INVARIANTS GENERATION FOR METHOD OVERRIDING USING ABSTRACT INTERPRETATION

SITI HAFIZAH AB. HAMID

THESIS SUBMITTED IN FULFILMENT
OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE &
INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2013

Abstract

Software verification is an important element of software reliability. The significance and importance of verification have been recognized by Bill Gates in his speech in WinHEC 2002. The software verification allows program's specification to be formally proved to ensure the specification verified the program before its execution time using static analysis. However, in the context of object-oriented program, studies show there is a need to have formal specifications for method overriding because the overriding feature plays important role in allowing program reusability. This thesis develops an abstract formal framework for invariant generation of static analysis for method overriding in object-oriented program using inheritance. It focuses on late bound method in the class invariants generation. There are two main problems arise during the process of generating class invariant which are reverification of class invariant and over-approximation of late binding call. In the context of method overriding, the problem of late binding call happens when the abstract semantic function uses behavioral subtyping that is restricted to the rule of contravariance and covariance. The abstract formal framework using abstract interpretation theory is proposed to overcome the problem above. The framework exploits the capability of abstract interpretation method in making program analysis automated. It also overcomes the problem of generating the invariants for late binding call of method overriding with less restrictions rules of lazy behavioral subtyping method. The use of lazy behavioral subtyping results to the overridden method semantics has a not over approximated invariant. The framework produces two equations for two invariants, which are modular invariants for inheritance and invariants for method overriding. A scenario based evaluation is conducted to validate the invariants and to compare the proposed

framework using lazy behavioral subtyping with the framework using behavioral subtyping.

Abstrak

Pentahkikan perisian adalah satu elemen penting dalam kebergantungan perisian. Signifikan dan kepentingan pentahkikan telah diiktiraf oleh Bill Gates dalam ucapannya di WinHEC 2002. Pentahkikan perisian membenarkan spesifikasi program diformalkan secara pembuktian untuk memastikan spesifikasi terebut mentahkik program sebelum masa laksananya menggunakan analisa statik. Akan tetapi, di dalam konteks program berorientasikan objek, kajian menunjukkan bahawa terdapat keperluan untuk mempunyai spesifikasi formal bagi *method overriding* kerana ciri *overriding* memain peranan penting dalam membenarkan keboleh-gunaan semula program. Tesis ini membina rangka kerja formal yang abstrak untuk menghasilkan *invariant* bagi analisa statik untuk *method overriding* dalam program berorientasikan-objek menggunakan pewarisan. Ia memfokus kepada kemodularan dan fungsi ikatan lewat dalam menghasilkan *class invariants*. Terdapat dua masalah utama yang berbangkit semasa proses menghasilkan *class invariants* iaitu pentahkikan ulangan *class invariants* dan anggaran melampau bagi panggilan ikatan lewat. Dalam konteks *method overriding*, masalah panggilan ikatan lewat berlaku ketika fungsi semantic abstrak menggunakan *behavioral subtyping* yang mengikut peraturan *contravariance* dan *covariance*. Satu rangka kerja formal yang abstrak dicadangkan untuk mengatasi masalah tersebut. Rangka kerja itu mengambil peluang keupayaan *abstract interpretation* dalam menjadikan analisa program automasi. Ia juga menyelesaikan masalah menghasilkan *invariants* untuk panggilan ikatan lewat bagi *method overriding* dengan kurang ketegasan peraturan oleh kaedah *lazy behavioral subtyping*. Penggunaan *lazy behavioral subtyping* memberi keputusan kepada semantik *overridden method* mempunyai anggaran *invariant* yang tidak melampau. Rangka kerja tersebut menghasilkan dua persamaan untuk dua *invariants*, iaitu *invariant* bermodular bagi pewarisan dan *invariant* untuk *method overriding*. Satu penilaian berasaskan senario dijalankan untuk mengesahkan *invariants* dan untuk membandingkan rangka kerja menggunakan *lazy behavioral subtyping* yang dicadangkan dengan rangka kerja menggunakan *behavioral subtyping*.

Acknowledgment

Alhamdulillah.

I would never have been able to finish my thesis without the guidance of my supervisor, help from friends, and the support from my family.

I would like to show my gratitude to my supervisor, Prof Dr Mohd Sapiyan Baba for his excellent guidance, advice, and patience throughout my time as his student. I would like to thank Associate Prof Dr Abdullah Gani who motivated me unconditionally and responded to my questions and queries so promptly. Special thanks to Prof Dr Wan Ahmad Tajuddin Wan Abdullah who gave me the best suggestion on references on subtyping. I would also like to thank Dr Anthony J.H. Simons who promptly answered my questions on behavioral subtyping and Associate Prof Chin Wei Ngan who gave me a great idea on static analysis of object-oriented languages.

Finally yet importantly, I would like to say a big thank you to my husband and kids who always believe in me.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

In the name of God,
who is Most Gracious and Most Merciful

Reliability is an important aspect of software. Softwares, when used, are usually assumed to be reliable. However, there is no guarantee that a software will function as intended or will not break. That is why most softwares come with disclaimers but not guarantees. Therefore, if a software damages consumer's data, there is no compensation to be made. To ensure reliability, softwares are checked for correctness before being deployed. This is done using software testing.

Software testing is a process designed to ensure that a program code does what it is meant to do (Myers, 2008). Consider the following simple example:

```
while (number<=3) {
if (number<=3) then
 number = number + 1; }
System.out.print(number);
```

The *if* statement adds 1 to *number* until it exceeds 3, then the program displays its value. A simple unit testing is done, for example, by initializing *number* to 0. Table 1.1 below shows the executed code for different values and their results:

Table 1.1: Executed code with values and results

Value for number	**Executed code**	**Result**
0	number = number + 1	number = 1
1	number = number + 1	number = 2
2	number = number + 1	number = 3
3	number = number + 1	number = 4
4	cout<<number	Printing value of number which is 4

The result shows that the program works successfully. However, if *number* is not initialized at the beginning of the code, the program will take a number from the heap memory for example -234987 to execute the code. The code will then display a similar output, but with longer execution time. Hence, software verification is used to check the program's correctness.

Software verification is a process of checking program correctness based on software specification of the software. The significance and importance of verification have been recognized by Bill Gates in his speech in WinHEC 2002:

> Things like even software verification has been the Holy Grail of computer science for many decades, but now in some key areas, for example, driver verification, Microsoft is building tools that can do actual proof about the software and how it works, in order to guarantee reliability.(Gates, 2002) .

The software verification allows program's specification to be formally proved, for all possible runs that is held during program execution (Parkinson, 2005). The checking process is based on set theory logic and abstract algebra. It is applied to various types of programming languages including object-oriented programming language.

1.1 Motivation

Software reliability in any software development is crucial to ensure that a software does without fail. Software systems continue to grow in size and complexity, as can be seen in retailing, manufacturing, telecommunication, and transportation that utilize aviation system, real-time system, concurrent system, hybrid system, reactive system, and web-based system. Sometimes, a program stops after running for certain duration. This is seen on flight billboard monitor at the airport, announcement or advertisement billboard, or shopping mall map system. However, this situation is unacceptable to any reactive system that needs to run continuously without stopping. It is crucial for the reactive system of medical hardware because a failure can be fatal. For example, the failures by Ariane 5 in 1996 to detect the coordinate of its location, results in more than USD370 million loss even though there is no human in the flight during the crash (Dowson, 1997). Such software failure happened again in 2004, when Mars Rover Spirit failed to send data to earth due to lack of flash memory capacity (Reeves, 2004). Both are prominent examples of software failure due to lack of software reliability and not because of hardware failure. In the current object-orientation software development, software is structured and developed in components. However, break downs still persist due to its complexity.

1.2 Statements of Problems

Due to the weakness of human being, errors are missed during the verification process. Thus, automated process is required to overcome the problem. It is done by having automated specification production where invariants generation can facilitate in producing accurate result for the verification process. However, for the invariants in object-oriented program, specifically in the context of method overriding, automation is achieved by considering one main issue; late bound method calls (or late binding call). There are two problems in developing the invariants for late binding call.

1. Restriction rule on the notion of behavioral subtyping – For the program method, the rule follows the notion of contravariant and covariant, which enforces properties of a method redefined in a subclass must satisfy all superclass properties.
2. Class invariant keeps on changing every time new sub class is added into the structure of inheritance (reverification of old classes) – when the inheritance hierarchy is extended with new subclass, the whole structure is verified again including the superclass and subclass that have been verified previously.

1.3 Research Objectives

This thesis aims to develop an abstract formal framework for static analysis of verification on method overriding. Based on the aim, the thesis objectives are as follows:

1. To analyze current frameworks on generating invariants in object-oriented programming language for static analysis focusing on programs with method overriding.
2. To design an abstract formal framework for verification on method overriding focusing on invariants generation using abstract interpretation and lazy behavioral subtyping.
3. To validate the formalization of the abstract formal framework using case studies.

1.4 Research Methodology

This research consists of four main tasks; 1) analysis of related works, 2) development of abstract formal framework, 3) proof of concept of the abstract formal framework, and 4) validation of the abstract formal framework. All of the tasks are based on the three objectives stated in §1.3. The methods used achieve each objective is summarized in Table 1.2 below. All objectives are achieved by contribution lists in the table.

Table 1.2: Research Methods

Objectives	Methods	Chapter/ Section	Contribution
1. To analyze current frameworks on generating invariants in object-oriented programming language for static analysis focusing on programs with method overriding.	– Review articles on program analysis, formal verification, and verification on object-oriented programs. – Summarize the importance of method overriding. – Summarize the concept of program static analysis. – Analyze methods of static analysis • Compare the methods of static analysis based on lines of code, human intervention, and concrete or abstract characteristics. – Conduct an analysis on related works of verification object-oriented programs with subtyping. • Compare the related works with criteria related to non-reverification feature of method overriding verification. • Compare the related works with techniques	§2.1-§2.4 §2.3 §2.4-§2.5 §2.6 §2.9	1. Analysis of works on verification program using static analysis 2. Analysis of static analysis methods

	used to verify method overriding.		
2. To design an abstract formal framework for verification on method overriding focusing on invariants generation using abstract interpretation and lazy behavioral subtyping.	– Define concrete semantics of object-oriented programming language (OOPL). – Define abstract domains – Define abstract semantics – Prove the abstract semantics – Develop equations on invariants generation for class, inheritance, and method overriding.	Chapter 4	Equations for: 1. invariant in inheritance 2. invariant in method overriding
3. To validate the formalization of the abstract formal framework using case studies.	– Validate the equations using two case studies. – Evaluate the result of invariants from the equation used lazy behavioral subtyping and behavioral subtyping using the case studies.	Chapter 5	1. Result analysis of case studies using invariant generation by behavioral subtyping and lazy behavioral subtyping.

1.5 Thesis Outline

This introduction chapter is followed by a literature review on automated software verification for method overriding in **Chapter 2**. The importance of method overriding is discussed. This includes the usage of method overriding as reusability and specialization in programming. Then, static analysis methods and related works are analyzed to identify improvements needed for better methods for verification and to understand problems involved during verification. In **Chapter 3,** problems on automated linear invariants generation are discussed using a language called methL. Based on the problems identified, list of potential solutions are studied. Therefore, a solution is proposed that shows methods and features to solve each problem. In **Chapter 4**, the formalization of abstract formal framework of invariants generation for method overriding is presented. The framework consists of an introductory example, concrete syntax domains, concrete semantic domains, abstract semantics, and abstract domains of object-oriented programming languages. **Chapter 5** discusses the results of the generated invariants using two case studies. Both case studies generate invariants using behavioral subtyping and lazy behavioral subtyping. Lastly, in **Chapter 6**, the thesis ends with an explanation on the overall work and future works that can be done. It also concludes its contribution to the body of knowledge in term of its strengths and limitations.

Chapter 2
Automated Software Verification for Method Overriding

History does not repeat itself, it does rhyme
-Mark Twain

This chapter provides the background information on automated software verification for method overriding and related works. Its aim is to show findings from reviews of literatures on suitable method for conducting static analysis. It discusses the important features and problems on automated method overriding verification and its importance towards software development. It also analyses methods and related works by comparing them using features and variables of static analysis. The analysis determines the methods and techniques in producing invariants.

2.1 Object-Oriented Programming Language (OOPL)

A programming language is a language to program a system or software executed by a computer or a machine-readable device. Simula is the first programming language that models objects of a simulation as program objects. Later, Stroustrup (1987) came out with the idea of managing programs based on class and object; which is adopted from Simula. A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics (OMG, 2001). An object is an instance that originates from a class. It is structured and behaves according to its class (OMG, 2001). Therefore, Stroustrup (1987, p.70) defined “object-oriented programming as a programming using inheritance”. Then, in 1997, an object-oriented design is defined as the construction of software systems as structured collections of abstract data type implementations (Meyer, 1997, p.59). By considering all definitions above, there are three characteristics of OOPL; abstraction, inheritance, and polymorphism. The thesis focuses on inheritance.

2.2 Inheritance

Programmers use object-oriented technique in their design and program due to its program reusability for software maintenance (Engels and Groenewegen, 2000). The main characteristic that supports program reusability is inheritance. Inheritance in a program means the program must consists at least two classes which are superclass and subclass (Dahl et al., 1966). Superclass acts like a parent class where it has data and methods that are inherited by the subclass that acts like a child class. Figure 2.1 illustrates the inheritance relationship. The Figure 2.1 shows there are two classes called as *GeometricFigure* and *Rectangle*. The *GeometricFigure* has one data named *side* that is type of integer and one method named *calcArea()* that return an integer value. The *Rectangle* has one data named *area* that is of type integer. It also has one method named *calcArea()* that returns an integer value. The arrow shows a direction from *Rectangle* to *GeometricFigure*, which means *Rectangle* is subclass to *GeometricFigure*.

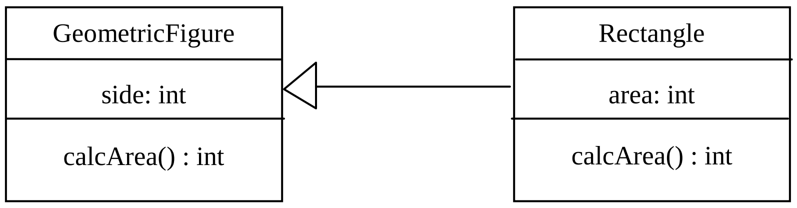


Figure 2.1: Inheritance Relationship

Taivalsaari (1996) defined the basic idea of inheritance as the fact that new object definitions can be based on existing ones; when a new object class is defined, only those properties that differ from the properties of the specified existing classes need to be declared explicitly, while the other properties are automatically extracted from the existing classes and included in the new class. Using the incremental modification mechanism proposed by Wegner et al. (1988) and Taivalsaari (1996), inheritance is presented using a maxim

$$R = P \oplus \Delta R$$

where R is newly defined object or class,

P is properties inherited from existing object or class,

ΔR is incrementally added new properties that differentiate R from P,

$\oplus$ is an operation to combine ΔR with the properties of P.

Therefore, the operation $\oplus$ makes R contains the properties of P and its own properties. However, the incremental modification of ΔR may introduce properties that override those of P so as to redefine or cancel certain properties of P.

Meyer (1997) defined inheritance using two different views: module view and type view.

(1) A module consists of a set of program services to be used by the end users. With inheritance, every new service is provided without defining all the services that have

been developed using the module. It is done by simply adding new services to the existing modules. Inheritance, as in the module view, is meant for reusability purpose where the inheritance is used to start from the designing phase of software development.

(2) A type consists of a set of objects with operations. Using the type view, inheritance is meant for reusability and extendibility represented the is-a relation using dynamic binding. Dynamic binding or dynamic dispatch is a process when a method of an object is generated or called not at compile time but at run time. The object uses inheritance hierarchy to decide what method to apply to itself. Since the process of binding object and method occurs later during run time, the process is also called as late binding call.

Inheritance allows programmers to modify their code incrementally. For example, by referring to Figure 2.1, the programmers code a system by having a superclass named *GeometricFigure* and a subclass named *Rectangle*. The system can calculate area of other geometric figures by adding new subclasses, e.g., *Circle* and *Triangle*. Therefore, the system is a complete system where it can calculate area of any geometric figures. The easy modification process is important to fulfill the program's later requirement. It is not only classes that are added to the system but the modification can also be applied to the class methods. Methods that have the same purpose with the same name can generate different outputs based on their definitions. For example, method *calcArea()* exists in both class *GeometricFigure* and *Rectangle* where both of them has one purpose which is to calculate area of a figure. However, both methods give different outputs. This is called method overriding.

2.3 Method Overriding

Objects use methods in a class to perform operations. According to Martin and Odell (1998), a method is a processing specification for an operation. It determines the behavior of the object. There are two type of method in OOPL, which are method overloading and method overriding.

Method overloading is when two or more methods have the same name but different argument or parameter. It is used when the methods have the different requirement represents same conceptual operation using same method name (Gil and Lenz, 2012). For example, in Figure 2.2, there are two *imprint()* methods. The first one has one argument named *radius* and the second one has two arguments named *x* and *y*. Even though they use the same method name, each method can be called at different time based on their arguments.

```
void imprint (int radius) {
  System.out.print("draw a circle");}

void imprint (int x, int y) {
  System.out.print("draw a rectangle") ;}
```

Figure 2.2: *imprint*() methods

On the other hand, method overriding is when methods with the same name, arguments or parameters, and return types but different operation in both superclass and subclass. It is one of the essences of inheritance that allows methods in subclass to override the implementation of already defined superclass (Wegner et al., 1988). This allows the implementation to be specialized and still reusable. For example, in Figure 2.3 for the class

Staff, we define salary for all staff by adding the basic salary with 10% or 5% of the profit. However, class *SalesPerson* has extra salary with extra commission.

```
public class Staff {
  private int basicSalary;
  public Staff() { }
  public void giveComission(int profit) {
      basicSalary = basicSalary + 0.1 * profit; }
};

public class SalesPerson extends Staff {
  private int salary;
  public SalesPerson () {  }
  public void giveComission(int profit) {
      salary = basicSalary + 0.05 * profit; }
};
```

Figure 2.3: class *Staff*

The implementation of method overriding realizes the is-a relationship of inheritance. The relationship describes an object as a special type of another superclass. Not only can the subclass methods share properties from its superclass method, they can also redefine the superclass method's operation in the subclass method. The capability of sharing the properties is called subclassing, whereas the capability of redefining is called subtyping. Formally, "subclassing is an implementation mechanism for sharing code and representation" (Taivalsaari, 1996, p.446). Subtyping acts as a type signature that exists in inheritance with substitutability principle.

2.3.1 Use of Method Overriding

Method overriding is applicable for the purpose of reusability and specialization. Reusability is the capacity for something to be used more than once. In object-oriented programming, the programmers can use created object many times for different scenarios.

The object can be a program, component, or interface. Specialization means making something suitable for a specific aim. In object-oriented programming, a specialized object is instantiated using specific data and methods based on its requirement. It is more specific compared to the object it is generated from.

2.3.1.1 Method Overriding for Reusability

Reusability is a key element of object-oriented. In inheritance, the reusability feature is implemented by creating a subclass that uses data or method of superclass. Reuse concept is most beneficial in object-oriented because it saves a lot of time and energy in coding and understanding code. A programmer can reuse the existing code by modifying the code to suit new applications. In fact, there are many objects that is easily called up and combined together to produce applications. The ability to reuse code relies on the ability of the programmer to develop a big application from existing smaller components. Therefore, the programmer has to know how to install, manage or package the components. With method overriding, the programmer can use the same name of the existing method in the superclass to appear again in subclass. However, the method has different definition based on its requirement. The overridden method can call the method in the superclass as part of its definition. Therefore, the programmer does not have to think of other names for the method if the action is more or less the same. For the method *draw()* in Figure 2.4 below, it can draw different shape depending on the definition.

```
public class Shape {
  private int side1;
  public Shape() { }
  public int getData() {
      return side1; }
  public void setData(int x) {
      this.side1=x; } };

public class Square extends Shape {
  private int side2;
  public Square() {  }
  public int getData() {
      return side2; }
  public void setData(int a) {
      this.Shape::setData(a);
      this.side2 = a; }
  public void draw() {
      for(int i=1; i<=this.Shape::getData(); i++) {
       for(inti=1; i<=this.getData();i++)
        System.out.print("*");
      System.out.println("\n"); }  }
```

Figure 2.4: *Shape* class and *Square* class

Shape class is a superclass of *Square* class where the code segment is to draw a square shape by using asterisk '*' as shown in Figure 2.4. The *Shape* class has one data as *side1* and three methods: constructor, get value of *side1* (*getData*()) that gives a value to *side1* (*setData*(*int x*)). Square class has one data which is *side2* and four methods which are constructor, get value of *side2* as defined in *getData*(), set same value for both *side1* and *side2* as defined in *setData*(*int a*), and draw a square as defined in *draw*(). Overridden method from subclass *Square* which is *setData*() reuses code from method *setData*() of super class by calling *this.Shape::setData(a).* The purpose is to give the same value for both sides.

2.3.1.2 Method Overriding for Specialization

Specialization is implemented in subclass in order to make the subclass's behavior more specific. It gives a privilege to superclass to define a method as general as it can be. The overridden method in the subclass has a full definition of what action the method exactly has to do. *Shape* class literally is understood as any shape. It can be circle, rectangle or triangle. It cannot draw any shape until it is fully defined by its subclass. Figure 2.5 shows class *Rectangle* that has been defined so that it can draw a rectangle.

```
public class Shape {
 private int side1;
 public Shape() { }
 public int getData() {
      return side1; }
 public void setData(int x) {
      this.side1=x; } };

public class Rectangle extends Shape{
 private int side2;
 public Rectangle() {  }
 public int getData() {
      return side2; }
 public void setData(int a) {
      this.Shape::setData(a*5);
      this.side2 = a; }
 public void draw() {
      for(int i=1; i<=this.Shape::getData();i++) {
      for(inti=1; i<=this.getData();i++)
      System.out.print("*");
      System.out.println("\n"); }  }
```

Figure 2.5: Class *Shape* and subclass *Rectangle*

The subclass *Rectangle* is implemented to specialize the behavior of class *Shape* which is to draw a rectangle. It has one data, which is *side2*, and four methods, which are method constructor, method to get value of *side2*, method to give new value to *side1* of super class by multiplying five with the value, and method to draw a rectangle. Overridden method of *setData*() changes the original operation of super class from setting an integer number to *side1* to five times the value of *side1*.

With the explanation in §2.3, we cannot deny the importance of method overriding in software development. Thus, there is a need to ensure the code is well-written and correct to avoid unexpected termination or behavior from the program. To achieve the above, a programmer has to check the program's correctness using software verification.

2.4 Software Verification

Software verification is a formal process to check specification of a program. The idea of software verification is in response to the question: "are we building the program right?". In contrast, software validation is in response to the question; "are we building the right program?" (Baresi et al., 2006). The process of verification is to detect programming errors or to prove the absence of errors. It applies formal method to formalize the program in term of its grammatical well-formedness of the syntax, interpreting the semantics of coded statements in a meaningful and precise way, and inferring information from the formal specification (Lamsweerde, 2000). Hence, it is called formal software verification.

Formal specification used for software verification is an expression of mathematical description of software. At the abstraction level, it is a collection of properties a system should satisfy. It proves the program's correctness by checking the consistency between programs and the expected properties (Lamsweerde, 2000). In 1960s, the specification is done by annotating the code with the states based properties at specific points in the program (Floyd, 1967). These properties are customized with special techniques to cater different kinds of program, e.g., data structured program, concurrent program, and object-oriented program.

In order to conduct software verification, there are two types of program analysis that can be done: static analysis and dynamic analysis. Static analysis uses proven formal specification for correctness purpose. Therefore, a programmer has a chance to correct the program before failure happens. Dynamic analysis, also known as software testing, is used to check program behaviors at actual execution time. The analysis is on the exact code of the program. The programmer does not need to approximate or abstract the behavior of the program. However, the analysis results cannot be generalized for future executions. Therefore, there is no guarantee that the test covers all possible program executions (Ernst, 2003).

2.5 Static Analysis

Jackson et al. (2000) defined static analysis as “the process of examining program code without executing the program in order to obtain information that is valid for all possible executions” (p.133). It offers “static compile-time techniques for predicting safe and computable approximations to the set of values or behaviors arising dynamically at run-time when executing a program on a computer” as explained by Nielson, et al. (2005, p.1). It is used for program optimization and program correctness to ensure software reliability. It enables the checking of the behaviors of the program for all input vectors (D'Silva et al., 2008). The potential errors cannot be found during testing process, but it may appear after the program has been executed for a certain period. The typical examples are null pointer, array bound, division by zero, and buffer overflow.

Three main techniques are used in conducting static analysis. They are assertions, model checking, and abstract interpretation. To illustrate each technique consider the simple code using method overriding in Java in Figure 2.6.

```
public class Shape {
private int side1 = 3;
public Shape(){}
public int getData() {
   return side1;}
public void setData(int x){
   this.side1 = x;}};

public class Square extends Shape{
private int side2=3;
   public Square() {}
   public int getData(){
     return side2;}
   public void setData(int a){
     super.setData(a*5);
     this.side2 = a;}
   public void draw(int i){
     int j;
     for (i=1; i<super.getData();i++){
      for (j=1; j<this.getData();j++)
       System.out.print("*");
      System.out.println("\n");} } }
```

Figure 2.6: Sample Program

2.5.1 Assertion

Assertion is a predicate statement inserted at specific point of a program (Hoare,1981). In 1967, Floyd used assertions as foundation to static proof of program correctness. He specified assertions at the point of the program code to ensure its correctness. Using this idea, Hoare (1969) came out with a set of axioms and rules of inference to proof the assertions or properties of the program using axiomatic semantics, later known as Hoare Logic. The idea is that each program statement must have a precondition and a postcondition using predicate logic expression as {P} S {Q}; where P is a precondition, Q is a postcondition, and S is a statement. The expression is interpreted as “if the assertion P is true before the initiation of a program S, then the assertion Q will be true on its completion” (Hoare, 1969, p.577).

The illustration of assertion is seen as in comment (/* */) in Figure 2.7. The assertion is written using Java Modeling Language (JML). JML is used to check the correctness of Java program. To represent precondition and postcondition, JML uses a keyword named *requires* for precondition and *ensures* for postcondition. The codes of method *draw()* are statements that are checked by *requires* and *ensures*. The statement *requires i>=1 && j>=1* means the method *draw()* is only executed if *i* and *j* are greater or equal to 1. The statement *ensures i<MAX_LENGTH && j<MAX_LENGTH* means the method produces an output where the value of *i* and *j* are not greater than maximum length of the program’s memory.

```
/*@ requires i>=1 && j>=1
    ensures i<MAX_LENGTH && j<MAX_LENGTH
@*/
public void draw(int i){
  int j;
  for (i=1; i<super.getData();i++){
   for (j=1; j<this.getData();j++)
 System.out.print("*");
  System.out.println("\n");} } }
```

Figure 2.7: Sample Code with Assertions

Tools such as Daikon, LOOP, Julia, Boogie, and ESC/java use Hoare logic for different types of programming languages to check program correctness. For C language, Ernst’s group in MIT has developed a tool named Daikon to discover invariant in C program (Ernst et al., 2007). The tool infers invariants from a program automatically. It captures all inputs in a program and traces all relevant variables with values. Therefore, a programmer does not have to specify the program in order to verify it. For Java language, a tool named LOOP reasons sequential Java codes (Van Den Berg et al., 2001). It is strongly typed and is applied to JavaCard API. However, it does not verify Java bytecode. Java bytecode verification is handled by a tool called Julia (Spoto, 2010). There are many tools to develop programs using C#.Net language (Softworks, 2012). However, there is only one tool that supports verification on C#.Net, which is Boogie (Barnett et al., 2006). It is originally an automatic program verifier for Spec# programs. Spec# programming language is a superset of C# language. It has specification features which named as pre-, post- and object invariant.

2.5.2 Model checking

Model checking is a technique for verifying correctness of a computer program based on a model of states of computation, where it tests automatically whether the model meets the specification of the computer program or otherwise. It is an automatic technique for verifying finite-state reactive systems (Clarke, 1997). The specification is written in temporal logic formula. Temporal logic handles propositions whose truth value evolves over time (Monin, 2003). The reactive system is modeled as a state transition graph. In order to determine whether the state transition graph is satisfied or otherwise, an efficient search procedure is used.

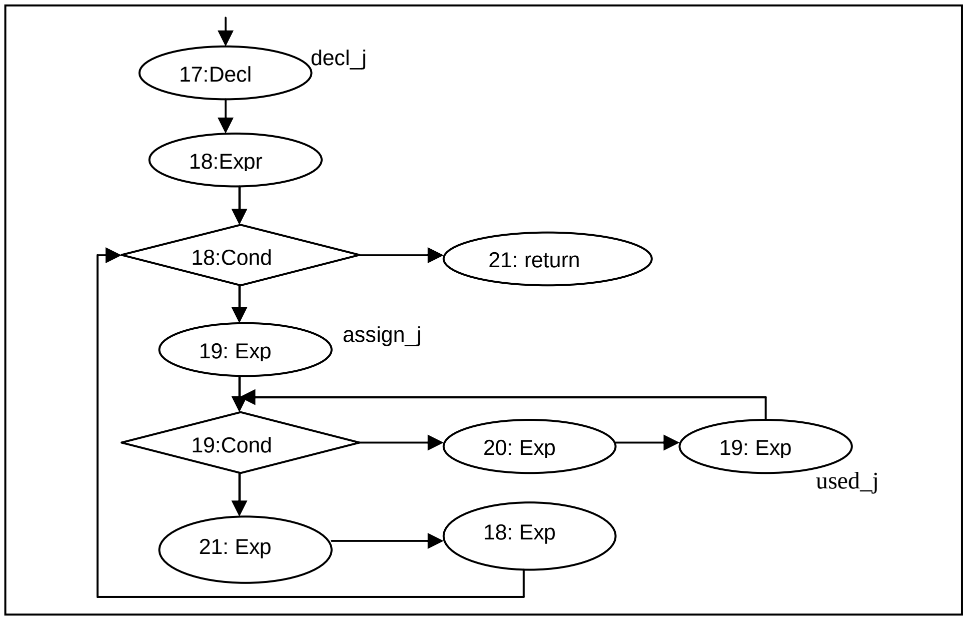


Figure 2.8 : Fragment of the Annotated Control Flow Graph (CFG)

Figure 2.8 illustrates a fragment of the annotated control flow graph (CFG) based on method *draw*() from Figure 2.6. The annotated CFG becomes the foundation of the program analysis using model checking. Based on the CFG, variable *j* is detected. The purpose of this model checking is to detect uninitialized variable in the program. There are three variables which are *decl_j*, *assign_j*, and *used*_j. All three are annotated using CFG. By using NuSMV (Fehnker et al., 2007), a fragment of code is produced as in Figure 2.9.

Every model checking code starts with a *main* module, followed by variables and the flow of the program's graph. The code specifies the program by temporal logic formulas *SPEC AG decl_j => (A ~used_j W assign_j)*, where *AG* is an acronym for *always generally*. The *decl_j* is not to be used until it has a value assigned; otherwise it is not used at all. Therefore, based on the temporal logic formula, the method *draw()* does not produce any warning when the method fulfills the specification. Programmers have to learn how NuSMV works and the syntax for the module *main*. When the programmers write a code inside the module *main* on the specification, the specification determines the error it checks. There are more than 13 lines of code (the case part is not complete) to specify and check the method *draw()*.

```
MODULE main
VAR location : {loc17, loc18, .., loc21}
next (location) :=
 case
  location = loc17 : {loc18};
  location = loc18 : {loc18};
  location = loc18 : {loc19,loc21};
  ....
 esac
DEFINE
  decl_j := location in {loc17};
  used_j := location in {loc19};
  assign_j := location in {loc19};

SPEC AG decl_j => (A ~used_j W assign_j)
```

Figure 2.9: NuSMV code

There are tools for model checking. One of them is called Blast. It is a verification tool to check the safety of C programs (Henzinger et al., 2005). It receives inputs in a specification language, with C like syntax and produces outputs that indicate whether the program satisfies the safety property or otherwise. It implements a lazy abstraction algorithm, which integrates automatic abstraction refinement and model checking. For Java language programs, a tool called Java Pathfinder uses model checking to verify the Java programs safety (Havelund et al., 2000). It translates Java to Promela, which is the modeling language of SPIN model checker. After the program is translated into Promela, Java Pathfinder model checks the program using SPIN. Java Pathfinder has been used in NASA in the research area of space, aviation, and robotics (NASA, 2012).

2.5.3 Abstract interpretation

Cousot (2007) defined abstract interpretation as "a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems". "The specification of an analyzer is an approximation of a semantics, where concrete or exact properties are replaced by abstract or approximate properties" (Cousot, 1996, p.73). For example, the semantics S of a programming language L associates a semantic value in the semantic domain D to each program P in L written as $S[\![P]\!] \in D$ (Cousot, 1996). Many analyses are formalized by abstract interpretation. However, among those associated with semantics are static analysis, data flow analysis, control flow analysis, types, predicate abstraction, and class analysis.

Abstract interpretation uses fixed point of Tarski's theorem to model all possible behaviors of the program (Cousot, 1996). The program is formalized as graphs or transition systems and the behaviors are represented as a set of states, Σ. The states that represent the transition systems use partial trace semantics to execute trace of states; $s_0, s_1, s_2, .., s_n; \{s_n \in \Sigma \mid n \in \mathbb{Z}\}$, s_0 is an initial state, and $\mathbb{Z}$ is the set of integers. The intermediate states $(s_1, s_2, .. s_{n-1})$ is a transition move from one state s_i to the next s_{i+1}, such that $\langle s_i, s_{i+1} \rangle \in t$, where t is a transition relation between one state to its successor state. Then, the partial trace semantics (concrete semantics) is replaced by the reflexive transitive closure (abstract semantics) using Galois connection. Since this abstraction is undecidable (non-computable), a widening or narrowing is used to approximate the semantic abstraction (Cousot, 1977). The process of abstraction is generated by a library for the program. For the method *draw()* taken from Figure 2.6, the invariant that the abstract interpretation produces is

```
//invariant i>=1 && i<MAX_LENGTH , j>=1 && j<MAX_LENGTH
```

Therefore, there is no warning produced on method *draw*() because the *i* and *j* have been assigned to 1 and both conditions do not exceed maximum length of program's memory. Programmers do not need to type the specification statement. The program analyzer produces the invariant statement. Based on the method *draw*(), there is only one specification statement in the form of invariant produced. It is a statement of specifying the minimum and maximum value of *i* and *j*.

The abstract interpretation has been applied to many languages, e.g., Prolog (Mellish, 1986; Bourdoncle, 1993; Marriott et al., 1994; Charlier et al., 2002; Barbuti et al., 2003) and C (Ball et al., 2001; Loding et al., 2008; Michiel et al., 2008). However, for Java, the verification process is made by using its small scale language, e.g. Featherweight Java (Igarashi et al., 2001). Every single small language is made to verify specific property of Java, e.g., class invariant and generalization structure of inheritance as proposed by Logozzo (2007) and a flexible type and effect inference of Java as proposed by Skalka et al. (2005). Bernardeschi et al. (2002), Avvenuti et al. (2003), and Barbuti et al. (2010) verified Java bytecode in term of its security, information flow, and space efficiency. In addition, Pollet et al. (2005) and Distefano et al. (2008) verified automatically to complete Java scale without Java concurrency. For concurrent programming, Codognet et al. (1995) has proposed a verification framework using abstract interpretation and constraint system. However, other researchers have proposed a verification technique to problems related to concurrent programming such as trace semantic (Barbuti et al., 1999), information flow (Bernardeschi et al., 2003), and race condition (Barbuti et al., 2003). Abstract interpretation has also been used to verify applications such as timed concurrent system (Falaschi et al., 2009), mobile communication (Feret, 2001; Nielson et al., 2003; Albert et al., 2005; Barthe et al., 2008), and database system (Toman, 1997; Bailey and Poulovassilis, 1999; Halder et al., 2010; Halder et al., 2011).

2.6 Analysis on the static analysis methods

As explain in previous sub-sections, there are three methods on static analysis which are assertion, model checking, and abstract interpretation. In order to analyze the methods, three features that are important to achieve the thesis's objectives are explained briefly. All three features are applied to the method *draw*() of the code program in Figure 2.6 to compare the capability of each method. Later, comparison of the three static analysis methods given in Table 2.1 provides the justification for the chosen method in verifying method overriding in this thesis.

2.6.1 Features of Static Analysis Methods

All of the static analysis methods are compared based on lines of code, human intervention, and characteristics of concrete and abstract semantics. The three features above are considered because they are generic, performance effective and less error prone.

- Lines of code

  Lines of code are an important element in determining performance of a program. In software verification, it refers to the number of lines need to be verified from the program code. The more lines used, the more time required for the verification (Fenton and Pfleeger, 1998).

- Human intervention

  It refers to the need of programmer's annotation to verify the program code. If the method needs human intervention, it means the method is manual. However, if the method can verify the program code by itself, it means the method is automatic. This feature is important because it avoids human errors during annotation process.

- Characteristics

  There are two types of characteristics that are used to interpret the semantics of the program code; concrete and abstract. For a concrete method, all lines of code or all states of code behavior have to be verified explicitly. However, the abstract method summarizes the code or the states of code behavior; making it simpler.

Table 2.1: Comparison of the static analysis methods

Method name	Lines of code	Human intervention	Characteristics
Assertion	2	Yes	Concrete
Model checking	> 13	No	Concrete
Abstract interpretation	1	No	Abstract

2.6.2 Comparison of static analysis methods

By referring to Table 2.1, model checking asserts more than 13 lines of NuSMV code to verify the method *draw*(). Assertion uses two lines of JML code and abstract interpretation uses only one line of code. Model checking needs more lines of code because after translating the program code into states, it verifies every flow of the code which uses all lines. Abstract interpretation interprets the program code in an abstract way using theorems. No matter which types of semantics the methods use, all three methods produce the same result over the *draw*() method, i.e. *i* and *j* are more or equal to 1 and not more than the maximum length of heap memory capacity.

Only assertion method needs human annotation during the verification process. Therefore, the programmer must know how the syntax of the assertion works. In addition, the programmer has to know which part of the program needs to be annotated. Any missing assertion leads to a less precise result on the verification. Model checking and abstract interpretation are done automatically. Abstract interpretation method is more complicated (heavyweight form of static analysis) compared to model checking because it uses mathematical statements of abstract algebra which is hard to learn (Hall, 1990; Siu, 2001). Model checking method is easy to understand and use, as it uses temporal logic for finite state machine of a program (Schnoebelen, 2002). However, model checking is unable to summarize or simplify the verified state. The programmer has to know every single state that he wants to verify. However, abstract interpretation is able to simplify the verified state by abstracting the verification state. Therefore, the verification process covers all possible states from the codes of the program.

Only abstract interpretation uses the abstract method. Both assertion and model checking method use concrete method. Using a concrete method, the programmers know which code or code behavior is verified. Therefore, the results from both methods are always precise. However, since assertion asserts many lines and model checking checks the code by using many temporal logics, the verification process is slower than abstract interpretation. Also, an abstract interpretation process is faster than both methods because its mathematical logic only covers the substance behavior of the program. However, its speed is at the expense of the accuracy because it tends to miss important codes in the program.

Assertion and abstract interpretation use one to two lines of code to verify the program. This is minimal compare to model checking where it needs more than 13 lines of code. Even though they are minimal, they use different technique to conduct verification; assertion uses concrete technique, which verifies every line of code. Abstract interpretation uses abstract technique, which verifies targeted line of code only. Therefore, the number of lines of code does not reflect the choice of the characteristics and human intervention during verification. However, abstract interpretation offers automation same as model checking, which can avoid human errors.

Abstract interpretation has all features needed to conduct static analysis in automated way. It can verify the program code without programmers' annotation and uses abstract method with formal mathematical logic. The abstract method makes the representation of the program's behavior generic, which can cover a variety of possible behaviors. Since it uses one line of code, the verification is fast.

2.7 Verification for Method Overriding

The process of verification on method overriding involves two main concepts. They are invariant and subtyping. Both use Unified Modeling Language (UML) for explanation. In Figure 2.10, a class *Person* has a data member declared as name. It has a subclass *Worker* that has a data member declared as *tSalary*; the total salary a worker earns. The subclass

Worker has subclass *Manager* where it has a data member *manages*. The superclass *Person* has a method *writeName*(), the class *Worker* has a method *writeSalary*() and the class *Manager* has a method *writeManager*(), where all of the methods display their data member accordingly.

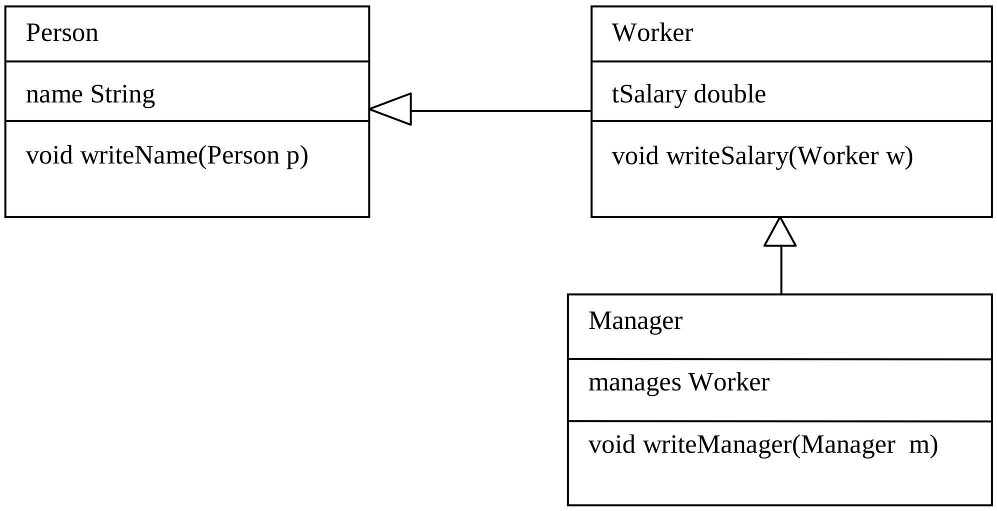


Figure 2.10: UML diagram of class Person, Worker, and Manager

2.7.1 Invariant

Invariant is a concept taken from Mathematics. It is described as the value of expression that does not change during program execution. In OOPL, there are four types of invariants, which called as class invariant, object invariant, type invariant, and loop invariant (Parkinson, 2007). No matter where the invariant is in the program, the purpose is only one; to become a property that is true for all expressions of a given code at all time. Therefore, for object-oriented programming, "class invariant is a property that is true for all objects of a given class at all times" (Webber, 2001, p.87) . The problem with class invariant related to reverification of existing classes is explained in detail in §3.3. An example of class invariant is typed in bold below, which means data *name* cannot be null value:

```
public class Person {
  private char* name; //invariant name != null;
  public Person() {
    name = new char ("Aliyah");    }
  public void writeName(Person* p) {
    cout<< p->name; }  }
```

2.7.2 Subtyping

Basic subtyping principle is substitutability, a situation when a datatype can be substituted by another datatype (that is supertype). Liskov et al. (1994) explained subtyping also known as Liskov Substitutability Principle, which is used to reason program's semantics as:

> What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T. (p.23)

Then, Liskov et al. (2001) formalized her statement using invariants and constraints by the statement:

> Subtype requirement: Let $\phi(x)$ be a property provable about objects x of type T, then $\phi(y)$ should be true for objects y of type S where S is a subtype of T. (p.1812)
>
> or
>
> symbolized as $S <: T$, which means S is subtype of T. (p.1823 & 1827)
>
> or
>
> visualized as

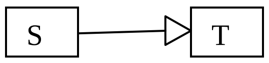


Figure 2.11: S is subtype of T

For example, when type *integer* is a subtype of type *double*; i.e. *integer<:double*, then number, say 10 that is declared as *integer* is received as *double* as well, just as in Java language. By considering contravariance and covariance, substitutability has better notion under behavioral subtyping (Castagna, 1995; Liskov and Wing, 2001). Liskov and Wing (2001) stated that methods must be contravariant and covariant because the methods determine how different types of data work or function. However, it becomes a problem during verification process which will be explained in §3.4.

Another example of subtype in inheritance program is the use of a pseudovariable; named *super*. When a method sends a message using a *super* method, the process starts from the immediate superclass possessing that method. However, if the method exists in the superclass itself, self-reference technique occurs. *Self* is another pseudovariable that realizes self reference in subtype of inheritance. It is defined in term of itself. It is used in recursive function, procedure, method or datatype. Figure 2.12 and Figure 2.13 illustrate the differences between *super* and *self* reference. It changes and modifies the state and behavior of the object at the later stage of execution. This is called late binding. Late binding allows the properties of objects to be reused and redefined without any textual copying or editing. It minimizes the process of code duplication (Cook, 1989).

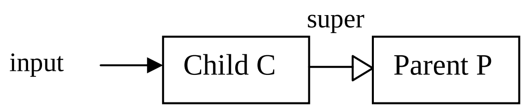


Figure 2.12: Super

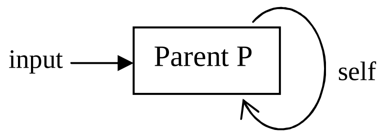


Figure 2.13: self reference

2.8 Features of Automated Verification on Method Overriding

Based on the analysis of static analysis methods (in Table 2.1) and concepts on verification in method overriding (in §2.7), the main component of verification is automation invariant generation. The production of the automated process equations must consider: (1) non-reverification of old classes and (2) modularity of invariant statements. The automated process means the process of verification needs neither programmer intervention nor annotation of specification. This is important to deal with human error during the process. However, the process is not easy because it always leads to over approximation of the invariant statements, which is explained in detail in the next chapter.

2.8.1 Non-reverification

Reverification happens when the verification process executes more than once on the same code. For example, in the beginning, a program has one class. The verification verifies the class to check its correctness. Then, a programmer modifies the program by extending the class with another new class. The new class and the old class have a relationship called inheritance. Therefore, the old class becomes a superclass and the new class becomes a subclass. When the programmer verifies the program, the verification process will verify the old and new classes. Thus, the old class has been verified twice. If, in the future, the programmer adds a new class to the subclass, then, during the verification, the old class has to be verified again. As the extension of the program becomes larger, the verification also takes a longer time. The non-reverification means only new subclass is verified after program extension. Therefore, verification process will be faster.

2.8.2 Modularity

A program analysis is considered modular when the analysis is decomposed into segments to be analyzed according to requirements. Modularity in analysis means the analysis is on program fragments or modules which contain only related variables (Banerjee, 1997). Modularity is also related to relationship between segments. There are segments that cannot be executed separately (dependency). In object-oriented program analysis, the program specification is decomposed into classes, methods, objects, components, or loops. This technique helps to understand and give better performance to the program analysis. It can also manage generalization structure of classes and avoid reverification because the verified class with invariant will not be verified again when new class is added. According to Meyer (1997), there are five criteria of modularity which are:

1. Decomposability

   A software construction method satisfied modular decomposability if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work proceed separately on each of them.

2. Composability

   A method satisfies modular composability if it favors the production of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed.

3. Understandability

   A method favors modular understandability if it helps produce software in which a human reader can understand each module without having to know the others, or at worst, by having to examine only a few of the others.

4. Continuity

   A method satisfies modular continuity if, a small change in a problem specification will trigger a change in just one module, or a small number of modules.

5. Protection

   A method satisfies modular protection if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighboring modules.

From the five criteria above, we only use three: decomposability, composability, and understandability. The reason is even though the invariants for a whole program, the invariants are generated according to class and method. The invariants are decomposed in such a way that the generated invariants are easier to manage and manipulate. In addition, the decomposability method avoids complexity during the verification process when generalization structure in inheritance involved. The process of decomposed and composed invariants helps the programmer to understand the program analysis because the analysis is small to trace and read. The program analysis does not consider continuity and protection because every generated invariant is independent. Therefore, any changes to the invariant will not affect other invariant even though they are in the same generalization structure.

2.9 Related Works

There are different methods and techniques applied and manipulated by other research works to generate invariants for programs with method overriding. They are used for verifying semantics of object-oriented programs using subtyping. Table 2.2 is the result of comparison between related works and techniques they used to ensure invariants are not reverified. Table 2.3 is a summary on related works and their techniques in verifying method overriding focusing on subtyping and generating invariants. All works concentrated on object-oriented programming languages, which have inheritance and dynamic binding. In order to find a good solution in verifying method overriding, all works are compared using five criteria; subtyping, invariant, non-reverification, modularity, and automated. Techniques for subtyping and invariant are analyzed to find each work’s strengths and weaknesses.

2.9.1 Analysis of Related Works Techniques on Non-Reverification

The performance of verification process on object-oriented programs depends on non-reverification feature in the equation of program analysis. If the verification process allows reverification of previous invariants, then more time is needed to conduct the process. The result shows that there are 50% of the chosen related works (from Table 2.3) apply non-reverification feature as shown in Table 2.2. The other 50% that does not have non-reverification feature because their works concentrated on specific element in OOPL, e.g., ownership, hybrid types, proof environment, memory location frames, and model fields. It means the non-reverification feature is not one of their main concerns.

All five of the non-reverification related works used behavioral subtyping to reason the semantics of method overriding. This can be done because behavioral subtyping makes sure preconditions of methods are weak and postconditions are strong. It is conducted by making sure superclasses that have verified will not reverified. If the verified superclasses are reverified, there is a possibility methods have strong preconditions due to the changes of the superclasses. Therefore, each technique used assumptions and enforcement to have non-reverification feature. Observable, specification inheritance, and modular technique enforce specifications are only analyzed on superclasses. That is the reason the specification superclass must be valid for specification subclass. Even though the techniques used same method, their names are different because observable implemented using abstract interpretation, specification inheritance using assertion, and modular using less mathematical equation approach.

Invariants produced by specification subsumption and extended abstract predicate family have same techniques on achieving non-reverification. The reason is both used same logic, which is separation logic. The static specification is only on superclass, which means the specification superclass is not allowed to be changed. However, Chin (2008) has different technique on generating invariants because he enabled to solve the problem of producing format to capture objects of classes without losing their information. Therefore, all techniques do not allow specification superclass be modified in order to conduct static checking. In addition, subclasses have to preserve specification superclass and only new methods are verified. So that, every time new subclass or method is added into a program, only the new one is verified.

Table 2.2: Comparison of Related Works and Criteria Related to Non-Reverification

			Non-Reverification		
Related Works	Subtyping	Invariant	Technique	Superclass	Subclass
Logozzo (Logozzo, 2004)	Behavioral subtyping	Observable behavior	Observable	Analyze on superclass only	Preserve superclass properties
Leavens (Leavens, 2006)	Behavioral subtyping	Modular specification	Specification inheritance	Specification for fields must valid for subclass	Extended Specification definition
Chin (Chin et al., 2008)	Behavioral subtyping	Specification subsumption	Checking inherited static specification	Static specification	Check method specification with new static specification for subclass
Parkinson (Parkinson et al., 2008)	Behavioral subtyping	Extended Abstract Predicate Family	Static and dynamic specification	Static specification	Check method specification with new static specification for subclass
Cheon (Cheon et al., 2012)	Behavioral subtyping	Cleanroom Software Engineering	Modular	Overridden method must behave like overriding method	Verify only new code

2.9.2 Analysis of Evaluated Related Works

Table 2.3 shows the result of features used by related works towards verification process on method overriding. All related works in the used the notion of behavioral subtyping except Dovland's (2008) to support modularity in the presence of subclassing and late binding. This is because it is the only way to reason semantic operation in inheritance. The rule of contravariance and covariance restricts the capability of the program to be reused (will explain in §3.4). Therefore, Dovland et al. (2008) proposed lazy behavioral subtyping that does not follow the behavioral subtyping. This affects the verification process which the reasoning semantics can easily verify polymorphism of OOPL. It involves inheritance, method overriding, and late binding call. His notion has been applied to multiple inheritance successfully. However, the verification process needs a programmer to annotate and it is not modular.

All authors used variety of ways to generate or annotate invariants onto object-oriented program. For example, Müller (2002), Leavens (2006), and Dovland et al. (2009) used Hoare logic. Even though they used the same logic, they approached the problem in the verification process differently. Muller (2002) used object ownership, Leavens (2006) used modular specification, and Dovland et al. (2009) used proof environment accordingly. Parkinson and Bierman (2008) extended the capability of Hoare logic by considering pointer during invariant generation which is called separation logic. By using separation logic, they formulated reasoning for inheritance and method overriding which they named abstract predicate family. Chin et al. (2008) applied the separation logic on inheritance and subtyping which he called specification subsumption. The specification subsumption focuses on distinction and relation between specifications to support behavioral subtyping in class invariant. Both superclass and subclass are not verified at the same time. Cheon et al. (2012) argued that Hoare logic reasons program as backward from postcondition to precondition which hard to learn and apply. Therefore, he adapted Cleanroom software engineering method to support forward reasoning in order to verify object-oriented program. All research works used Hoare logic as their technique of verification whether the logic is straightforward used or extended version. However, only Logozzo (2004) used abstract interpretation to conduct verification. This is done to automate the process of generating invariants which hard to achieve using Hoare logic. He implemented observable behavior technique using abstract interpretation for subtyping.

Based on non-reverification component, there are five works do not reverify old verified classes. Even though, they use behavioral subtyping of Liskov and Wing (1994) to solve problem of subtyping, they change the specification definition to suit with the problem they solve. Chin et al. (2008) specification subsumption has two conditions of contravariance and covariance that check same subclass where they are included in the behavioral subtyping. This method avoids re-verification of existing class by directly inserting the previous generated invariant into the new invariant of the new subclass. Leavens (2006) focused on methods that are similar by having definition on pre-behavioral subtyping, strong and weak behavioral subtyping. The weak behavioral subtyping definition allowed the less restrict constraints rule the methods, which allows object to be aliased and mutated due to method definition. However, unexpected behavior can happen due to less control of the methods. Parkinson et al. (2008) used separation logic to avoid reverification where it allows derived classes exist without reverifying the base class with the assumption methods do not modify the variables containing the arguments in the method body. The rule does not use method body because it works at the specifications level only. Logozzo (2004) used observable behavior which he defined classes that have relationship as a domain of observables which he refined when new classes added into the hierarchy.

Based on modularity component, all works are modular. Since the programming language is object-oriented that is based on modularity, it is easier to make the invariants modular as well. This allows behavioral subtyping being applied to reason the program semantics. Therefore, modularity is a required component in producing invariants for OOPL because the invariants produced are managed by class or method. In addition, modularity feature enables the invariants produced for scalable programs.

Only Logozzo (2004) verified object-oriented program automatically. This is due to the used of abstract interpretation theory and behavioral subtyping for verifying method overriding. His technique on method overriding verification is on superclass only because overridden method that exists in subclass is already in the superclass. He converted concrete classes and methods into abstract domains, which resulted in over approximation of the method semantics. Other works practice invariants based on programmer's annotation, which is prior to the code. If the programmer does not accidently assert errors, the result is accurate which close to the program behavior. However, the programmer's invariants are not generic and depended on the capability of the programmer's interpretation of the program. Even though, the automation process has been done since 2004, other researchers mentioned here did not extend the work because they did not use abstract interpretation as method of static analysis. In fact, the heavyweight of static analysis due to the use of abstract algebra by abstract interpretation makes researchers did not fully implement it but incorporate it with other methods. For example, Boogie tool used both assertion and abstract interpretation to verify object-oriented programs (Barnett et al., 2006).

Table 2.3: Comparison of Evaluated Related Works

Related Works	Techniques used to Verify Method Overriding				
	Subtyping	Invariant	Non-Reverification	Modularity	Automated
Muller (Müller, 2002)	Behavioral subtyping	Object Ownership	x	✓	x
Logozzo (Logozzo, 2004)	**Behavioral subtyping**	**Observable behavior**	**✓**	**✓**	**✓**
Flanagan (Flanagan et al., 2006)	Behavioral Subtyping	Hybrid types	x	✓	x
Leavens (Leavens, 2006)	Behavioral subtyping	Modular specification	✓	✓	x
Chin (Chin et al., 2008)	Behavioral subtyping	Specification subsumption	✓	✓	x
Parkinson (Parkinson et al., 2008)	Behavioral subtyping	Extended Abstract Predicate Family	✓	✓	x
Dovland (Dovland et al., 2009)	Lazy behavioral subtyping	Proof Environment	x	✓	x
Smans (Smans et al., 2010)	Behavioral Subtyping	Dynamic frames	x	✓	x
Balint (Balint et al., 2011)	Behavioral subtyping	Model fields	x	✓	x
Cheon (Cheon et al., 2012)	Behavioral subtyping	Cleanroom Software Engineering	✓	✓	x

Taken as a whole, there is only one work that has all three components: that of Logozzo’s. Other works such as Leavens’s (2006), Chin’s (2008), Parkinson’s (2008), and Cheon et al. (2012), have both components of non-reverification and modularity even though theirs were not automated. This is because they used Hoare style logic instead of abstract interpretation where the techniques need the programmers to learn the syntax of specification language in order to annotate the program. However, since the annotation is on the program behavior, it can ensure the program verification works as it intended to be. On the other hand, Logozzo’s (2004) equations on method overriding semantics produced over-approximation value because the overridden method's invocation is hard to trace before run time. The over-approximation value is a safe value for overriding and overridden method where the method semantics reasoned by behavioral subtyping. He argued that the over-approximation allowed the equations to cover all values from datatype of method parameters and arguments.

2.10 Conclusion

This chapter has provided a brief background information on method overriding at the beginning and related features to verify method overriding. It overviewed three techniques of static analysis: assertion, model checking, and abstract interpretation. An analysis on related works discussed on techniques used, strengths, and weaknesses in producing invariant generation. The finding shows technique used by Logozzo (2007) called abstract interpretation using observable behavior fulfilled all requirements needed to verify method overriding using invariant generation without human intervention. However, the technique produced problems related to class invariant and late binding call. Therefore, in the following chapter we will analyze the problems, which regards to automated linear invariant generation.

Chapter 3
Automated Linear Invariant Generation

'… and, Our Lord do not make us bear a burden
for which we have no strength...'
-Al-Baqarah verse 286

This chapter examines problems of developing invariants generation in both classes and methods for programs with method overriding. Its aim is to find a method and technique of solution for the problems. The problems are identified and discussed using program logic. There are two major components; class invariant and late binding call. The class invariant has a problem with reverification of verified invariants. The late binding call has a problem with restriction of semantics reasoning rule. Therefore, possible methods of solution for each problem is discussed in §3.3 and §3.4.

3.1 Automated Software Verification

Automated software verification becomes a grand challenge after Sir Charles Anthony Richard Hoare stated the importance of having automated program verifier as the main objective to achieve reliable softwares and systems (Hoare, 2007). The automated program verifier enables to ensure the absence of runtime errors, which avoids the unexpected result by the softwares. In the context of object-oriented program, works have been done to develop the automated program verifier (D'Silva et. al,2008) as well. For example, Astree

(Cousot, et.al, 2007), jStar (Distefano, 2008), and Polyspace (Little and Moler, 2013). However, researchers face challenges in generating invariants for dynamic allocated data structure, shared-variable concurrency, different code environments, and object mutation. All these problems are due to change of states in term of platform, data, environment and object. Therefore, until now, there is no such full automated software verification because of the difficulty to capture states of programs. As static analysis enables to help verification process to inspect programs during compile time, it is categorized into two methods; 1) type system and 2) formal verification. Type system requires programmers to annotate the program code with type information. Formal verification as explained in §2.4 can generate two types of invariants; 1) polynomial loop invariant and 2) linear invariant. Polynomial loop invariant is an inductive invariant for initial and consecutive location of the loop program (Rodriguez-Carbonell, 2007). Linear invariant is the invariant that is always true at the initial program and throughout the program execution. The current thesis limits to linear invariant generation.

3.2 Linear Invariants Generation

In order to have automated linear invariant generation, static analyzer must able to generate correct invariants for the whole code of the program. However, the program enables to scale using inheritance. Therefore, the invariants scale up as the program expands. This can affect the performance during static analysis process. Therefore, it is important to module the program to ease the process. In general, the program modules as class, method, and program structure. Researchers successfully analyze statically class and program structure (Logozzo, 2004 and Dovland, 2009). However, researchers face problems in generating linear invariant for method as it can be method overriding. The overriding method has the privilege to be invoked by any object as long as the object in the same inheritance hierarchy structure. The invocation allows program to be reused and changed states according to the program's requirements. Therefore, it is important to analyze statically the method overriding to achieve full automated software verification.

3.3 Methods for Problem Analysis

Initially, the problems related to automated invariant generation is represented using a small language called Method Language (methL) which is based on Featherweight Java (FJ) (Igarashi et al., 2001). The purpose of using a small language is to understand another language (Hoare, 1981). The syntax of methL is as below.

(*class definition*) $L ::= class\ C\ extends\ C\ \{\ \bar{F}\ \bar{M}\}$
(*field definition*) $F ::=\ \tau\ f$
(*method definition*) $M ::=\ \tau\ m\left(\overline{\tau\ f}\right) : (p,q)\{\bar{b}\}$
(*body statement*) $b ::= x \mid new\ C(\bar{e}) \mid e.f \mid e; e \mid e.m(\bar{e}) \mid m(\bar{e}) \mid C :: m(\bar{e})$
(*return type*) $\tau\ ::= C \mid void \mid int \mid bool$
$(expression)$ $e ::= F \mid x \mid null$

Here, *C* represents a class name where methL language is a language with inheritance. Overbar notation denotes there is a list; for example, $\bar{F}$ means a list of data members. A program consists of a list of class definitions. The definition of inheritance is an extension from one class to another. We do not consider multiple inheritance in this language. The body of declaration has data fields or member $\bar{F}$ and methods $\bar{M}$. Data field is declared using types τ and variables f. Type τ can be a class, integer, Boolean, or void. Methods in this language are methods that can change behavior of the class. We omit super method for simplicity. The methods precondition and postcondition are specified as (p, q). Every single method consists of

1. a variable x
2. a new object $new\ C(\bar{e})$
3. a data member call $e.f$
4. a sequential composition of expression $e; e$
5. three types of method calls, which are
    a. external method call, $e.m(\bar{e})$
    b. internal static call, $C :: m(\bar{e})$
    c. internal late bound call, $m(\bar{e})$

The external method call happens when an object calls the method as $e.m(\bar{e})$. The internal static call method $C :: m(\bar{e})$ happens in class C where the compiler compiles and binds the method at compile time. The internal late bound call only happens at run time when the actual object has been determined.

A visual representation of inheritance using UML diagram is shown in Figure 3.1 which is taken and extended from Figure 2.10. All classes are extended with additional data members and methods for explanatory purpose. In this figure, there is a class named *Person* that has two data type called *name* and *bSalary* as basic salary. It has a subclass named *Worker* that has a data type named *tSalary*. The subclass *Worker* has a subclass named *Manager* where it has a data type named *manages*. A method called *calc*() appears three times in the diagram. All classes; *Person*, *Worker*, and *Manager* have the method *calc*(), which *calc*() in class *Worker* and *Manager* can override *calc*() in class *Person*. There is another method *calc*() which is called from inside method *salary*() in class *Person*.

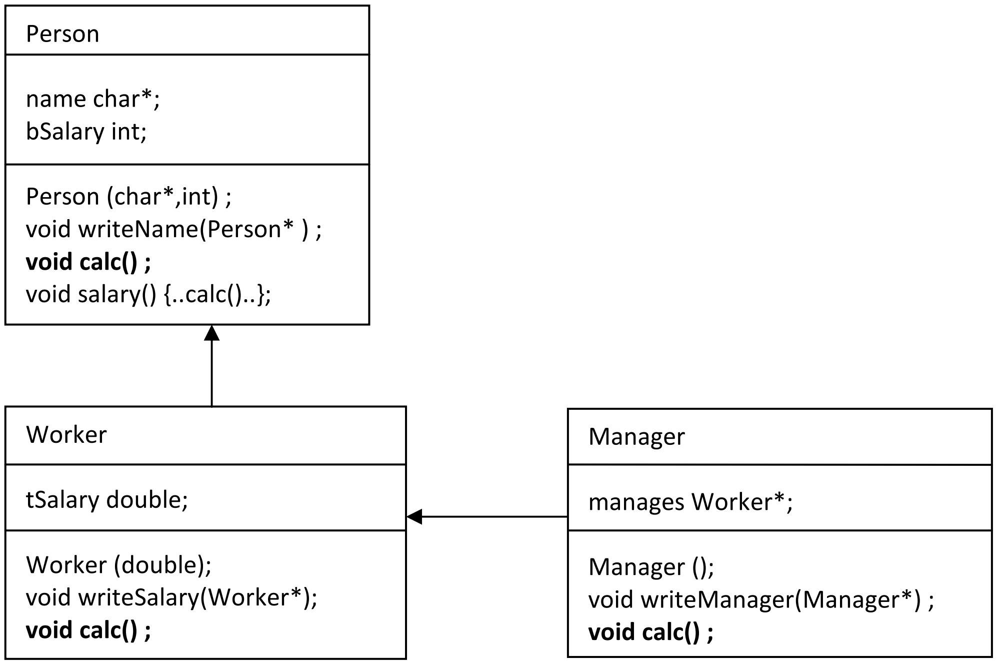


Figure 3.1: Example of inheritance with overridden method *calc()*

3.4 Problem 1: Class Invariant

In OOPL, generating class invariant is a difficult task especially in the presence of inheritance because class invariants are meant for single objects. However, in inheritance there is an generalization structure that involves two or more objects. There is one main problem arise during verification on object-oriented program. It is called reverification. It affects the performance of the compilation due to the repetitive nature of the verification process, which is exaggerated when the program scales up.

For example, let class Person of Figure 3.1 be coded and extended as in Figure 3.2. The class invariant restricts the data to a certain amount of limit to avoid memory overload especially for array. It also avoids mathematical operations on data member that has no value. The class invariant adds another class invariant for class *Worker*, a subclass to class *Person* because inheritance allows subclass to inherit data member from superclass, e.g. *sal* in method *calc*() in class *Worker*. In this situation, there is no error because the class invariant in superclass *Person* has specified subclass *Worker*. However, problems arise when class *Manager* is added later. If the class *Manager* is to be verified, the verification process has to start from the beginning. To avoid reverification of class invariant, there are five techniques, from Table 2.2 that are used in the program verification.

The idea of a class invariant that was first proposed by Hoare (1969) has been extended, so that the inheritance structure of classes and objects are easily verified. Parkinson et al. (2007) proposed the use of a more general foundation of verification which is Hoare logic to specify the properties of generalization structure, after considering the complexity of peer invariants of Leino and Muller (2004) and history invariants of Leino and Schulte (2007). Based on existing invariants generation techniques, Xing et al. (2010) present a technique where invariants are generated at each statement to ensure all properties are safe and terminated. Banarjee (2009) merges non-computer related technique, which is called clonal selection theory with a program verification process to predict program invariant shapes. However, all these techniques limit to programs with no method overriding.

```
public class Person {

public class Person {
  private String name = "Adam";
  private int bSalary = 100;
  public int testSalary = 200;

  public Person(String n, int s){
    name = n;
    bSalary = s;
  }
  public void writeName(Person p){
    System.out.print("The employer name is " + name);
  }
  public void calc(){
    bSalary = 2100;
    System.out.println("Person::calc()");
  }
  public void salary(){
    calc();
  } } //end of class Person

public class Worker extends Person{
  private int testSalary = 300;
  private double tSalary;

  public Worker(String nama, int gaji, double tot){
   super(nama,gaji);
   tSalary = tot;
  }
  public void writeSalary(Worker w){
   System.out.println(w.tSalary);
  }
  public void calc(){
   tSalary += bSalary;
   System.out.println( tSalary);
  } } //end of class Worker

public class Manager extends Worker{
  private String address = "Malaysia";
  Worker manages = new Worker("Aliyah",1000,0);

  public Manager(String nama, int gaji, double tot, String add){
    super(nama,gaji,tot);
    address = add;
  }
  public void writeManager(Manager m){
    System.out.print("\n" + m.name + " has a worker named " +
                     manages.name + " whose salary is RM");
    manages.calc();
  }
  public void calc(){
    tSalary = bSalary + 4000;
    System.out.print(name + "'s total salary is RM" + tSalary);
  }  } //end of class Manager
```

Figure 3.2: Salary System

3.4.1 Specification Subsumption and Extended Abstract Predicate Family

The emergence of separation logic by Parkinson (2005) and behavioral subtyping by Liskov and Wing (1994) produces a novel *specification subsumption* that avoids reverification during program analysis (Chin et al., 2008). Parkinson (2005) applied his separation logic to come out with a predicate for inheritance, which is called abstract predicate family. The specification of subclass comes with its superclass' specification to show the relationship between the classes. However, this technique produced reverification problem. Therefore, Parkinson has extended the abstract predicate family with static and dynamic specification (Parkinson et al., 2008). Chin et al. (2008) also use the same idea in verifying inheritance where static specification is used for new inherited methods and subclasses and dynamic specification is used for overriding method to ensure behavioral subtyping. They proposed a mechanism called specification subsumption that focuses on distinction and relation between specifications to support behavioral subtyping in class invariant. The word subsumption used by Chin et al. (2008), in the context of OOPL means "the ability to emulate an object by means of another object that has more refined methods" (Abadi and Cardelli, 1994, p.1). Both superclass and subclass are verified at the same time by considering their behavioral subtyping and method overriding. After the enhancement, the specification subsumption mechanism enables to ensure contravariance of precondition and covariance on postconditions using this inference rule:

$$\frac{preA \vdash preB * \Delta \qquad postB * \Delta \vdash postA}{\big((preB *\!\!\rightarrow\ postB) <:\ (preA\ *\!\!\rightarrow\ postA)\big)}$$

The inference rule consists of $preA$, $preB$, $postA$, $postB$, and Δ. $preA$ and $preB$ are precondition of A and B respectively. $postA$ and $postB$ are postcondition of A and B

respectively. Δ is the residual heap from the contravariance check on preconditions. It is used later for covariance check on postconditions. It is included in the inference rule because the rule enables to reason semantics of program with pointers. Annotation of $(preB \ast\!\!\rightarrow\ postB)$ is a subtype of $(preA\ \ast\!\!\rightarrow\ postA)$ if 1) the annotation's precondition has $preA$ that involved $preB$ and Δ, and 2) the annotation's postcondition has $postB \ast \Delta$ that involved $postA$. This means A is always has B and its residual heap because A is a supertype of B.

3.4.2 Observable Behavior

Observable behavior preserves behavior of objects especially superclass to be used later when new subclass is added. The observable behavior is a method on how objects react to messages based on its early specification (America, 1991). Logozzo (2004) used it in his framework of abstract interpretation to avoid reverification. However, the invariants are limited to superclasses and not to subclasses when it comes to method overriding. The idea of this technique is that since the precondition and postcondition of overridden and overriding methods are the same, the class invariant is generated only on superclass. Later, the additional invariants are added when new subclasses with new methods are created. Logozzo treated the technique as a domain that keeps old specification of precondition and postcondition. He did not manipulate the observable behavior domain to ensure the domain follows the rule of behavioral subtyping.

3.4.3 Cleanroom Software Engineering

The idea of Cleanroom software engineering was first published by Mills et al. (1987). The main objective of the method is to achieve the high quality in software with statistical quality control using mathematical verification. By using the concept, Cheon (2010) minimized the mathematical used during specification by proposing the use of concurrent assignment notation with intended function. The name of the method is *intended function* because the annotation is function-like way, which is similar to the way a function is typed in a program. He used the notion of behavioral subtyping to reason subtyping. Therefore, the technique only considered overridden method that existed in subclass every time a program with new subclass is verified. Therefore, the technique is modular. However, the annotation code has to be learned by the programmer even though the syntax follows function like syntax. Due to the syntax, the annotation code is long and there is a need to process the function mapping which affects the program's execution time during compilations. However, the annotation code assists programmers in minimizing of learning new specification language because the annotation uses Java's expression syntax.

3.4.4 Modular Specification

Modular specification applied by Leavens (2006) was an extension version of better JML. He applied behavioral subtyping in reasoning inheritance using the concept of refinement, which defined the binary relation on method specification. The purpose was to ensure modularity to avoid reverification. Even though the language of JML is straightforward, the programmer has to learn on how to apply the language in the situation of method overriding. Therefore, it opens the verification process to human error.

All five techniques for class invariants generation are taken from Table 2.2. They have one common feature, which avoids reverification. To achieve it, all the techniques must be in modular. The modularity allows the programmer to manipulate the class invariants; whether to reverify or not to. Every single technique has its own advantage that merged and manipulated to produce better verification. For example, modular specification uses a concept called supertype abstraction, which assumes all objects of subtypes can be treated uniformly. However, specification subsumption and separation logic reason an object-oriented program semantics using static and dynamic specification. This avoids loss of information because extra variables are used to capture important information. Therefore, every type is treated accordingly. Logozzo (2004) used the same concept, which his method he called observable.

3.5 Problem 2: Late Binding Call

Method overriding enables the subclass method to change the semantics of its superclass which affects the behavioral properties of objects. When a method is redefined, its behavior may change and may contradict its specification. The process of changing the object's behavior is only known at runtime. This is called late binding call. The late binding call happens when a method body is called during execution depending on the callee's actual class (Dovland et al., 2008).

For example, by referring to Figure 3.2, if method *calc*() from class *Worker* is selected for execution using an object that is an instance of class *Worker*, the method *calc*() from class *Worker* is executed and not from the class *Person*. However, if method *salary*() were selected by the instance of class *Person*, the late bound invocation of method *calc*() would be bound to the method *calc*() in class *Person*. Consider main method below as a demo method for Figure 3.2:

```
1  public static void main (String[] args) {
2    Person clerk = new Manager (“Adam”,2000,100,”Shah Alam”);
3    clerk.calc();           //output: 2100.00 - late binding
4      ((Manager)clerk).writeManager ((Manager)clerk);
5    System.out.print(”The test salary is : ”+clerk.testSalary);
     //output: 200 - early binding
6  }
```

The output:

```
Adam's total salary is RM6200.0
Adam has a worker named Aliyah whose salary is RM1000.0
The test salary is : 200
```

From the code above, it shows that the statement in line 3 produces 2100.00 from the method in class *Worker*. However, statement 5 produces 200, the value coming from *testSalary* of class *Person* not from class *Worker*. This is because object *clerk* belongs to class *Person* before runtime. Then, it knows the object *clerk* also belongs to class *Worker* when statement 2 is executed during runtime, which the statement is bound later after compilation. During the runtime, the method *calc*() of class Worker substitutes the definition of method *calc*() of class *Person* where the program uses the concept of subtyping. In order to verify the semantics of method *calc*() operation (method overriding), behavioral subtyping of Liskov Substitutability Principle is commonly used.

3.5.1 Behavioral Subtyping

There are three notions of behavioral subtyping: 1) object of subtype must be substitutable for its supertype, 2) precondition for a supertype entails the precondition for subtype, and 3) postcondition for subtype entails the postcondition for the supertype. For the purpose of program safety, behavioral subtyping is defined in two ways (Liskov and Wing, 1994). The first definition treats subtype relations as constraints. The constraints are annotated by using history rule. The history rule has a property that keeps constraints of methods, which are method's pre condition and post condition. The property is called history property. History properties cannot be changed, as they cannot be deduced. The deduction enables the programmer to monitor subclasses in invariant.

In the second definition, there is an extension map to define all new methods in the subtype. The extension map has extension rule that states each method has diamond rule to follow. The diamond rule is used to relate abstract value to method calling or executing the program. However, for method overriding where it has late bound method call, behavioral subtyping is less flexible due to its constraint rules (Mihancea and Marinescu, 2009).

To ensure a method type is specific, the method follows the rule of contravariance and covariance. The contravariance rule is when the method argument has a more general (wider) type. The covariance rule is when the method's return value has a more specific (narrower) type. The contravariant rule becomes a problem when verifying inheritance because the general type of method argument becomes very general as the declaration of the method argument can come from many superclasses.

Referring to Figure 3.2, considers extending the code with a higher-order function named *printing*() and a main function. This example uses C++ code instead of Java to show the example of subtyping using method call. Java does not allow method to be an argument. The *printing*() function has two arguments which are method of *Worker* and instance of type of *Worker*. The *Worker* is chosen because the class has a superclass and a subclass where we can see the access capability to the both classes. In the main function, there is an instance of *Worker* called *workerOfTheMonth*.

```
void printing( void (*action)(Worker*), Worker* worker) {
  (*action)(worker);
}
void main() {
  Worker* workerOfTheMonth;
  printing (writeSalary(), workerOfTheMonth);
  printing (writeName(), workerOfTheMonth);
}
```

From the code above, both *printing*() functions in the main function have no compile-time error because *writeSalary()* and *writeName()* are accessed under the declaration of class *Worker*. However, even though *Manager* is a subclass of *Worker*, *printing*() function is unable to call *writeManager*() because it prevents behavioral subtyping. Therefore, if we want the *printing*() function executes *writeManager*(), the function must be defined on an instance of *Manager*.

The late binding call problem can also be explained using reasoning system as the problem of calling correct methods is related to semantics. Figure 3.3 shows related rules to syntax methL language introduced in §3.3. For simplicity, the reasoning system tracks data type using Hoare logic of $\{p\}b\{q\}$, where p is precondition and q is postcondition to the statement b. The internal and the external late binding call use the same reasoning. Subtyping is represented using $\subseteq$. All of the subclasses denoted as $e_i\colon E$ are bound to each other in $bind(C_0^{cls}, m)$, where

C_0 , single class

cls, all superclasses belong to C_0

m, method

Therefore,

$$bind\left(C_0^{cls}, m\right) \overset{\text{def}}{=} if\ m \in M\ then\ C_0\ else\ bind\left(C^{cls}, m\right)\ end$$

For example, let method *calc*() be called from method *salary*() as illustrated in Figure 3.2. (p_1, q_1) is specification of superclass Person and (p_2, q_2) is specification of subclass *Worker*. If method *calc*() has specification in the form of $\{r\}calc(\quad)\{s\}$, then the inference rules in Figure 3.3 validates the $\{p_1 \wedge p_2\}\ calc(\quad)\{q_1 \vee q_2\}$ expression which is inferred from the rule of (*body*) and (*lateCall*). However, if a new subclass is added to the code, e.g. class *Manager*, the previous verification is not valid anymore, because the rule is changed to $\{p_1 \wedge p_2 \wedge p_3\}\ calc()\{q_1 \vee q_2 \vee q_3\}$ at call site.

$$(body)\ \frac{p \Rightarrow p_1 \quad \{p_1\}\ b\ \{q_1\} \quad q_1 \Rightarrow q}{\{p\}\ b\ \ \{q\}}$$

$$(var)\{p\}x\{p\}$$

$$(new)\{-\}new\ C(\bar{e})\{q\}$$

$$(sequential)\frac{\{p\}e\{c\} \quad \{c\}e\{q\}}{\{p\}e;\ e\{q\}}$$

$$(staticCall)\frac{\{p\}b_{m(\bar{x} \vee \bar{F})}^{bind(this:C,m)}\{q\}}{\{p[x,F]\}\ \ C :: m(\bar{e})\ \ \{q[\bar{b},\tau]\}}$$

$$(lateCall)\frac{\forall cls \in\ (\subseteq_1^n C) \qquad \{p_{cls} \wedge c{:}\,C\}\ \bar{b}_{c.m(\bar{x} \vee \bar{F})}^{bind(C_0^{cls},m)}\ \ \{q_{cls}\}}{\{\wedge_{cls}\ (p_{cls}[\bar{e}\,/\,\bar{x}\ \vee\ \bar{F}])\ \wedge\ \bar{e}\} \quad m(\bar{e}) \quad \{\vee_{cls}\ (q_{cls}[\bar{e},\tau\,])\}}$$

$$(externalCall)\frac{\forall cls \in\ (\subseteq_1^n C) \qquad \{p_{cls} \wedge c{:}\,C\}\ \bar{b}_{c.m(\bar{x} \vee \bar{F})}^{bind(C_0^{cls},m)}\ \ \{q_{cls}\}}{\{\wedge_{cls}\ (p_{cls}[\bar{e}\,/\,\bar{x}\ \vee\ \bar{F}])\ \wedge\ \bar{e}\} \quad c.m(\bar{e}) \quad \{\vee_{cls}\ (q_{cls}[\bar{e},\tau\,])\}}$$

Figure 3.3: Inference Rules

With the restriction of behavioral subtyping, several approaches, e.g. plug-in matching (America, 1991) and relaxed plug-in (Nunes, 2004), have been used to restrict the new definition of methods. However, the restriction on how the object behaves beat the purpose of having object-orientation methodology in software development, which is code-reuse. Therefore, Dovland (2008) used a lazy method to reason subtyping using open closed principle of object-oriented design where programs are open to be reused without programmers do not have to worry the specification that changes due to program modification. It is called lazy behavioral subtyping.

3.5.2 Lazy Behavioral Subtyping

Dovland et al. (2008) produced a novel lazy behavioral subtyping (LBS) method that considers superclasses and their subclasses when analyzing methods in object-oriented programs. LBS uses open world assumption concept, which the classes are extended and reused over time. The classes are incremental reasoned on the class hierarchies using LBS. The open world assumption allows the program not only being gradually expanded but also leads to potential bindings using method overriding. The LBS reasons the program's method definition and method call by assigning specifications for the purpose of static analysis. The specifications (better known as assertion in LBS) are defined for the methods using an assertion entailment. The method definition's assertion is represented as p and q and the method call is represented as r and s. The (p,q) and (r,s) are employed based on Hoare logic of precondition and postcondition. Entailment for both assertions is defined by (a formal presentation is in §4.8)

1. A method definition assertion entails a method call assertion, $(p, q) \rightarrow (r, s)$.

The assertion of method call consists of assertion of method definition and the assertion themselves.

2. The sets of method definition assertion U (denotes $\{(p_i, q_i)|1 \leq i \leq n\}$) entail a method call assertion, $U \rightarrow (r, s)$.

The assertion of method call consists of the sets of method definition assertion, which the sets come from more than one method definition of different superclasses.

3. The sets of method definition assertion entail the sets of method call assertion V (denotes $\{(r_i, s_i)|1 \leq i \leq n\}$), $U \rightarrow V$.

The sets of method call assertion consist of the sets of method definition assertion when there is more than one assertion of method definition for many method call assertion called from different subclasses.

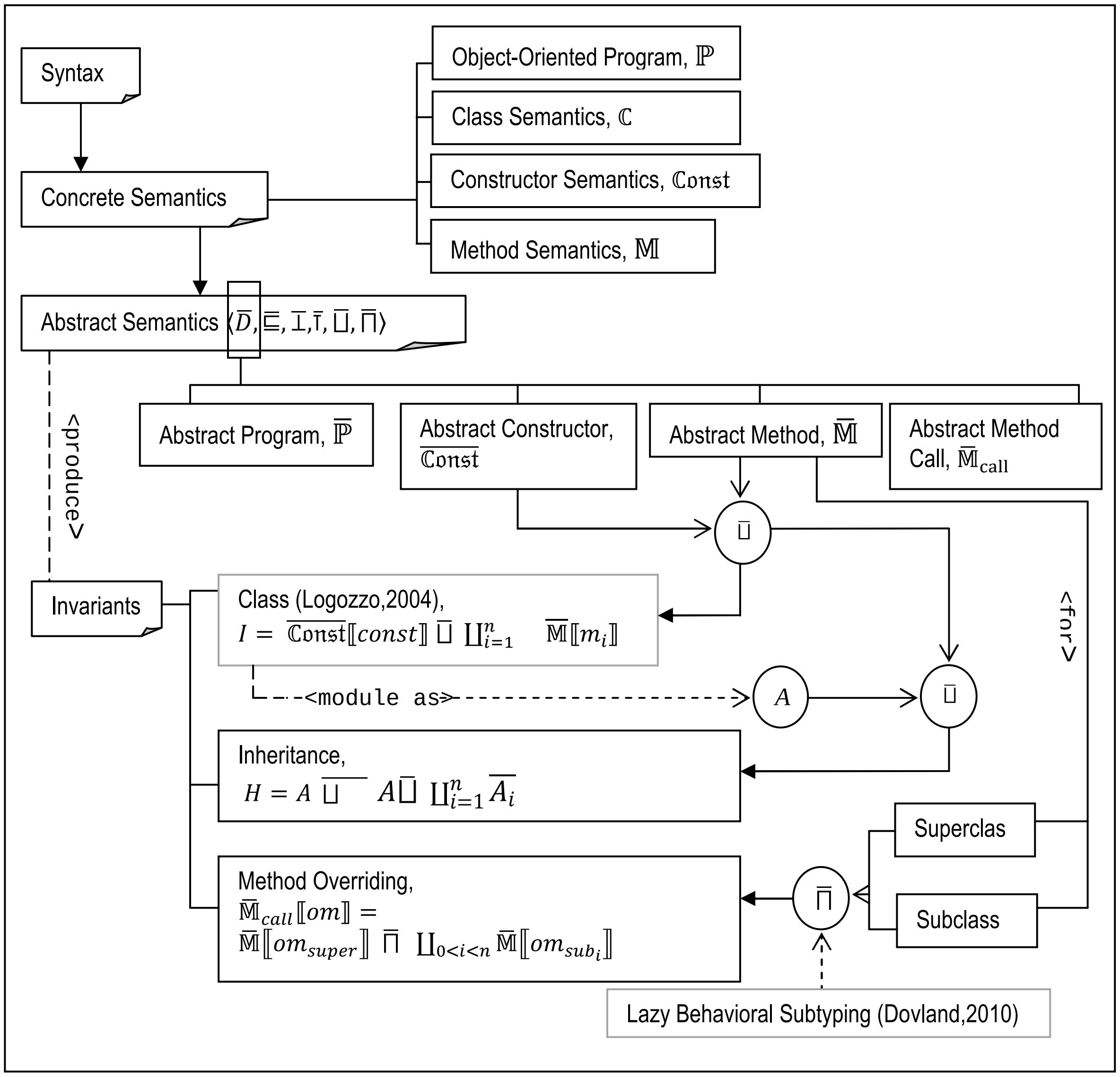
The definitions above show that LBS focus on the method specification which allows method overriding be reasoned statically. All method definitions have specification as well as method calls. The method call that is called within method definition holds specification from itself and also all specifications for its method definitions. Therefore, whenever the method is called, the analysis uses all specifications already specified by LBS, which covers all possibilities statically. In addition, for the method overriding, its method definition only uses specification that has been defined without considering specification from its superclass. This rule contradicts from behavioral subtyping that whenever a method is redefined in subclass, its new method definition must satisfy superclass specifications. As of 2012, there are only Dovland and his collegue’s papers prove the use of LBS (Dovland et.al, 2010). They have applied the concept for distributed concurrent objects successfully (Johnsen and Owe, 2007) without formally published the LBS at the time and multiple inheritance (Dovland et.al, 2009). However, the work limits to manual specification and Hoare style logic programming. As a consequence, the implementation needs programmers’ intervention for the verification process. Therefore, with the strength of LBS over behavioral subtyping, this thesis adopts LBS using abstract interpretation to design an abstract formal framework to achieve automation program verification focusing on method overriding.

3.6 Proposed Abstract Formal Framework

Figure 3.4 shows the illustration of proposed abstract formal framework, which the detail framework is in chapter 4. The framework based on abstract interpretation uses Java language as a basis for the program syntax. The syntax helps in explaining the program semantics that focuses on the use of data field in the presence of method overriding. Therefore, the syntax consists of a class, main class, and library. For the concrete semantics, the framework bases on object-oriented program semantics, class semantics, constructor semantics, and method semantics. These concrete semantics have domains to define each concrete semantics using Fixpoint Tarski’s theorem. The theorem traces the changes of states for each concrete semantics, so that the traces can be abstracted using abstract interpretation theory. The domains are input and output values, environment, store and state.

The conversion of the concrete semantics to abstract semantics is by abstract interpretation. The abstract interpretation uses Galois connection to ensure the abstraction and concretization of the program semantics in monotonic function. The structure of the program semantics orders in partial order set, so that the semantics always in lattice form. The abstract semantics has four domains which are abstract program, abstract constructor, abstract method, and abstract method call. Abstract constructor and abstract method are merged using union to produce class invariant. The creation of the class invariant is adopted from Logozzo (2004). Then, the class invariant is composed in an invariant namely *A* to ensure the invariant is not reverified when new subclasses added. The composition of each invariant produced is kept in modular. Therefore, the technique solved the problem of reverification by class invariant. Then, the class invariant is used for invariant in inheritance. The *H* represents invariant in inheritance. It is a convergence of the *A*, which is the old invariant that has been verified and new invariants created for new subclasses.

The abstract method is used to create abstract method call in the presence of method overriding. The abstract method call generates from abstract method of overriding and overridden method from superclass and subclass. The abstraction produces invariants from the overriding and overridden method, which the invariants is used when the method call is invoked. This technique is adapted from the notion of lazy behavioral subtyping. By using lazy behavioral subtyping, invariants can be generated for method call compare to behavioral subtyping that does not analyzed method call. Even though lazy behavioral subtyping allows analysis on method call, the invariants used specify by the programmer. Therefore, this framework lets the invariants be generated by abstract interpretation.



Syntax
Object-Oriented Program, $\mathbb{P}$
Class Semantics, $\mathbb{C}$
Constructor Semantics, $\mathfrak{Const}$
Concrete Semantics
Method Semantics, $\mathbb{M}$
Abstract Semantics $\langle \bar{D}, \bar{\sqsubseteq}, \bar{\bot}, \bar{\top}, \bar{\sqcup}, \bar{\sqcap} \rangle$
Abstract Program, $\bar{\mathbb{P}}$
Abstract Constructor, $\overline{\mathfrak{Const}}$
Abstract Method, $\bar{\mathbb{M}}$
Abstract Method Call, $\bar{\mathbb{M}}_{\text{call}}$
<produce>
Invariants
Class (Logozzo,2004), $I = \overline{\mathfrak{Const}}[\![const]\!] \bar{\sqcup} \coprod_{i=1}^{n} \bar{\mathbb{M}}[\![m_i]\!]$
<module as>
A
<for>
Inheritance, $H = A\ \overline{\sqcup}\ A\bar{\sqcup} \coprod_{i=1}^{n} \overline{A_i}$
Superclas
Subclass
Method Overriding, $\bar{\mathbb{M}}_{call}[\![om]\!] = \bar{\mathbb{M}}[\![om_{super}]\!]\ \bar{\sqcap}\ \coprod_{0<i<n} \bar{\mathbb{M}}[\![om_{sub_i}]\!]$
Lazy Behavioral Subtyping (Dovland,2010)

Figure 3.4: Proposed Abstract Formal Framework

3.7 Conclusion

In this chapter, we have examined problems of automated invariant generation using a language called methL based on program logic. There are two main problems, which are reverification of class invariant and over-approximation value of invariants in late binding method call. There are five solutions for reverification of class invariant problem proposed by other works, which are specification subsumption, extended abstract family, observable behavior, cleanroom software engineering, and modular specification. For the problem of over-approximation value of invariants in late binding method call, there are two solutions, namely behavioral subtyping and lazy behavioral subtyping. To achieve the objective of automated invariants generation that is modular and non reverification, abstract interpretation (taken partly from observable behavior technique), and lazy behavioral subtyping method are chosen as the solution. With this solution, the development of abstract formal framework for invariants generation is described in chapter 4 using abstract interpretation.

Chapter 4
Formalization of Invariants in Method Overriding

There is no abstract art.
You must always start with something.
Afterward, you can remove all traces of reality.
-Pablo Picasso

This chapter designs an abstract formal framework for verification on method overriding. Its aim is to produce proposed equations to produce invariants for inheritance and method overriding with late binding call. The framework consists of the formalization of equations for invariants developed using abstract interpretation theory and lazy behavioral subtyping, which any research has been done before. The framework adopts class invariant of Logozzo (2004), Fixpoint Tarski's theorem (1955), Fages lemma (2008), and Galois connection to develop the framework using abstract interpretation.

4.1 Preliminary Notation

To facilitate our discussion, we introduce mathematical concepts and notations for lattice theory, fixed point theory, and abstract interpretation theory required for the study.

4.1.1 Sets

We denote sets with capital letters and element of sets with small letter, italic cambria math font. For example, e is a member of the set E, written as $e \in E$. We use bar to represent the abstraction of the element or set of element. For example, the abstract domain D written as $\overline{D}$. We also denote the set of natural numbers as $\mathbb{N}$, the set of integer numbers as $\mathbb{Z}$, the set of Boolean values as $\mathbb{B}$, the set of String value as $\mathbb{S}$, and let $[x..y]$ be the set of $\{i \in \mathbb{Z} |\ i \geq x\ \wedge i \leq y\}$.

Given two sets A and B, their Cartesian product is denotes by $A \times B$ where $A \times B = \{(a, b): a \in A \wedge b \in B\}$. A relation r between A and B is a subset of their Cartesian product, i.e. $r \subseteq A \times B$, and a relation r on A is $r \subseteq A \times A$.

4.1.2 Partially Ordered Sets

A partial ordering of a set S is given by a relation $\leq$ such that it is

1. Reflexive: $a \leq a$ for all $a \in S$
2. Antisymmetric: if $a \leq b$ and $b \leq a$,then $a = b$
3. Transitive: if $a \leq b$ and $b \leq c$, then $a \leq c$

We denote the partially ordered set (poset) as $\langle D, \sqsubseteq \rangle$ instead of $\langle D, \leq \rangle$. The top element of $\langle D, \sqsubseteq \rangle$ is $\top$ iff $\top \in D\ \wedge\ \forall d \in D. d \sqsubseteq \top$. The bottom element of $\langle D, \sqsubseteq \rangle$ is $\bot$ iff $\bot \in D\ \wedge\ \forall d \in D\ . \bot \sqsubseteq d$.

We say that $d \in D$ is the least upper bound of A denoted by $\sqcup A$, if $\forall d' \in D$ such that $d \sqsubseteq d'$. Symmetrically, we denote the greatest lower bound of A by $\sqcap A$.

A poset $\langle D, \sqsubseteq \rangle$ is called a lattice if any two elements of D have both a greatest lower bound and a least upper bound. For a complete lattice, we write $\langle D, \sqsubseteq, \bot, \top, \sqcap, \sqcup \rangle$. The poset $\langle D, \sqsubseteq \rangle$ satisfies the ascending chain condition (ACC), if every ascending chain $d_1 \sqsubseteq d_2 \sqsubseteq ..$ of elements in D is eventually stationary, i.e., $\exists i \in \mathbb{N} \,.\, \forall m > n \,.\, d_m = d_n$.

4.1.3 Functions

A function is a relation r such that if $(a, b) \in r$ and $(a, b') \in r$, then $b = b'$. We specify functions using λ notation, e.g. $\lambda x. Expr$. It defines a function with an input x and an output produces by expression, $Expr$. Let f be a function, a an element in its domain and b an element in its co domain. Therefore, $f[a \mapsto b]$ is a function that accepts a as input and returns b as output. We denote $f[A \longrightarrow B]$ as the domain of the function f is included in A, and its co domain is included in B. Let $f[A \longrightarrow B]$ and $g[X \longrightarrow Y]$, then $f \circ g \in [A \longrightarrow Y]$, which represents the composition of function f and g, i.e., $\lambda x. g\big(f(x)\big)$. Let two posets be $\langle A, \sqsubseteq_a \rangle$ and $\langle B, \sqsubseteq \rangle$, a function $f[A \longrightarrow B]$ is

1. Monotonic: $\forall a_1, a_2 \in A \,.\, a_1 \sqsubseteq_a a_2 \iff f(a_1) \sqsubseteq_a f(a_2)$
2. Join-morphism: $\forall a_1, a_2 \in A \,.\, f(a_1 \sqcup a_2) \iff f(a_1) \sqcup_a f(a_2)$

4.1.4 Fixed points

Let function f be $f \in [D \rightarrow D]$. A fixpoint of f is an element $d \in D$ such that $f(d) \in d$. If we define f over partial order set $\langle D, \sqsubseteq \rangle$, then the element d is the least fixpoint with $d = f(d)$ and all next element of d , d' is $\forall d' \in D \,.\, d' = f(d') \Rightarrow d \sqsubseteq d'$.

Let f be a function defined over partial order set $\langle D, \sqsubseteq \rangle$, an element $d \in D$, and the order $\sqsubseteq$ is larger than d. Least fixpoint of f, denoted as $lfp_d^{\sqsubseteq} f$. The least and greatest fixpoints on a monotonic function is guaranteed by the Tarski's fixpoint theorem.

Theorem 4.1 (Fixpoint Tarski's Theorem (Tarski, 1955))

Let $\langle D, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ be a complete lattice and let $f \in [D \rightarrow D]$ be a monotonic function. Then, the set of fixpoints is a non empty complete lattice, and:

$$lfp_\bot^{\sqsubseteq} f = \sqcap \{d \in D | f(d) \sqsubseteq d\}$$

$$gfp_\bot^{\sqsubseteq} f = \sqcup \{d \in D | f(d) \sqsupseteq d\}$$

4.1.5 Traces

Let Σ be a set of states with an internal state as σ, a non state is $\varepsilon \notin \Sigma$, and a trace is a function $\tau \in [\mathbb{N} \rightarrow \Sigma \cup \{\varepsilon\}]$ with prefix condition $\forall n \in \mathbb{N} . \tau(n) = \varepsilon \Rightarrow \forall i > n . \tau(i) = \varepsilon$. We denote the sets of traces over Σ with $\mathcal{T}(\Sigma)$.

The length of a trace, $len\mathcal{T}$ is $len\mathcal{T} \in [\mathcal{T}(\Sigma) \rightarrow \mathbb{N}]$ where $len\mathcal{T} = \lambda\tau. \min(n) \in \mathbb{N} \mid \tau(n) = \varepsilon \Rightarrow \forall i > n . \tau(i) = \varepsilon$.

DEFINITION 1(Fixpoint partial traces semantics (Cousot et al., 1979))

Let

Σ be a set of states (σ),

$\Sigma_0 \subseteq \Sigma$ be a set of initial states,

$\rightarrow_P \subseteq \Sigma \times \Sigma$ be the transition relation from one state to another state, and

function F be

$$F \in \left[\mathcal{P}(\Sigma) \rightarrow \mathcal{P}\big(\mathcal{T}(\Sigma)\big) \rightarrow \mathcal{P}\big(\mathcal{T}(\Sigma)\big)\right] \tag{1}$$

Consequently, with X is a variable, equation (1) can be defined with fixpoint as :

$$F(\Sigma_0) = \lambda \mathrm{X}. \Sigma_0 \cup \{\sigma_0 \rightarrow \cdots \sigma_\mathrm{n} \rightarrow \sigma_{\mathrm{n}+1} | \, \sigma_0 \rightarrow \cdots \sigma_\mathrm{n} \in \mathrm{X} \wedge \sigma_\mathrm{n} \rightarrow_P \sigma_{\mathrm{n}+1}\}$$

Then the fixpoint partial trace semantics of program P is

$$\mathcal{T}[\![P]\!](\Sigma_0) = lfp_{\bot}^{\sqsubseteq}(\Sigma_0) = \bigcup_{i \leq w} F^i(\Sigma_0)$$

4.1.6 Abstract Interpretation

Abstract interpretation formalizes the approximation between the program concrete semantics and abstract semantics. We use the formalization to conduct static analysis for the purpose of verifying program with method overriding. This theory has been applied not only to verify a language but also to bytecode (Barbuti et al., 2010), networking (Borghuis et al., 2000), and code safety (Albert et al., 2005). Here, the concrete semantics is a concrete semantic domain, D which is partially ordered set $\langle D, \sqsubseteq \rangle$. The abstract semantic domain is represented as $\langle \overline{D}, \overline{\sqsubseteq} \rangle$. The concept of abstract interpretation is to define the program semantics as the fixed points of a monotonic function.

DEFINITION 2 (Galois Connections) (Jaoua and Elloumi, 2002)
For two partial orders $\langle D, \sqsubseteq \rangle$ and $\langle \overline{D}, \overline{\sqsubseteq} \rangle$, the abstraction $\alpha \in [D \rightarrow \overline{D}]$ and concretization $\gamma \in [\overline{D} \rightarrow D]$ be Galois connections iff

$$\forall d \in D \,.\, \forall \bar{d} \in \overline{D} \,.\, \alpha(d) \,\overline{\sqsubseteq}\, \bar{d} \iff d \sqsubseteq \gamma(\bar{d}) \tag{2}$$

Equation (2) is written as

$$\langle D, \sqsubseteq \rangle \;\; {}_{\gamma}\!\rightleftarrows^{\alpha} \;\; \langle \bar{D}, \bar{\sqsubseteq} \rangle \tag{3}$$

Equation (3) has the following properties:

1. α and γ are monotonic functions
2. $\alpha \circ \gamma$ is reductive where $\forall \bar{d} \in \bar{D}. \alpha \circ \gamma(\bar{d}) \bar{\sqsubseteq} \bar{d}$
3. $\gamma \circ \alpha$ is extensive where $\forall d \in D.\, d \sqsubseteq \gamma \circ \alpha(d)$

DEFINITION 3 (Lattice of Abstract Interpretations) (Tarski,1955)

Let

$\langle D, \sqsubseteq \rangle$ be a complete lattice,

abstract interpretation of the domain $AI(D)$ be $\left\{\langle \bar{D}, \bar{\sqsubseteq} \rangle \mid \exists \langle \alpha, \gamma \rangle. \langle D, \sqsubseteq \rangle \;\; {}_{\gamma}\!\rightleftarrows^{\alpha} \;\; \langle \bar{D}, \bar{\sqsubseteq} \rangle\right\}$,

order $\sqsubseteq_{AI}$ on $AI(D)$ be

$$\sqsubseteq_{AI} = \left\{\langle \langle \bar{D}_1, \bar{\sqsubseteq}_1 \rangle, \langle \bar{D}_1, \bar{\sqsubseteq}_2 \rangle \rangle \mid \exists \langle \alpha, \gamma \rangle. \langle \bar{D}_1, \bar{\sqsubseteq}_1 \rangle \;\; {}_{\gamma}\!\rightleftarrows^{\alpha} \;\; \langle \bar{D}_1, \bar{\sqsubseteq}_2 \rangle\right\}$$

Then, $\langle AI(D), \sqsubseteq_{AI} \rangle$ is a complete lattice.

For the powerset of concrete and abstract domain, they are connected using Galois connection to formalize the abstraction of the domains. This is to ensure two domains are corresponded to each other:

Lemma 1 (Fages,2008):

Let concrete domain be $c \in \mathfrak{C}$, where $\mathfrak{C}$ is a set of concrete domains, and abstract domain be $a \in \mathfrak{A}$, where $\mathfrak{A}$ is a set of abstract domains, such that $\alpha \in [\mathcal{P}(\Sigma) \rightarrow \bar{D}]$ where $\alpha\left(\mathcal{P}(\Sigma)\right) = \bigcup \gamma^{-1}\left(\mathcal{P}(\Sigma)\right)$ and $\gamma(\bar{D}) = \bigcup \gamma(\{d\})$. Then, $\mathcal{P}(\mathfrak{C}) \;\; {}_{\gamma}\!\rightleftarrows^{\alpha} \;\; \mathcal{P}(\mathfrak{A})$.

Proof:

Let a be $a = \bigcup \gamma^{-1}\left(\mathcal{P}(\Sigma)\right)$ and c is $\bigcup \gamma(\{d\})$. Then,

$$a = \alpha\left(\mathcal{P}(\Sigma)\right) = \bigcup \gamma^{-1}\left(\mathcal{P}(\Sigma)\right) = \bigcup_{d \in c} \gamma^{-1}(\{d\}) = \bigcup_{d \in \gamma(\{d\})} \gamma^{-1}(\{d\})$$

For each d, where $d \in \{D \mid D = \mathcal{P}(\Sigma)\}$ and $\mathcal{P}(\Sigma) \subseteq \overline{D}$ such that

$\alpha(\{d\}) \subseteq a$

$\Rightarrow \alpha\left(\mathcal{P}(\Sigma)\right) \subseteq a$

$\Rightarrow \gamma^{-1}\left(\mathcal{P}(\Sigma)\right) \subseteq a$

$\Rightarrow \gamma^{-1}(\{d\}) \subseteq a \quad \text{q.e.d}$

4.2 Syntax

Object-oriented program is a program that uses class and object as the paradigm for program development. A simple implementation has one main class and a library. In Java, the program consists of a class, a main class and a library. In C++, an object-oriented program has one class, a main method, and a library as in C#.net. Therefore, for simplicity, the thesis takes a program as consisting of one main library and one main class. With class C, an object-oriented program P consists of two elements $\{C, C_{main}, L\}$ where C is the class, $C_{main} \in C$ is the main class and $L \subseteq C$ is the library used in program. However, in the current thesis C_{main} is also C because class for main uses the same syntax as other classes.

"A class is a software element describing an abstract data type and its partial or total implementation" (Meyer, 1997, p.23). By considering both abstract data type and implementation, the class consists of data members or fields, a constructor, and methods.

Destructors are not included because there is a garbage collector that can manage the unused data. Class C is a tuple $\langle f, const, m\rangle$, where *f* is field declaration, *const* is the class constructor, and *m* is method. Program *P* produces states; $\sum$, which is <*E*, *S*>, where *E* is environment and *S* is store. State; $\sum$ consists of many internal states; σ, that come from objects in the program ($\sigma \in \sum$). An environment; *E*, is a map from variables; *Var*, to memory addresses; A as $E \stackrel{\text{def}}{=} Var \mapsto A$. A store; *S*, is a map of from addresses; *A*, to values; *Val* as $S \stackrel{\text{def}}{=} A \mapsto Val$, where values can be integer, Boolean, and null; $Val = \{int, bool, null\}$.

4.3 Semantic Domains

A semantic domain is a domain used to describe the meaning of the concrete semantics It describes the semantics of class, constructor, and method. All set of these semantic domains are represented by the powerset $\mathcal{P}(\quad)$. For example, $\mathcal{P}(\sum)$ means all sets of the set of state, $\mathcal{P}(\mathcal{P}(\sigma))$. These are elements involved in defining semantic domains:

1. Input value, D_{in}; output value, D_{out}

   The set of input value, D_{in} and output value, D_{out} are integer, boolean, or null.

2. Environment, E

   A set of environments $\mathcal{P}(E)$ is a map denoted as $[Var \mapsto A]$; variable Var is a string, $Var \in \mathbb{S}$; and the address, A is a natural number, $A \in \mathbb{N}$.

3. Store, S

   The set of stores $\mathcal{P}(S)$ are a map of $[A \to Val]$, where the value is $Val = \{int, bool, null\}$.

4. State, $\sum$

   The set of states $\mathcal{P}(\sum)$ are products of environment and store, $E \times S$.

4.4 Concrete Semantics

The goal of static analysis is to provide an effective computable approximation of the concrete semantics (Cousot et al., 1977). This is achieved by first defining the properties involved in the program execution through concrete semantics. Then, the concrete semantics is converted to abstract domain semantics using lattice theory. Every change of the semantic domains are traced using semantics traces where the semantics domains are represented using fixpoint theorem (Tarski, 1955) and the relation between the domains are represented using Galois connection (Jaoua et al., 2002).

DEFINITION 4 (Object-Oriented Program Semantics, $\mathbb{P}[\![-]\!]$)

Let

$Person$ be a class name,

main is the main method in class $Person$,

L is the library used in the class,

OO be $\langle Person_{main}, L\rangle$, and

$\longrightarrow \subseteq (\sum) \times (\sum)$ be a trace from one state to another state.

The semantics of object-oriented program is

$$\mathbb{P}[\![OO]\!] \in [\mathcal{P}(\sum) \longrightarrow \mathcal{P}(\mathcal{T}(\sum))] \tag{4}$$

Consequently, equation (4) when defined as fixpoint is,

$\mathbb{P}[\![OO]\!](\sigma_{init}) = lfp^{\subseteq}_{null} \curlywedge Y.\sigma_{init} \cup$

$$\{\sigma_0 \to \cdots \sigma_n \to \sigma_{n+1} | (\sigma_0 \to \cdots \sigma_n \in Y) \wedge (\sigma_{n+1} \in \sum) \wedge (\sigma_n \longrightarrow (\sum))\}$$

where, a set of program initial states is $\sigma_{init} \in \mathcal{P}(\sum)$, such that all initial method (m) states in a program with Val_{in} be input value is $\forall\, \sigma_0 \in \sigma_{init}\,.\ \sigma_0(currentMethod) = m \wedge \sigma_0(Val) = Val_{in}$.

DEFINITION 5(Class Semantics, $\mathbb{C}[\![-]\!]$)

Let

$constr$ be constructor,

m be method,

v be value,

s be store,

$\mathcal{T}$ be the trace of the states,

f be data field with element $\langle D_{in}, D_{out}\rangle$, and

class *Person* be $Person =< f, constr, m >$.

Therefore, the class semantics for *Person* is

$$\mathbb{C}[\![Person]\!] \in \mathcal{P}\big(\mathcal{T}(\Sigma)\big) \tag{5}$$

Definition of (5) using fixpoint is

$\mathbb{C}[\![Person]\!] = lfp^{\subseteq}_{null}\, \lambda\mathcal{T}.Semantic_0\langle v, s\rangle$

$$\cup \left\{\sigma_0 \xrightarrow{<m,v>_0} \sigma_1 \to \cdots \xrightarrow{<m,v>_{n-1}} \sigma_n \mid \left(\sigma_0 \xrightarrow{<m,v>_0} \sigma_1 \to \cdots \xrightarrow{<m,v>_{n-1}} \sigma_n \in \mathcal{T}\right)\right\}$$

To illustrate next definitions, an example of each definition is given using a program sample as shown in Figure 4.1.

```
class Worker {
double salary;
Worker (double sal) {
   salary = sal;  }
void writeSalary(Worker w) {
   System.out.print(w.salary); }
void calc(double sal) {
   System.out.print(“weekly wage
is:”+salary=sal/4); }
```

Figure 4.1: Class Worker

DEFINITION 6 (Constructor Semantics, $\mathbb{C}\mathfrak{onst}[\![-]\!]$)

Let

$D_{in} \subseteq Val$ be a semantic domain for the input values,

$e_0 \subseteq E$ be the initial environment,

a_{in} *and* a_{lc} be the memory address for the constructor’s input (*in*) data fields and the location (*lc*),

val be the value, and

inputs, pc_{const} be the constructor’s entry point.

Then the constructor semantics is

$$\mathbb{C}\mathfrak{onst}[\![const]\!] \in [D_{in} \times S \rightarrow \mathcal{P}(\sum)]$$

with

$\mathbb{C}\mathfrak{onst}[\![const]\!] = \lambda\ (v_{in}, s).\, let\ \sigma_0$

$$= \langle e_0[v \mapsto a_{in}, lc \mapsto a_{\,lc}], s[a_{in} \mapsto val_{in}, a_{lc} \mapsto lc_{const}]\rangle$$

Example 4.1 (Constructor Semantics)

The class *Worker* has one data member, which is salary that is the input value for the class. The constructor receives a value named *sal*. The semantics of constructor *Worker* is

$$\mathfrak{Const}[\![Worker()]\!]$$

$$= \curlywedge\ (sal, s).\left\{\langle e_0\left[salary \mapsto a_{salary}, lc \mapsto a_{lc}\right], s\left[a_{salary} \mapsto sal, a_{lc} \mapsto 4\right]\rangle\right\}$$

DEFINITION 7 (Method Semantics, $\mathbb{M}[\![-]\!]$ (and Method Call Semantics $\mathbb{M}_{call}[\![-]\!]$))

Let

$D_{in}, D_{out} \subseteq Val$ be semantic domains for input and output values,

m be a method,

$a_{in}\ and\ a_{pc}$ are the memory address for the constructor's data fields, and

inputs, lc_m be the method's entry point at a line of code.

Then the semantics of method, m is

$$\mathbb{M}[\![m]\!] \in [(D_{in} \times E \times S) \rightarrow \mathcal{P}(D_{out} \times E \times S)]$$

with

$$\mathbb{M}[\![m]\!] = \curlywedge\ (v_{in}, e, s).\ let\ \sigma_0 = \langle e[v \mapsto a_{in}, lc \mapsto a_{lc}], s[a_{in} \mapsto val_{in}, a_{lc} \mapsto lc_m]\rangle$$

Method semantics consists of input value and output value, regardless it is method definition or method call. The values expresses invariants for the methods. Using lazy behavioral subtyping, the method definition represents the specification and the method call represents the requirements. Therefore, there is no different in definition for the method definition and method call. The only different is the method call is called within a method, which the call can be instantiated by superclass and subclass.

Example 4.2 (Method Semantics)

Method *calc(double sal)* does receive one variable which is *sal*. However, the method does not change any object environment but only prints out the result for the *salary*.

$$\mathbb{M}[\![calc]\!] = \lambda\, (v_{in}, e, s).\{sal, e, s[\ a_{lc} \mapsto 8, e(salary) \mapsto v/4]\}$$

4.5 Abstract Semantics

Abstract semantics is a superset of program concrete semantics. The abstract semantics represents all possible cases of the program execution. Cousot (1996) states the program concrete semantics is safe whenever the abstract semantics is safe. The safety of the program is hold by the invariants to ensure there is no unlimited or over-range data.

DEFINITION 8 (Abstract semantics)

Let abstract semantics $\langle \overline{D}, \overline{\sqsubseteq}, \overline{\bot}, \overline{\top}, \overline{\sqcup}, \overline{\sqcap} \rangle$ be a complete lattice and let concrete domain be $\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle$. The abstract and concrete semantics are connected by Galois Connection as

$$\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle \ {}_{\gamma}\!\rightleftarrows^{\alpha}\ \langle \overline{D}, \overline{\sqsubseteq}, \overline{\bot}, \overline{\top}, \overline{\sqcup}, \overline{\sqcap} \rangle$$

where the abstract domain $\overline{D}$ is defined as object-oriented program, constructor, method, and method call.

DEFINITION 9 (Order, $\overline{\sqsubseteq}$)

Let t_i and $\overline{\mathrm{T}}$ be a trace of states; $\{\forall t_i \in \overline{\mathrm{T}} | \forall i \in \{1..n\}. \overline{\mathrm{T}} = \bigcup_{\mathrm{i=1}}^{\mathrm{n}} t_i\}$. The correspondence points is $\overline{D}_{\mathrm{p}} \in \mathcal{P}(\mathcal{T}(\Sigma))$ where

$\forall \overline{\mathrm{T}}_{\mathrm{p}} \in \overline{D}_{\mathrm{p}}$,

$t_1, t_2 \in \overline{\mathrm{T}}$, and

$t = \lambda\sigma(m, v) \,.\left\{m \in \mathbb{M}, v \in Var.\, \sigma_1 \xrightarrow{(m,v)} \sigma_2 = \sigma_1 \subseteq \sigma_2\right\}.$

Then, by using t_1, t_2, order $\bar{\sqsubseteq}$ is

$$t_1 \bar{\sqsubseteq} t_2 = \left(t_1 = \sigma_1 \xrightarrow{(m_1,v_1)} t_1 \wedge t_2 = \sigma_2 \xrightarrow{(m_2,v_2)} \bar{t} \wedge \sigma_1 \subseteq \sigma_2\right)$$

DEFINITION 10 Join, $\bar{\sqcup}$

The join of two or more points $\bar{\sqcup}_p \in \left[\overline{D} \times \overline{D} \rightarrow \overline{D}\right]$, defined as, for the trace, join means $\overline{\mathrm{T}}_{\mathrm{p}} \bar{\sqcup}_p \overline{\mathrm{T}}_{\mathrm{p}} = \mathcal{T}(\mathcal{P}(\Sigma)) \times \mathcal{T}(\mathcal{P}(\Sigma))$, and for the state, join means $\sigma_1 \cup \sigma_2 = \sigma_1(m, v) \cup \sigma_2(m, v)$.

DEFINITION 11 Top, $\bar{\top}$

The top of semantics points $\bar{\top} \in \overline{D}$, defined as $\bar{\top} = \bigcup_{\mathrm{i} \in 1..\mathrm{n}} \bar{\mathrm{t}}_{\mathrm{i}}$, such that $\forall\, \bar{\top}_{\mathrm{p}} \in \overline{\mathrm{D}}_{\mathrm{p}} \,.\, \overline{\mathrm{T}}_{\mathrm{p}} \subseteq \overline{\mathrm{T}}$.

DEFINITION 12 Meet, $\bar{\sqcap}$

The meet of two or more points $\bar{\sqcap}_p \in \left[\overline{D} \times \overline{D} \rightarrow \overline{D}\right]$, defined as, for the trace, meet means $\overline{\mathrm{T}}_{\mathrm{p}} \bar{\sqcap}_p \overline{\mathrm{T}}_{\mathrm{p}} = \mathcal{T}(\mathcal{P}(\Sigma)) \times \mathcal{T}(\mathcal{P}(\Sigma))$, and for the state, meet means $\sigma_1 \cap \sigma_2 = \sigma_1(m, v) \cap \sigma_2(m, v)$.

DEFINITION 13 (Abstract program, $\overline{\mathbb{P}}$)

Let

Σ_0 be an initial state, and

$\overline{\mathbb{P}}$ be $E \times S \times \mathcal{P}(\mathrm{A})$ be a domain of abstract program.

Then, an abstraction function of program semantics is

$$\alpha_{\mathbb{P}} : \mathbb{P}[\![\quad]\!] \rightarrow \overline{\mathbb{P}}$$

where $\alpha_{\mathbb{P}}$ is the union of all states such that $\alpha_{\mathbb{P}}\,(\Sigma_0 \cup \Sigma_\text{n})_{n\in\{1..\mathbb{N}\}}$.

Proposition:

Let $\gamma_{\mathbb{P}}$ be $\bigcup_1^{n\in\mathbb{N}} \alpha_{\mathbb{P}}^{-1}$, then $\langle \mathbb{P}[\![\quad]\!], \subseteq, \emptyset, \Sigma, \cup, \cap\rangle \; {}_{\gamma_{\mathbb{P}}}\!\rightleftarrows^{\alpha_{\mathbb{P}}} \; \langle \overline{\mathbb{P}}, \overline{\sqsubseteq}, \overline{\bot}, \overline{\top}, \overline{\sqcup}, \overline{\sqcap}\rangle$ is a sound approximation by a Galois connection.

Proof: The proof is shown applying the Definition 4 and Lemma 1.

Let concrete program be $\mathbb{P}[\![\quad]\!] \in \mathfrak{C}$ and abstract program be $\overline{\mathbb{P}} \in \mathfrak{A}$. Then,

$\overline{\mathbb{P}} = \alpha_{\mathbb{P}}\,(\mathbb{P}[\![\quad]\!]) = \bigcup \gamma_{\mathbb{P}}{}^{-1}(\mathbb{P}[\![\quad]\!])$ and

$\gamma_{\mathbb{P}}(\overline{\mathbb{P}}) = \bigcup \gamma_{\mathbb{P}}(\{\mathbb{P}[\![object - oriented\;program]\!]\})$ then

$\mathcal{P}(\mathbb{P}[\![\quad]\!]) \; {}_{\gamma_{\mathbb{P}}}\!\rightleftarrows^{\alpha_{\mathbb{P}}} \; \mathcal{P}(\overline{\mathbb{P}})$ q.e.d

DEFINITION 14 (Abstract constructor, $\overline{\mathfrak{Const}}$)

Let

k represents numbers of object in the main method of the program, and

$\overline{\mathfrak{Const}}$ be $\mathcal{P}(E \times S) \rightarrow \mathcal{P}(E \times S)$ be a domain of abstract constructor.

Then, an abstraction function of constructor semantics is

$$\alpha_{\mathfrak{Const}} : \mathfrak{Const}[\![\quad]\!] \rightarrow \overline{\mathfrak{Const}}$$

where $\alpha_{\mathfrak{Const}}$ is the initial states Σ_0 for each object exists in the program such that $\alpha_{\mathfrak{Const}}\left(\Sigma_0^\text{k}\right)_{k\in\{1..\mathbb{N}\}}$.

Proposition:

Let $\gamma_{\mathfrak{Const}}$ be $\bigcup_1^{n\in\mathbb{N}} \alpha_{\mathfrak{Const}}^{-1}$, then $\langle \mathfrak{Const}[\![\quad]\!], \subseteq, \emptyset, \Sigma, \cup, \cap\rangle \; {}_{\gamma_{\mathfrak{Const}}}\!\rightleftarrows^{\alpha_{\mathfrak{Const}}} \; \langle \overline{\mathfrak{Const}}, \overline{\sqsubseteq}, \overline{\bot}, \overline{\top}, \overline{\sqcup}, \overline{\sqcap}\rangle$ is a sound approximation by a Galois connection.

Proof: The proof is shown applying the Definition 5 and Lemma 1.

Let concrete program be $\mathfrak{Const}[\![\quad]\!] \in \mathfrak{C}$ and abstract program be $\overline{\mathfrak{Const}} \in \mathfrak{A}$. Then,

$\overline{\mathfrak{Const}} = \alpha_{\mathfrak{Const}}(\mathfrak{Const}[\![\quad]\!]) = \cup\, \gamma_{\mathfrak{Const}}{}^{-1}(\mathfrak{Const}[\![\quad]\!])$ and

$\gamma_{\mathfrak{Const}}(\overline{\mathfrak{Const}}) = \cup\, \gamma_{\mathfrak{Const}}(\{\mathfrak{Const}[\![constructor\ method]\!]\})$ then

$\mathcal{P}(\mathfrak{Const}[\![\quad]\!])\ {}_{\gamma_{\mathfrak{Const}}}\!\rightleftarrows^{\alpha_{\mathfrak{Const}}}\ \mathcal{P}(\overline{\mathfrak{Const}})$ q.e.d

DEFINITION 15 (Abstract method, $\bar{\mathbb{M}}$)

Let domain of abstract method $\bar{\mathbb{M}}$ be $\mathcal{P}(E \times S) \rightarrow \mathcal{P}(E \times S)$. Then, an abstraction function of method semantics be $\alpha_{\mathbb{M}} : \mathbb{M}[\![\quad]\!] \rightarrow \bar{\mathbb{M}}$ where $\alpha_{\mathbb{M}}$is a set of states such that $\alpha_{\mathbb{M}}(\mathcal{P}(\Sigma))$.

Proposition:

Let $\gamma_{\mathbb{M}}$ be $\bigcup_{1}^{n\in\mathbb{N}} \alpha_{\mathbb{M}}^{-1}$, then $\langle \mathbb{M}[\![\quad]\!], \subseteq, \emptyset, \Sigma, \cup, \cap \rangle\ {}_{\gamma_{\mathbb{M}}}\!\rightleftarrows^{\alpha_{\mathbb{M}}}\ \langle \bar{\mathbb{M}}, \bar{\sqsubseteq}, \bar{\bot}, \bar{\top}, \bar{\sqcup}, \bar{\sqcap} \rangle$ is a sound approximation by a Galois connection.

Proof: The proof is shown applying the Definition 7 and Lemma 1.

Let concrete program be $\mathbb{M}[\![\quad]\!] \in \mathfrak{C}$ and abstract program be $\bar{\mathbb{M}} \in \mathfrak{A}$. Then,

$\bar{\mathbb{M}} = \alpha_{\mathbb{M}}\ (\mathbb{M}[\![\quad]\!]) = \cup\, \gamma_{\mathbb{M}}{}^{-1}(\mathbb{M}[\![\quad]\!])$ and

$\gamma_{\mathbb{M}}(\overline{\mathbb{M}}) = \cup\, \gamma_{\mathbb{M}}(\{\mathbb{M}[\![method()]\!]\})$ then

$\mathcal{P}(\mathbb{M}[\![\quad]\!])\ {}_{\gamma_{\mathbb{M}}}\!\rightleftarrows^{\alpha_{\mathbb{M}}}\ \mathcal{P}(\bar{\mathbb{M}})$ q.e.d

DEFINITION 16 (Abstract method call, $\bar{\mathbb{M}}_{\text{call}}$)

Let domain of abstract method $\bar{\mathbb{M}}_{\text{call}}$ be $\mathcal{P}(E \times S) \rightarrow \mathcal{P}(E \times S)$. Then, an abstraction function of method semantics be $\alpha_{\mathbb{M}_{\text{call}}} : \mathbb{M}_{\text{call}}[\![\quad]\!] \rightarrow \bar{\mathbb{M}}_{\text{call}}$ where $\alpha_{\mathbb{M}_{\text{call}}}$is for one state Σ produced by the $\mathbb{M}_{\text{call}}[\![\quad]\!]$ such that $\alpha_{\mathbb{M}_{\text{call}}}(\Sigma)$.

Proposition:

Let $\gamma_{\mathbb{M}_{\text{call}}}$ be $\bigcup_{1}^{n\in\mathbb{N}} \alpha_{\mathbb{M}_{\text{call}}}^{-1}$, then $\langle \mathbb{M}_{\text{call}}[\![\quad]\!], \subseteq, \emptyset, \Sigma, \cup, \cap \rangle \ {}_{\gamma_{\mathbb{M}_{\text{call}}}}\rightleftarrows^{\alpha_{\mathbb{M}_{\text{call}}}} \ \langle \overline{\mathbb{M}_{\text{call}}}, \overline{\sqsubseteq}, \overline{\bot}, \overline{\top}, \overline{\sqcup}, \overline{\sqcap} \rangle$ is a sound approximation by a Galois connection.

Proof: The proof is shown applying the Definition 7 and Lemma 1.

Let concrete program be $\mathbb{M}_{\text{call}}[\![\quad]\!] \in \mathfrak{C}$ and abstract program be $\overline{\mathbb{M}_{\text{call}}} \in \mathfrak{A}$. Then,

$\overline{\mathbb{M}_{\text{call}}} = \alpha_{\mathbb{M}_{\text{call}}} (\mathbb{M}_{\text{call}}[\![\quad]\!]) = \cup\, \gamma_{\mathbb{M}_{\text{call}}}{}^{-1}(\mathbb{M}_{\text{call}}[\![\quad]\!])$ and

$\gamma_{\mathbb{M}_{\text{call}}}(\overline{\mathbb{M}_{\text{call}}}) = \cup\, \gamma_{\mathbb{M}_{\text{call}}}(\{\mathbb{M}_{\text{call}}[\![method()]\!]\})$ then

$\mathcal{P}(\mathbb{M}_{\text{call}}[\![\quad]\!]) \ {}_{\gamma_{\mathbb{M}_{\text{call}}}}\rightleftarrows^{\alpha_{\mathbb{M}_{\text{call}}}} \ \mathcal{P}(\overline{\mathbb{M}_{\text{call}}})$ q.e.d

4.6 Class Invariants

Class invariant gives specifications to class in order to check the class's correctness that cannot be checked by the compiler. The specifications use invariants, pre-condition, and post-condition to verify the behavior of the class. Hoare's style uses pre-condition and post-condition methods to check the program before execution time. Webber stated "a class invariant is a property that is true for all objects of a given class at all times" (Webber, 2001, p.87) . However, it is hard to have properties for objects that hold true value throughout program execution. The objects need weaker properties that allow to be temporarily broken in a method when the object has modification. Therefore, we use lazy behavioral subtyping method to produce properties (or specification) for methods that modify the objects. For the class invariant, we adopted Logozzo's work in 2004. This work is depicted because the proposed class invariant is rigorously proven and easily adapted to our proposed framework.

Theorem 4.2 Class Invariant (Logozzo, 2004)

Let $C = \langle const, f, m\rangle$ be a class with the set of states $\mathcal{P}(\Sigma)$ and $\overline{D}$ is an abstract domain such that

$$\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap\rangle \ {}_{\gamma}\!\rightleftarrows^{\alpha} \ \langle \overline{D}, \overline{\sqsubseteq}, \overline{\bot}, \overline{\top}, \overline{\sqcup}, \overline{\sqcap}\rangle.$$

The domain for the abstract constructor is

$$\overline{\mathbb{C}\mathfrak{onst}}[\![const]\!] \in [\mathcal{P}(E \times S) \rightarrow \mathcal{P}(E \times S)]$$

and the domain for the abstract method is

$$\coprod_{i=1}^{n} \overline{\mathbb{M}}[\![m_i]\!] \in [\mathcal{P}(E \times S) \rightarrow \mathcal{P}(E \times S)]$$

where $m \in M$ such that $\overline{\mathbb{C}\mathfrak{onst}}[\![const]\!] \subseteq \gamma(\overline{\mathbb{C}\mathfrak{onst}}[\![const]\!])$ and $\overline{\mathbb{M}}[\![m_i]\!] \subseteq \gamma\big(\overline{\mathbb{M}}[\![m_i]\!]\big)$.

Then, the class invariant *I* is based on the following recursive equation:

$$I = \overline{\mathbb{C}\mathfrak{onst}}[\![const]\!] \ \overline{\sqcup} \coprod_{i=1}^{n} \overline{\mathbb{M}}[\![m_i]\!] \qquad (6)$$

such that class semantics for class C is $\mathbb{C}[\![C]\!] \subseteq \gamma(I)$. As a tuple, the class invariant *I* is $\langle A, A_0, A_1, \dots, A_n\rangle \in \overline{D^{n+2}}$ where A is the class invariant, A_0 is the constructor semantics, and $A_1, \dots, A_n$ is the method semantics.

Proof: By formal definition of abstract interpretation (Cousot et al., 1977), the tuple of class invariant *I* complies with tuple of abstract interpretation where the set of abstract contexts is a complete lattice with ordering $\leq$. By fixpoint Tarski's theorem, both abstract semantic constructor and method are in the form of monotonic function, $f \in [D \rightarrow D]$. with the combination of constructor and method, the function becomes$[D^{n+2} \rightarrow D^{n+2}]$. With its least fixpoint, the equation (6) is a non empty complete lattice, where the least trace is the infimum, $\bot$ (the least value) will be taken from the concrete properties.

Example 4.1 : A class invariant for class Person

The class *Person* is taken from chapter 3, Figure 3.2. The class has two fields which are *name* and *bSalary* where *name* is for storing name of the person and *bSalary* is for storing the amount of salary of the person. The abstract domain for the class *Person* is based on the $\langle String, sign, sign, Esc\rangle$. The specification is to ensure the salary value which is hold by data field *bSalary* and *testSalary* are always positive number as salary must always be a positive value. The *String* is for the value of *name*, the *sign* is for the sign of *bSalary* and *testSalary* and *Esc* is for capturing the fields that may escape the object scope (return value). The iterations of the abstract domain is $\overline{D}^5$ with the abstract domain is

$$\overline{D} = String\ \times sign\ \times sign \times\ \mathcal{P}(\{name, bSalary, testSalary\ \})$$

The constructor, *calc* method and *salary* method are analysed to compute the class invariant because they modify object state. The method *writename* is included to show how the method does not change the state of data fields. Accessor method that starts with *get* word is also not included because the method does not modify the fields or data. Therefore, there is no such method in Figure 3.2. Using (6) as a tuple, the first element set is for class invariant, second element set is for constructor, and other element sets are for methods. The sign for *bSalary* with value 100 and *testSalary* with value 200 is positive (annotate as p).

The first iteration is the bottom value for all elements.

$$I^0 = \langle\langle \text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle, \langle\bot,\bot,\bot,\emptyset\rangle, \langle\bot,\bot,\bot,\emptyset\rangle, \langle\bot,\bot,\bot,\emptyset\rangle, \langle\bot,\bot,\bot,\emptyset\rangle\rangle$$

The second iteration corresponds to the abstract execution of the class constructor because the constructor is the first method call when an object is instantiated. It is

$$\begin{aligned}\mathbb{C}\mathfrak{onst}[\![Person(\quad)]\!](\langle \text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle) &= \langle \text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle \sqcup \langle \top, \top, \mathrm{p}, \emptyset\rangle \\ &= \langle \top, \top, \mathrm{p}, \emptyset\rangle\end{aligned}$$

The abstract executions to all methods are

$$\mathbb{M}[\![writeName]\!](\langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle) = \langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle \,\bar{\sqcup}\, \langle\bot, \bot, \bot, \emptyset\rangle$$

$$= \langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle$$

$$\mathbb{M}[\![calc]\!](\langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle) = \langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle \,\bar{\sqcup}\, \langle\bot, \bot, \bot, \emptyset\rangle$$

$$= \langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle$$

$$\mathbb{M}[\![salary]\!](\langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle) = \langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle \,\bar{\sqcup}\, \langle\bot, \bot, \bot, \emptyset\rangle$$

$$= \langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle$$

Union both I^0 that already has initial value for all data fields with current state of constructor and methods, the first class invariant's approximation is

$$I^1 = \left\langle \begin{matrix} \langle\top,\top, \mathrm{p}, \emptyset\rangle, \langle\top,\top, \mathrm{p}, \emptyset\rangle, \langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle, \langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle, \\ \langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle \end{matrix} \right\rangle$$

The second iteration is the post condition of the all methods with the $\langle\top,\top, \mathrm{p}, \emptyset\rangle$ taken from I^1. There is no change for method *writeName* because the method does not change any state of data field.

$$\mathbb{M}[\![writeName]\!](\langle\top,\top, \mathrm{p}, \emptyset\rangle) = \langle\top,\top, \mathrm{p}, \emptyset\rangle$$

The method *calc* reset the value of *bSalary* to 2100 as well as method *salary* which calls method *calc*. Therefore,

$$\mathbb{M}[\![calc]\!](\langle\top,\top, \mathrm{p}, \emptyset\rangle) = \langle\top,\top, \mathrm{p}, \emptyset\rangle \sqcup \langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle$$

$$= \langle\top,\top, \mathrm{p}, \emptyset\rangle$$

$$\mathbb{M}[\![salary]\!](\langle\top,\top, \mathrm{p}, \emptyset\rangle) = \langle\top,\top, \mathrm{p}, \emptyset\rangle \sqcup \langle\text{"Adam"}, \mathrm{p}, \mathrm{p}, \emptyset\rangle$$

$$= \langle\top,\top, \mathrm{p}, \emptyset\rangle$$

Therefore, the second class invariant's approximation is

$$I^2 = \left\langle \begin{matrix} \langle\top,\top, \mathrm{p}, \emptyset\rangle, \langle\top,\top, \mathrm{p}, \emptyset\rangle, \langle\top,\top, \mathrm{p}, \emptyset\rangle, \langle\top,\top, \mathrm{p}, \emptyset\rangle, \\ \langle\top,\top, \mathrm{p}, \emptyset\rangle \end{matrix} \right\rangle$$

The class invariant's approximation on the third iteration has same result as previous iteration. Then, the iteration stops at I^2.

In conclusion, from the static analysis of Figure 9.1 code, the class invariant produced is

//name:String, $bSalary \leq 0$ && $bSalary \geq 0, testSalary > 0$

From the class invariant, *bSalary* data field can be positive or negative that is wrong for the specification of the salary, which must positive. Therefore, the code needs conditions to ensure the *bSalary* is always positive value. For example,

```
Person(String n, int s){
    name = n;
    if (s>0)
      bSalary = s;
    else
     bSalary = 0;
  }
```

Figure 4.2: The new code for the constructor of class Person

With the if-else condition, *bSalary* can accept positive value only, which the class invariant is //name:String, $bSalary \geq 0, testSalary > 0$.

DEFINITION 17 (Method Invariant)

Let $D_{in}, D_{out} \subseteq \{\bar{D} \mid \bar{D} \subseteq (E \times S \times \mathcal{P}(A))\}$ be the semantic domain for parameter and return value of method. Then, the method invariant is $D_{in} \sqcup D_{out}$.

4.7 Invariants in Inheritance

Inheritance is the essence of object-oriented programming language. It allows classes to be reused by making the class properties generalized or specialized. Invariants in the presence of inheritance involve both superclass and subclass. The superclass keeps on expanding by having new subclasses. Therefore, the invariants are also changing based on new subclasses. In order to ensure the invariants are easy to monitor, they are generated in modular form. Modularity technique in generating invariants is a technique that the invariants are produced based on units, for example, class, method or subclass. The technique is mainly used in ESC/Modulo 3 and ESC/Java which are tools to find run time errors at compile time. ESC/Modula 3 is the predecessor of ESC/Java for checking C and C++ languages. As its name implies, ESC/Java is for Java language (now it has been extended to ESC/Java2). Being modular is crucial in generating invariants to support large programming codes.

Example 4.2 : Non-Modular-Based Invariants in Inheritance

Let us say, there are two classes named *Rodent* and *Mouse* where *Rodent* is the base class of *Mouse*. The static analysis starts with *Rodent* class invariant, $X = \overline{\mathfrak{Const}}[\![const_R]\!] \mathbin{\overline{\sqcup}} \coprod_{i=1}^{n} \overline{\mathbb{M}}[\![m_i]\!]$ where the class is defined as $\langle f, const, m \rangle$. The extension class for object-oriented language carries data from its superclass. When it is instantiated, it automatically calls the constructor and data members of the superclass. Therefore, it is able to act differently from its superclass but with its superclass feature. For the extended class, *Mouse*, the class invariant Y is defined as below if the equation is based on union operation ($Rodent\ \cup Mouse$):

$$Y = \overline{\mathfrak{Const}}[\![const_R]\!] \mathbin{\overline{\sqcup}} \overline{\mathfrak{Const}}[\![const_M]\!] \mathbin{\overline{\sqcup}} \coprod_{i=1}^{n} \overline{\mathbb{M}}[\![m_i]\!] \mathbin{\overline{\sqcup}} \coprod_{j=1}^{k} \overline{\mathbb{M}}[\![m_j]\!] \tag{7}$$

The *Rodent* class definition is $\langle f_R \cup f_M, const_R \cup const_M, m_R \cup m_M \rangle$. Fields of $f_R \cup f_M$ are data members that belong to both classes, however, for the constructors and methods, the data fields are variables that are used to support calculation in the constructor and methods. However, the equation becomes more complex as more subclasses are added to the superclass because the superclass is reverified every time a new subclass is added to its hierarchy. This can lead to low performance during the static analysis of verification as the code becomes more and more complex. Therefore, the old invariant or previous invariant of two predicates stores information of invariants that have been verified previously to be used again for the next process.

DEFINITION 18 (Modular-based Invariants in Inheritance)

In Theorem 4.2, the class invariant is extended as follows:

Let A be previous invariant with its initial abstract domains

$$\overline{\mathbb{Const}}[\![const_0]\!] \ \bar{\sqcup} \ \coprod_{i=0}^{n} \overline{\mathbb{M}}[\![m_{0_i}]\!] \qquad (8)$$

Then, the new abstract domains that come from new subclasses are

$$\overline{A_i} = \overline{\mathbb{Const}}[\![const_i]\!] \ \bar{\sqcup} \ \coprod_{i=0}^{n} \overline{\mathbb{M}}[\![m_i]\!], i = \{1..n\} \qquad (9)$$

Therefore, the invariant for inheritance is

$$H = A \ \bar{\sqcup} \ \overline{\mathbb{Const}}[\![const_i]\!] \ \bar{\sqcup} \ \coprod_{i=0}^{n} \overline{\mathbb{M}}[\![m_i]\!]\,(H) \qquad (10)$$

$$= \ A \ \bar{\sqcup} \ \coprod_{i=1}^{n} \overline{A_i} \qquad (11)$$

such that the new inheritance invariant is $\mathbb{C}[\![C]\!] \subseteq \gamma(H)$.

Proof: By comparing both equation of (7) and (11), it is stated that to improve execution time during static analysis on inheritance, equation (11) is preferable since equation (11) is not extending every time new subclasses are added as in equation (7).

The modular part of the equation is based on the merging of previous or first abstract constructor and abstract method as one module. From the above definition, the mergence is represented as A. Every time, new subclasses are added to the program, the A has no need to be verified. However, the invariant of the new subclasses are merged with the A as an invariant for inheritance as H. This mergence changes to A, when the program adds another new subclass, which later the invariant becomes H. This process repeats as new subclasses are added to the program. As a consequence, if the equation were implemented, logically, the process of generating new invariant is faster as old invariants are used when program extension happened.

Example 4.3 : Modular-Based Invariants in Inheritance

Class *Person* in Figure 3.2 consists of two data members which are *name* and *bSalary* (we omit *testSalary*, temporarily, for simplicity). It has a constructor and three methods which are *writeName*(), *calc*(), and *salary*(). For subclass *Worker*, it has one new data field named *tSalary*. The *tsalary* receives its initial value through constructor variable named *tot*. Using the same specification to check the value is a positive number, the abstract domain $\overline{D}^4$ chosen is

$$\overline{D} = sign \times \mathcal{P}(\{bSalary, tSalary\})$$

We abstract away name and *testSalary* as they do not change the state of *tSalary*. *bSalary* is included because it is not only can change the state the *tSalary* but also data field of superclass *Person*, which can be accessed by the subclass *Worker*. The first iteration is

$$I^0 = \langle\langle\bot,\emptyset\rangle,\langle\bot,\emptyset\rangle,\langle\bot,\emptyset\rangle,\langle\bot,\emptyset\rangle\rangle$$

The second iteration corresponds to the abstract execution of the class constructor;

$$I^1 = \langle\langle \top, \emptyset\rangle, \langle \top, \emptyset\rangle, \langle \bot, \emptyset\rangle, \langle \bot, \emptyset\rangle\rangle$$

The third iteration is

$$I^2 = \langle\langle \top, \emptyset\rangle, \langle \top, \emptyset\rangle, \langle \bot, \emptyset\rangle, \langle \top, \emptyset\rangle\rangle$$

The method *calc* has a statement *tsalary+=bSalary*. However, the union of positive value of *bSalary* with the least upperbound value is $\langle pos\rangle \sqcup \langle \top\rangle = \top$. The positive value of *bSalary* is used as in new code in Figure 4. Therefore, the class invariant produced for *tSalary* is

$$// \; tSalary \leq 0 \;\&\&\; tSalary \geq 0$$

With the class invariant, *tSalary* can accept any positive or negative value which does not correct for a salary number. However, for this example, we proceed as it is. Then, using (5), the invariant for both superclass and subclass is

$$\begin{aligned} H &= A \sqcup \coprod_{i=1}^{n} \overline{A_i} \\ &= bSalary \geq 0 \;\sqcup\; 0 \geq tSalary \geq 0 \end{aligned}$$

which the invariant is

$$// \; bSalary \geq 0, \; tSalary \geq 0 \;\&\&\; tSalary \leq 0$$

This is the result of the union of superclass invariant and subclass invariant. H is updated when a new subclass is added. However, there is no need to verify the superclasses that have been analyzed.

4.8 Invariants in Method Overriding

Method overriding exists in inheritance as an important tool to support reusability in object-orientation. Logozzo (2005) used best case over approximation of the method which was $\overline{\mathbb{M}}_{call}[\![m]\!](\top)$ because input of m was not known and

$$d \,\overline{\sqsubseteq}\, \overline{\top} \Longrightarrow \overline{\mathbb{M}}_{call}[\![m]\!](\overline{d}) \,\overline{\sqsubseteq}\, \overline{\mathbb{M}}_{call}[\![m]\!](\overline{\top}) \tag{12}$$

The best case over approximation is always the maximum value of the variable. In order to avoid the over approximation, Dovland et al. (2008) proposes a method called Lazy behavioral subtyping which considers all inputs and outputs of methods in method overriding. Lazy behavioral subtyping is a method to reason about late binding of method calls. It is developed with less restriction on pre- and post- condition of methods compared to conventional behavioral subtyping.

DEFINITION 19 (Lazy behavioral subtyping (Dovland et al., 2008))

Let (p, q) and (r, s) be assertion pairs and let U denote the sets $\{(p_i, q_i) | 1 \leq i \leq n\}$ and V denote the sets $\{(r_i, s_i) | 1 \leq i \leq m\}$. Entailment is defined over assertion pairs and sets of assertion pairs by

1. $(p, q) \rightarrow (r, s) \triangleq (\forall \bar{z}_1 . p \Longrightarrow q') \Rightarrow (\forall \bar{z}_2 . r \Longrightarrow s')$, where $\bar{z}_1$ and $\bar{z}_2$ are the logical variables in (p, q) and (r, s), respectively.
2. $U \rightarrow (r, s) \triangleq \left(\bigwedge_{1 \leq i \leq n} \left(\forall \bar{z}_i . p_i \Longrightarrow q'_i\right)\right) \Rightarrow (\forall \bar{z} . r \Longrightarrow s')$
3. $U \rightarrow V \triangleq \bigwedge_{1 \leq i \leq m} U \rightarrow (r_i, s_i)$

In the context of class analysis, method definition uses $S(Class, method)$ as a set of specifications. $R(Class, method)$ is a set of requirements for method call. It is used when an overridden method is called from another method. In inheritance, function $S \uparrow$ defined as a method f exists in subclass and its immediate superclass. Therefore, the specification generates from the superclass and the subclass, $S\uparrow(\text{subclass}, \text{f}) \stackrel{\text{def}}{=} S(\text{subclass}, \text{f}) \cup S\uparrow(\text{superclass}, \text{f})$. The function is a recursive function, if the superclass also has another superclass. The entailment rule extends to below in inheritance program. In general, the rules show that requirements of method call requires specifications from method definition of the method's superclass and subclass. This technique gives all possible invariants that can be used to analyze the method call.

1. $S\uparrow(Class, method) \Rightarrow R\uparrow(Class, method)$,
2. $S\uparrow(Class, method) = S(Class_1, method) \cup S(Class_2, method) \Rightarrow R\uparrow(Class, method)$,
3. $(r_i, s_i) \in R(Class, method)$,
4. $U \rightarrow V \triangleq \bigwedge_{1\le i\le m} U \rightarrow (r_i, s_i)$,

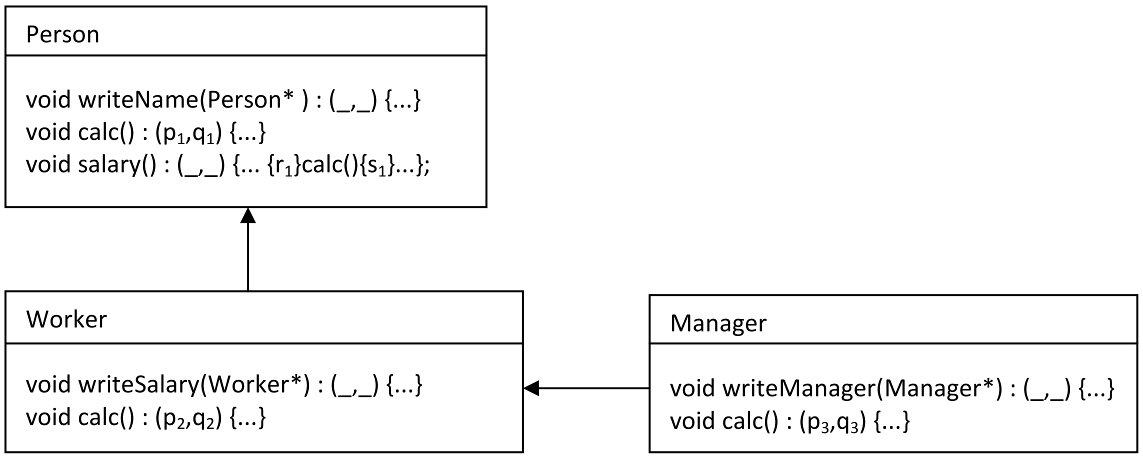

Person
void writeName(Person*) : (_,_) {...}
void calc() : (p1,q1) {...}
void salary() : (_,_) {... {r1}calc(){s1}...};
Worker
void writeSalary(Worker*) : (_,_) {...}
void calc() : (p2,q2) {...}
Manager
void writeManager(Manager*) : (_,_) {...}
void calc() : (p3,q3) {...}

Figure 4.3: Inheritance relationship with proof outline

Figure 4.3 illustrates a simple version of the class diagram using the same example of inheritance taken from §3.3 of Figure 3.2. The figure omits all constructors and lines of code for simplicity. Note that *p*, *q*, *r*, and *s* specify pre and post condition for the method and method call. The *p* and *q* represent specification for method where *p* is for pre-condition and *q* is for post-condition. The *r* and *s* represent pre-condition and postcondition for the method calls. The q' and s' are for new post-conditions. The post-conditions consists of old (q and s) and new post-conditions due to changes or new local variables in pre-conditions. Using abstract interpretation, all of *p, q, r,* and *s* are converted into $\overline{\mathbb{M}}[\![m_i]\!]$. All methods in the classes have method invariant as $B = D_{\text{in}} \sqcup D_{\text{out}}$ and D_{in}, $D_{\text{out}} \in \{\bar{D} \mid \bar{D} \subseteq (\mathrm{E} \times \mathrm{S} \times \mathcal{P}(\mathrm{A}))\}$. Using Definition 19, equations for invariants in method overriding is produced in Definition 20.

DEFINITION 20 (Invariants of Method Overriding)

Let methods in the classes be method invariant which is $M = D_{\text{in}} \sqcup D_{\text{out}}$ where D_{in}, $D_{\text{out}} \in \{\bar{D} \mid \bar{D} \subseteq (\mathrm{E} \times \mathrm{S} \times \mathcal{P}(\mathrm{A}))\}$. Method semantics is

$\overline{\mathbb{M}} \in [(\mathrm{D}_{\text{in}} \times \mathrm{E} \times \mathrm{S}) \rightarrow \mathcal{P}(\mathrm{D}_{\text{out}} \times \mathrm{E} \times \mathrm{S})]$ and all methods in class are represented by $\coprod_{i=1}^{n} \overline{\mathbb{M}}[\![m_i]\!]$. Then, the invariants of method overriding is

$$\overline{\mathbb{M}}_{call}[\![om]\!] = \overline{\mathbb{M}}[\![om_{super}]\!] \;\bar{\sqcap}\; \coprod_{0<i<n} \overline{\mathbb{M}}[\![om_{sub_i}]\!] \tag{13}$$

such that $\overline{\mathbb{M}}_{call}[\![om]\!] \subseteq \gamma(\overline{\mathbb{M}}[\![-]\!])$, $\overline{\mathbb{M}}[\![om_{super}]\!] \subseteq \gamma(\overline{\mathbb{M}}[\![-]\!](superclass))$, and $\overline{\mathbb{M}}[\![om_{sub_i}]\!] \subseteq \gamma(\overline{\mathbb{M}}[\![-]\!](subclass))$. The method semantic $\overline{\mathbb{M}}_{call}[\![om]\!]$ is used for overridden method (*om*) call which its definition is determined at run time. For example, whether the *calc()* method of Figure 4. is called from object *Person* or object *Worker*, the methods conjunction cover both situations of method calling.

Proof: By using the fixpoint theorem, the least abstract fixpoint by abstract interpretation is a sound approximation for its concrete fixpoint with Galois connection, $\langle \mathcal{P}(\sum), \subseteq, \text{null}, \sum, \cup, \cap \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \bar{\mathrm{D}}, \bar{\sqsubseteq}, \bar{\bot}, \bar{\top}, \bar{\sqcup}, \bar{\sqcap} \rangle$. Then, $\mathbb{M} \subseteq \gamma(M)$. Hence, the function is monotonic. Therefore, for each overridden method, it uses the concrete least fixpoint for its abstract domains.

4.9 Conclusion

In this chapter, we designed an abstract formal framework of invariants generation for the purpose of verification on method overriding. The framework developed equations using abstract interpretation. There are two equations produced and proved, which are invariants inheritance and method overriding in equation (11) and (13). The equations are used to generate invariants that able to verify program with inheritance and method overriding. To validate the equations, an experiment will be conducted to the equations on case studies.

Chapter 5
Result and Discussion

I didn't fail the test, I just found 100 ways to do it wrong.
-Benjamin Franklin

This chapter presents the evaluation of the proposed equation. It is important to apply the equation on object-oriented programs to check its reliability and validity. The reliability is achieved by applying the equation on two case studies, which are Salary System and Quadrilaterals System to generate invariants. The validity is achieved by discussing the result of the cases studies with Java Specification Language. Every case study has invariants that generated using behavioral subtyping and lazy behavioral subtyping method. Each result of each method is discussed and analyzed to compare the differences.

5.1 Case Study 1: Salary System

Figure 5.1 is a code of Salary System taken from Figure 3.2 with the modification for constructor of class Person and class Worker. The new code of the program is used to ensure the specification of salary value is always positive. The program is written in Java language. The purpose of method overriding in the program is to reuse data of its superclass. The method *calc()* that is used to demonstrate the late binding call of method overriding is method *calc().* The method *calc()* appears on both superclass and subclass which are called from superclass object and subclass object. Therefore, method *calc()* in class *Worker* will override definition of class *Person* whenever possible. In addition, the method *calc()* is also called from a method in superclass named *salary()*.

```
public class Person {

public class Person {
  private String name = "Adam";
  private int bSalary = 100;
  public int testSalary = 200;

  public Person(String n, int s){
    name = n;
    if (s>0)
      bSalary = s;
    else
      bSalary = 0;
  }
  public void writeName(Person p){
    System.out.print("The employer name is " + name);
  }
  public void calc(){
    bSalary = 2100;
    System.out.println("Person::calc()");
  }
  public void salary(){
    calc();
  } } //end of class Person

public class Worker extends Person{
  public int testSalary = 300;
  private double tSalary;

  public Worker(String nama, int gaji, double tot){
   super(nama,gaji);
   tSalary = tot;
  }
  public void writeSalary(Worker w){
   System.out.println(w.tSalary);
  }
  public void calc(){
   tSalary += bSalary;
   System.out.println( tSalary);
  } } //end of class Worker
```

Figure 5.1: Salary System Revisited

5.1.1 Invariants Generation Using Behavioral Subtyping

Let method invariant be $M = \overline{D}_{\text{in}} \square \overline{D}_{\text{out}}$, method semantics be $\mathbb{M} \in [(\mathrm{D}_{\text{in}} \times \mathrm{E} \times \mathrm{S}) \rightarrow \mathcal{P}(\mathrm{D}_{\text{out}} \times \mathrm{E} \times \mathrm{S})]$, and $\mathbb{M} \subseteq \gamma(\overline{\mathbb{M}})$. Method *calc()* of class *Person* has the invariant of

// *bSalary*≥ 0

where, $\overline{D}_{\text{in}}$ is $bSalary \geq 0$ and $\overline{D}_{\text{out}}$ is $bSalary \geq 0$ taken from *bSalary=2100* which is a positive value. Therefore, $M = \overline{D}_{\text{in}} \square \overline{D}_{\text{out}}$

$$= bSalary \geq 0 \square bSalary \geq 0$$

$$= bSalary \geq 0$$

The method *calc()* of class Worker has the invariant of

// *bSalary*≥ 0, *tSalary*≥ 0*&&* *tSalary*≤ 0

where, $\overline{D}_{\text{in}}$ is $bSalary \geq 0 \sqcup tSalary \geq 0 \sqcap tSalary \leq 0$ and $\overline{D}_{\text{out}}$ is $bSalary \geq 0 \sqcup tSalary \geq 0 \sqcap tSalary \leq 0$ taken from *tSalary += bSalary*. Based on the *Worker's* constructor, there is no limit for *tSalary*. Therefore, the *tSalary* has any value of positive and negative. The result violates the program's specification, which states the salary must be positive value. However, as an example, the program proceeds as it is. Then,

By using behavioral subtyping, method invariant for method *calc()* of class *Person* is $bSalary \geq 0 \sqcup tSalary \geq 0 \sqcap tSalary \leq 0$ where the statement derives from combination of invariant of *calc()* of class *Person* and class *Worker*; $bSalary \geq 0 \sqcup (bSalary \geq 0 \sqcup tSalary \geq 0 \sqcap tSalary \leq 0)$. However, if the code of method *calc()* of class *Worker* changes to *tSalary = testSalary + 1000* (from *tSalary +=bSalary*), the method invariant becomes $bSalary \geq 0 \sqcup (tSalary \geq 0 \sqcap tSalary \leq 0) \sqcup (testSalary \geq 0 \sqcap testSalary \leq 0)$, which is stated as

// *bSalary*≥ 0, *tSalary*≥ 0 *&&* *tSalary*≤ 0, *testSalary*≥ 0 *&&* *testSalary*≤ 0

The addition of *bSalary* due to the fact that the method *calc()* is an inherited method from its superclass, which it must implement all superclass specifications for *calc()*. When the code below is executed (using *tSalary = testSalary + 1000)*

```
public class Employment {
  public static void main(String[] args) {
    Person objW = new Worker (“Ali”,1800,0);
    objW.salary();
    Person objP = new Person (“Adam”,23000);
    objP.salary();
  }
   }
```

, the

$bSalary \geq 0 \sqcup (tSalary \geq 0 \ \sqcap \ tSalary \leq 0) \sqcup (testSalary \geq 0 \ \sqcap \ testSalary \leq 0)$

is used. This is a valid statement for any value for both data objects because they use maximum value of both positive and negative values. For *objP*, even though, there are only two data fields used, the checking takes *testSalary* into consideration, because that is the rule used in behavioral subtyping. The purpose is to avoid miss analyzed. However, as a result, over-approximation on method semantics occurs in the method overriding verification.

5.1.2 Invariants Generation Using Lazy Behavioral Subtyping

For the class analysis using lazy behavioral subtyping, the set of specification of method *calc()* in class *Person* is represented as $S(Person, calc)$. Using same class invariant and method semantics, $S(Person, calc)$ is $(bSalary \geq 0)$. The method invariant for *salary()* is also $(bSalary \geq 0)$ because there is no $\overline{D}_{\mathrm{in}}$ and $\overline{D}_{\mathrm{out}}$ for the method.

The requirements of *calc()* called in *salary()* is represented as $R(Person, calc)$. Since $S(Person, calc) \rightarrow R(Person, calc)$ as formalize in equation (13), then $(bSalary \geq 0) \in R(Person, calc)$. For the class *Worker*, the specification for *calc()* is

$(bSalary \geq 0 \sqcup tSalary \geq 0 \sqcap tSalary \leq 0) \in S(Worker, calc)$.

However, if the code of method *calc()* of class *Worker* changes to *tSalary = testSalary + 1000* (from *tSalary +=bSalary*), the method invariant becomes $(tSalary \geq 0 \sqcap tSalary \leq 0) \sqcup (testSalary \geq 0 \sqcap testSalary \leq 0)$, which is stated as

// tSalary≥ 0 && tSalary≤ 0, testSalary≥ 0 && testSalary≤ 0

The statement shows that there is no need to implement invariant of superclass as opposed to behavioral subtyping. Then, using lazy behavioral subtyping's entitlement rule of

$$S \uparrow (Worker, calc) \rightarrow R \uparrow (Person, calc)$$

, the requirements for internal call of *calc()* is

$\{(bSalary \geq 0), (bSalary \geq 0 \sqcup tSalary \geq 0 \sqcap tSalary \leq 0)\} \in R(Person, calc)$

If, we use *tSalary = testSalary + 1000* statement, the requirements for internal call of *calc()* becomes

$\{(bSalary \geq 0),$

$(tSalary \geq 0 \sqcap tSalary \leq 0) \sqcup$

$(testSalary \geq 0 \sqcap testSalary \leq 0)\} \in R(Person, calc)$

To show the effect of method overriding, let us say in the main method for these classes, has the following:

```
public class Employment {
 public static void main(String[] args) {
   Person objW = new Worker (“Aliyah”,1800,0);
   objW.salary();
 }
```

```
    }
```

The method *salary()* is called by using an object of *Worker*, called *objW*. The instantiation of $objW$ involves both class *Person* and *Worker* as both classes are related as in inheritance. When the $objW$ is instantiated, both class *Person* and *Worker* are activated. For the method *salary(),* its method invariant is $\overline{\mathbb{M}}[\![salary]\!]$. However, method *calc()* that is hidden inside method *salary()* has both specification from method invariant of method *calc()* of class *Person* and of class *Worker*, which is

$$\overline{\mathbb{M}}_{call}[\![calc]\!] = \overline{\mathbb{M}}[\![calc_{Person}]\!] \,\overline{\sqcap}\, \overline{\mathbb{M}}[\![calc_{Worker}]\!].$$

So, it becomes

$$S \uparrow (Worker, calc) \rightarrow R \uparrow (Person, calc)$$
$$= \{(bSalary \geq 0), (bSalary \geq 0 \ \sqcup \ tSalary \geq 0 \sqcap \ tSalary \leq 0)\}$$

Therefore, for *objW.salary();*,where the $objW$ instantiates by the *Worker* object, the invariant invokes

$$//bSalary \geq 0\ , tSalary \geq 0 \ \&\& \ tSalary \leq 0$$

Let us say in the main method, there are below codes. In this code, an object named *objP* instantiates from class *Person*. Through *objP*, method *salary()* is called.

```
public class Employment {
  public static void main(String[] args) {
    Person objP = new Person (“Ali”,1800);
    objP.salary();
  }}
```

Therefore, for *objW.salary();*,where the $objW$ is a *Person* object, the invariant is

$$//bSalary \geq 0$$

From the invariants produced, it shows that invariants used are relaxed to object call only. The analysis using lazy behavioral subtyping does not implement all invariants of superclass. Therefore, the approximation value from the invariants limit to data fields used only.

5.2 Case Study 2: Quadrilaterals System

The Quadrilateral system is a simple program that draws a shape of four sides with 90 degree angle only. The specification is to ensure the sides are in positive value. The program can draw two basics four-sided shapes, which are square and rectangle. By referring to Figure 5.2, there are two main classes that have inheritance relationship, which are *Shape* class and *Rectangle* class. Both classes have *getData()* and *setData()* method. Both methods from *Rectangle* class override methods in *Shape* class accordingly. The *setData()* method of *Shape* class has new value after overriding by *Rectangle* class where the *side1* is not a single value but multiply by 5. Therefore, this program will always draw a rectangle instead of square because the side value is changed in the program.

```
public class Shape {
  private int side1 = 9;

   public Shape(int s1){
     if (s1>0)side1 = s1;
     else side1 = 0;
   }
   public int getData(){
     return side1;
   }
   public void setData(int x){
     side1 = x;
   }}

public class Rectangle extends Shape{
  private int side2 = 9;
   public Rectangle(int s1,int s2){
     super(s1);
     if (s2>0) side2 = s2;
     else side2 = 0;
   }
   public int getData(){
     return side2;
   }

   public void setData(int a){
     super.setData(a*5);
     side2=a;
   }

   public int draw(){
     int total = (2*super.getData()) + (2*this.getData());
     for (int i=1; i<=super.getData(); i++)
       System.out.print("*");
     for (int j=1; j<=(this.getData()-2); j++){
       System.out.print("\n*");
     for (int i=1; i<=(super.getData()-2); i++)
       System.out.print(" ");
     System.out.print("*");
     }
     System.out.print("\n");
     for (int k=1; k<=super.getData(); k++)
       System.out.print("*");
     return total;
   }}

public class DrawShape {
   public static void main(String[] args) {
     Shape c = new Rectangle(2,2);
     c.setData(6);
     ((Rectangle)c).draw();
   }}
```

Figure 5.2: Quadrilaterals System

5.2.1 Invariants Generation Using Behavioral Subtyping

There are two overriding methods in the system which are *getData()* and *setData()*.For the class *Shape*, the *getData()* has no $\overline{D}_{\mathrm{in}}$ but has $\overline{D}_{\mathrm{out}}$ from *side1*. Therefore, the method invariant taken from the class *Shape* invariant, which is

$$//side1 \geq 0$$

The *setData()* contains data field *side1* and variable *x*, so the method invariant is

$$//(x \geq 0 \; \&\& \; x \leq 0), (side1 \geq 0)$$

The data field *side1* in always positive number as the code has checked the input value using if-else condition. However, there is no condition to control the value of *x*. Nevertheless, the data field is always positive regardless of the *x* value.

For subclass *Rectangle* of *Shape*, using behavioral subtyping, the *getData()* invariant is based on data fields from superclass and subclass. Therefore, the *getData()* in *Shape* changes as well as to avoid miss analyzed during verification. Then, the method invariant for both *getData()* is

$$//side1 \geq 0, side2 \geq 0$$

For the *setData()* of *Rectangle*, the method invariant is

$$//(a \geq 0 \ \&\& \ a \leq 0), (side2 \geq 0)$$

The convergence of the method invariant for both superclass and subclass makes *setData()* changes to

$$//(x \geq 0 \ \&\& \ x \leq 0), (side1 \geq 0), (a \geq 0 \ \&\& \ a \leq 0), (side2 \geq 0)$$

Below is the sample code of an object *Shape* and *Rectangle* calls method *setData(int x).* For both situations, the same invariant is employed due to behavioral subtyping is engaged to generate the invariants. Therefore, there is over-approximation values from invariants existed even though the invariants are not required for the analysis process.

```
public class DrawShape {
  public static void main(String[] args) {
    Shape c = new Rectangle(2,2);
    c.setData(6);
    Shape d = new Shape(2);
    d.setData(3);
  }
}
```

5.2.2 Invariants Generation Using Lazy Behavioral Subtyping

Using same technique as explained in §5.1.2, the class *Shape*'s *getData()* has no $\overline{D}_{\text{in}}$ but has $\overline{D}_{\text{out}}$ from *side1*. Therefore, the method invariant adopted from the class *Shape* invariant is

$$//side1 \geq 0$$

Method invariant for *Shape*'s *setData()* is

$$//(x \geq 0\ \wedge x \leq 0), (side1 \geq 0)$$

Method invariant for *Rectangle*'s *getData()* is

$$//side2 \geq 0$$

Method invariant for *Rectangle*'s *setData()* is

$$//(a \geq 0\ \wedge a \leq 0), (side2 \geq 0)$$

Using lazy behavioral subtyping, there is no need to merge invariants as in behavioral subtyping. Therefore, the invariant used depends on object call. The lazy technique limits the expansion of the invariant generation as the program can extend by having new subclasses. Using the same sample code in §5.2.1,

```
public class DrawShape {
  public static void main(String[] args) {
    Shape c = new Rectangle(2,2);
    c.setData(6);
    Shape d = new Shape(2);
    d.setData(3);
  }
}
```

, the invariant produced for *c.setData(6)* is

$$//(a \geq 0\ \wedge a \leq 0), (side2 \geq 0)$$

and the invariant for *d.setData(3)* is

$$//(x \geq 0\ \wedge x \leq 0), (side1 \geq 0)$$

The invariants generated show that there are different invariants produced depending on the data fields involved in the method. In fact, there is no mergence of the invariant from superclass and subclass for method overriding as in behavioral subtyping. The reason is lazy behavioral subtyping reduces the invariant that can expand when the inheritance structure grows.

5.3 Analysis of the Case Studies

By referring to Table 5.1, the data is taken from two previous case studies explained in previous section; §5.1-§5.2. They are Salary System and Quadrilaterals System. Salary System is a system that applied method overriding to reuse data from superclass. In addition, there is a call for overridden method from superclass and subclass within other method. The call method is only known at runtime using an object. Quadrilateral System is a system that implemented method overriding for the purpose of specialization. There is an overriding method of subclass specializes the definition of overridden method of a superclass. Table 5.1 is divided into invariants produced for two variables where each case study compares using the method of behavioral subtyping (BS) and lazy behavioral subtyping (LBS). The methods are used because behavioral subtyping is the current method researchers mostly used as explained in §2.10 and lazy behavioral subtyping as a solution method in proposed abstract formal framework. The case studies have been analyzed statically using invariant for inheritance as in equation (11) (pg.87) and invariant for method overriding as in equation (13) (pg.92). Three values produced from each case study. It can be zero (0), positive value (+veVal), and negative value (-veVal). The double dotted (..) represents the range between one value to another value.

The findings of Table 5.1 show that invariants generated for three methods of Salary System have different value using method BS and LBS. Method *calc* and *salary* are method definition and ..*calc*.. is a method call. Both *salary* and ..*calc*.. method only appears in superclass. Therefore, boxes in subclass are in grey. For method *calc* that exists in superclass and subclass, produces same value for the invariant generated using BS. However, different result produced using LBS. The superclass consists of value for *bSalary* in superclass and has values for *bSalary* and *tSalary* in subclass. The value shows that LBS result relaxes to the data fields used for the method definition in particular class only; without considering other classes. For the method *salary*, both methods produce same result, which is 0 until positive value. For method call ..*calc*.., there is no invariants generated using BS because the method limits its rule to method invariant not method call invariant. The reason is the method can be called by any object, which is hard to predict. Then, the LBS solves the problem by generating invariant by adopting invariants produced by the method's superclass and subclass. Therefore, any object can invoke the method.

For the case study of Quadrilaterals System, there is no different value for the invariants of method *getData* and *setData* using BS and LBS. For the method *getData*, using BS, the invariants in superclass are same as in subclass. The same result produced for method *setData*. Using LBS, for method *getData*, there is no invariant for tSalary in superclass and no invariant for *bSalary* in subclass. The same result generated for invariants in method *setData* as well. The reason is LBS only produced invariant based on the data fields of the method without considering its immediate superclass's or subclass's invariant. This contradicts to BS where the superclass's invariant changes as subclass added.

By comparing both case studies, it shows that the generated invariants for method overriding using behavioral subtyping enable to produce over-approximated value. The over-approximated value means invariants are generated to the superclass and subclass as long as there is a new subclass added to the program structure. To ensure any object's method definition can be verified, the BS believes it is safe to cover all data fields for the invariant. However, the over-approximated value from unnecessary invariants can cause overflow, if the exceeded invariant is increased because of the program scalability. We can compare the exceeded invariant with exceeded value in integer number as stated in ISO/IEC 9899:201x and Java Language Specification to see the danger if the program has exceeded invariant. According to ISO/IEC 9899:201x (Jones, 2009), the value resulting from an instance of integer overflow in C or C++ programming language needs not be detected. The undetected mechanism leads to stack overflow problem during program execution. The same mechanism is used for Java which Joy et al. (2000) state clearly in Java Language Specification; "the built-in integer operators do not indicate overflow or underflow in any way" (p.44). This situation does not occur if the invariants are generated using lazy behavioral subtyping because its rules allow method definition has invariant based on its data field and not affected by the inheritance hierarchy structure of the program. In addition, LBS allows method call has invariant that cover both invariants from superclass and subclass method. Therefore, any object that invokes the method call is analyzed statically.

The findings of the present study are regarded as a solution of previous studies. Previous research managed to conduct static analysis on late binding call (Privat et al., 2005) as well as multiple inheritance (Dovland et al., 2009) using Hoare-style logic programming. Privat (2005) suggested type system analysis to verify object-oriented languages together with coloring and binary tree dispatching technique for language compilation. However, Dovland (2009) demonstrated a technique to use lazy behavioral subtyping onto an inference system to verify multiple inheritance. The study reported here differs from previous studies in one important aspect: it applied abstract interpretation to conduct late bound verification. The application of abstract interpretation allows the technical implementation of the proposed framework can be done in automated manner. The automation is hard to implement using Hoare-style logic programming as automated program verifier is still a major problem in software verification (Hoare, 2007). Even though, the proposed abstract formal framework is a heavyweight framework, the outcome of having automation verifier is worthwhile in the future.

Table 5.1: Analysis on Case Studies

Case Studies	Method	BS/ LBS	Methods in Superclass		Methods in Subclass	
			bSalary	tSalary	bSalary	tSalary
Salary System	calc	BS	0..+veVal	-veVal..+veVal	0..+veVal	-veVal..+veVal
		LBS	0..+veVal	-	0..+veVal	-veVal..+veVal
	salary	BS	0..+veVal	-	▨	▨
		LBS	0..+veVal	-	▨	▨
	..calc..	BS	-	-	▨	▨
		LBS	0..+veVal	-veVal..+veVal	▨	▨
			side1	side2	side1	side2
Quadrilaterals System	getData	BS	0..+veVal	0..+veVal	0..+veVal	0..+veVal
		LBS	0..+veVal	-	-	0..+veVal
	setData	BS	0..+veVal	0..+veVal	0..+veVal	0..+veVal
		LBS	0..+veVal	-	-	0..+veVal

BS : Behavioral Subtyping, LBS : Lazy Behavioral Subtyping, +veVal: positive value, -veVal : negative value, .. : range, Grey box : not applicable

5.4 Conclusion

This chapter shows the validation of the invariants generated using proposed equations using two case studies. The result shows that the chosen method, which is lazy behavioral subtyping, produced better value compare to behavioral subtyping. The value is in a range of integer number not a maximum value. The maximum value results stack overflow problem during program execution as stated by ISO/IEC 9899:201x and Java Specification Language. Therefore, the study has verified that there is a solution to verify late method call in object-oriented programs.

Chapter 6
Conclusion

Praise belongs to God.
-Al- Fatihah verse 1

6.1 Summary of the Study

This thesis investigates invariants generation on a program with method overriding using abstract interpretation and lazy behavioral subtyping. The thesis has achieved the objectives to solve two problems of generating invariants in method overriding which are problems of class invariants and late binding call. The investigation starts with the definition of static analysis, the purpose of static analysis for program analysis, and methods available in conducting static analysis. All three methods of static analysis; namely (1) assertion (2) model checking, and (3) abstract interpretation; are analyzed based on their capability to generating invariants automatically, lines of code needed, and whether the methods are concrete or abstract. Abstract interpretation is chosen because the method fulfills all the requirements needed. To justify the importance of method overriding, a literature exploration is done on types of method overriding usage in OOPL. Then, the investigation using a small language that we created called Method Language or methL, is made to analyse the problems during verification using static analysis on method overriding. In the context of method overriding where the focus is on subtyping, there are two main problems that occur during the process of generating class invariants for inheritance and late binding method calls. They are

(1) reverification when new subclasses are added into the inheritance hierarchy

(2) over approximation on abstract method semantics due to unknown method calls. Then, behavioral subtyping and lazy behavioral subtyping are analyzed based on their specification and related works that have applied them. The aim of the analysis is to find the most suitable method to solve the problem related late binding method calls. Therefore, lazy behavioral subtyping is chosen due to its specification on method overriding that can be generated in both overriding and overridden method, and on new subclasses.

Then, a framework using abstract interpretation has been developed by applying lazy behavioral subtyping method for method overriding. Its application on the model solves the problem of over approximation value on method calls. To merge both methods;

(1) abstract interpretation and

(2) lazy behavioral subtyping,

the framework of class invariants generation must be modular. Modularity on class invariants makes the model easily to apply lazy behavioral subtyping for the method calls because every invariant is stored as a module that is combined and manipulated whenever any equation is called. The framework has produced three equations to generate invariants for class, inheritance, and method overriding. The equations have been validated using two case studies namely Salary System and Quadrilaterals System. Then, the result of each variable for each case study has been tabulated to compare with the value produced by the same case studies using invariants generated using behavioral subtyping. The result has achieved to show the comparison between invariant generation using behavioral subtyping and lazy behavioral subtyping.

6.2 Contributions to Body of Knowledge

The main contribution of this thesis is the proposed equations to generate invariant for a program with method overriding. The research has shown that the application of lazy behavioral subtyping using abstract interpretation results to not-overapproximated value for the invariants. Therefore, the program has overcome the possible an integer overflow problem using the method. An analysis between behavioral subtyping and lazy behavioral subtyping has been conducted to find the value produced using these two methods. The results of this analysis show that there is a possibility for the invariant produced using invariants generated from behavioral subtyping to have a maximum value. In contrast, the value produced using invariants generated from lazy behavioral subtyping is a value within specific range.

The study has shown that there are limitations of techniques used in conducting program verification by related works, which indicates objective one has been achieved. The result has been tabulated in Table 2.3. The first major finding showed that Logozzo (Logozzo, 2004) scores all evaluated criteria with behavioral subtyping and observable behavior as techniques of verification. Even though Logozzo (2004) scores all evaluated criteria, behavioral subtyping made over approximated value for class invariant. The second major finding was that lazy behavioral subtyping proposed by Dovland (2009) enables to solve the problem faced by Logozzo (2004), despite the technique is non implemented in non-reverification and automated purposes.

The objective two set out to design an abstract formal framework for verification on method overriding focusing on invariants generation. The most obvious finding to emerge from this objective is that three equations have been produced from the framework. They are equations for class invariant, invariant in inheritance, and invariant in method overriding as in §4.6-4.7. These equations have been validated to check their reliability using behavioral subtyping and lazy behavioral subtyping on two case studies. The result analysis in §5.3 has shown that invariants produced by equations using lazy behavioral subtyping created invariants with value in specific range. The results of this analysis indicate that objective three has been achieved.

6.2.1 Strengths

Probably the main strength of the thesis is the application of lazy behavioral subtyping using abstract interpretation theory. The development of the framework based on the theory for the semantic analysis of programs leads to automatized applications for the program reliability (Cousot, 1978). There are two strengths of the framework.

1. The application solved the problem of over-approximation invariants produced using behavioral subtyping. The over-approximated invariants can give overflow problem to the program which can result to unexpected behavior from the program; e.g. nonstop execution. There are two equations involved to solve the problem. They are equations for inheritance and method overriding.
2. All two equations of invariants are in modular to avoid reverification on new subclasses. The equation is merged from abstract semantics where it comes from concrete semantics that consist of object-oriented program semantics, class semantics, constructor semantics, and method semantics.

6.2.2 Limitations

Due to time constraint, the equations produced have two limitations.

1. The abstract semantics lacks of states of behaviors of the object. If the abstract semantics includes the states of behaviors, then the objects are traced even though they mutate during the execution.
2. A further study on the implementation of the static analysis tool for the equations might provide practical insights of abstract interpretation with lazy behavioral subtyping method on object-oriented programs. Its practicality will produce an automatic static analyzer for program verification, which can be used during software development.

6.3 Future Works

This study focuses only on the abstract formal framework for a program with method overriding. There are three prominent future works can be done, which related to current trend of technology.

1. The generated invariants are for parallel computing programs. Parallel computing is one of the features of cloud computing where computers are executed in parallel to perform one big task. It would be interesting to discover techniques on how to verify objects mutation in a parallel program that has race conditions problem.

2. The application of the abstract formal framework implements on other languages, e.g. scripting programming languages and mobile programming languages. The mobile programming languages are important recently as consumers towards mobile

applications. In addition, programmers use the scripting programming languages used to conduct unit testing in agile methodology development.

3. A future work on full implementation on an automatic verification tool has to be worked out. Even though the study focuses only method overriding, there are other elements that contribute to program with polymorphism; e.g. method overloading, mutated objects, and single dispatch. Consequently, reliable software can be produced in the future if the verification tool enables to verify all features of object-orientation; i.e., encapsulation, inheritance, polymorphism, and abstraction.

References

Abadi, M. & Cardelli, L. (1994). A theory of primitive objects. In Sannella, D. (Ed.). *Programming languages and systems—esop'94*. Lecture notes in computer science, vol. 788. (pp. 1-25). Springer Verlag.

Albert, E., Puebla, G. & Hermenegildo, M. (2005), An abstract interpretation-based approach to mobile code safety. *Electronic Notes on Theoretical Computer Science*, *132(1)*, 113-129.

America, P. (1991), Designing an object-oriented programming language with behavioural subtyping. *Journal of Foundations of Object-Oriented Languages*, 60-90.

Avvenuti, M., Bernardeschi, C. & Francesco, N. D. (2003), Java bytecode verification for secure information flow. *SIGPLAN Notices*, *38(12)*, 20-27.

Bailey, J. & Poulovassilis, A. (1999), Abstract interpretation for termination analysis in functional active databases. *Journal of Intelligence Information System*, *12(2-3)*, 243-273.

Balint, M. & Minea, M. (2011). Automatic inference of model fields and their representation. *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs held on 26 July at the Lancaster, United Kingdom* (pp. 1-6). ACM.

Ball, T., Majumdar, R., Millstein, T. & Rajamani, S. (2001), Automatic predicate abstraction of c programs. *ACM SIGPLAN Notices*, *38(5)*, 203-213.

Banerjee, A. (1997), A modular, polyvariant and type-based closure analysis. *SIGPLAN Notices*, *32(8)*, 1-10.

Barbuti, R., Francesco, N. D., Santone, A. & Vaglini, G. (1999), Abstract interpretation of trace semantics for concurrent calculi. *Journal of Information Processing Letters*, *70(2)*, 69-78.

Barbuti, R., Cataudella, S. & Tesei, L. (2003), Abstract interpretation against races. *Journal of Fundamental Information*, *60(1-4)*, 67-79.

Barbuti, R., Francesco, N. D. & Tesei, L. (2010), An abstract interpretation approach for enhancing the java bytecode verifier. *Journal of Computer*, *53(6)*, 679-700.

Baresi, L. & Pezze, M. (2006), An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, *148*, 89-111.

Barnett, M., Chang, B. Y., DeLine, R., Jacobs, B. & Leino, K. (2006). Boogie: A modular reusable verifier for object-oriented programs. *Proceedings of the Formal Methods for Components and Objects*. (pp. 364-387). Springer.

Barthe, G. & Kunz, C. (2008). Certificate translation in abstract interpretation. *Proceedings of the Programming languages and systems (Theory and practice of software)* (pp. 368-382). Budapest, Hungary, Springer-Verlag.

Bernardeschi, C. & Francesco, N. D. (2002). Combining abstract interpretation and model checking for analysing security properties of java bytecode. In Goos, G. et al (Eds.). *Third international workshop on verification, model checking, and abstract interpretation: Revised papers/vmcai'02*. Lecture notes in computer science, vol. 2294. (pp. 1-15). Venice, Italy: Springer-Verlag.

Bernardeschi, C., Francesco, N. D. & Lettieri, G. (2003), Concrete and abstract semantics to check secure information flow in concurrent programs. *Fundamental Information*, *60(1-4)*, 81-98.

Borghuis, T. & Feijs, L. (2000), A constructive logic for services and information flow in computer networks. *The Computer Journal*, *43(4)*, 274-289.

Bourdoncle, F. (1993), Abstract debugging of higher-order imperative languages. *ACM Sigplan Notices*, *28(6)*, 46-55.

Castagna, G. (1995), Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems 17(3)*, 431-447.

Charlier, B. L., Rossi, S. & Hentenryck, P. V. (2002), Sequence-based abstract interpretation of prolog. *Theory Pract. Log. Program.*, *2(1)*, 25-84.

Cheon, Y. (2010). Functional specification and verification of object-oriented programs. *Departmental Technical Reports (CS)*. http://digitalcommons.utep.edu/cs_techrep/23.

Cheon, Y., Yeep, C. & Vela, M. (2012), The cleanjava language for functional program verification. *International Journal of Software Engineering*, *5(1)*, 47-68

Chin, W. N., David, C., Nguyen, H. H. & Qin, S. (2008), Enhancing modular oo verification with separation logic. *ACM SIGPLAN Notices*, *43(1)*, 87-99.

Clarke, E. (1997). Model checking. In Ramesh, S. et al (Eds.). *Foundations of software technology and theoretical computer science*. Lecture notes in computer science, vol. 1346. (pp. 54-56). Germany: Springer Berlin / Heidelberg.

Codognet, C. & Codognet, P. (1995). A generalized semantics for concurrent constraint languages and their abstract interpretation. In Meyer, M. (Ed.). *Constraint processing, selected papers*. Lecture notes in computer science, vol. 923. (pp. 39-49). Germany: Springer-Verlag.

Cook, W. R. (1989). A denotational semantics of inheritance. PhD thesis. Brown University, Rhode Island, USA.

Cousot, P. & Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the*

International Conference on the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (pp. 238-252). New York, ACM Press, New Yor.

Cousot, P. (1978). Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes. PhD thesis. Université Scientifique et Médicale de Grenoble, France.

Cousot, P. & Cousot, R. (1979). Systematic design of program analysis frameworks. *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (pp. 269-282). Los Angeles, USA, ACM Press, New Yor.

Cousot, P. (1996), Abstract interpretation. *ACM Computing Surveys (CSUR), 28(2),* 324-328.

Cousot, P. (2007). The verification grand challenge and abstract interpretation. In Meyer, B. et al (Eds.). *Verified software: Theories, tools, experiments*. Lecture notes in computer science, vol. 4171. (pp. 227-240). Germany: Springer Verlag.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., & Rival, X. (2007). Combination of abstractions in the ASTRÉE static analyzer. Advances in Computer Science-ASIAN 2006. Secure Software and Related Issues, 272-300.

D'Silva, V., Kroening, D. & Weissenbacher, G. (2008), A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *27(7)*, 1165-1178.

Dahl, O. & Nygaard, K. (1966), Simula, an algol-based simulation language. *ACM Communication, 9*, 671-678.

Distefano, D. & J, M. J. P. (2008), Jstar: Towards practical verification for java. *SIGPLAN Not., 43(10)*, 213-226.

Dovland, J., Johnsen, E. B. & Owe, O. (2005). Verification of concurrent objects with asynchronous method calls. *Proceedings of the IEEE international Conference on Software, Science, Technology and Engineering*. (pp. 141-150). Herzelia, Israel, IEEE.

Dovland, J., Johnsen, E. B., Owe, O. & Steffen, M. (2008), Encapsulating lazy behavioral subtyping. *Journal on Specification Transformation Navigation*, 72-88.

Dovland, J., Johnsen, E., Owe, O. & Steffen, M. (2009). Incremental reasoning for multiple inheritance. In Leuschel, M. et al (Eds.). *Integrated formal methods*. Lecture notes in computer science, vol. 5423. (pp. 215-230). Germany: Springer-Verlag.

Dowson, M. (1997), The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes, 22(2)*, 84.

Engels, G. & Groenewegen, L. (2000). Object-oriented modeling: A roadmap. *Proceedings of the Proceedings of the Conference on the Future of Software Engineering*. (pp. 103-116). ACM.

Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. *Proceedings of the Software Engineering (Workshop on Dynamic Analysis)*. (pp. 24-27). Citeseer.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., et al. (2007), The daikon system for dynamic detection of likely invariants. *Science of Computer Programming, 69(1-3)*, 35-45.

Fages, F., & Soliman, S. (2008). Abstract interpretation and types for systems biology. Theoretical Computer Science, 403(1), 52-70.

Falaschi, M., Olarte, C. & Palamidessi, C. (2009). A framework for abstract interpretation of timed concurrent constraint programs. *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming held on 7 - 9 September at the Coimbra, Portugal* (pp. 207-218). ACM.

Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M. & Rauch, F. (2007). Model checking software at compile time. *Proceedings of the IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*. (pp. 45-56). Shanghai, China.

Fenton, N. E. & Pfleeger, S. L. (1998). *Software metrics: A rigorous and practical approach*. PWS Publishing Co.

Feret, J. (2001). Abstract interpretation-based static analysis of mobile ambients. In Cousot, P. (Ed.). *The 8th international symposium on static analysis*. vol. 2126. (pp. 412-430). Paris, France: Springer-Verlag.

Ferrara, P. (2006). Jail: Firewall analysis of java card by abstract interpretation. *Proceedings of the*. (pp., Citeseer.

Flanagan, C., Freund, S. N. & Tomb, A. (2006). Hybrid types, invariants, and refinements for imperative objects. *Proceedings of the International Workshop on Foundations and Developments of Object-Oriented Languages held on 11-13 January at the South Carolina, USA* (pp. 1-11). ACM.

Floyd, R. W. (1967), Assigning meanings to programs. *Mathematical aspects of computer science, 19(19-32)*, 1.

Gates, B. (2002). Keynote address at windows hardware engineering conference (winhec) 2002. http://research.microsoft.com/en-us/projects/slam/.

Gil, J. Y. & Lenz, K. (2012), Keyword- and default- parameters in java. *Journal of Object Technology, 11(1)*, 1-17.

Halder, R. & Cortesi, A. (2010). Abstract interpretation for sound approximation of database query languages. *Proceedings of the 7th International Conference on Informatics and Systems (INFOS)*. (pp. 1-10).

Halder, R. & Cortesi, A. (2011). Cooperative query answering by abstract interpretation. *Proceedings of the Proceedings of the 37th international conference on Current trends in* *theory and practice of computer science held on 22 - 28 January at the Smokovec, Slovakia* (pp. 284-296). Springer-Verlag.

Hall, A. (1990), Seven myths of formal methods. *Software, IEEE, 7(5)*, 11-19.

Harrold, M. J. (2000). Testing: A roadmap. *Proceedings of the Proceedings of the conference on the future of software engineering*. (pp. 61-72). ACM.

Havelund, K. & Pressburger, T. (2000), Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT), 2(4)*, 366-381.

Henzinger, T., Jhala, R. & Majumdar, R. (2005). The blast software verification system. In Godefroid, P. (Ed.). *Model checking software*. Lecture notes in computer science, vol. 3639. (pp. 25-26). Germany: Springer Verlag.

Hoare, C. A. R. (1969), An axiomatic basis for computer programming. *ACM Communications 12(10)*, 576-580.

Hoare, C. A. R. (1981). The 1980 ACM Turing Award Lecture. *Journal of Communications*, 24(2), 75-83.

Hoare, T. (2007). The Ideal of Program Correctness: Third Computer Journal Lecture. The Computer Journal, 50(3), 254-260.

Igarashi, A., Pierce, B. C. & Wadler, P. (2001), Featherweight java: A minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems 23(3)*, 396-450.

Jackson, D. & Rinard, M. (2000). Software analysis: A roadmap. *Proceedings of the Conference on The Future of Software Engineering held on 4 - 11 June at the Limerick, Ireland* (pp. 133-145). ACM.

Jaoua, A. & Elloumi, S. (2002), Galois connection, formal concepts and galois lattice in real relations: Application in a real classifier. *Journal of Systems and Software, 60(2)*, 149-163.

Johnsen, E., & Owe, O. (2007). An Asynchronous Communication Model for Distributed Concurrent Objects. Software & Systems Modeling, 6(1), 39-58.

Jones, L. (2009). *WG14 N1401 Committee Draft ISO/IEC 9899: 201x*, International Standards Organization

Joy, B., Steele, G., Gosling, J. & Bracha, G. (2000). *The java language specification*. Addison-Wesley.

Lamsweerde, A. v. (2000). Formal specification: A roadmap. *Proceedings of the Proceedings of the Conference on The Future of Software Engineering held on 4 - 11 June at the Limerick, Ireland* (pp. 147-159). ACM.

Leavens, G. (2006). Jml’s rich, inherited specifications for behavioral subtypes. In Liu, Z. et al (Eds.). *Formal methods and software engineering: 8th international conference on formal engineering methods*. Lecture notes in computer science, vol. 4260. (pp. 2-34). Macao, China: Springer Verlag.

Leino, K. R. M. & Müller, P. (2004). Object invariants in dynamic contexts. *Ecoop 2004–object-oriented programming*. Lncs, vol. 3086. (pp. 95-108).

Leino, K. R. M. & Müller, P. (2005). Modular verification of static class invariants. *Formal methods*. Lecture notes in computer science, vol. 3582. (pp. 26-42). Germany: Springer Verlag.

Leino, K. R. M., & Schulte, W. (2007). Using history invariants to verify observers. In R. De Nicola (Ed.). *Programming Languages and Systems.* Lecture notes in computer science, vol. 4421. (pp. 80-94). Germany: Springer-Verlag.

Liskov, B. H. & Wing, J. M. (1994), A behavioral notion of subtyping. *ACM Transaction on Programming languages and Systems, 16(6),* 1811-1841.

Liskov, B. H. & Wing, J. M. (2001). Behavioural subtyping using invariants and constraints. In Bowman, H. et al (Eds.). *Formal methods for distributed processing: A survey of object-oriented approaches*. (pp. 254-280). UK: Cambridge University Press.

Little, J., & Moler, C. (1994-2013). Static Analysis with Polyspace Product, from http://www.mathworks.com/products/polyspace/

Loding, H. & Peleska, J. (2008), Symbolic and abstract interpretation for c/c++ programs. *Electron. Notes Theor. Comput. Sci., 217,* 113-131.

Logozzo, F. (2004). Modular static analysis of object-oriented languages. PhD thesis. Ecole Polytecnique, France.

Logozzo, F. & Cortesi, A. (2004). Semantic class hierarchies by abstract interpretation. *Research Report CS-2004-7,* Department of Computer Science, University Ca' Foscari of Venice, Italy. http://www.dsi.unive.it/~cortesi/paperi/CS_2004_7.pdf.

Logozzo, F. (2005), Class invariants as abstract interpretation of trace semantics. *computer languages, systems and structures*.

Logozzo, F. (2007). Cibai: An abstract interpretation based static analyser for modular analysis and verification of java classes. In Cook, B. et al (Eds.). *Verification, model checking and abstract interpretation*. Lecture notes in computer science, vol. 4349. (pp. 283-298). Germany: Springer Verlag.

Marriott, K., Søndergaard, H. & Jones, N. D. (1994), Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS), 16(3),* 607-648.

Martin, J. & Odell, J. J. (1998). *Object-oriented methods: A foundation*. vol. 2nd Edition. USA: Prentice Hall.

Mellish, C. (1986). Abstract interpretation of prolog programs. In Shapiro, E. (Ed.). *Third international conference on logic programming*. Lecture notes in computer science, vol. 225. (pp. 463-474). London: Springer Verlag.

Meyer, B. (1997). *Object-oriented software construction*. New Jersey: Prentice Hall PTR.

Michiel, M. d., Bonenfant, A., Cass, H. & Sainrat, P. (2008). Static loop bound analysis of c programs based on flow analysis and abstract interpretation. *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications held on 25-27 August, 2008 at the Kaohisung, Taiwan* (pp. 161-166). IEEE Computer Society.

Mihancea, P. F. & Marinescu, R. (2009). Discovering comprehension pitfalls in class hierarchies. *Proceedings of the 13th European Conference on Software Maintenance and Reengineering, CSMR '09 held on 24-27 March 2009 at the Kaiserslautern, Germany* (pp. 7-16). IEEE Computer Society.

Mills, H. D., Dyer, M. & Linger, R. C. (1987). Cleanroom software engineering. *The Harlan D.Mills Collection*. http://trace.tennessee.edu/utk_harlan/18.

Monin, J.-F. (2003). *Understanding formal methods*. London: Springer-Verlag.

Müller, P. (2002). *Modular specification and verification of object-oriented programs*. Lecture notes in computer science. vol. 2262 Springer-Verlag.

Myers, G. J. (2008). *The art of software testing*. India: Wiley.

NASA. (2012). "Robust software engineering." 2012, from http://ti.arc.nasa.gov/tech/rse/vandv/jpf/.

Nielson, F., Hansen, R. R. & Nielson, H. R. (2003), Abstract interpretation of mobile ambients. *Sci. Comput. Program., 47(2-3)*, 145-175.

Nielson, F., Nielson, H. R. & Hankin, C. (2005). *Principles of program analysis*. Germany: Springer-Verlag.

Nunes, I. (2004), Method redefinition--ensuring alternative behaviors. *Information Processing Letters, 92(6)*, 279-285.

OMG. (2001). Omg unified modelling language specification version 1.4.

Parkinson, M. J. (2005). Local reasoning for java. PhD thesis. University of Cambridge.

Parkinson, M. J. (2007). Class invariants: The end of the road? *Proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*. (pp. 9-10). Berlin, Germany, Citeseer.

Parkinson, M. J. & Bierman, G. M. (2008), Separation logic, abstraction and inheritance. *ACM SIGPLAN Notices*, *43(1)*, 75-86.

Pollet, I. & Charlier, B. L. (2005), Towards a complete static analyser for java: An abstract interpretation framework and its implementation. *Electron. Notes Theor. Comput. Sci.*, *131*, 85-98.

Privat, J. & Ducournau, R. (2005), Link-time static analysis for efficient separate compilation of object-oriented languages. *SIGSOFT Softw. Eng. Notes*, *31(1)*, 20-27.

Reeves, G. (2004). "The mars rover spirit flash anomaly." from http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/39361/1/04-3354.pdf

Rodriguez-Carbonell, E., & Kapur, D. (2007). Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1), 54-75.

Schnoebelen, P. (2002). The complexity of temporal logic model checking. In P. Balbiani et al (Eds.). *Advances in modal logic*. vol. 4. (pp. 393-436).

Siu, M. K. (2001). Why is it difficult to teach abstract algebra. In Vincent, J. et al (Eds.). *The future of the teaching and learning of algebra*. (pp. 541-547). Hong Kong: Hong Kong University.

Skalka, C., Smith, S. & Horn, D. V. (2005), A type and effect system for flexible abstract interpretation of java. *Electron. Notes Theor. Comput. Sci.*, *131*, 111-124.

Smans, J., Jacobs, B., Piessens, F. & Schulte, W. (2010), Automatic verification of java programs with dynamic frames. *Formal Aspects of Computing*, *22(3)*, 423-457.

Softworks, A. (2012). "C # tools." 2012, from http://www.csharptools.com/.

Spoto, F. (2010). The nullness analyser of julia. In Clarke, E. M. (Ed.). *Logic for programming, artificial intelligence, and reasoning*. Lecture notes in computer science, vol. 6355. (pp. 405-424). Springer Verlag.

Stroustrup, B. (1987). What is object-oriented programming? *Proceedings of the ECOOP'87 European Conference on Object-Oriented Programming*. (pp. 51-70). Paris, Springer Verlag.

Sun, J. & Dong, J. S. (2005). Extracting fsms from object-z specifications with history invariants. *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*. (pp. 96-105). IEEE.

Taivalsaari, A. (1996), On the notion of inheritance. *ACM Computing Surveys*, *28(3)*, 438-479.

Tarski, A. (1955), A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics, 5(2)*, 285-309.

Toman, D. (1997). Constraint databases and program analysis using abstract interpretation. *Second international workshop on constraint database systems, constraint databases and their applications*. Lecture notes in computer science, vol. 1191. (pp. 246-262). Germany: Springer-Verlag.

Van Den Berg, J. & Jacobs, B. (2001). The loop compiler for java and jml. *Tools and algorithms for the construction and analysis of systems*. Lecture notes in computer science, vol. 2031. (pp. 299-312). Springer Verlag.

Webber, A. B. (2001). What is a class invariant? *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*. (pp. 86-89). Utah, USA, ACM.

Wegner, P. & Zdonik, S. B. (1988). Inheritance as an incremental modification mechanism or what like is and isn't like. *Proceedings of the ECOOP'88 European Conference on Object-Oriented Programming*. (pp. 55-77). Oslo, Norway, Springer Verlag.

Xing, J., Li, M. & Li, Z. (2010). Automated program verification using generation of invariants. *Proceedings of the the International Conference on Quality Software*. (pp. 300-305). China, IEEE.