are lower than Plug-In Match.

5.3 Conclusion

Through analyzing a list of examples, we have a further understanding of the algorithm proposed in article [1] and the relationship between the algorithm now is more clearly and understandable. It has proved the algorithm apply to our project perfectly.

Chapter 6 Evaluation

In this chapter we will show the performance of our solution, present our results as well as analyze it and the application of this project. We're also going to make a compare to solutions which already have.

6.1 Performance of the Solution

The match algorithm we used is proposed by Amy Moormann Zaremski and Jeannette M. Wing's on "Specification Matching of Software Components" [1]. In Fig 2.4, we can have an overview of different levels of matching. While actually, in Amy Moormann Zaremski and Jeannette M. Wing's latest article, two more matching methods are add in Fig 2.4——Guarded Plug-In Match and Guarded Post Match, these two matches are both belong to Generic Pre/Post Match. The definition of Guarded Plug-In Match is: `(pre2==>pre1) && (pre1&&pos1==>pos2)`, and the definition of Guarded Post Match is: `pre1 && (pre1&&pos1==>pos2)`, we didn't analyze this two matching method in this thesis. For the analyze method is exactly the same as we do in Chapter 5, using truth table and give an example could explain the matching method well.

One question reader maybe concerned about is does our tool generate correct code and how often the two Spec# programs' specification will be similar about the output. As we know, the process is signature matching, rename, print and then verification. The rename and print process do not make any influence on "how often the Spec# programs' specification part similar", while the signature matching does. As we mentioned before, what's we use here is exact signature, it has both advantages and disadvantages. The disadvantages would be it may miss some part of match while the advantage is the point here; it improves the success chance of match as well as much more accurate.

To have a general idea of how this tool works, we can have a look at figure 4.3, we apply the Plug-In Match rule on the program (as it should be), figure 4.3 shows the output of our tool, and then we paste the print out program on the on line Spec# verify tool to verify it (we can also do it automatically by using web knowledge, as it is not the main part of our project, we just paste the program on the website manually), as we expect, there is no warning no errors, it is correct.

And then we apply the exact pre/post rule to it, the ensure part change from Ensures1 to Ensures2:

Ensures1:

```
ensures ((y>=5) ==>(3<=y)) &&((result*result <= y && y < (result+1)*(result+1))
==>(result*result <= y && y < (result+1)*(result+1)));
```

Ensures2:

```
ensures    ((y>=5)    <==>(3<=y))    &&((result*result    <=    y    &&    y    <
(result+1)*(result+1))<==>(result*result <= y && y < (result+1)*(result+1)));
```

While verify program with Ensures2 on the online Spec# verify tool, there is a warning here, said Method Test1.ISqrt (int y), unsatisfied postcondition: ((y>=5) <==> (3<=y)); so it detected the error automatically as we expected. This example shows us the different between different levels of match.

For Exact Pre/Post Match the two programs are exactly the same expect that the parameter name may not the same, while for the Plug-In Match, it applies to the rule (Pre2 pre1) && (post1 post2); which means that the precondition of the program2 should be no weaker than the precondition of the Spec# program1, and the postcondition of the Spec# program1 should be no weaker than the precondition of the Spec# program2. It relaxes the conditions comparing to the Exact Pre/Post Match. It also explains Fig 2.4 Lattice of function specification matches why that level 1 is above level 2 and so on. For in Fig 2.4 the topper the position is the stronger the condition is.

6.2 Comparing to Solutions Already Have

Our tool could deal with variable renaming as well as subtype. And the two Spec# programs would not be exactly the same, even they are similar our tool could recognize and gives the relevant similar level output. It is more precise.

The main idea of the tool is based on the algorithm proposed by Amy Moormann Zaremski and Jeannette M. Wing. In their article "Specification Matching of Software Components" they proposed Lattice of function specification matches. Comparing to this article, we work out examples for every match method and do case study to show the difference between these matches, our examples listed in this article also proved that the strength of the relationship between different levels of matching. Amy Moormann Zaremski and Jeannette M. Wing give examples from their implementation of match specifications using the Larch Prover. While in this project, we apply this algorithm on Spec# match specifications. This is also a good proof of this algorithm can be applied to another language. So there is a big possibility that the algorithm proposed by Amy Moormann Zaremski and Jeannette M. Wing can be applied to many other different languages. While the way to realize the algorithm on Spec# programs could be used on other languages.

For example, it maybe apply to java code match. There is already a Class named ASTMatcher for java code match in eclipse, public class ASTMatcher extends Object. Concrete superclass and default implementation of an AST subtree matcher. For example, to compute whether two ASTs subtrees are structurally isomorphic, use n1.subtreeMatch (new ASTMatcher(), n2) where n1 and n2 are the AST root nodes of the subtrees. Subclasses may override (extend or reimplement) some or all of the match methods in order to define more specialized subtree matchers. However this method it offers here can only compare the AST which are exactly the same or one AST is a subtree of another AST, it could not compare in precise and it also use structural matching,

if use semantic match method which proposed in this article [1], it would be more precise.

6.3 Application

6.3.1 Preparing for C# Code Match

The C# Language Specification is the definitive source for C# syntax and usage. The Spec# contains detailed information about all aspects of the language, including many points that the documentation for Visual C# doesn't cover [8]. C# code with Spec# language as its specification, for the target of code reuse, the Spec# specification match and C# code match could benefit each other.

One idea is Spec# specification match could be as a filter the C# code match, in the above-mentioned, as the Spec# specification match using the signature matching as a filter, and how the signature matching plays an important role in the whole match, in this project, we use the exact signature match, the exact signature match here means the signature return type, parameter number, and parameter type should be exactly the same, that means if we use this rule of signature match here, and using the Spec# match specifications as the filter of C# code match, we will miss some part.

While we still use the exact signature for the shortage we proposed here could be offset by C# code. C# code match can do separately, so C# code match and Spec# specification match could complement each other. The shortage of using exact signature matching is two parts. Apart from Example<2> which the Parameter types do not match, another part it may miss is while the parameter number is not matched, the specification part could still be similar.

Example<3>

(Parameter number does not match):

```
Program 1:
Public class Test1
{
    int add (int x, int y, int z)
    Requires x>0 && y>0;
    Ensures result >0 ;
      {
        Return x+y;
      }
}
```

```
Program 2:
Public class Test2
{
    double add (int x, int y)
    Requires x>0 && y>0;
    Ensures result >0;
    {
      Return x+y;
    }
}
```

Fig 6.1: Another Example which can’t detected by our Tool

These two programs in example<3> do the similar thing, while using the Spec# match specifications as a filter for C# code match, the output will be the

two C# code do not similar, it is not what's we expected (the same problem as in example2, that the parameter type do not match).

Yes, if we insist using the Spec# specifications matching result as a filter for C# code match, there are two ways to solve this problem, one is while matching Spec# specifications we do not use the exact signature matching as a filter, the second method is that we do specification match first directly without using signature matching as a filter, and then implement the process proposed in chapter 4—— signature matching, rename, print and verify. But there is no need to do such a job, as we mentioned before.

That we could not use the Spec# match specifications as a filter for C# code match, we combine Spec# match specifications with C# code match as complementary, that we put them in the same level as for retrieving for each other, as in figure 2.3. So when the Spec# specification matched the related C# code must have some relationship, in reverse, when the C# code matched the Spec# specification related must have some relationship.

6.3.2 Application in Other Areas

In Amy Moormann Zaremski and Jeannette M. Wing's article "Specification Matching of Software Components", it has studied the match specifications application in three areas: retrieval for reuse, substitution for subtyping, and determining interoperability in details.

In this thesis, we focus on the implementation of these algorithms on matching Spec# specifications. In chapter 4, we show the whole process of how to realize it in details. As a lot of languages have common characteristics so could use the process we proposed in this thesis in other languages. For example, boogie, there is also a class for printing out, if the printer in boogie likes the one in Spec# which doesn't print the thing we actually need, we could use the text operator method as we suggested in this thesis.

Another application of using the implementation method is JML. JML is the specification language of java, the relationship between JML and java is quite similar with the relationship between Spec# specification and C# code. Java like C# is also a very popular language, if we could make java code reusable, it can be applied in many areas and of great significant. Actually there already something has been done related to this [16].

6.4 Conclusion

In this chapter, we've made an evaluation for our tool through its performance and comparing to other related methods and tools, discussed both the advantage and disadvantage of our tool. We also show the possible application area for our tool.

Chapter 7 Conclusions

7.1 Summary

This project aimed to match Spec# specifications. We have developed a tool which can detect how similar two Spec# specifications is based on algorithm proposed Amy Moormann Zaremski and Jeannette M. Wing on article "Specification Matching of Software Components". We also list examples to explain different levels of match, show the difference of these matches as well as the relationships between different levels of match.

There are some limitations of our tool too. As mentioned before, in example<2> and example<3> even the specification part is exactly the same. However, the output shows they are not similar. We have two ways to solve this problem but there is no need so we give up using the way in this project. There is a big space to improve it to make the match more accurate. Finding the correspondence in Spec# match specifications and C# code matching and how they two could help each other in software reuse is another space which could improve in further study.

We didn't do the last step either——paste the Spec# program on the Spec# on line verify tool automatically and run it automatically to show the result, for it is not an important part in our project and study, as well as the tension of time if I am familiar with the knowledge of website, it would not spend that much time. Our main target in this project is to find a way to match Spec# specification, and to promise the result is as accurate as it could be; we've done perfectly on this point. Though the idea of doing Spec# specification match was originally motivated by doing the preparation work for the C# code matching, it is also applicable to other areas of software engineering as well, for example, JML matching. The matching method we used in this quite normal, so it is supposed can be used in many different kinds of specification matching.

7.2 Future Work

In the future work, establishing the link between Spec# specification match and C# code match is a very meaningful job. What's more there is a big space to expand our tool. For example, now the output of the supposed interface is in which level the Spec# specification part match, in future, it may improve into showing how similar they are in more details, such as using an exact number 78% similar. The tool in the future may also generate a list of report to show clearly the similar part and different part, the best thing is the tool could give the user some suggestions automatically. Finally, we may also build a library of Spec# programs in the future, than we can use it for retrieve, and then it is more

convenient and accurate for us to do the Spec# specification matching.

7.3 Conclusions

Based on the examples and the analyze we do in chapter 5, the algorithm fit on our method for Spec# specification quite well. It is quite accurate, and when it said these two Spec# specifications are similar, it must be.

We do the match with the semantic retrieve process based on a very good paper——Specifications Matching of Software Components. It more effective and accurate than structure matching as we has explained before, and by building a Spec# specification match engine, we demonstrated the feasibility of the ideas that mentioned in Amy Moormann Zaremski and Jeannette M. Wing's article, the method we use here is very effective. For the printer part, the text operator method is much easier than the formal print method.

Reference

[1] Amy Moormann Zaremski and Jeannette M. Wing: Specification Matching of Software Components. ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 4, October 1997.

[2] Mike Barnett, Manuel fähndrich, K. Rustan M. Leino, Peter Müller, Herman Venter. Specification and Verification: The spec# experience: June 2011, Communications of The ACM.

[3] Wei-Jin Park, Doo-Hwan Bae: A two-stage framework for UML specification matching. Information & Software Technology 53(3): 230-244 (2011).

[4] W.B. Frakes, K. Kang, Software reuse research: status and future, IEEE Transactions on Software Engineering 31 (7) (2005) 529 – 536.

[5] Stefan Haefliger , Georg von Krogh, Sebastian Spaeth: Code Reuse in Open Source Software. Published online before print November 9, 2007.

[6] W. Yang. Identifying Syntactic differences Between Two Programs. Software - Practice and Experience, 21(7):739–755, 1991.

[7] Karina Robles, Anabel Fraga*, Jorge Morato, Juan Llorens.Towards an ontology-based retrieval of UML Class Diagrams: Information and Software Technology 54 (2012) 72 – 86.

[8] Fausto Giunchiglia, Mikalai Yatskevich, Pavel Shvaiko. Semantic Matching: Algorithms and Implementation: Journal on Data Semantics IX, 2007 – Springer.

[9] Ioannis T. Kassios, Peter M¨uller, and Malte Schwerhoff. Comparing Verification Condition Generation with Symbolic Execution: An Experience Report: Springer-Verlag Berlin Heidelberg 2012.

[10] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness: Microsoft Research 2010.

[11] C. Hoare, J. Misra, G. T. Leavens, and N. Shankar. The verified software initiative: A manifesto. ACM Comput. Surv., 41(4):1-8, 2009.

[12] Mike Barnett, Manuel Fähndrich and K. Rustan M. Leino : Specification and Verification: The Spec# Experience. 2009.

[13] Greg Nelson. Techniques for Program Verification: CSL-81-10 JUNE 1981.

[14] ZAREMSKI, A. M., AND WING, J. M. Signature Matching: a Tool for Using Software Libraries: ACM TOSEM (Apr. 1995).

[15] K. Rustan M. Leino, Peter Müller. Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs: Advanced Lectures on Software Engineering, 2010 – Springer.

[16] Johannes Henkel, Amer Diwan. Discovering Algebraic Specifications from Java Classes: Springer-Verlag Berlin Heidelberg 2003.

[17] Frakes, W.B. and Kyo Kang, (2005), "Software Reuse Research: Status and Future", IEEE Transactions on Software Engineering, 31(7), July, pp. 529-536.

[18] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview: In CASSIS 2004, LNCS vol. 3362, Springer, 2004.

[19] K. Rustan M. Leino, Rosemary Monahan. Program Verification Using the Spec# Programming System :
http://www.cs.may.ie/~rosemary/ETAPS-SpecSharp-Tutorial.pdf