2008

Analysis and Transformation of Post-Conditions

Abhinav Vasikarla
Wright State University

ANALYSIS AND TRANSFORMATION OF POST-CONDITIONS

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

By

Abhinav Vasikarla
Bachelor of Technology in Computer Science and Engineering,
Kakatiya Institute of Technology and Science, INDIA 2006

2008
Wright State University

WRIGHT STATE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

August 15, 2008

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Abhinav Vasikarla ENTITLED Analysis and Transformation of post-conditions BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Computer Science.

Thomas C. Hartrum, Ph.D.
Thesis Co-Director

Mateen M. Rizki, Ph.D.
Thesis Co-Director

Thomas Sudkamp, Ph.D.
Department Chair

Committee on
Final Examination

Thomas C. Hartrum, Ph.D.

Mateen M. Rizki, Ph.D.

Krishnaprasad Thirunarayan, Ph.D.

Joseph F. Thomas, Jr., Ph.D.
Dean, School of Graduate Studies

ABSTRACT

Software engineers have been trying for years to develop a software synthesis system that can transform a formal specification model to a design model from which executable code can be generated. AFIT wide spectrum object modeling environment (AWESOME) is one result of their research. AWESOME presents a formal model as an abstract syntax tree. This model consists mainly of object class specifications. The methods in these classes are specified using pre and post-conditions.

The intent of this thesis is to support the transformation of post-conditions to code statements. A post-condition is first categorized as dependent or independent relative to other post-conditions. Post-conditions are further divided into actions and constraints. Actions can be converted to executable statements. Constraints can be converted to pre-conditions using weakest pre-condition analysis. Functions have been designed to categorize the post-conditions. Transforms have been designed to simplify the post-conditions and to determine the weakest pre-condition and add it to the method. The result is a design model from which executable code can be generated.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

It is my pleasure to thank many people who made this thesis possible. It is difficult to overstate my gratitude to my thesis advisor, Dr. Thomas C Hartrum. This thesis would not have been possible without his kind support and remarkable patience. I am very thankful to my committee members Dr. Mateen M.Rizki and Dr. Krishnaprasad Thirunarayan for their invaluable feedback on this work. I remain indebted to my parents who blessed me with the courage and strength to achieve my goals. I would also like to thank all my friends for their support.

1. INTRODUCTION

The automatic generation of code from a formal specification is a topic of interest in the field of software engineering. The general idea involved is to define a problem in a formal specification language and then apply a series of transforms to generate executable code. This requires all the requirements to be formally specified. The generation of executable code involves a series of steps where the application engineer develops a formal specification model as an abstract syntax tree (AST). Transforms are applied to change this AST to a design model. More transforms are applied to the design model to change it to executable code. The methods in the classes are specified using pre-conditions and post-conditions. Transforms are to be designed to convert the post-conditions into statements in the AST from which executable code can be generated. This thesis deals with different post-conditions to be transformed to executable code. A post-condition is a Boolean expression that must be true after the method is executed. There is a single expression for the post-condition, but this expression may be a conjunction of one or more simpler expressions. Post- conditions in this thesis are categorized as

- Dependent and Independent Post-Conditions.
- Constraints and Actions
- Simplified and Non-Simplified Post-Conditions.

INTRODUCTION TO AWESOME

AWSOME stands for the AFIT Wide Spectrum Object Modeling Environment [1]. It is the result of AFIT's efforts in designing a transformation system that uses a formal specification model to generate Java code. The AFIT transformation system is object-oriented, based on the Object Modeling Technique (OMT) and Unified Modeling Language (UML). This tool is based on a meta-model represented using abstract syntax trees (AST). The formal language AFIT Wide-spectrum Language (AWL) is used to represent the formal specifications. The formal AWL model is parsed into the Abstract Syntax Tree (AST). Then a series of transforms are performed on the AST to change it from a specification AST to a design AST from which executable code can be generated. The goal of AWSOME is to transform any formally correct representation of an object model into executable code. Currently the system has many transforms developed to change the AST from the specification model to the design model.

PROBLEM STATEMENT

The primary goal of this thesis is to examine all of the post-conditions in a method and get them into a format that can be directly converted to executable statements, or by changing them to appropriate pre-conditions. We first analyze the post-condition and determine each post-condition to be dependent or independent and their order of execution if they are dependent. We also differentiate between a constraint and an action. A *constraint* shows the relationship between two variables at the same time or between different values of the same variable at different times and actions are those which are not constraints .There are different ways to handle existing transforms which require post-conditions to be in a certain format that may require logical simplification of the original post-condition .This thesis will address some of these specifications and other simplifying transforms. After resolving the dependent post-conditions they are separated to actions and constraints. Ultimately weakest-precondition analysis is applied to constraints to replace them with their weakest-pre-condition.

DEPENDENT AND INDEPENDENT POST-CONDITIONS

Detection and combination of the dependent and independent post-conditions is our first concern. If we have an expression X’=X+1 and Y’=X’+1 then these post-conditions are said to be dependent as the value of Y’ is dependent on the value of X’ and Y’ cannot be determined unless X’ is determined, so their order of execution is of primary concern. The expressions X’=X+1 and Y’=Y+1 are said to be independent as neither the value of X or Y is dependent on each other and hence they can be executed in any order. This thesis intends to detect independent and dependent post-conditions and focuses on their order of execution.

CONSTRAINTS AND ACTIONS

An action is post-condition that can uniquely be made true by executing program statements.

EXAMPLE:

X’=10

This is an action as this can be made true by assigning 10 to X in the program. However for X'<10, there are many ways to make it true, hence it is a constraint and not an action.

SIMPLIFICATION

This thesis will address some of the NOT simplifications. The entire NOT operator post-conditions evaluate to a Boolean value. Some of the transforms that are considered include

NOT (A AND B) is equivalent to NOT A OR NOT B

NOT (A OR B) is equivalent to NOT A AND NOT B

WEAKEST-PRECONDITION

Weakest pre-condition is one of the strategies to verify program correctness. Arbitrarily there can be many pre-conditions which can be true for a given program but there is only one pre-condition which has the maximum set of states for which the execution of the program results in satisfying the post-condition for that pre-condition. Such a pre-condition is determined to be the weakest pre-condition.

EXAMPLE

{x > 5} z := x {z > 0} where x > 5 is the pre-condition

{x > 3} z := x {z > 0} the pre-condition is weaker

{x > 0} z := x {z > 0} has the weakest pre-condition

Hence x>0 is the weakest pre-condition.

.

LIMITATIONS OF THE THESIS

This thesis focuses on differentiating the post-conditions as dependent and independent post-conditions and then further dividing them into actions or constraints. The weakest pre-condition analysis is performed on the constraints so that the post-conditions are converted into pre-conditions. Determining when and where to apply these transforms requires human interaction. It is assumed that the necessary transforms are already applied to the input AWL file to get it into the form required to apply the transforms designed in this thesis. This research work provides computer help in the form of transforms and utility function. The goal is not to develop a fully automated system.

APPROACH

The approach used in the thesis is an object oriented approach. This thesis follows a step by step approach. The initial phase is the requirement analysis phase in which the existing AWSOME tool is studied and the problem statement is carefully looked at in order to identify the transforms that are to be designed. The problem to be solved in this thesis is discussed here and the limitations of the thesis are also stated. In the next phase we analyze the existing transforms and figure out what needs to be done to meet our goal. The transforms that are identified in the analysis phase are designed. Design of each transform is explained in detail in the design phase. The transforms designed are applied to different input specifications and the output model is analyzed for correctness.

DOCUMENT ORGANIZATION

This thesis document starts with an **Introduction** chapter which gives a brief description of the AFIT Wide Spectrum Object Modeling Environment (AWSOME). It presents the problem statement and limitations of the thesis. The approach involved to solve is outlined here and also an idea of how the thesis document is organized. In the **Background** chapter AWESOME is discussed in detail for how the transformation is done from the specification model to a design model. This chapter makes understand about AWESOME and the ongoing research work in this field. The next chapter **Requirement analysis** covers the problem statement and the necessary steps needed to resolve the problem statement are discussed. The transforms to be designed are identified in this chapter. The identified transforms are designed in the Design chapter. Each transform class is explained in detail in this chapter. The next chapter is **Implementation and Testing** where the transforms designed are implemented and are tested for validation. This chapter presents the test cases used to test the transforms and the output obtained for each transform. The final chapter **Conclusion and Future Work** reviews the work done and the limitations of the implemented transforms, along with the future work that can be done to improve the design of the transforms.

2. BACKGROUND

This chapter covers some of the background details regarding AWSOME, AWL and weakest pre-condition. These details should be sufficient enough for the reader to understand the rest of the document.

INTRODUCTION TO AWESOME and AWL

AWESOME is a formal-based software synthesis system initially developed at AFIT[1] . It is an object oriented software synthesis system. The requirements of AWESOME are to develop a formal specification of a problem; then some correctness preserving transforms are applied to it to change it to a design model; then executable code is generated from this design. It is a semi-automated system in the sense that the software engineer makes the decisions and the system makes necessary changes to the model [8]. AWESOME is a tool based on a meta-model represented using the abstract syntax tree. An abstract syntax tree is a compiler's internal representation of a computer program while it is being optimized and from which code generation is performed [6]. It is the data structure representing the parsed input. AWESOME is based on a single abstract syntax tree. It is a serial process with the input as the formal specification and the target language program as the output. AWESOME is a platform for the development and manipulation of object-oriented models. The AWSOME tool transforms are built in Java which makes it easy to extend with GUI based tools making use of extensive Java class libraries. AWESOME also supports all models from the requirements phase until the code phase [6].

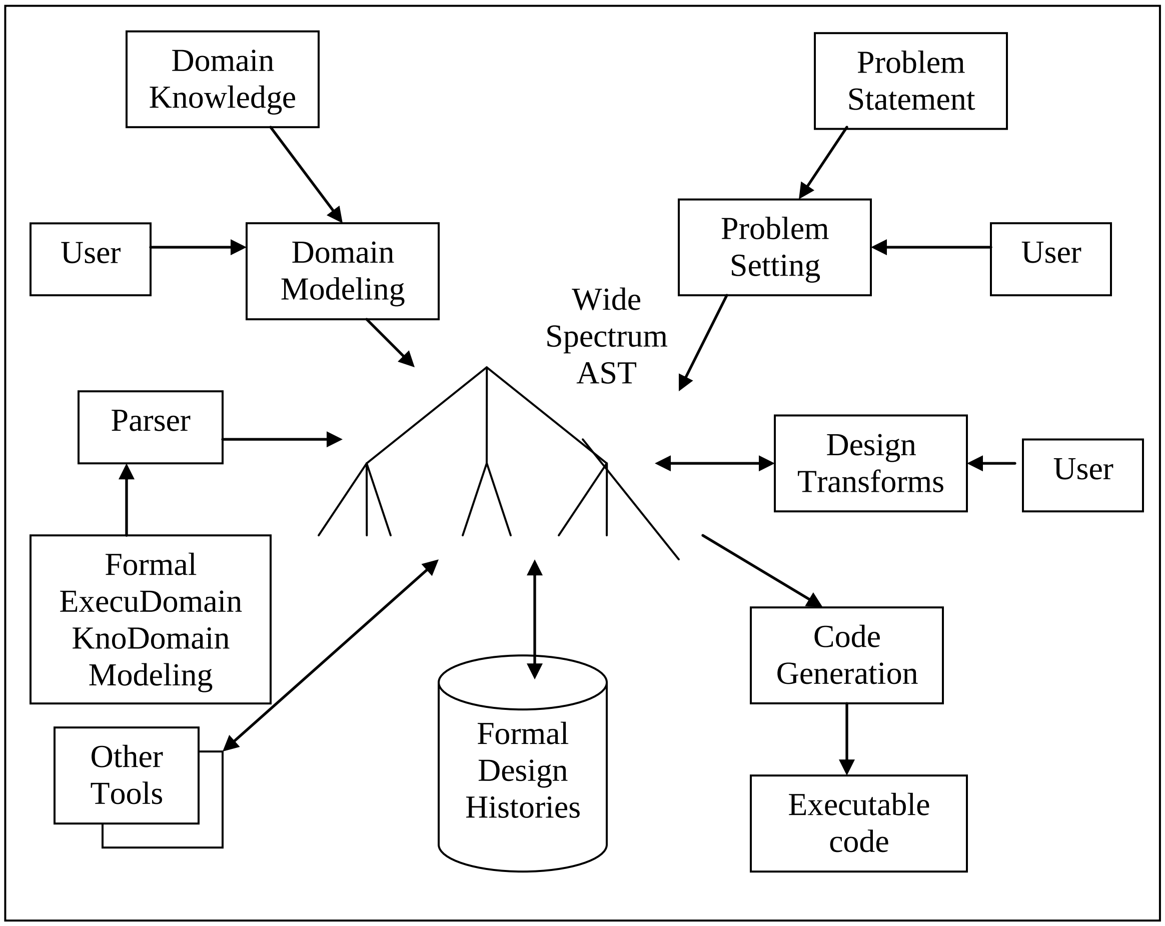


Figure 2.1 AWESOME Transformation System.

AWL

AFIT developed the AFIT Wide-spectrum Language (AWL) to support the forward synthesis of software from formal specifications as well as reverse engineering of existing programs. The language is defined by its surface syntax and by its corresponding abstract syntax tree (AST) meta-model [2]. It is an easy user interface to AWESOME. The specification defines what function is provided without saying how it will be provided. It was designed to be a simple input specification language. The major objective of AWL was the translation to and from other imperative languages like

COBOL and C. AWL includes the major concepts of the object oriented paradigm like classes, inheritance and polymorphism. It also supports pre and post-conditions and class invariants expressed using first order predicate logic and set theory. The syntax for the AWL parser was developed using JavaCC, a Java-based compiler-compiler [6]. AWL has the ability to save any modified model in AWESOME in a parsable format.

AWESOME META-MODEL

A parse tree is produced when an input specification is parsed. The parser initially checks for the syntax errors in the input AWL file and then parses it to an AST by eliminating irrelevant information. Transforms are applied to this AST to generate the executable code.

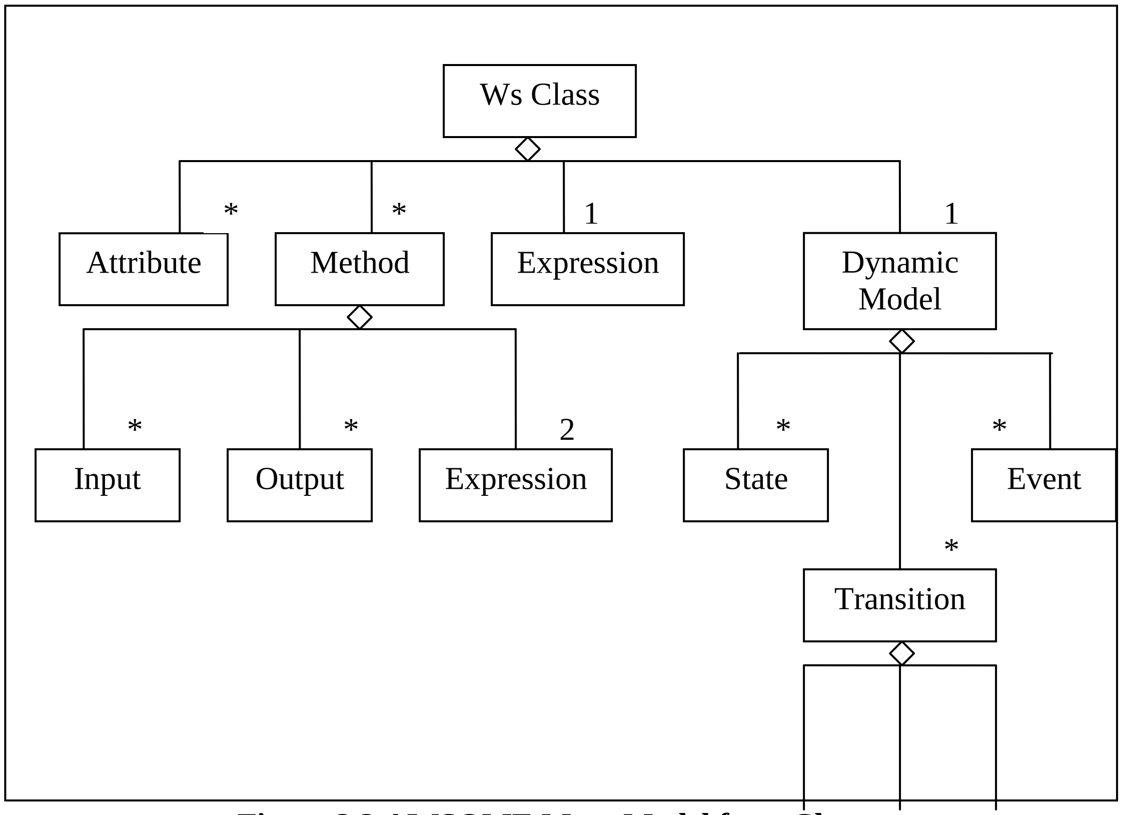


Figure 2.2 AWSOME Meta-Model for a Class

AWESOME is a transformation system built using Java and is based on a single Abstract Syntax Tree (AST). This AST allows various tools to access the model in any stage shown in Figure 2.1 [1]. The AWESOME model consists of a set of object classes. Each object class has a structural model (contains a set of attributes), a functional model (contains a set of methods) and a dynamic model (set of states and transitions) as shown in figure 2.2 [7].

METHODS

The functional model of a class consists of a set of methods. A method is defined by the following: name, a set of input parameters, a set of output parameters, a precondition and a post condition. A pre-condition defines constraints on inputs or data the program must handle. A Post-condition defines what the program must achieve. It is a constraint on the final state of the program.

WEAKEST PRE-CONDITION [16]

A program's structure is defined by its syntax and its elements referred to as *s-units*. A correct program must conform to a specification

- {P}S {Q} - Referred to as a *Hoare triple*.
- This says "if the initial program state satisfies pre-condition **P** then a correct program **S** will terminate in a state satisfying post-condition **Q**."
- Can be applied to individual s-units

There are verification methods using inference rules of Hoare Logic which are used to verify that the program meets the required specifications. One of the strategies for program verification is the "weakest precondition predicate transformer" (*wp*) developed by E. W. Dijkstra .This approach is based on Hoare Logic.

Let us assume we want to verify a program where we know the post-condition but *not* the precondition:

{?}S {Q}

In general, there could be arbitrarily many pre-conditions *P* which are valid for the program *S* and a post-condition *Q*. However, there is precisely one precondition describing the *maximal* set of possible initial states such that the execution of *S* leads to a state satisfying *Q*. This *P* is called the *weakest precondition*. (A condition *P* is *weaker* than Q if Q => *P*.)

EXAMPLE

Given:

A program *S*: $y := x * x$, post-condition *Q*: $y >= 4$

Find:

The weakest pre-condition *Q*.

Solution:

P: (x<=2) or (x >=2)

The precondition $x >= 2$ would also guarantee that *P* is valid after execution. Even stronger preconditions like $x >=3$, $x = 3$, etc. would be valid preconditions as well.

However, the weakest precondition is *P: (x<=2) or (x >=2)* because it has the maximal set. The weakest precondition of a program *S* and a post-condition *Q* is denoted by ***wp(S,Q)*** and is a predicate which describes the set of all initial states that will guarantee termination of *S* in a state satisfying *Q*. This can also be expressed as a Hoare Triple:

{*wp (S, Q)* } *S {Q}*

So if one wants to verify {P*} S {Q}* using *wp*, one can first determine *wp*(*S*,Q) and then prove

P => ***wp*(*S, Q*)**

(Note:We are not trying to verify the program, but want to replace the post-condition with the weakest pre-condition. We assume the program is correct.) For a fixed statement *S*, the function *wp* can be viewed as a function taking only a predicate (the post-condition) and returning another predicate (the weakest precondition). Therefore *wp* is also called **a *predicate transformer.*** The **weakest pre-condition transform must be defined for each statement S in the language.** We will have a brief study of the available program statements for the weakest pre-condition in the next section.

GUARDED COMMAND LANGUAGE

The following weakest pre-condition definitions are based on the definitions of the guarded command language. The syntax for each statement type is introduced as needed.

The available statements for the weakest pre-condition are:

- Assignment statement
- Sequence of statements
- If else statements

- If statements
- Procedural calls
- Loop statements

ASSIGNMENT EXPRESSION

An assignment of an expression to a variable X can be denoted by X: =e. When an assignment expression is considered, to find the weakest pre-condition for that expression we follow the textual substitution.

i.e. **wp(x:= e) ≡ Q_e^x** - Textual substitution where

Q_e^x refers to the predicate with all the free occurrences of x simultaneously replaced by the expression.

EXAMPLE

We have a post-condition Q: x + x * y = c + y and the assignment statement **s** x: = y + c. We need to replace the value of x in the post-condition with the x in the expression (y+c).

wp(x := y + c, x + x * y = c + y) ≡

(y+c) + (y+c) * y = c + y ≡

y*(y+c) = 0

SEQUENTIAL STATEMENTS

The instruction sequence consists of a series of statements which should be executed in a sequence order. If we have a sequence of instructions S1; S2; S3 and the post-condition Q then the weakest-pre-condition for the sequence of the instructions is

$$\mathrm{wp}(S_1 ; S_2 ;S3,Q) \equiv \mathrm{wp}(S_1;\mathrm{wp}(S2,\mathrm{wp}(S3, Q)))$$

The weakest pre-condition for the statement S3 is resolved, which then becomes the post-condition for S2. Then the weakest-pre-condition for S2 is calculated which becomes the post-condition for instruction S1. Finally the weakest-precondition for S1 is resolved.

EXAMPLE

Program cube (n)

```
    P: n > 5
    x: = n * n
    x: = x * n
    Q: x = n * n * n
    return(x)
```

Solution:

Program cube (n)

```
    P: n > 5
    wp2 ≡  n * n * n = n * n * n
    x: = n * n
    wp1 ≡ x * n = n * n * n
    x: = x * n
    Q: x = n * n * n
    return(x)
```

The final weakest pre-condition i**s** n * n * n = n * n * n

IF ELSE STATEMENTS

The if else statements have a condition and depending on the condition the instructions will be executed. The syntax of an if else statements follows as

Syntax: if B S_1 else S_2

where B: condition and S_1,S_2 : instructions

The weakest-pre-condition for the if else statements can be resolved as

$$wp\ (if\ B\ S_1\ else\ S_2,\ Q) \equiv B \Rightarrow wp\ (S_1,\ Q)\ AND\ (NOT\ B) \Rightarrow wp\ (S_2,\ Q)$$

The process of resolving the sub weakest pre-conditions wp (S_1, Q) and wp (S_2, Q) is by using the assignment expression and if it consists of a sequence of instructions we can use the sequence instruction to find the weakest pre-condition.

If the if else statement doesn't contain the else part in some conditions, i.e. it is an If condition without an else, its weakest-precondition can be found as

wp (if B S_1, Q) ≡ B=>wp (S_1, Q) AND (NOT B) => Q

EXAMPLE

```
if x == 0
 x: = x + 1;
else
    x: = x/x;
Q: x ≥ 0
```

Solution:

wp (if x == 0 x: = x + 1; else x: = x/x, x >= 0) ≡

x = 0 => wp(x: = x + 1, x >= 0) AND NOT (x = 0) => wp(x: = x /x, x >= 0) ≡

[x=0 => x+1>=0] AND [NOT x=0 => x/x >=0]

If we consider the same example for the If statement without an else the solution would be

```
if x == 0
 x: = x + 1;
Q: x ≥ 0
```

Solution:

wp (if x==0 x: =x + 1, x>=0) ≡

x=0 => wp(x: = x+1, x>=0) AND NOT (x=0) => (x>=0) ≡

[x=0 => x+1 >=0] AND[NOT (x=0)=> (x>=0)]

PROCEDURAL CALLS

The weakest pre-condition for procedural calls depends on its input and output parameters. The definition for the procedural call is

Proc procname (formal parameter)

S

The set of actual parameters is the union of the input and output parameters. The subset I' are the inputs and the subset I" are the output. The weakest pre-condition computation for the procedural call is

wp (procname (actual parameters), Q) ≡ wp (I':=i; S; o: =I', Q)

All the input parameters I are copied to the local variable subset I' and then the sequence of statements S is executed. The result of the computation I" is copied to the output parameters.

EXAMPLE

The called function swap as the statements

```
Proc swap(int a,int b)
{
int temp;
temp=a;
a=b;
b=temp;
}
```

In the program the calling function with the actual parameters

swap(x,y);

To calculate the weakest pre-condition for this procedure call we need to assign the formal parameter to the actual parameters first

wp(swap(x,y)) ≡

```
a=x;
b=y;
temp=a;
a=b;
b=temp;
x=a;
y=b;
```

Now after this expansion of the function we follow the sequence of statements to calculate the weakest-precondition for the procedure.

LOOP INVARIANTS

Loops are difficult to resolve for correctness because they represent an unknown number of paths. Partial correctness can be proven using loop invariants. Loop invariant is a constraint on variables that is always true at loop initialization, loop iteration and loop termination. If we are given an appropriate loop invariant we can prove the loop to be correct.

The concept of loops and procedure calls is beyond the scope of this thesis and hence will not be addressed.

SUMMARY

This chapter provides the necessary information for the readers to get familiar with AWESOME and AWL. We have also discussed about methods, pre and post-conditions and weakest pre-condition which are useful for the reader to understand the rest of the document.

3. REQUIREMENTS ANALYSIS

This chapter presents a detailed description of the different post conditions and some possible ways to categorize them for a better processing. The term *method* refers to a piece of code that is exclusively associated either with a class (called class methods, static methods, or factory methods) or with an object (called instance methods). A method in AWESOME contains a name, a set of input parameters, local variables, a set of output parameters, pre-conditions, and post-conditions. This thesis aims at supporting the transforms that are applied to the post-conditions, which generate a sequence of statements to replace a post-condition. A post-condition is a Boolean expression that must be true after the method is executed. We discussed earlier that post-conditions could be categorized in three ways. We will further study them in this chapter. The first overall requirement is to determine the category for each post-condition. The second requirement is to process them once categorized.

INDEPENDENT POST-CONDITIONS

Two post-conditions are said to be independent if they can be resolved in any order and the outcome is always true. The order is not of primary concern.

Example: P1: X’ = X+1 P2: Y’=Y+1

The outcome of X’ and Y’ remains the same if the post-conditions are executed in any order.

<u>DEPENDENT POST-CONDITIONS</u>

Two post-conditions are dependent if the order in which the post-conditions are resolved has a true effect on the outcome, or post-conditions are said to be dependent if no order exists for the solution.

Dependent-on:

P1 is dependent on P2 if P2 must be resolved first.

We will now examine the different cases of dependency

Case 1:

P1: X'=X+1 P2: Y'=X+1

Post-condition P1 is dependent on post-condition P2 as P2 uses the old value of X which P1 will change. This dependency is of the form **P1 dependent on P2**.

Case 2:

P1: X'=X+1   **P2**: Y'=X'+1

P2 is dependent on P1 as P2 needs the value of X' which it can get only after P1 is resolved. This dependency is of the form **P2 dependent on P1**.

Case 3:

P1: X'=Y+1 P2: Y'=X+1

This is a case of dependency in which P1 and P2 are dependent on each other. Neither of them can be resolved first. We need to use a temporary variable, an approach similar to swap logic.

Considering the different cases of post-conditions mentioned above, we analyze them based on the primed and unprimed variables that the post-conditions have and then resolve them to be dependent if they satisfy any one of the following.

CONDITION 1: Both the post-conditions contain the same primed values.

CONDITION 2: A variable is primed in one post-condition and unprimed in the other.

Considering the above two conditions we can underlay a more formal definition for dependency.

DEPENDENCY

Two post-conditions exhibit dependency if they both contain the same primed value or a primed value in one post-condition and an unprimed value in the other.

ACTIONS AND CONSTRAINTS

The task of resolving the dependent and independent post-conditions is done in the section above; after these dependencies are resolved, we get the dependent post-conditions and independent post-conditions separated. Our task is now to determine whether these post-conditions are actions or constraint.

ACTION: An action is a post-condition that can uniquely be made true by executing program statements. Every method needs at least one action in the post-condition.

CONSTRAINT: A constraint is a post-condition that cannot be uniquely made true by executing program statements. A post-condition which is not an action is a constraint.

Examples:

- X=5 is an action as this can be made true by assigning 5 to X in the program
- X<10 and X>8 is an action as X uniquely has a value 9.
- X<20 is a constraint as X can have any value less than 20 which cannot be uniquely determined.
- X<10 and X>7 is a constraint as X can have 8 or 9 as its value which cannot be uniquely determined.

These are some of the basic cited examples to illustrate the action and constraint behavior. There are several other Boolean operators, some of which can be classified as a constraint or an action and some operators with special conditions which need to be simplified for it to be an action or a constraint. We will now classify the different Boolean operators on the basis of a constraint or an action.

CLASSIFICATION OF BOOLEAN OPERATORS

The Boolean operators have been tabulated in Table 1 and they provide information about the post-conditions that are constraints and those that are actions.

Table 1: Classification of Boolean Operators

OPERATOR	CONSTRAINT	ACTION
LESS THAN (<)	YES	NO
LESS THAN OR EQUALS TO (<=)	YES	NO
GREATER THAN (>)	YES	NO
GREATER THAN OR EQUALS TO (>=)	YES	NO
EQUALS TO (=)	Equals Rule	Equals Rule
NOT EQUAL TO (/=)	YES	NO
NOT	SIMPLIFY	SIMPLIFY
IN	SIMPLIFY	YES
IMPLIES	SIMPLIFY	YES
FORALL	NO	YES
EXISTS	NO	YES
SUBSET	YES	NO
SUBSETEQ	YES	NO
AND	SIMPLIFY	SIMPLIFY
OR	YES	SIMPLIFY

BOOLEAN OPERATORS

Boolean operators define the relationships between words or groups of words. Table 1 and the following analyze the Boolean operators available in AWESOME and AWL.

COMPARISION OPERATORS

Comparison operators, as their name implies, allow you to compare two values. A post-condition may contain one or more comparison operator (in combination with other Boolean operators.). The comparison operators **(<, <=, >=, >)** will be treated as a constraint. The operator *equals* (=) has an EQUALS rule to be applied and then the resulting output determines it to be a constraint or an action. Operator **not equals** (/=) can be resolved as constraint.

EQUALS RULE

If the resolved post-condition has an *equals* operator and is independent and has all the variables in the post-condition ticked then the post-condition with the *equals* operator is termed to be a constraint. If it violates any of these rules then it is an action.

Example: Let us consider the post-conditions obtained after our dependent and independent analysis

b’=bb and a’=b’

a’=cc

a’=b’

z’=z+1

The first post-condition is dependent and all the other post-conditions are resolved to be independent. All of the three independent post-conditions have an *equals* operator, but all the variables are ticked in only the third post-condition, hence it is resolved as a constraint and all the other post-conditions are resolved as actions.

Hence, after our analysis of the EQUALS rule the post-conditions can be resolved as:

b'=bb and a'=b'- (***Dependent and Action***)

a'=cc - (***Independent and Action***)

a'=b' - (***Independent and Constraint (Equals rule)***)

z'=z+1 - (***Independent and Action***)

<u>NOT</u>

The **NOT** operator can only be used with an operand that evaluates to a Boolean value. It is one of those operators that can be both a constraint and an action. If the post-condition is of the form **NOT** A **SUBSET** B or **NOT** A **SUBSETEQUAL** B then the post-condition is said to be a constraint. We need to simplify all the other post-conditions and convert them to another form which can be resolved as a constraint or an action. The post-conditions, with **not** as the top-level operator, can sometimes be split and simplified using de Morgan's laws. In addition, other simplification rules can be applied [7]. Consider the following post-conditions:

- **NOT** (A **AND** B) is equivalent to **NOT** A **OR NOT** B
- **NOT** (A **OR** B) is equivalent to **NOT** A **AND NOT** B
- **NOT** (A **=>** B) is equivalent to A **AND NOT** B

- **NOT** (**NOT** (A')) is equivalent to A'
- **NOT** (A**<** B) is equivalent to **(**A >= B)
- **NOT** (A <=B) is equivalent to (A>B)
- **NOT** (A > B) is equivalent to (A<=B)
- **NOT** (A >=B) is equivalent to (A<B)
- **NOT** (A =B) is equivalent to A/=B
- **NOT** (A /=B) is equivalent to **NOT** (A=B)
- **NOT(FORALL (X: MYINT) (EXP1))** is equivalent to **EXISTS (X:MYINT) (NOT(EXP1)**
- **NOT(EXISTS (X: MYINT) (EXP1))** is equivalent to **FORALL (X:MYINT) (NOT(EXP1)**

IN

The elements of the IN operator are of set type. It results in a Boolean. Existing transforms XformArjun10 and XformArjun11 change all the post-conditions that have IN as top-level operator into procedure call statements [8].

IMPLIES

The `implies` operator evaluates to true either if the left hand side is false or if both sides are true. If the post-condition is of the form **P IMPLIES NOT P** or **NOT P IMPLIES NOT Q** then the post-condition needs to be simplified to be resolved as an action or a constraint. All the other post-conditions with implies are ***actions***.

Transforms have been developed to change a post-condition with implication as the top-level operator. Transform XformArjun7 removes all the post-conditions that have **IMPLICATION** as the top-level operator and changes them to **IF THEN** statements. [8]

<u>FORALL</u> <u>and</u> **<u>EXISTS</u>**

These operators are resolved as actions in all the post-conditions.

<u>SUBSET</u> <u>and</u> **<u>SUBSETEQ</u>**

The result of an operation A **SUBSET** B or A **SUBSETEQ** B is always a constraint and the output is a Boolean value which is true if every element in A is present in B and false if otherwise.

<u>AND</u>

If two or more post-conditions are joined using **AND** then they can be split accordingly using the dependent and independent analysis. Then all the dependent post-conditions which are dependent are separated from those post-conditions which are independent. Then these post-conditions are resolved to be a constraint or an action depending on the operators these post-conditions possess.

<u>OR</u>

The OR transform can be used to combine two post-conditions.

Note 1: A transform XformArjun8 was developed to transform the form (expA **AND** expB) **OR** (**NOT** (expA) **AND** expC) to (expA => expB) **AND** (**NOT** (expA) => expC)[8].

If the post-condition is of the above form then it can be resolved to be an ***action*** as it follows the action rule for implies.

Note 2: If the transform is of the form (P OR NOT P), then it is evaluated to true.

If neither of these cases occur using the OR operator, then we can resolve the post-condition to be a ***constraint.***

WEAKEST-PRECONDITION

The available statements for the weakest pre-condition in AWESOME are assignment statements, sequence of statements, if statements, if else statements. This thesis aims at designing transforms that will determine the weakest pre-condition for assignment statements, sequence of statements, if statements, if else statements and procedure calls. Sequence of statements consists of sequence of AWESOME statements that have to be recursively transformed to get the weakest pre-condition. The if else statements have a condition and a body consisting of a sequence of instructions. The transform should also be designed to determine the pre-condition even if the if statement does not have the else part.

Assignment Statements

AWESOME does not change the syntax representation of an assignment expression. The representation of an assignment statement is X: =exp

Sequence of statements:

The sequence of statements consists of a list of AWESOME statements, since these are each terminated by a “**;**” the syntax representation in AWESOME remains the same.

If else statements:

The syntax for the if else statements differ slightly in AWESOME and its syntax is

```
if condition then
 S1
else
S2
end if
```

The condition must always be a Boolean type.

Procedure Call:

Procedures are user defined statements. The syntax of the procedure call is represented as follows

```
Procedure Identifier (sequence of formal parameters)
[ assumes precondition ]
[ guarantees post-condition ]
[ Is
[ local object declarations ]
begin
sequence of statements
end;
```

The pre and post-conditions can see only the parameters. The body of the subprogram must satisfy the pre and post-conditions whenever all are specified. The procedure parameters can have mode in, out, or in out[2]

These are the syntax representations of the available statements in AWESOME for the weakest pre-condition.

SUMMARY

This requirement analysis chapter provides information regarding dependent and independent post-conditions, actions and constraints. A formal definition of dependency is overlaid. The existing transforms are handled by treating the post-conditions to be independent , but now the dependent and independent transforms are separated and then transforms are applied respectively. This chapter presents all the simplifying **NOT** transforms. Finally the weakest pre-condition predicate transformer with its allowable program statements is discussed.

4. DESIGN

The previous chapter provides the necessary information regarding the different types of post-conditions. We analyzed the dependency of the post-conditions and resolved the difference between an action and a constraint. This chapter provides the design to meet the requirements mentioned in the last section of the previous chapter.

WORKING DEFINITIONS:

In the requirements analysis chapter, we categorized the post-conditions into dependent/independent and action/constraint. The present section provides working definitions to identify the different types of post-conditions of interest using the mathematical definitions of sets for dependency and categorizing the Boolean operators, which can be resolved to an action or constraint.

DEPENDENCY: Two post-conditions exhibit dependency if they both contain the same primed values or a primed value in one post-condition and the same unprimed value in the other.

The analysis we have made suggests that the dependency of the post-conditions is focused on the primed and unprimed set of variables that they contain. If we could obtain the set of primed and unprimed variables and then establish a relation between them, it would give us the exact termed definition for dependency. Hence we will follow the mathematical analysis of sets to determine the dependency of the post-conditions and then overlay the exact definition of dependent post-conditions and independent post-conditions.

<u>MATHEMATICAL DEFINITIONS:</u>

A set is a collection of objects of the same type or of a different type. The variables in the post-conditions are grouped into sets depending on their primed and unprimed values.

ANALYSIS:

Let us consider two sets **U (exp)** and **P (exp)** where

U (exp) = Set of variables that are unprimed in expression exp.

P (exp) = Set of variables that are primed in expression exp.

Considering the above sets, we will define a mathematical definition for the post-conditions.

DEPENDENT:

Two post-conditions are said to be dependent if any one of the conditions cited below is true.

CONDITION 3:

The intersection of both the primed sets of a post-condition is not a NULL set.

$$P\ (exp1) \cap P\ (exp2) \neq \text{NULL SET}$$

CONDITION 4:

The intersection of the unprimed set of post-condition 1 and the primed set of post-condition2 is not a NULL set.

$$U\ (exp1) \cap P\ (exp2) \neq \text{NULL SET}$$

CONDITION 5:

The intersection of the primed set of post-condition 1 and the unprimed set of post-condition 2 is not a NULL set.

$$P\ (exp1) \cap U\ (exp2) \neq \text{NULL SET.}$$

Example: Determining the post-conditions to be dependent or independent

$$P1: X' = X+1$$

$$P2: Y' = X+1$$

We will apply the analysis made so far and then determine the dependency between them.

$$P\ (exp1) = \{X\}$$

$$P\ (exp2) = \{Y\}$$

$$U\ (exp1) = \{X\}$$

$$U\ (exp2) = \{X\}$$

1. Applying CONDITION 3 we have

$$P\ (exp1) \cap P\ (exp2) \neq \text{NULL SET}$$

$$\{X\} \cap \{Y\} = \text{NULL SET}$$

Hence our CONDITION 3 is ***false***

2. Applying CONDITION 4 we have

$$U\ (exp1) \cap P\ (exp2) \neq \text{NULL SET}$$

$$\{X\} \cap \{Y\} = \text{NULL SET}$$

Hence our CONDITION 4 is ***false***

3. Applying CONDITION 5 we have

$$P\ (exp1) \cap U\ (exp2) \neq \text{NULL SET}$$

$$\{X\} \cap \{X\} \neq \text{NULL SET}$$

Hence our CONDITION 5 is ***true***

Therefore we determine that post-condition p1 and p2 are dependent.

If any one of the conditions stated above is true, then the post-conditions are said to be dependent. Hence, by the mathematical analysis of sets we can determine the dependency of the post-conditions.

The class **XgetVarAbhinav** is designed to determine the primed and unprimed variables in the post-condition. This class consists of three library functions they are

1. **returnPrimeVar(exp, tick):** The function returns a vector containing all the primed variables in a post-condition.

2. **returnUnPrimeVar(exp):** This function returns the Unprimed variables in the post-condition .

3. **returnPrUnPrimeVar(exp,tick):** This function returns both the primed and unprimed variables in the post-condition

<u>ACTIONS AND CONSTRAINTS</u>

ACTION: An action is a post-condition that can uniquely be made true by executing program statements. Every method needs at least one action in the post-condition.

CONSTRAINT: A constraint is a post-condition that cannot be uniquely made true by executing program statements, or a post-condition that is not an action is a constraint.

The classification of Boolean operators provides the necessary information to resolve the post-condition to be an action or constraint. Hence, we will list classified Boolean operators according to the table discussed in chapter 3 and then resolve the post-conditions. Note that in many cases the decision is based on the result of existing transforms, or they are obvious from an intuition sense.

<u>BOOLEAN OPERATORS</u>

In the previous chapter, Table 1 classifies all the Boolean operators that we are concerned about. We will list each operator and then resolve the operator to be a constraint or an action.

<u>COMPARISON OPERATORS</u>

The comparison operators are used to compare the operands and hence it does contain a range of values, which cannot be uniquely made true by executing a sequence of statements. Hence by our definition of action, we can resolve the post-condition that have comparison operators to be constraints. The comparison operators from Table 1 are:

- LESS THAN (<)
- LESS THAN OR EQUALS TO (<=)
- GREATER THAN (>)
- GREATER THAN OR EQUALS TO (>=)
- EQUALS TO (=) – (***Exceptional Case***)
- NOT EQUAL TO (/=)

If the post-condition contains any of these comparison operators then we can resolve the post-condition to be a **constraint**.

EXAMPLES

- X<100 is a constraint.
- X<=200 is a constraint.
- X>=10 and X<100 is a constraint.

All these examples do not contain a unique value that can be made true and hence are resolved as constraints.

<u>EQUALS TO (=)</u>

The EQUALS operator is an exceptional case, which needs to undergo an ***erule*** test to be resolved to a constraint or an action. Apply the rules below when an EQUALS operator is present in the post-condition.

MATHEMATICAL DEFINITIONS:

- ***ERULE1:*** If post-condition is independent and has an EQUALS operator and if the unprimed set of the post-condition is a NULL set and the primed set contains at least 2 variables, then the post-condition is a constraint, else it is an action

U (exp) =NULL and size((exp))>=2

The EQUALS operator should be evaluated with ERULE1 before it can be resolved.

<u>FORALL</u> <u>and</u> **<u>EXISTS</u>**

These operators contain a range of values that can be uniquely made true by executing a sequence of statements and hence by our definition these operators are resolved as actions in all the post-conditions. Hence, the post-conditions with these operators can be directly resolved to an action.

XNotAbhinav8 Transform is designed to transform the post-condition of the form **NOT(FORALL (X: MYINT) (EXP1)) to EXISTS (X:MYINT) (NOT(EXP1).**

XNotAbhinav9 Transform is designed to transform the post-condition of the form **NOT(EXISTS (X: MYINT) (EXP1)) to FORALL (X:MYINT) (NOT(EXP1)**

<u>IMPLIES</u>

All the post-conditions which contain the IMPLIES operator are resolved to be actions except for the two cases.

***Case 1*: A => NOT A**

This post-condition evaluates to true and hence it can be considered true and can be removed.

XImpliesAbhinav2 transforms the post-condition of the form **e1 -> NOT e2 to TRUE**

***Case 2*: NOT A => NOT B ≡ B => A**

This form of post-condition needs to be simplified in order to be resolved.

XImpliesAbhinav1 Transforms the post-conditions of the form **NOT e1 => NOT e2** to **e2=> e1.**

AND

The post-conditions which contain AND cannot be directly resolved as an action or a constraint. The existing transform XformArjun5 directly splits all the post-conditions which have AND. Then by using the dependent and independent analysis, all the post-conditions which are dependent are separated from those post-conditions which are independent. Then these post-conditions are resolved to be a constraint or an action depending on the operators these post-conditions possess.

EXAMPLE

If the post-condition is of the form:

(a' = this') and (e' = a'^[i'].field[c']^) and (a' = b'[c']) and (a' = t'.c) and (a' = ptr'^)

by applying XformArjun5 the following results

a' = this'

e' = a'^[i'].field[c']^

a' = b'[c']

a' = t'.c

a' = ptr'^

OR

The post-conditions which contain OR as an operator can be directly resolved to constraints but there are some exceptional cases which need to be resolved. The cases are

Case 1: (P **AND** Q) **OR** (NOT P **AND** R)

If the post-condition is of the above form then Xformarjun8 can be applied to it, which results in

(P **AND** Q) OR (NOT P **AND** R) ≡ (P **=>** Q) **AND** (NOT P **=>** R)

The above post-condition can now be resolved to an action.

Case 2: (P **OR NOT** P)

This post-condition evaluates to true P **OR NOT** P ≡ **TRUE**

XOrAbhinav transforms the post-condition of the form **(P OR NOT P)** to **TRUE.**

SIMPLIFICATIONS

Some of the NOT simplifications using all the operators are specified in Table 1.

- **NOT (A < B) ≡ A >= B**
- **NOT (A <= B) ≡ A > B**
- **NOT (A > B) ≡ A <= B**
- **NOT (A >= B) ≡ A < B**
- **NOT (A = B) ≡ A ≠ B**
- **NOT (A ≠ B) ≡ A = B**
- **NOT (NOT A) ≡ A**
- **A => NOT A ≡ TRUE**
- **NOT A => NOT B ≡ B => A**

- **NOT (FORALL x) (Px) ≡ (EXISTS x) (NOT Px)**
- **NOT (EXISTS x) (Px) ≡ (FORALL x) (NOT Px)**
- **NOT (A AND B) ≡ NOT A OR NOT B**
- **NOT (A OR B) ≡ NOT A AND NOT B.**

All the simplifications are implemented in the AWESOME tool and are checked for correctness.

- **XNotAbhinav1** transforms the post-condition of the form

  NOT(A<B) to **A>=B .**

- **XNotAbhinav2** transforms the post-condition of the form

  NOT(A<=B) to **A>B**

- **XNotAbhinav3** transforms the post-condition of the form

  NOT(A>B) to A<=B

- **XNotAbhinav4** transforms the post-condition of the form

  NOT(A>=B) to A<B

- **XNotAbhinav5** transforms the post-condition of the form

  NOT(A=B) to A!=B

- **XNotAbhinav6** transforms the post-condition of the form

NOT(A!=B) to A=B

- **XNotAbhinav7** transforms the post-condition of the form

  NOT(NOT A) to A

- **XNotAbhinav8** transforms the post-condition of the form

  NOT(FORALL (X: MYINT) (EXP1)) to **EXISTS (X:MYINT) (NOT(EXP1).**

- **XNotAbhinav9** transforms the post-condition of the form

  NOT(EXISTS (X: MYINT) (EXP1)) to **FORALL (X:MYINT) (NOT(EXP1)**

- **XNotAbhinav10** transforms the post-condition of the form

  NOT (A AND B) to NOT A OR NOT B.

- **XNotAbhinav11** transforms the post-condition of the form

  NOT(A AND B) to NOT A OR NOT B.

<u>WEAKEST PRE-CONDITION</u>

In the previous chapter we discussed about the different available statements in weakest pre-condition. This section discusses the different ways to design these transforms to be implemented in the tool.

WEAKEST PRE-CONDITION ALGORITHM

The following algorithm is being used before using the weakest pre-condition analysis. Since AWESOME and AWL are primed variables for "after" values the post-conditions must be converted to a form similar to the guarded command language.

1. For each ticked variable $x^{'}$, define a local variable x_pre which is similar to the OCL's x@pre.
2. **For** each such variable, add an assignment statement x_pre: =x ***at the beginning of the program.***
3. **In** the post-condition(s) replace each unticked variable with the corresponding "_pre" version (e.g $x^{'}$ **=** x + 1 becomes $x^{'}$ = x_pre + 1)
4. **In** the post-condition(s), replace each ticked variable with the corresponding unticked version (e.g. $x^{'}$ = x + 1 becomes x = x_pre + 1). Note that this is basically the OCL post-condition syntax.
5. Proceed with the weakest-precondition analysis as usual, moving the post-condition back through each statement.

A transform class **XAddLocalAbhinav** is designed to perform the necessary task. It adds a local variable "_pre" to the unprimed variable and changes the variable " x' " to x. This transform also adds an assignment statement at the beginning of the method. It calls the **XWpAbhinav** class and adds the weakest pre-condition to the subprogram. The **XWpAbhinav** class calls the required functions of the program statements. **XWpAbhinav** calls **WeakestPrecondition(WsAssignment, WsExpression),**

WeakestPrecondition(WsSelection,WsExpression),WeakestPrecondition(Vector,Ws Expression) functions which have been discussed below.

ASSIGNMENT EXPRESSION

In the assignment expression all the occurrences of the variable are being replaced by the expression (right hand side of the expression). If x: =exp, all the occurrences of x are being replaced by exp. The syntax for the assignment expression can be represented as **wp(x:=exp, Q) = Q^x_e** means the predicate Q with all free occurrences of x simultaneously replaced by e.

WeakestPrecondition(WsAssignment, WsExpression) determines the weakest-precondition for assignment statements where **WsAssignment** contains the assignment expression and **WsExpression** with all occurrences of the LHS of the **WsAssignment**.

IF-ELSE STATEMETS

In if-else statements a condition is specified in the **If** part which is followed by the **then** part which contain statements to be executed depending on whether the condition is true or false. The syntax of if-else statements is represented as

wp(if B S1 else S2 , Q) ≡ B → wp(S1, Q) AND (NOTB) → wp(S2, Q)

Special Case:

There are some if statements which do not have the else parts. The syntax of these statements is different, as they do not contain the else part. The syntax for only if statement is

(if B S1 else S2, Q) ≡ B => **wp(S1, Q) AND NOT B => Q**

WeakestPrecondition(WsSelection,WsExpression) determines the weakest pre-condition for selection statements where **WsSelection** contains the selection statements and **WsExpression** with occurrences of LHS of **WsAssignment**.

SEQUENCE OF STATEMENTS

It resembles the assignment expression but it is being applied repeatedly for a sequence of statements and hence termed sequence of statements. The syntax for the sequence of statements can be represented as wp**(S1 ; S2 , Q)=wp(S1, wp(S2 , Q))** where s1, s2 are the instruction sequence and the post-condition of S1 must exactly match the weakest pre-condition of S2 establishing Q.

The desired post-condition is considered and the variable in it is replaced with the expression and weakest pre-condition is calculated. We termed it as wp1 and then wp2 is calculated with the next desired variable substituted with the expression. This sequential process is carried out until we are out of expressions and the desired pre-condition is obtained.

WeakestPrecondition(Vector,WsExpression) determines the weakest-precondition for sequence of statements. The Vector may contain **WsAssignment** statements or **WsSelection** statements or sequence of statements which recursievely calls itself.

PROCEDURAL CALLS AND LOOP INVARIANTS

The design for the procedure calls and loop invariants is out of the scope this thesis and is not being addressed in this thesis.

SUMMARY

The design chapter provides a detailed description of the design decisions involved in the design of the transforms. The needed library functions and transforms have been discussed. The dependencies between the transforms are designed using the mathematical sets analysis. The post-conditions are then separated to actions or constraints using Table1. Some simplifications have to be done to the post-conditions before they can be determined as actions or constraints. The special cases for the operators mentioned in Table1 have been designed and discussed. Finally the weakest pre-condition analysis which contains different program statements have been discussed. These transforms designed may be clearer after explaining the test case examples in the next chapter.

5. IMPLEMENTATION AND TESTING

The previous chapter presented the design of different transforms. This chapter discusses the implementation of these transforms and the output obtained after applying the transforms. The main goal of this chapter is to present the approach of testing the transforms. The validity of the transforms and the different test cases used to test the validity of the transform and the output obtained is presented in this chapter.

TESTING ENVIRONMENT

To test the transforms a case tool shown in Fig 5.1 is used. It can hold all the tools required for testing. To use a tool we just need to click on it. The most important tool is the **ToolMethod1** which opens a GUI application that plays a key part in the testing. It has all the necessary functions required. Some of the supporting tools used during the testing are **ToolLoader** which will parse and load the input file, **ToolToAWL** writes the AST back to **AWL** and **ToolToOutline** prints the **AST** to a file in outline format. The name of the output file can be specified by the user. All these tools are a part of a package called **toolbox** [4].

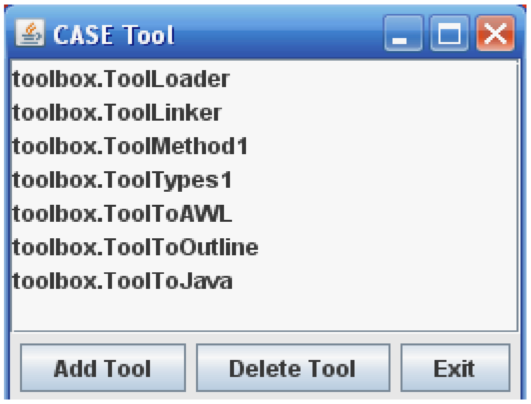
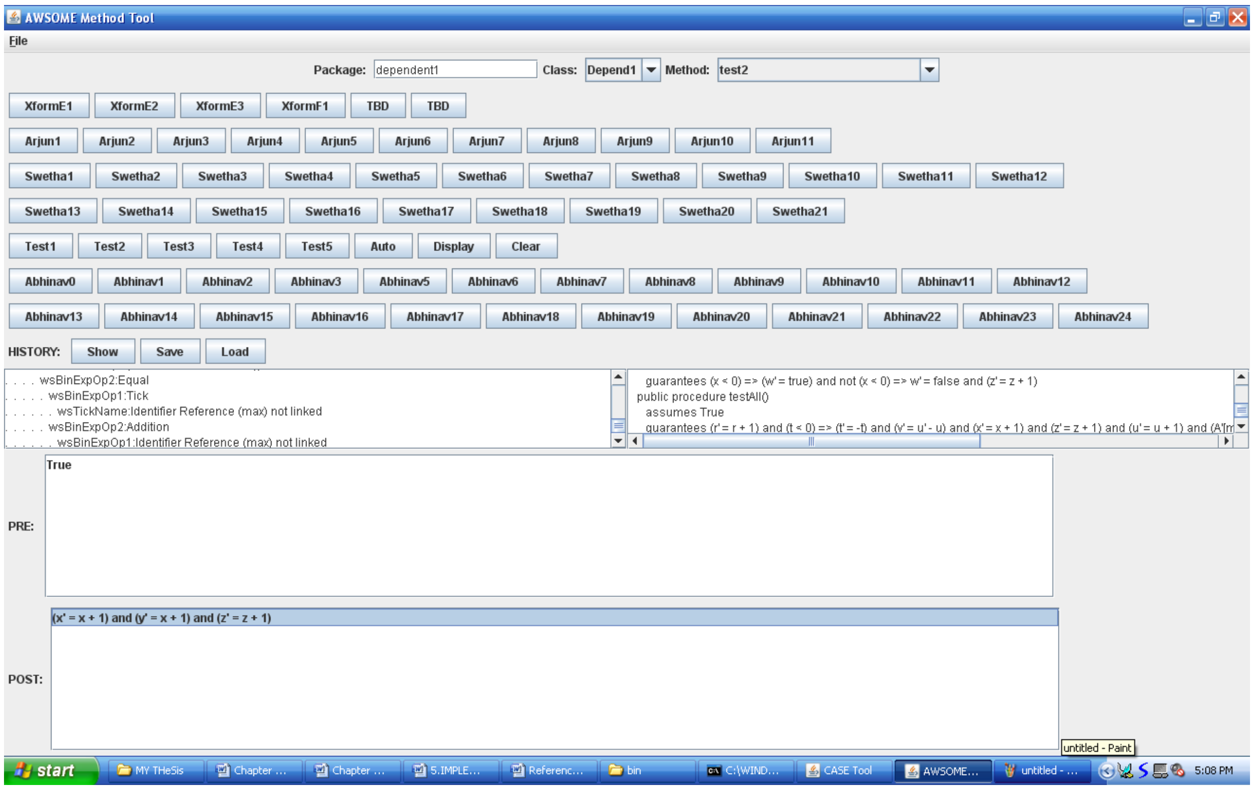


Figure 5.1 Case Tool

The **ToolMethod1** GUI is an interface which has a drop menu at the top to select the required package, class and method name from the list of available options. It contains lots of buttons where each corresponds to a definite transform and that transform is applied when it is clicked. The center section of the GUI is comprised of two text areas. The right text area displays the target class (including all the attributes and the subprograms) and the text area on the left displays the AST of the target class in outline format [6]. The last two text areas of the tool contain the selected method's pre and post-conditions. The post-condition which we need to transform is selected in the post-conditions text area and then the required transform button is applied to the post-condition to get it transformed. A screen shot (Figure5.2) of **ToolMethod1** is provided for better understanding


AWSOME Method Tool
File
XformE1
XformE2
XformE3
XformF1
TBD
TBD
Arjun1 Arjun2 Arjun3 Arjun4 Arjun5 Arjun6 Arjun7 Arjun8 Arjun9 Arjun10 Arjun11
Swetha1 Swetha2 Swetha3 Swetha4 Swetha5 Swetha6 Swetha7 Swetha8 Swetha9 Swetha10 Swetha11 Swetha12
Swetha13 Swetha14 Swetha15 Swetha16 Swetha17 Swetha18 Swetha19 Swetha20 Swetha21
Test1 Test2 Test3 Test4 Test5 Auto Display Clear
Abhinav0 Abhinav1 Abhinav2 Abhinav3 Abhinav5 Abhinav6 Abhinav7 Abhinav8 Abhinav9 Abhinav10 Abhinav11 Abhinav12
Abhinav13 Abhinav14 Abhinav15 Abhinav16 Abhinav17 Abhinav18 Abhinav19 Abhinav20 Abhinav21 Abhinav22 Abhinav23 Abhinav24
HISTORY: Show Save Load
start
MY THeSis
Chapter ...
Chapter ...
5.IMPLE...
Referenc...
bin
C:\WIND...
CASE Tool
AWSOME...
untitled - ...
untitled - Paint
5:08 PM

Package: dependent1 Class: Depend1 Method: test2

```
. . . . wsBinExpOp2:Equal
. . . . . wsBinExpOp1:Tick
. . . . . . wsTickName:Identifier Reference (max) not linked
. . . . . wsBinExpOp2:Addition
. . . . . . wsBinExpOp1:Identifier Reference (max) not linked
```

```
  guarantees (x < 0) => (w' = true) and not (x < 0) => w' = false and (z' = z + 1)
public procedure testAll()
  assumes True
  guarantees (r' = r + 1) and (t < 0) => (t' = -t) and (v' = u' - u) and (x' = x + 1) and (z' = z + 1) and (u' = u + 1) and (A'[m
```

PRE: True

POST: (x' = x + 1) and (y' = x + 1) and (z' = z + 1)

Figure 5.2 Screen Shot of ToolMethod1

TESTING METHODOLOGY

The test cases are written in a formal specification AWL file. Each transform is being tested with different test cases for accuracy. The required test case is loaded in the tool by invoking **ToolLoader**. The **ToolLoader** will check for any syntax errors and displays a successful parse message after being parsed, or prompts an error message if there are any syntax errors. The syntax errors have to be cleared before **ToolMethod1** is loaded from the case tool. When **ToolMethod1** is selected the GUI is displayed with all the buttons which have the required transforms. The required package class, and method name have to be selected before transforming a post-condition. After their selection the required post-condition which needs to be transformed is selected. Applying the transform will call the *applicable()* method and the result is either a transformed post-condition or code statements. To check the modifications either the **ToolToAWL** or the **ToolToOutline** can be used. The **ToolToAWL** will write the AST back to an AWL file which will reflect the changes made to the input AWL file. This is mostly used since it is in a user readable form. The **ToolToOutline** will print the AST to a file which will reflect the changes made to the AST. This is used to verify the internal changes to the AST as needed. This description of the tool should be sufficient for the user to use the tool and know its functions.

TESTING CLASSES

Class XgetVarAbhinav:

Function: Gets the names of the primed and unprimed variables from a post-condition and places them in different vectors.

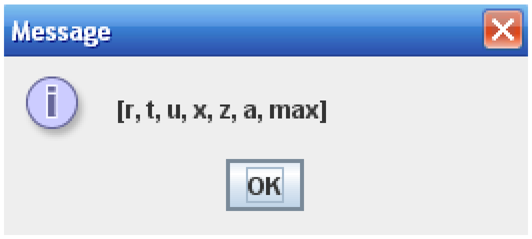
This class mainly has 3 functions .

1. **returnUnPrimeVar(exp, tick):** The function returns a vector containing all the unprimed variables in a post-condition. The required post-condition is passed as WsExpression and a Boolean variable tick is passed as second parameter. If tick is false the variable in the post-condition is unprimed and is moved into the unprimed vector. The testcase for XgetVarAbhinav is shown in figure 5.3. It contains a testAll() method which contains a post-condition with primed and unprimed variables and this function displays the unprimed variables as shown in figure 5.4

```
public procedure testAll()
    assumes True
    guarantees r' = r + 1 and (t < 0 => t' = -t) and v' = u' - u
        and x' = x + 1 and z' = z + 1 and u' = u + 1
        and A'[max'] = a and s' = r + 1 and y' = x' + 1
        and (t >= 0 => t' = t) and max' = max + 1
```

Figure 5.3: Input to test the xgetVarAbhinav class

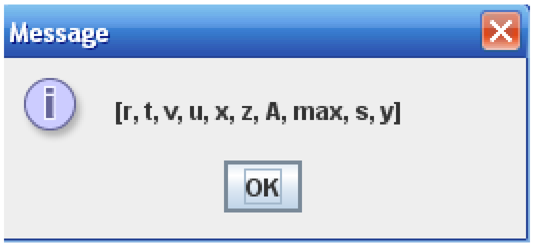
.



[r, t, u, x, z, a, max]

Figure 5.4 : Output of the function returnUnPrimeVar(exp, tick)

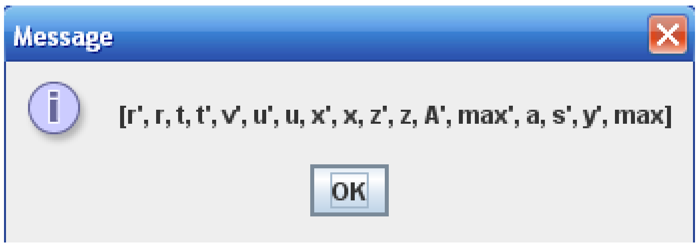
2. **returnPrimeVar(exp):** This function returns the primed variables in the post-condition. The input post-condition is passed as WsExpression as the post-condition may be WsBinaryExpression, WsUnaryExpression, WsIdentifierRef, WsQuantifiedExpression and many other forms which are accepted in AWSOME. The output of the function is displayed in figure 5.5 for the input of the figure 5.3.



```
[r, t, v, u, x, z, A, max, s, y]
```

Figure 5.5: Output of the function returnPrimeVar(exp)

3. **returnPrUnPrimeVar(exp,tick):** This function returns both the primed and unprimed variables in the post-condition. This function is not used in the thesis but, it is being written for future reference. The output of the function is shown in figure 5.6.



```
[r', r, t, t', v', u', u, x', x, z', z, A', max', a, s', y', max]
```

Figure 5.6: Output of the function returnPrUnPrimeVar(exp,tick)

Class XDependentAbhinav:

Function: Returns true if the selected post-conditions are dependent.

This class contains a single function which tests the selected post-conditions to be dependent and returns true if they are dependent. This function uses the functions defined in the **XgetVarAbhinav** class to display the dependency of the post-conditions.

areDependent(exp1,exp2): This function has the selected post-conditions as its two arguments which are passed as WsExpressions. The primed and unprimed variables of the post-conditions are passed into separate vectors and depending on the **CONDITION RULES** mentioned in chapter 4 the dependency of the post-conditions is determined.

```
//dependent.awl - test post-condition dependencies
//09/06/08
package dependent is
type Integer is range 0..100;

  class Depend is
    public r : Integer;
    public s : Integer;
    public t : Integer;
    public u : Integer;
    public v : Integer;
    public w : boolean;
    public x : Integer;
    public y : Integer;
    public z : Integer;
    public k : Integer;
    public l : Integer;
    public g : Integer;
    public h : Integer;
    public max : Integer;

    //P1,p2 and P3 independent
    public procedure test1()
      assumes True
      guarantees x' = x + 1 and y' = y + 1 and z' = z + 1

   //p1, p2 and p3 independent
    public procedure test1a()
      assumes True
      guarantees x' = x + k' and y' = y + 1 and z' = z + 1
```

Figure 5.7 Input specification file for testing Dependency

```
//p1, p2, p3, p4 and p5 independent
      public procedure test1b()
            assumes True
      guarantees x' = x + k' and y' = y + J' and z' = z + l' and s'= f' + u' and h = n' +
g

//P1 dependent on P2
    public procedure test2()
      assumes True
      guarantees x' = x + 1 and y' = x + 1 and z' = z + 1
//P2 dependent on P1
    public procedure test3()
      assumes True
      guarantees x' = x + 1 and y' = x' + 1 and z' = z + 1

//P2 & P1 dependent on each other
    public procedure test4()
      assumes True
      guarantees x' = y + 1 and y' = x + 1 and z' = z + 1

    //P2 & P1 dependent on each other
    public procedure test5()
      assumes True
      guarantees x' = x + 1 and y' = x' - x and z' = z + 1

    //p1,P2 and p3 dependent but P1, P3 are mutually independent
    public procedure test6A(a : in Integer)
      assumes True
      guarantees x' = a and x'*x' + y'*y' = z'*z' and y' = y

    //P1,P2 are dependent, P2,P3 are dependent but p1,p2,p3 independent
    public procedure test6B(a : in Integer)
      assumes True
      guarantees x' = a and x'*x' + y'*y' = z'*z' and y' = z'

    //p1,P2,P3 are dependent, P2,P3 are mutually independent
    public procedure test7()
      assumes True
      guarantees x' = x + 1 and y' = x' + 1 and z' = x' + 2

    //P2 dependent on P1, P3 dependent on P2
    public procedure test8()
      assumes True
      guarantees x' = x + 1 and y' = x' + 1 and z' = y' + 1

    //P1 and P2 independent
    public procedure test9()
      assumes True
      guarantees (x >= 0 => x' = x) and
                 (x < 0 => x' = -x)
```

Figure 5.7 continued…

```
//P2 and P1 dependent on each other
   public procedure test10()
     assumes True
     guarantees (light = red => light' = green) and
              (light = green => light' = red)

//p1,P2 are dependent
   public procedure test11A(a : in Integer)
     assumes True
     guarantees max' = max + 1 and A'[max'] = a

//P1,p2 are dependent
   public procedure test11B(a : in Integer)
     assumes True
     guarantees max' = max + 1 and A'[max] = a

//p1,p5 and p2,p11 and p3,p8 and p4,p10 and p6,p9 are dependent and oher
are mutually indeoendent
   public procedure testAll()
     assumes True
     guarantees r' = r + 1 and (t < 0 => t' = -t) and v' = u' - u
           and x' = x + 1 and z' = z + 1 and u' = u + 1
           and A'[max'] = a and s' = r + 1 and y' = x' + 1
           and (t >= 0 => t' = t) and max' = max + 1

  end class;

end package;
```

Figure 5.7 continued

The dependency of the post-conditions in every method is determined and their dependency is commented above every function. This function will only individually determine the selected post-conditions to be dependent and does not give the dependent post-conditions from a group of post-conditions as commented above every function in the figure 5.7.

3.XConstraintAbhinav:

Function: This class determines the specific post-condition to be a constraint.

isConstraint(exp): This function takes the post-condition as the input argument in the form of WsExpression and checks the instance of the post-condition according to Table1 cited in chapter 3 and returns true if the post-condition is a constraint. The input specification file is shown in figure 5.8. *Instanceof()* operator is used to determine the expression from the list of **WsClasses.**

All the post-conditions in the input AWL file are termed to be constraints and hence ***true*** is returned as the output

```
package testing_action_constraint is

type Integer is range 0..100;

  class Teesting_Action_Constraint is

public procedure test_constraint()
     assumes True
     guarantees x' < x + 1 and y'> y + 1 and z'<=z + 1 and w'>=w+1 and h'=0 and
h'/= g + 1 and k subset l and k subseteq h and ((a' < b') or (c' <= d'))
```

Figure 5.8: Input AWL specification for testing Constraints

3. XActionAbhinav:

Function: This class determines the specific post-condition to be an action.

isAction(exp): This function takes the post-condition as the input argument in the form of WsExpression and checks the instance of the post-condition according to Table1 cited in chapter 3 and returns true if the post-condition is an action. The input specification file is shown in figure 5.9. *Instanceof()* operator is used to determine the expression from the list of **WsClasses.**

```
package testing_action_constraint is
type Integer is range 0..100;

 class Teesting_Action_Constraint is

public procedure test_constraint()
     assumes True
     guarantees x' < x + 1 and y'> y + 1 and z'<=z + 1 and w'>=w+1 and h'=0 and h'/= g + 1
and k subset l and k subseteq h and ((a' < b') or (c' <= d'))
```

Figure 5.9 Input AWL specification for testing Action

All the post-conditions in the input AWL file are termed to be actions and hence **TRUE** is returned as the output.

Some special cases need to follow the ***Equals Rule*** specified in chapter 3. The cases with **AND, OR, IMPLIES** have to be simplified before they can be determined to be an action or constraint. We will discuss the special cases in detail.

Transform XOrAbhinav:

Function: Transforms the post-condition of the form **P OR NOT P** to **TRUE**

This transform is applied to the post-conditions which contain the OR operator. Most of the post-conditions which contain OR operator are termed to be constraints. But, there is a special case of OR to which this **XOrAbhinav** transform is applied to convert it to TRUE.

The post-condition of the form P OR NOT P is evaluated to TRUE. The *applicable()* method checks for the syntax of the post-condition. It should contain OR as the operator.

After verifying it *execute()* is applied and is evaluated to TRUE only if *applicable()* is TRUE. The input specification of the AWL file is shown in figure 5.10

```
package testing_action_constraint is

type Integer is range 0..100;

  class Teesting_Action_Constraint is

//special case for or
    public procedure test_or()
      assumes True
      guarantees ( u and v ) or ( not u and w )
```

Figure 5.10 Input Specification of AWL file for OR

```
package testing_action_constraint is

type Integer is range 0..100;

class Teesting_Action_Constraint is

//special case for or
public procedure test_or()
assumes True
guarantees (u => v) and (not u => w)
```

Figure 5.10a Input Specification of AWL file for OR

<u>**TESTING SIMPILFICATION TRANSFORMS**</u>

XImpliesAbhinav:

Function:

(1) Transforms the post-condition of the form **NOT P IMPLIES NOT Q** to **Q IMPLIES P**

(2) Transforms the post-condition of the form **P IMPLIES NOT P** to **TRUE**

This transform is applied to the post-conditions which contain the **IMPLIES** operator. All the post-conditions that contains **IMPLIES** operator are termed to be actions. But, there are two special cases of **IMPLIES** which need to be transformed to an alternate post-condition which can be determined to be an action.

(1) The post-condition of the form **NOT P IMPLIES NOT Q** is checked by the ***applicable()*** function and when it returns TRUE the ***execute*()** function is called to transform the post-condition to **Q IMPLIES P** which can be determined to be an action. The ***execute()*** method in turn calls **replacePostCondition (postCond, exp1, exp2)** which takes the post-condition needed and is transformed to a new post-condition **Q IMPLIES P** with exp1 and exp2 as its operands.

(2) The post-condition of the form **P IMPLIES NOT P** is checked by the ***applicable()*** function and when it returns **TRUE** the ***execute*()** function is called to transform the post-condition to **TRUE**. The ***execute()*** method in turn calls **replacePostCondition(postCond, exp1, exp2)** which takes the post-condition needed and is transformed to a new post-condition which evaluates to **TRUE**.

The input AWL specification file for the IMPLIES operator is shown in figure 5.11 and the output is shown in figure 5.12.

```
package testing_action_constraint is
type Integer is range 0..100;

  class Teesting_Action_Constraint is

//special case for implies
    public procedure test_implies()
      assumes True
      guarantees (a => not a) and (not (x' * y ) => not (z' * w))
```

Figure 5.11 Input AWL specification for IMPLIES

```
package testing_action_constraint is
type Integer is range 0..100;

  class Teesting_Action_Constraint is

//special case for implies
    public procedure test_implies()
      assumes True
      guarantees true and (z' * w ) => (x' * y)
```

Figure 5.12 Transformed output of AWL file in Figure 5.11

AND Transforms:

This thesis does not have any **AND** transforms but the transform XformArjun5 is applied which iteratively splits all the post-conditions in the method CheckIn, which have **AND** as the top-level operator into individual post-conditions. The resulting post condition set is modified and holds the individual post conditions.

```
package testing is
   type INT is range -65000 .. 65000;
   class TestingAND is
      public a : INT;
      public b : INT;
      public c : INT;
      public d : INT;
      public proedure split()
         assumes true
         guarantees a' > b' and c' >= d' and a' = b' and c' /= 5 and a' < b'
   end class;
end package;
```

Figure 5.13 Input AWL specification for AND

```
public proedure split()
         assumes true
         guarantees a' > b'
                    c' >= d'
                     a' = b'
                    c' /= 5
                     a' < b'
```

Figure 5.14 Output specification for AND using xformArjun5

NOT Transforms:

This thesis presents all the **NOT** transforms which are used in **AWSOME**. There are 11 **NOT** transforms which have been dealt with and each transform has its own specific post-condition which needs to be transformed. The post-conditions which have **NOT** as the top level operator is considered and are being transformed.

Each of the Not transforms have been discussed briefly.

XNotAbhinav1:

Function: This transform transforms the post-condition of the form **NOT(A<B)** to **A>=B**

XNotAbhinav2:

Function: transform transforms the post-condition of the form **NOT(A<=B)** to **A>B**

XNotAbhinav3:

Function: This transform transforms the post-condition of the form **NOT(A>B) to A<=B**

XNotAbhinav4:

Function: This transform transforms the post-condition of the form **NOT(A>=B) to A<B**

XNotAbhinav5:

Function: This transform transforms the post-condition of the form **NOT(A=B) to A/=B**

XNotAbhinav6

Function: This transform transforms the post-condition of the form **NOT(A/=B) to A=B**

The ***applicable()*** method off all these transforms checks for the post-condition which has **NOT** as the top level operator and is in the form **NOT(A<B)** for **XNotAbhinav1**, **NOT(A<=B)** for **XNotAbhinav2, NOT(A>B)** for **XNotAbhinav3, NOT(A>=B)** for **XNotAbhinav4, NOT(A=B)** for **XNotAbhinav5, NOT(A/=B)** for **XNotAbhinav6**. It returns **TRUE** if the verified post-condition is an instance of **NOT** operator and has the required operator.

The ***execute()*** method uses **replacePostCondition(postCond, exp1, exp2)** which takes the input post-condition, the right and left hand side operands(expressions) and replaces the post-condition with instance of **WsGreaterThanOrEqual(A>=B)** for **XNotAbhinav1, WsGreaterThan(A>B)** for **XNotAbhinav2, WsLessThanOrEqual(A<=B)** for **XNotAbhinav3, WsLessThan(A<B)** for **XNotAbhinav4, WsNotEqual(A/=B)** for **XNotAbhinav5, WsEqual(A=B)** for **XNotAbhinav6** with exp1 and exp2 as its operands.

```
package testing_Not_Simplification is
public procedure not_simplification1()
     assumes True
     guarantees (not(a < b))
end class;
end package;
```

Figure 5.15 Input AWL specification for NOT(A < B)

```
package testing_Not_Simplification is
public procedure not_simplification1()
     assumes True
     guarantees (a>=b)
end class;
end package
```

Figure 5.16 output AWL file for NOT(A < B)

```
package testing_Not_Simplification is
public procedure not_simplification2()
assumes True
guarantees (not(a <= b))
end class;
end package;
```

Figure 5.17 Input AWL specification for NOT(A <= B)

```
package testing_Not_Simplification is
public procedure not_simplification2()
    assumes True
    guarantees (a>b)
end class;
end package
```

Figure 5.18 output AWL file for NOT(A <= B)

```
package testing_Not_Simplification is
public procedure not_simplification2()
    assumes True
    guarantees (a>b)
end class;
end package
```

Figure 5.19 Input AWL specification for NOT(A > B)

```
package testing_Not_Simplification is
public procedure not_simplification3()
    assumes True
    guarantees (a<=b)
end class;
end package
```

Figure 5.20 output AWL file for NOT(A > B)

```
package testing_Not_Simplification is
public procedure not_simplification1()
      assumes True
      guarantees (a>=b)
end class;
end package
```

Figure 5.21 Input AWL specification for NOT(A >=B)

```
package testing_Not_Simplification is
public procedure not_simplification4()
      assumes True
      guarantees (a<b)
end class;
end package
```

Figure 5.22 output AWL file for NOT(A >= B)

```
package testing_Not_Simplification is
public procedure not_simplification5()
      assumes True
      guarantees (not(a=b))
end class;
end package
```

Figure 5.23 Input AWL specification for NOT(A =B)

```
package testing_Not_Simplification is
public procedure not_simplification5()
      assumes True
      guarantees ((a/=b))
end class;
end package
```

Figure 5.24 output AWL file for NOT(A = B)

```
package testing_Not_Simplification is
public procedure not_simplification6()
     assumes True
     guarantees (not(a/=b))
end class;
end package
```

Figure 5.25 Input AWL specification for NOT(A !=B)

```
package testing_Not_Simplification is
public procedure not_simplification6()
     assumes True
     guarantees ((a=b))
end class;
end package
```

Figure 5.26 output AWL file for NOT(A != B)

The input and output AWL specifications of the NOT simplifications have been presented for each of the NOT transform implemented.

XNotAbhinav7:

Function: This transform transforms the post-condition of the form **NOT(NOT A) to A**

The ***applicable()*** method checks for the post-condition which has **NOT** as the top level operator and is in the form **NOT(NOT A).** It returns **TRUE** if the verified post-condition is an instance of **NOT** operator and is also an instance of **WsNot**.

The ***execute()*** method uses **replacePostCondition(postCond, exp1)** which takes the input post-condition, the right operand(expressions) as **WsNot** is Unary and replaces the post-condition with A. It is the transformed final post-condition.

```
package testing_Not_Simplification is
public procedure not_simplification7()
     assumes True
     guarantees (not(not(a))
end class;
end package
```

Figure 5.27 Input AWL specification for NOT(NOT(A))

```
package testing_Not_Simplification is
public procedure not_simplification7()
     assumes True
     guarantees (a)
end class;
end package
```

Figure 5.28 output AWL file for NOT(NOT(A))

XNotAbhinav8:

Function: This transform transforms the post-condition of the form

NOT(FORALL (X: MYINT) (EXP1)) to **EXISTS (X:MYINT) (NOT(EXP1)**

The ***applicable()*** method checks for the post-condition which has **NOT** as the top level operator and is in the form **FORALL (X: MYINT) (EXP1).** After the top level **NOT** operator is found it will search for an instance of **Universal Quantifier** class. It returns **TRUE** if the verified post-condition is an instance of **NOT** operator and is also an instance of **WsNotEqual**.

The ***execute()*** method replaces the **Universal quantifier** with **Existential quantifier** and the post-condition is transformed to **EXISTS (X:MYINT) (NOT(EXP1).**

```
package testing_Not_Simplification is
public procedure not_simplification8()
     assumes True
     guarantees ((not( forall (i : INT) ((i' > 0) => (k' /= h')))))
end class;
end package
```

Figure 5.29 Input AWL specification for not(forall (i : INT) ((i' > 0) => (k' /= h')))

```
package testing_Not_Simplification is
public procedure not_simplification8()
     assumes True
     guarantees ((exists (i : INT) (not((i' > 0) => (k' /= h'))))
end class;
end package
```

Figure 5.30 output AWL file for not(forall (i : INT) ((i' > 0) => (k' /= h')))

XNotAbhinav9:

Function: This transform transforms the post-condition of the form

NOT(EXISTS (X: MYINT) (EXP1)) to **FORALL (X:MYINT) (NOT(EXP1)**

The ***applicable()*** method checks for the post-condition which has **NOT** as the top level operator and is in the form **NOT(EXISTS (X: MYINT) (EXP1)).** After the top level **NOT** operator is found it will search for an instance of **Existential Quantifier** class. It returns **TRUE** if the verified post-condition is an instance of **NOT** operator and is also an instance of **WsNotEqual**.

The ***execute()*** method replaces the **Existential quantifier** with **Universal Quantifier** and the post-condition is transformed to **FORALL (X: MYINT) (NOT(EXP1).**

```
package testing_Not_Simplification is
public procedure not_simplification9()
     assumes True
     guarantees  (not( exists (i : INT) ((i' > 0) => (k' /= h'))))
end class;
end package
```

Figure 5.31 Input AWL specification for (exists (i : INT) ((i' > 0) => (k' /= h'))))

```
package testing_Not_Simplification is
public procedure not_simplification9()
assumes True
guarantees  ((forall (i : INT) (not((i' > 0) => (k' /= h'))))
end class;
end package
```

Figure 5.32 output AWL file for (exists (i : INT) ((i' > 0) => (k' /= h'))))

XNotAbhinav10:

Function: This transform transforms the post-condition of the form

NOT (A AND B) to **NOT A OR NOT B.**

The ***applicable()*** method checks for the post-condition which has **NOT** as the top level operator and is in the form **NOT(A AND B).** It returns **TRUE** if the verified post-condition is an instance of **NOT** operator and is also an instance of **WsAnd**.

The ***execute()*** method uses **replacePostCondition(postCond, exp1, exp2)** which takes the input post-condition, the right and left hand side operands(expressions) and replaces the post-condition with instance of **WsOr** with exp1 and exp2 as its operands and transforms to **NOT A OR NOT B**.

```
package testing_Not_Simplification is
public procedure not_simplification10()
     assumes True
     guarantees (not(a and b))
end class;
end package
```

Figure 5.33 Input AWL specification for NOT(A AND B)

```
package testing_Not_Simplification is
public procedure not_simplification10()
     assumes True
     guarantees (not a or not b)
end class;
end package
```

Figure 5.34 output AWL file for NOT(A AND B)

XNotAbhinav11:

Function: This transform transforms the post-condition of the form

NOT(A OR B) to NOT A AND NOT B.

The ***applicable()*** method checks for the post-condition which has **NOT** as the top level operator and is in the form **NOT(A OR B).** It returns **TRUE** if the verified post-condition is an instance of **NOT** operator and is also an instance of **WsOr**.

The ***execute()*** method uses **replacePostCondition(postCond, exp1, exp2)** which takes the input post-condition, the right and left hand side operands(expressions) and replaces the post-condition with instance of **WsAnd** with exp1 and exp2 as its operands and transforms to **NOT A AND NOT B**.

```
package testing_Not_Simplification is
public procedure not_simplification11()
     assumes True
     guarantees (not(a or b))
end class;
end package
```

Figure 5.35 Input AWL specification for NOT(A OR B)

```
package testing_Not_Simplification is
public procedure not_simplification11()
     assumes True
     guarantees (not a and not b)
end class;
end package
```

Figure 5.36 output AWL file for NOT(A OR B)

All the transforms cited so far categorize the post-conditions to actions or constraints. But, the post-conditions which have an *equals* operator remain uncategorized. The next task is to categorize the post-conditions which contain the *equals* operator to action or constraints and then test their dependency. A button is added in **ToolMethod1** which will call **doAbhinav0a()** function to perform the required task.

doAbhinav0a:

We have three vectors **VEquals, VConstraint, VAction** which will store the required post-conditions. If the post-condition is an instance of *WsEqual* it is moved to the **VEquals** vector. The library functions ***isConstraint(post-condition)*** and ***isAction(post-condition)*** are used to move the post-conditions to their respective **VConstraint** or **VAction** vectors.

The second step involves checking the dependency among the post-conditions in the **VEquals** vector. For each **VEquals** post-condition we check the dependence against all the remaining post-conditions using ***areDependent(exp1,exp2)*** library function. If they are independent of all other post-conditions they are further examined to be an action or constraint and removed from **VEquals** vector**.** The **VEquals** vector then contains the remaining post-conditions where are recursively checked against all other post-conditions for dependency. If they are dependent we call the conjunct transform and replace both the post-conditions in conjucted form in the **VEquals** vector. The post-condition left in the **VEquals** vector is a fully conjucted, independent post-condition and is moved to **VAction** vector. The **VEquals** should be empty at this stage. Now each of the post-conditions in **VAction** call transforms to covert them to code.

At this stage all the actions and constraints are separated in to two different vectors. The weakest-precondition analysis is performed on the constraints to get the weakest pre-condition.

WEAKEST PRE-CONDITION

The statements in the methods consist of assignment statements, sequence of assignment statements, if statements and if else statements. This thesis aims at determining the weakest-precondition for these allowable program statements.

Using the weakest-precondition algorithm as discussed in chapter 4, we define a local variable **x_pre** for each ticked variable x' and an assignment statement **x_pre:=x** is added at the beginning of the program. A transform class **XAddLocalAbhinav** is designed to perform the necessary task.

XAddLocalAbhinav:

Function: Adds a local variable "_pre" to the unprimed variable and changes the variable " x' " to x. This transform also adds an assignment statement at the begining of the method. It calls the XWpAbhinav class and adds the weakest pre-condition to the subprogram.

The class iterates through the post conditions of the method to change the expression of the type x'=x+1 to x = x_pre+1.It adds an assignment x_pre=x at the begining of the method. The *applicable()* makes sure that there is at least one selected post-condition and the *execute()* function defines "_pre" variable for each ticked variable in the post-condition. The duplicates of ticked variables are checked and are removed by **getLocalVar(method ,string_name)** where **string_name** is the name of the duplicate variable present in the **method.** Then the name and type of the **WsVariable** to be added is set and the required assignment statement is added to the AWL file using **WsAssignment** class. A copy visitor is used to copy the initial tree.

ChangeRefNewVisitor class changes the **WsTick** variable to AWSOME syntax i.e changes the *ticked* variable to the *unticked* version as discussed in the weakest pre-condition algorithm discussed in chapter 4. This class uses the **visitBinary(WsBinaryExpression, Object)** and **visitUnary(WsUnaryExpression, Object)** functions which consider unary and binary expressions and search through the post-condition and replace the *ticked* version of the variable with the *unticked* version using the **visit(WsIdentifierRef, Object)** function.

ChangeRefNameVisitor class changes the variable to "_pre" version as discussed in the weakest pre-condition algorithm. This class uses the **visitBinary(WsBinaryExpression, Object)** and **visitUnary(WsUnaryExpression, Object)** functions which will change the name of the required variable to "_pre" version. This class can handle both unary and Binary expressions. Once the assignment statements are created we proceed with the weakest-precondition analysis by moving the post-condition back through each statement. The function **WeakestPrecondition(Methods body,exp)** is called which will determine the weakest pre-condition. There are three allowable program statements that have been implemented in this thesis.

XTypeNameAbhinav class gives the type name of the class variable. The **getTypeName(WsClass ,String)** function returns the typr of the variable by taking the class name and the name of the variable.

XWpAbhinav:

Function: Adds the weakest pre-condition at the beginning of the method by transforming the post-condition.

. All the allowable program statements are considered and transforms are designed to handle them. We use three different methods to determine the weakest pre-condition. They are

1. **WeakestPrecondition(WsAssignment,WsExpression)**

2. **WeakestPrecondition(Vector,WsExpression)**

3. **WeakestPrecondition(WsSelection,WsExpression)**

1.WeakestPrecondition(Vector,WsExpression)

Function: Determines the weakest-precondition for sequence of statements.

A sequence of post-conditions is handled by this function. It takes a **Vector** and **WsExpression** as the input. If the input has a WsAssignment in the vector it calls the **WeakestPrecondition(WsAssignment,WsExpression)** function and it handles the assignment statements as discussed below. If the input is a **WsSelection(discussed in next part)** it calls the **WeakestPrecondition(WsSelection,WsExpression)** function which will determine the weakest pre-condition for selection statements. If the input is a vector itself it recursively calls itself and then depending on the input the assignment or the selection functions are called.

```
if(ws instanceof WsAssignment)

current=WeakestPrecondition((WsAssignment)ws,current);

else if(ws instanceof WsSelection)

  current=WeakestPrecondition((WsSelection)ws,current);

else if(ws instanceof Vector)
```

Figure5.37 code for sequence of statements

The *instanceof()* operator is used to determine the statements to be an assignment, sequence of statements or selection statements. Once the input statement is selected it is passed to *current* which is returned by the function.

2. WeakestPrecondition(WsAssignment, WsExpression):

Function: Determines the weakest-precondition for assignment statements.

The assignment statements LHS is a **WsName** and RHS is a **WsExpression.** The assignment statements LHS is searched for in the entire expression and if it is found it is replaced by the corresponding RHS WsExpression.

```
WsName rName=wa.getWsAssignLHS();

 WsExpression eExp=wa.getWsAssignRHS();

      WsIdentifierRef rRef=(WsIdentifierRef)rName;

      String LhsName=rRef.getName();

      //System.out.println("LHS NAME:"+LhsName);

WsVisitor ne=new FindRefNameVisitor(LhsName,eExp);  e.acceptVisitor(ne,null);
```

Figure 5.38 sample code for Assignment statement

FindRefNameVisitor class searches for the LHS side of the expression which needs to be replaced with the RHS of the expression to determine the weakest pre-condition. This class uses **visitBinary(WsBinaryExpression , Object)** and **visitUnary(WsUnaryExpression , Object)** to find the instance of the variable and replace it with the expression to its RHS.

FindRefNameVisitor class searches for the LHS side of the expression which needs to be replaced with the RHS of the expression to determine the weakest pre-condition. This class uses **visitBinary(WsBinaryExpression , Object)** and **visitUnary(WsUnaryExpression , Object)** to find the instance of the variable and replace it with the expression to its RHS.

```
public procedure test_weakest_precondition1()
      assumes True
      guarantees x' < x + 1 and y' < m + 1 and  z'<z+1 and y'<y+1 and y>k+1
      is
      temp: Integer;
      begin
      temp:= x+1;
      x:=temp;
      end;
```

Figure 5.39 Input AWL file for Assignment statements

The input AWL file consists of the sequence of assignment statements. The weakest pre-condition of the assignment statement x:=temp is determined by calling the **WeakestPrecondition(Vector,WsExpression)** that it in turn calls **WeakestPrecondition(WsAssignment, WsExpression).** This weakest pre-condition becomes the post-condition for temp:=x+1. The weakest pre-condition of x:=temp+1 is determined. The same process is continued until we are out of the assignment statements in the body. The resulting output is displayed in Figure 5.40

```
public procedure test_weakest_precondition1()
      assumes True and ((x + 1) < (x + 1 + (1))) and (y < m + 1) and (z < z + 1)
               and (y < y + 1) and (y > k + 1)
      guarantees (x' < x + 1) and (y' < m + 1) and (z' < z + 1) and (y' < y + 1)
               and (y > k + 1)
      is
        temp : Integer;
        x_pre : Integer;
        y_pre : Integer;
        z_pre : Integer;
      begin
        z_pre := z;
        y_pre := y;
        x_pre := x;
        temp := x + 1;
        x := temp;
      end;
```

Figure 5.40 : Ouput of weakest precondition for sequence and assignment statements

3.WeakestPrecondition(WsSelection,WsExpression)

Function: Determines the weakest pre-condition for selection statements.

This function handles two types of **WsSelection**. The first type includes the else part and the second type does not contain the else part. This function stores the condition of **WsSelection** in **WsExpression** and it contains two vectors where one of them will take the *then* part and the other takes the *else* part. As the syntax specified

wp(if B S1 else S2 , Q) ≡ B => **wp(S1, Q) AND (NOTB)** => **wp(S2, Q**)

it creates two implies objects. The first implies **B** => **wp(S1, Q)** takes the input condition **B** as the left operand and the right operand is the weakest pre-condition of the assignment statement in the than part. The **WeakestPrecondition(WsAssignment, WsExpression) function is called** is called and the pre-condition is determined. The second implies **(NOTB)** => **wp(S2, Q**) has WsNot with the condition set as the left operand and the right operand is the weakest pre-condition of the assignment statement in the else part. This also calls the **WeakestPrecondition(WsAssignment, WsExpression)** function to determine the weakest pre-condition. In some cases the selection statement doesn't have an else part or if can have nested selection statements. These have been handled by the condition **if(elseParts.size()>0).** If else part is not present then the weakest pre-condition becomes **wp(S1, Q) AND NOT B => Q.** It is similar to the one with else part but does not determine the weakest pre-condition for the second implies. Nested selection statements have also been handled by calling the function recursively.

The input and output of the selection program statements are shown in figures 5.41 and 5.42.

```
public procedure test_weakest_precondition2()
      guarantees tax'<10000
          is
          begin
      if income < 10000.00 then
       tax:=0.15 * income;
          end if;
      end;


    public procedure test_weakest_precondition3()
      guarantees tax'<10000
          is
          begin
      if income < 10000.00 then
       tax:=0.15 * income;
           else
          tax:=0.50 * income;
      end if;
      end;
```

Figure 5.41 Input AWL file for Selection statements

```
     public procedure test_weakest_precondition4()
       guarantees tax'<10000
            is
            begin
       if income < 10000.00 then
        tax:=0.15 * income;
              else
            if income > 10000.00 then
             tax:=0.25 * income;
              else
             tax:=0.50 * income;
       end if;
            end if;
       end;
```

Figure 5.41 continued….

```
public procedure test_weakest_precondition2()
         assumes (income < 10000.0) => ((0.15 * income) < (10000)) and not (income <
10000.0) => tax < 10000
         guarantees tax' < 10000
         is
            tax_pre : Float;
         begin
            tax_pre := tax;
            if income < 10000.0 then
               tax := 0.15 * income;
            else
            end if;
         end;

      public procedure test_weakest_precondition3()
         assumes (income < 10000.0) => ((0.15 * income) < (10000)) and not (income <
10000.0) => (0.5 * income) < (10000)
         guarantees tax' < 10000
         is
            tax_pre : Float;
         begin
            tax_pre := tax;
            if income < 10000.0 then
               tax := 0.15 * income;
            else
               tax := 0.5 * income;
            end if;
         end;
```

Figure 5.42 Output AWL specification for WsSelection

```
public procedure test_weakest_precondition4()
      assumes (income < 10000.0) => ((0.15 * income) < (10000)) and not (income <
10000.0) => (income > 10000.0) => ((0.25 * income) < (10000)) and not (income >
10000.0) => (0.5 * income) < (10000)
      guarantees tax' < 10000
      is
        tax_pre : Float;
      begin
        tax_pre := tax;
        if income < 10000.0 then
          tax := 0.15 * income;
        else
          if income > 10000.0 then
            tax := 0.25 * income;
          else
            tax := 0.5 * income;
          end if;
        end if;
      end;
```

Figure 5.42 Continued

SUMMARY

This chapter describes the testing environment and the testing approach used. Appropriate test cases are developed and all the transforms are tested. The results obtained show that the transforms meet the desired design goals.

6. CONCLUSIONS AND FUTURE WORK

The purpose of this thesis is to design transforms that can determine dependent and independent post-conditions and then categorize them to actions and constraints and then determine the weakest pre-condition. An object oriented approach is followed in this thesis. The transforms that are needed are first identified and then appropriately designed. Implementation followed the design and the result is a set of working transforms that can be applied to post-conditions. The transforms that are considered simple to implement were designed first and then the more complex ones were designed.

The first two chapters introduce the reader to AWSOME and AWL and provide the background information necessary to understand the rest of the document. The analysis chapter discusses the dependent and independent post-conditions, actions, constraints, Boolean operators in AWL, NOT transforms and weakest pre-condition analysis. This thesis focuses on differentiating the post-conditions as dependent and independent post-conditions and then further dividing them in to actions or constraints. The weakest pre-condition analysis is performed on constraints so that the post-conditions are converted into pre-conditions. The design chapter outlines the transforms that were needed. The functionality of all the transforms is discussed in the design chapter. Following this design all the transforms were implemented and carefully tested. The implementation and testing chapter acquaints the reader with the testing environment and the approach used to test the transforms. Appropriate test cases were written in AWL for testing the transforms. These test cases, along with the output obtained on applying the transforms, are presented in the testing chapter. Table 6.1 lists the transforms developed and their functionality.

Table 2: Summary of classes, Transforms and its description.

CLASS	DESCRIPTION
XgetVarAbhinav	Gets the names of the primed and unprimed variables from a post-condition and places them in different vectors.
XDependentAbhinav	Returns true if *areDependent(exp1,exp2)* function returns true. The primed and unprimed variables of the post-conditions are passed in as separate vectors and depending on the **CONDITION RULES**
XConstraintAbhinav	Determines a specific post-condition to be a constraint. *isConstraint(exp)* returns true if the post-condition is constraint..
XActionAbhinav	Determines the specific post-condition to be an Action. *isActiont(exp)* returns true if the post-condition is an action.

TRANSFORM	**DESCRIPTION**
XOrAbhinav	Transforms the post-condition of the form **P OR NOT P** to **TRUE** postCond is the vector that contains post-condition to be replaced.This function transforms **P OR NOT P** to **TRUE.**
XImpliesAbhinav1	Transforms the post-condition of the form **NOT P IMPLIES NOT Q** to **Q IMPLIES P.**
XImpliesAbhinav2	Transforms the post-condition of the **P IMPLIES NOT P** to **TRUE**
XNotAbhinav1	Transforms the post-condition of the form **NOT(A<B) to A>=B**
XNotAbhinav2	Transforms the post-condition of the form **NOT(A<=B) A>B**
XNotAbhinav3	Transforms the post-condition of the form **NOT(A>B) to A<=B**
XNotAbhinav4	Transforms the post-condition of the form **NOT(A>=B) to A<B**
XNotAbhinav5	Transforms the post-condition of the form **NOT(A=B) to A/=B**
XNotAbhinav6	Transforms the post-condition of the form **NOT(A/=B) to A=B**
XNotAbhinav7	Transforms the post-condition of the form **NOT(NOT A) to A**

XNotAbhinav8	Transforms the post-condition of the form **FORALL (X: MYINT) (EXP1) to EXISTS (X:MYINT) (NOT(EXP1)**
XNotAbhinav9	Transforms the post-condition of the form **EXISTS (X: MYINT) (EXP1) to FORALL (X:MYINT) (NOT(EXP1)**
XNotAbhinav10	Transforms the post-condition of the form **NOT (A AND B) to NOT A OR NOT B**
XNotAbhinav11	Transforms the post-condition of the form **NOT (A OR B) to NOT A AND NOT B**
XAddLocalAbhinav	Adds a local variable "_pre" to the unprimed variable and changes the variable " x' " to x. This transform also adds an assignment statement at the begining of the method. it also adds the weakest pre-condition to the subprogram.
XWpAbhinav	Determines and adds the weakest pre-condition at the starting of the method
ChangeRefNameVisitor	This Visitor traverses through all the hierarchy of the expression and changes the name of the oldvariable to “_pre” version.
FindRefNameVisitor	This Visitor traverses through all the hierarchy of the expression and finds the variable which has to be replaced with the RHS i.e the expression in weakest pre-condition
ChangeRefNewVisitor	This Visitor traverses through all the hierarchy of the expression and if it finds the name it is replaced by new name.

FUTURE WORK

For this thesis, dependency among the post-conditions is determined. We can say that two post-conditions are dependent, but the dependency relationship is not established. We can say that p1 and p2 are dependent, but we cannot determine that p1 is dependent on p2 or p2 is dependent on p1.

Transforms are developed to determine the weakest pre-condition. All the allowable program statements have been designed and implemented except procedure calls and loop invariants. The necessary background work is done but they need to be implemented. More work has to be done to transform them accordingly.

The weakest pre-condition is determined in a straight-forward way, and no simplification is done. Transforms are needed to simplify the results.

The work done in this thesis combined with the past and future works can perfect the AWSOME system and provide us with a transformation system that can successfully transform the formal specification model into executable code.

REFERENCES

[1] T. C. Hartrum and R. P. Graham, Jr., "The AFIT Wide Spectrum Object Modeling Environment: an AWSOME Beginning," *Proceedings of the IEEE 2000 National Aerospace & Electronics Conference (NAECON 2000)*, Dayton, OH, October 2000.

[2] R. P. Graham, Jr. and T. C. Hartrum, *AWSOME Wide-Spectrum Language*(AWL), Air Force Institute of Technology, Language Reference Manual, Edition 1.0, July 14th 2003, unpublished.

[3] T. C. Hartrum, *AWSOME Transforms*, Wright State University, Reference Manual, January 17th 2001, unpublished.

[4] T. C. Hartrum, *AWSOME Tool Box*, Wright State University, Reference Manual, Edition 1.0, January 18th 2005, unpublished.

[5] R. Balzer, T. E. Cheathan, Jr., and C. Green, "Software Technology in the 1990's: Using a New Paradigm", *IEEE Computer*, November, 1983.

[6] Parvathaneni, Swetha, *Transformation of Formally Specified Post-Conditions into Target Language Statements*, Masters Thesis, Wright State University, Dayton, OH, 2007.

[7] Manubolu, Pratap, *Transformation of Formally Defined User Data Types Into a Target Language*, Masters Thesis, Wright State University, Dayton, OH, 2006.

[8] Singam, Nagarjuna, *Automated Code Generation From a Formally Specified Post-Condition*, Masters Thesis, Wright State University, Dayton, OH, 2005.

[9] Bhooma Raghunathan, *Automated code synthesis from a formal state machine model.* Wright State University, Dayton OH, Masters Thesis, 2004.

[10] Preeti Subhedar, *Automated design transforms for the object-oriented structural model.* Wright State University, Dayton OH, Masters Thesis, 2005.

[11] Sarvepalli, Venkata, *Automated Design Transforms From a Formally Specified Class Definition*, Masters Thesis, Wright State University, Dayton, OH, 2005

[12] J.B Wordsworth, *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*, IBM United Kingdom Ltd., Addison Wesley 1994.

[13] James Rumbaugh, et al., *Object Oriented Modeling and Design*, Prentice Hall, Inc., 1991.

[14] B. Bruegge, and A.H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterens, and Java*, 2nd ed., Upper Saddle River: Prentice hall, 2004.

[15] Erich Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[16] Nikolaj Popov, *Verification Using Weakest Precondition Strategy*, Research Institute for Symbolic Computation,4232 Hagenberg, Austria.

[17] Wikipedia : Online encyclopedia (date: 04/16/2006)
http://en.wikipedia.org/wiki/Method_%28computer_science%29

Appendix

Java Code

The following section includes just the most important methods in the Transforms and leaves out the full source code. For each of the transforms, the *applicable* and the *execute* methods are outlined in this section. The structure of the complete source code is discussed in design chapter.

```
/*********************************************
* Source: XgetVarAbhinav.java
* FileName :This class gets the primed and
            unprimed variables from the *post-conditions.
/***************************************************\
/*******************************************************************
******
This function returns a vector containing all the unprimed variables
/*******************************************************************
******
public static Vector returnUnPrimeVar(WsExpression exp, boolean tick)
{
Vector v=new Vector();
boolean found = false;
String tempName = null;
if(exp instanceof WsBinaryExpression)
{
Vector v1=returnUnPrimeVar(((WsBinaryExpression)exp).getWsBinExpOp1(),
tick);
//System.out.println("VISITED OP1 SUCCESS");
Vector v2=returnUnPrimeVar(((WsBinaryExpression)exp).getWsBinExpOp2(),
tick);
//System.out.println("VISITED OP2 SUCCESS");
//Hartrum - Remove duplicates.
for (int i = 0; i < v1.size(); i++)
{
tempName = (String) v1.get(i);
found = false;
for (int j = 0; j < v.size(); j++)
{
if (tempName.equals((String) v.get(j)))
found = true;
}//for all v
if (!found)
v.add(tempName);
}//for all v1
for (int i = 0; i < v2.size(); i++)
{
tempName = (String) v2.get(i);
found = false;
```

```
for (int j = 0; j < v.size(); j++)
{
if (tempName.equals((String) v.get(j)))
found = true;
}//for all v
if (!found)
v.add(tempName);
}//for all v2
}//WsBinaryExpression
else if (exp instanceof WsUnaryExpression)
v=returnUnPrimeVar(( (WsUnaryExpression) exp).getWsUnaryExpOp(), tick);
else  if( exp instanceof WsIdentifierRef)
{
//System.out.println("NOW IN IDENTIFIER REF");
v.add(((WsIdentifierRef)exp).getName());
//System.out.println("AFTER CREATING IR");
}//WsIdentifierRef

//Added - Hartrum
else if (exp instanceof WsQuantifiedExpression)
v=returnUnPrimeVar(( (WsQuantifiedExpression)
exp).getWsQuantExpConstraint(), tick);
else if (exp instanceof WsContainerFormer)
v=returnUnPrimeVar(( (WsContainerFormer)
exp).getWsContainerConstraint(), tick);
else if (exp instanceof WsLiteralContainer)
{
Vector v1 = ((WsLiteralContainer) exp).getWsLitContainerElements();
for (int k = 0; k < v1.size(); k++)
{
Vector v2=returnUnPrimeVar( (WsExpression) v1.get(k), tick);
for (int i = 0; i < v2.size(); i++)
{
tempName = (String) v2.get(i);
found = false;
for (int j = 0; j < v.size(); j++)
{
if (tempName.equals((String) v.get(j)))
  found = true;
}//for all v
if (!found)
v.add(tempName);
}//for all v2
}//for all k
}//WsLiteralContainer
else if (exp instanceof WsFunctionCall)
{
Vector v1 = ((WsFunctionCall) exp).getWsSubprogCallArgs();
for (int k = 0; k < v1.size(); k++)
{
Vector v2=returnUnPrimeVar( (WsExpression) v1.get(k), tick);
for (int i = 0; i < v2.size(); i++)
{
tempName = (String) v2.get(i);
found = false;
for (int j = 0; j < v.size(); j++)
{
```

```
if (tempName.equals((String) v.get(j)))
  found = true;
}//for all v
if (!found)
v.add(tempName);
}//for all v2
}//for all k
}//WsFunctionCall
else if (exp instanceof WsAccess)
v=returnUnPrimeVar(( (WsAccess) exp).getWsAccessObject(), tick);
else if (exp instanceof WsTypeConversion)
v=returnUnPrimeVar(( (WsTypeConversion) exp).getWsTypeConvExp(), tick);
else  if( exp instanceof WsDereference)
v=returnUnPrimeVar(( (WsDereference) exp).getWsDerefName(), tick);
else if (exp instanceof WsIndexedComponent)
{
v=returnUnPrimeVar(( (WsIndexedComponent) exp).getWsIndxCompName(),
tick);
Vector v1=returnUnPrimeVar(( (WsIndexedComponent)
exp).getWsIndxCompIndex(), tick);
for (int i = 0; i < v1.size(); i++)
{
tempName = (String) v1.get(i);
found = false;
for (int j = 0; j < v.size(); j++)
{
if (tempName.equals((String) v.get(j)))
found = true;
}//for all v
if (!found)
v.add(tempName);
}//for all v1
}//WsIndexedComponent
else  if( exp instanceof WsSelectedComponent)
v=returnUnPrimeVar(( (WsSelectedComponent) exp).getWsSelCompName(),
tick);
else  if( exp instanceof WsThis)
v.add(((WsThis)exp).getName());
else  if( exp instanceof WsTick)
{
if (tick)
v.add(((WsTick)exp).getName());
}//tick
return v;
}//returnUnPrimeVar



/****************************************************************
This function Returns a vector containg all the primed variables
/*****************************************************************
public static Vector returnPrimeVar(WsExpression exp)
{
Vector v =returnPrUnPrimeVar(exp,true);
Vector vp= new Vector();
for (int k = 0; k < v.size(); k++)
{
```

```
String tempName = (String) v.get(k);
String tempName1=null;
int mytick;
mytick=tempName.indexOf("'");
if(mytick!=-1)
{
tempName1=tempName.substring(0,mytick);
   vp.add(tempName1);
}
}

return vp;

}//returnPrimeVar

Class XDependentAbhinav
/*******************************************************************
* Source file: XDependentAbhinav.java
* Purpose:Returns True if the post-conditions are dependent
*******************************************************************/
public static boolean areDependent(WsExpression exp1, WsExpression
exp2)
    {
            Vector primeexp1 = XgetVarAbhinav.returnPrimeVar(exp1);
            Vector primeexp2 = XgetVarAbhinav.returnPrimeVar(exp2);
            Vector unprimeexp1 =
XgetVarAbhinav.returnUnPrimeVar(exp1,false);
            Vector unprimeexp2 =
XgetVarAbhinav.returnUnPrimeVar(exp2,false);
             System.out.println("The primed set of exp1 is"+primeexp1);
             System.out.println("The unprimed set of exp1
is"+unprimeexp1);
             System.out.println("The primed set of exp2 is"+primeexp2);
             System.out.println("The unprimed set of exp2
is"+unprimeexp2);

 boolean atleastOneIsThere = false;

           //check the intersection of both the primed sets is not null

              for (int i=0;i<primeexp1.size();i++)
              {
                        String tempName2=(String) primeexp1.get(i);

                  for(int j=0;j<primeexp2.size();j++)
                    {
                              if(tempName2.equals(primeexp2.get(j)))
                         {


                            atleastOneIsThere = true;
                            System.out.println(atleastOneIsThere);
                                   break;
                              }
                          }
                   }
```

```
                    boolean atleastOneUnPrimeIsThere = false;

                  //checks the intersection of unprimed set of exp1 and
primed set of exp2 is not null

                    for (int i=0;i<unprimeexp1.size();i++)
                      {
                           String tempName1=(String) unprimeexp1.get(i);

                         for(int j=0;j<primeexp2.size();j++)
                          {


                            if(tempName1.equals(primeexp2.get(j)))
                              {


                              atleastOneUnPrimeIsThere = true;
                              break;
                        }
                    }
              }


              boolean athah = false;

            //checks the intersection of primed set of exp1 and unprimed
set of exp2 is not null

              for (int i=0;i<primeexp1.size();i++)
              {

                     String tempName=(String) primeexp1.get(i);

                      for(int j=0;j<unprimeexp2.size();j++)
                   {
                               if(tempName.equals(unprimeexp2.get(j)))
                       {

                          athah = true;
                          break;
                       }
                   }
              }



              boolean uthah = false;


                //checks the intersection of primed set of exp1 and
unprimed set of exp2 is not null

                for (int i=0;i<unprimeexp1.size();i++)
                {

                     String tempName=(String) unprimeexp1.get(i);
                     for(int j=0;j<unprimeexp2.size();j++)
                         {
                               if(tempName.equals(unprimeexp2.get(j)))
                               {

                                  uthah = true;
                                  break;
                               }
                         }
            }




               //Returns true if any of them is true

              if(atleastOneIsThere || atleastOneUnPrimeIsThere ||athah
 || uthah)
               {

                         return true;
                 }
                         else

                   {

                         return false;
                   }


               }

 }
```

```
Class 3: XConstraintAbhinav

/**********************************************************************
* Source file: XConstraintAbhinav1.java
* Purpose:Returns True if the post-condition is a constraint
/*********************************************************************
/*******************************************************************
This functions returns true if the post-condition is a constraint
/*******************************************************************

public static boolean isConstraint(WsExpression exp1)
    {

      boolean constraint = false;
      if(exp1 instanceof WsClasses.WsLessThan)

        constraint=true;

        else if(exp1 instanceof WsClasses.WsLessThanOrEqual)
```

```
  constraint=true;
  else if(exp1 instanceof WsClasses.WsGreaterThan)
  constraint=true;
  else if(exp1 instanceof WsClasses.WsGreaterThanOrEqual)
  constraint=true;
  else if(exp1 instanceof WsClasses.WsEqual) {
Vector v1= XgetVarAbhinav.returnUnPrimeVar(exp1,false);
Vector v2= XgetVarAbhinav.returnPrimeVar(exp1);
       if(v1.size()==0 || v1.size()==1)
       constraint=true;
         else
         constraint=false;

}  else if(exp1 instanceof WsClasses.WsNotEqual)
  constraint=true;
  else if(exp1 instanceof WsClasses.WsNot)
  constraint=true;
  else if(exp1 instanceof WsClasses.WsIn)
  constraint=false;
  else if(exp1 instanceof WsClasses.WsImplication)
  constraint=false;
  else if(exp1 instanceof WsClasses.WsSubset)
  constraint=true;
  else if(exp1 instanceof WsClasses.WsAnd)
  constraint=true;
else if(exp1 instanceof WsClasses.WsOr)
  constraint=true;
  else if(exp1 instanceof WsClasses.WsUniversal)
  constraint=false;
```

```
        else if(exp1 instanceof WsClasses.WsProperSubset)

        constraint=true;

        else if(exp1 instanceof WsClasses.WsIdentifierRef)

        constraint=true;

        else

        constraint=false;

      return constraint;
    }
}

Class XActionAbhinav

/**********************************************************************
******
* Source file: XActionAbhinav.java
* Purpose:Returns true if the post-condition is an action
/**********************************************************************
******
/******************************************************************
This functions returns true if the post-condition is an action
/*********************************************************************
*

public static boolean isAction(WsExpression exp1)
    {

      boolean action;
      if(exp1 instanceof WsClasses.WsLessThan)

        action=false;

        else if(exp1 instanceof WsClasses.WsLessThanOrEqual)

        action=false;

        else if(exp1 instanceof WsClasses.WsGreaterThan)

        action=false;

        else if(exp1 instanceof WsClasses.WsGreaterThanOrEqual)

        action=false;

        else if(exp1 instanceof WsClasses.WsEqual)
      {

       Vector v1= XgetVarAbhinav.returnPrimeVar(exp1);
       Vector v2= XgetVarAbhinav.returnUnPrimeVar(exp1,false);

       if(v1.size()>=1 || v2.size()>0 )
```

```
        action=true;

        else

        action=false;

        }  else if(exp1 instanceof WsClasses.WsNotEqual)

         action=false;

         else if(exp1 instanceof WsClasses.WsNot)

         action=false;

         else if(exp1 instanceof WsClasses.WsIn)

         action=true;

         else if(exp1 instanceof WsClasses.WsImplication)

         action=true;

         else if(exp1 instanceof WsClasses.WsSubset)

         action=false;

         else if(exp1 instanceof WsClasses.WsAnd)

         action=false;

       else if(exp1 instanceof WsClasses.WsOr)

         action=false;

         else if(exp1 instanceof WsClasses.WsUniversal)

         action=true;

         else if(exp1 instanceof WsClasses.WsProperSubset)

         action=false;

         else if(exp1 instanceof WsClasses.WsIdentifierRef)

         action=false;

         else

         action=true;

       return action;
     }
}

Transform XOrAbhinav
```

```
/*******************************************************************
******
* Source file: XOrAbhinav.java
* Purpose:Transforms the post-condition of the form (P OR NOT P) to
TRUE
/*******************************************************************
******

/***************************************************************
 * checks to make sure that there is atleast one post condition of the
 * form (P OR NOT P)
 * Parameters:
/ **************************************************************/
public static boolean applicable(Object target, Object params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                  Vector postConditions =
subprog.getWsPostConditionSet();

                  WsExpression exp;

                  if(postConditions.size() > 0){
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp OR NOT exp)*/
                        for(int i=0; i < postConditions.size();i++){
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof WsClasses.WsOr){
             WsBinaryExpression BinaryExp = (WsBinaryExpression) exp;

             WsExpression e1= BinaryExp.getWsBinExpOp1();
            WsExpression e2= BinaryExp.getWsBinExpOp2();

             if(e2 instanceof WsClasses.WsNot)
             return true;


                              }
                        }

                  }
                  else
                        return false;


                  return false;

             } //End of applicable
```

```
/*************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form (P OR NOT P)
*
* Then it changes the post condition to TRUE
*
* Parameters: NONE
*
*************************************************************/
public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){

      exp = (WsExpression) postConditions.get(index);


      if(exp instanceof WsClasses.WsOr)
               {

                        WsBinaryExpression BinaryExp =
(WsBinaryExpression) exp;

                        WsExpression e1 = BinaryExp.getWsBinExpOp1();

                        WsExpression e2 = BinaryExp.getWsBinExpOp2();


                        if(e2 instanceof WsClasses.WsNot)

                        {


                        WsExpression e2Not =
((WsUnaryExpression)e2).getWsUnaryExpOp();

                             postConditions.remove(index);
                           replacePostCondition(postConditions,e2Not);


                      }


            }

         index++;

            }

return true;
}
/*******************************************************************
*
      * It creates a new expression of the form TRUE
      * using the parameters passed to it and adds it to the post-
condition
      * Vector of the target method.
/*******************************************************************
*/
      private void replacePostCondition(Vector
postCond,WsClasses.WsExpression exp2)
            {
                  WsClasses.WsIdentifierRef newIdentifier =  new
WsIdentifierRef("True");

                  postCond.add(newIdentifier);

                  WsSubprogram subprog =
targetmethod.getWsMethodSubprogram();

                  subprog.setWsPostConditions(postCond);

          }
```

Class XImpliesAbhinav1

```
/*******************************************************************
*****
* Source file: XImpliesAbhinav1.java
* Purpose:Transforms the post-conditions of the form NOT e1 -> NOT e2
to e2-> e1
/*******************************************************************
******

/****************************************************************
checks to make sure that there is atleast one post condition of the
form (NOT e1 -> NOT e2)
/****************************************************************/
             public static boolean applicable(Object target, Object
params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                  Vector postConditions =
subprog.getWsPostConditionSet();

                  WsExpression exp;

                  if(postConditions.size() > 0){
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp -> exp)*/
                        for(int i=0; i < postConditions.size();i++){
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof
WsClasses.WsImplication){
                  WsBinaryExpression BinaryExp = (WsBinaryExpression)
exp;

                  WsExpression e1= BinaryExp.getWsBinExpOp1();

                  WsExpression e2= BinaryExp.getWsBinExpOp2();

      if((e1 instanceof WsClasses.WsNot) && (e2 instanceof
WsClasses.WsNot))
                                                return true;


                              }
                        }

                  }
                  else
                        return false;


                  return false;

            } //End of applicable




/*****************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form (NOT e1 -> NOT e2).
*
* Then it changes the post condition to the form...
*     e2 -> e1
*
*
* Parameters: NONE
*
*****************************************************************/
public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){

      exp = (WsExpression) postConditions.get(index);
      if(exp instanceof WsClasses.WsImplication)
               {

                         WsBinaryExpression BinaryExp =
(WsBinaryExpression) exp;

                         WsExpression e1 = BinaryExp.getWsBinExpOp1();

                         WsExpression e2 = BinaryExp.getWsBinExpOp2();


      if(e1 instanceof WsClasses.WsNot && e2 instanceof
WsClasses.WsNot)

                         {


                         WsExpression e1Not =
((WsUnaryExpression)e1).getWsUnaryExpOp();

      WsExpression e2Not = ((WsUnaryExpression)e2).getWsUnaryExpOp();

                              postConditions.remove(index);

replacePostCondition(postConditions,e1Not,e2Not);

                       System.out.println(e1Not.toString());
                       System.out.println(e2Not.toString());



                   }


            }

         index++;

            }

return true;
}


      /*************************************************************
*******
      * It creates a new expression of the form e2 -> e1
      * using the parameters passed to it and adds it to the post-
condition
      * Vector of the target method.
      *************************************************************
******/
      private void replacePostCondition(Vector
postCond,WsClasses.WsExpression exp1,WsClasses.WsExpression exp2)
      {
            WsClasses.WsImplication newImplication =  new
WsImplication(exp2, exp1);
```

```
            postCond.add(newImplication);

            WsSubprogram subprog =
targetmethod.getWsMethodSubprogram();

            subprog.setWsPostConditions(postCond);

      }

Class XimpliesAbhinav2
/*******************************************************************
******
* Source file: XImpliesAbhinav2.java
* Purpose: Transforms the post-condition of the form e1 -> NOT e2 to
TRUE
/*******************************************************************
******

/****************************************************************
checks to make sure that there is atleast one post condition of the
form (e1 -> NOT e2)
/ ***************************************************************/
             public static boolean applicable(Object target, Object
params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                  Vector postConditions =
subprog.getWsPostConditionSet();

                  WsExpression exp;

                  if(postConditions.size() > 0){
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp -> exp)*/
                        for(int i=0; i < postConditions.size();i++){
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof
WsClasses.WsImplication){
             WsBinaryExpression BinaryExp = (WsBinaryExpression) exp;

            WsExpression e1= BinaryExp.getWsBinExpOp1();

            WsExpression e2= BinaryExp.getWsBinExpOp2();

                  if((e2 instanceof WsClasses.WsNot))
                                          return true;


                              }
                        }
```

```
                }
                else
                      return false;


                return false;

            } //End of applicable




/*************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form (e1 -> NOT e2).
*
* Then it changes the post condition to the form...
*     TRUE
*************************************************************/
public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){

      exp = (WsExpression) postConditions.get(index);


      if(exp instanceof WsClasses.WsImplication)
               {

                         WsBinaryExpression BinaryExp =
(WsBinaryExpression) exp;

                         WsExpression e1 = BinaryExp.getWsBinExpOp1();

                         WsExpression e2 = BinaryExp.getWsBinExpOp2();


                         if(e2 instanceof WsClasses.WsNot)

                         {

           WsExpression e2Not =
((WsUnaryExpression)e2).getWsUnaryExpOp();
                     postConditions.remove(index);
                           replacePostCondition(postConditions,e2Not);

                       }
            }

         index++;

            }

return true;
}


      /***************************************************************
*******
      * It creates a new expression of the form TRUE
      * using the parameters passed to it and adds it to the post-
condition
      * Vector of the target method.
      ***************************************************************
******/
      private void replacePostCondition(Vector
postCond,WsClasses.WsExpression exp2)
      {
            WsClasses.WsIdentifierRef newIdentifier =  new
WsIdentifierRef("True");

            postCond.add(newIdentifier);

            WsSubprogram subprog =
targetmethod.getWsMethodSubprogram();

            subprog.setWsPostConditions(postCond);

      }


Transform XNotAbhinav1

/**********************************************************************
******
* Source file: XNotAbhinav1.java
* Purpose:Transforms the post-condition of the form NOT(A<B) to A>=B
/**********************************************************************
******
/*******************************************************************
checks to make sure that there is atleast one post condition of the
form NOT(A<B)
/*******************************************************************

public static boolean applicable(Object target, Object params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                  Vector postConditions =
subprog.getWsPostConditionSet();
```

```
                  WsExpression exp;

                  if(postConditions.size() > 0){
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp -> exp)*/
                        for(int i=0; i < postConditions.size();i++){
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof WsClasses.WsNot){

                                    WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                                    WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                        if(e1Unary instanceof WsClasses.WsLessThan){

                                          return true;
                               }



                              }
                        }

                  }
                  else
                        return false;


                  return false;

             } //End of applicable

/******************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form NOT(A<B)
* Then it changes the post condition to the form...A>=B
/*******************************************************************
public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){

      exp = (WsExpression) postConditions.get(index);
      if(exp instanceof WsClasses.WsNot)
               {
                        WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                        WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                 if(e1Unary instanceof WsClasses.WsLessThan){

                        WsBinaryExpression BinaryExp=
(WsBinaryExpression) e1Unary;

                        WsExpression e1Binary=
BinaryExp.getWsBinExpOp1();

                        WsExpression e2Binary=
BinaryExp.getWsBinExpOp2();

                      postConditions.remove(index);


replacePostCondition(postConditions,e1Binary,e2Binary);



                 }


             }

         index++;

             }

return true;
}
/*********************************************************************
*
      * It creates a new expression of the form A>=B
      * using the parameters passed to it and adds it to the post-
condition
      * Vector of the target method.
*********************************************************************
/
      private void replacePostCondition(Vector
postCond,WsClasses.WsExpression exp1,WsClasses.WsExpression exp2)
      {
            WsClasses.WsGreaterThanOrEqual newGreater =  new
WsGreaterThanOrEqual(exp1, exp2);

            postCond.add(newGreater);

            WsSubprogram subprog =
targetmethod.getWsMethodSubprogram();

            subprog.setWsPostConditions(postCond);
      }


Transform XNotAbhinav2

/**********************************************************************
******
* Source file: XNotAbhinav2.java
* Purpose:Transforms the post-condition of the form NOT(A<=B) A>B
/**********************************************************************
******
/*****************************************************************
 checks to make sure that there is atleast one post condition of them
form NOT(A<=B) A>B
/*****************************************************************

public static boolean applicable(Object target, Object params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                  Vector postConditions =
subprog.getWsPostConditionSet();

                  WsExpression exp;

                  if(postConditions.size() > 0){
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp -> exp)*/
                        for(int i=0; i < postConditions.size();i++){
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof WsClasses.WsNot){

                                    WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                                    WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                        if(e1Unary instanceof
WsClasses.WsLessThanOrEqual){

                                          return true;
                               }



                              }
                        }

                  }
                  else
```

```
                        return false;


                  return false;

             } //End of applicable

/*****************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form NOT(A<=B)
*
* Then it changes the post condition to the form A>B
/*****************************************************************
public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){

      exp = (WsExpression) postConditions.get(index);


      if(exp instanceof WsClasses.WsNot)
             {
                         WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                         WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                 if(e1Unary instanceof WsClasses.WsLessThanOrEqual){

                         WsBinaryExpression BinaryExp=
(WsBinaryExpression) e1Unary;

                         WsExpression e1Binary=
BinaryExp.getWsBinExpOp1();

                         WsExpression e2Binary=
BinaryExp.getWsBinExpOp2();

                       postConditions.remove(index);


replacePostCondition(postConditions,e1Binary,e2Binary);



                }
```

```
            }

         index++;

            }

return true;
}


      /**************************************************************
*******
      * It creates a new expression of the form A>B
      * using the parameters passed to it and adds it to the post-
condition
      * Vector of the target method.
      **************************************************************
******/
      private void replacePostCondition(Vector
postCond,WsClasses.WsExpression exp1,WsClasses.WsExpression exp2)
      {
            WsClasses.WsGreaterThan newGreater =  new
WsGreaterThan(exp1, exp2);

            postCond.add(newGreater);

            WsSubprogram subprog =
targetmethod.getWsMethodSubprogram();

            subprog.setWsPostConditions(postCond);

      }
```

Transform XNotAbhinav3

```
/*******************************************************************
******
* Source file: XNotAbhinav3.java
* Purpose:Transforms the post-condition of the form NOT(A>B) to A<=B
/*******************************************************************
******

/******************************************************************
checks to make sure that there is atleast one post condition of theform
NOT(A>B)
/******************************************************************

public static boolean applicable(Object target, Object params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                  Vector postConditions =
subprog.getWsPostConditionSet();
```

```
                  WsExpression exp;

                  if(postConditions.size() > 0){
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp -> exp)*/
                        for(int i=0; i < postConditions.size();i++){
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof WsClasses.WsNot){

                                    WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                                    WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                        if(e1Unary instanceof WsClasses.WsGreaterThan){

                                          return true;
                               }



                              }
                        }

                  }
                  else
                        return false;


                  return false;

             } //End of applicable

/*****************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form NOT(A>B)
*
* Then it changes the post condition to the form...
*     A<=B
/******************************************************************

public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){

      exp = (WsExpression) postConditions.get(index);
      if(exp instanceof WsClasses.WsNot)
               {
                         WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                         WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                 if(e1Unary instanceof WsClasses.WsGreaterThan){

                         WsBinaryExpression BinaryExp=
(WsBinaryExpression) e1Unary;

                         WsExpression e1Binary=
BinaryExp.getWsBinExpOp1();

                         WsExpression e2Binary=
BinaryExp.getWsBinExpOp2();

                       postConditions.remove(index);


replacePostCondition(postConditions,e1Binary,e2Binary);



                }


            }

         index++;

            }

return true;
}


      /**************************************************************
*******
      * It creates a new expression of the form A<=B
      * using the parameters passed to it and adds it to the post-
condition
      * Vector of the target method.
      **************************************************************
******/
      private void replacePostCondition(Vector
postCond,WsClasses.WsExpression exp1,WsClasses.WsExpression exp2)
      {
            WsClasses.WsLessThanOrEqual newLesser =  new
WsLessThanOrEqual(exp1, exp2);

            postCond.add(newLesser);
            WsSubprogram subprog =
targetmethod.getWsMethodSubprogram();

            subprog.setWsPostConditions(postCond);

      }
```

Transform XNotAbhinav4

```
/*******************************************************************
******
* Source file: XNotAbhinav4.java
* Purpose:Transforms the post-condition of the form NOT(A>=B) to A<B
/*******************************************************************
******
/******************************************************************
checks to make sure that there is atleast one post condition of theform
NOT(A>=B)
/******************************************************************

public static boolean applicable(Object target, Object params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                  Vector postConditions =
subprog.getWsPostConditionSet();

                  WsExpression exp;

                  if(postConditions.size() > 0){
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp -> exp)*/
                        for(int i=0; i < postConditions.size();i++){
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof WsClasses.WsNot){

                                    WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                                    WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                        if(e1Unary instanceof
WsClasses.WsGreaterThanOrEqual){

                                          return true;
                               }



                              }
                        }
                  }
                  else
                        return false;


                  return false;

             } //End of applicable

/*************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form NOT(A>=B)
*
* Then it changes the post condition to the form to A<B
/*************************************************************

public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){

      exp = (WsExpression) postConditions.get(index);


      if(exp instanceof WsClasses.WsNot)
               {
                         WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                         WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                 if(e1Unary instanceof WsClasses.WsGreaterThanOrEqual){

                         WsBinaryExpression BinaryExp=
(WsBinaryExpression) e1Unary;

                         WsExpression e1Binary=
BinaryExp.getWsBinExpOp1();

                         WsExpression e2Binary=
BinaryExp.getWsBinExpOp2();

                       postConditions.remove(index);


replacePostCondition(postConditions,e1Binary,e2Binary);
```

```
                 }


             }

          index++;

             }

return true;
}


      /************************************************************
*******
      * It creates a new expression of the form A<B
      * using the parameters passed to it and adds it to the post-
condition
      * Vector of the target method.
      ************************************************************
******/
      private void replacePostCondition(Vector
postCond,WsClasses.WsExpression exp1,WsClasses.WsExpression exp2)
      {
            WsClasses.WsLessThan newLesser =  new WsLessThan(exp1,
exp2);

            postCond.add(newLesser);

            WsSubprogram subprog =
targetmethod.getWsMethodSubprogram();

            subprog.setWsPostConditions(postCond);

      }
```

Transform XNotAbhinav5

```
/******************************************************************
******
* Source file: XNotAbhinav5.java
* Purpose:Transforms the post-condition of the form NOT(A=B) to A!=B
/******************************************************************
******
/*****************************************************************
checks to make sure that there is atleast one post condition of the
form NOT(A=B)
/******************************************************************

public static boolean applicable(Object target, Object params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();
```

```
                  Vector postConditions =
subprog.getWsPostConditionSet();

                  WsExpression exp;

                  if(postConditions.size() > 0){
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp -> exp)*/
                        for(int i=0; i < postConditions.size();i++){
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof WsClasses.WsNot){

                                    WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                                    WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                        if(e1Unary instanceof WsClasses.WsEqual){

                                          return true;
                               }



                              }
                        }

                  }
                  else
                        return false;


                  return false;

             } //End of applicable

/******************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form NOT(A=B)
*
* Then it changes the post condition to the form.A!=B
/******************************************************************

public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){
```

```
      exp = (WsExpression) postConditions.get(index);


      if(exp instanceof WsClasses.WsNot)
               {
                        WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                        WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                 if(e1Unary instanceof WsClasses.WsEqual){

                        WsBinaryExpression BinaryExp=
(WsBinaryExpression) e1Unary;

                        WsExpression e1Binary=
BinaryExp.getWsBinExpOp1();

                        WsExpression e2Binary=
BinaryExp.getWsBinExpOp2();

                       postConditions.remove(index);


replacePostCondition(postConditions,e1Binary,e2Binary);



                 }


             }

          index++;

             }

return true;
}


      /*****************************************************************
******
      * It creates a new expression of the form A!=B
      * using the parameters passed to it and adds it to the post-
condition
      * Vector of the target method.
      *****************************************************************
******/
      private void replacePostCondition(Vector
postCond,WsClasses.WsExpression exp1,WsClasses.WsExpression exp2)
      {
            WsClasses.WsNotEqual newNotEqual =  new WsNotEqual(exp1,
exp2);
```

```
            postCond.add(newNotEqual);

            WsSubprogram subprog =
targetmethod.getWsMethodSubprogram();

            subprog.setWsPostConditions(postCond);

      }
```

Transform XNotAbhinav6

```
/*****************************************************************
******
* Source file: XNotAbhinav6.java
* Purpose:Transforms the post-condition of the form NOT(A!=B) to A=B
/*****************************************************************
******
/**************************************************************
checks to make sure that there is atleast one post condition of the
form NOT(A!=B)
/**************************************************************

public static boolean applicable(Object target, Object params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                  Vector postConditions =
subprog.getWsPostConditionSet();

                  WsExpression exp;

                  if(postConditions.size() > 0){
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp -> exp)*/
                        for(int i=0; i < postConditions.size();i++){
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof WsClasses.WsNot){

                                    WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                                    WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                        if(e1Unary instanceof WsClasses.WsNotEqual){

                                          return true;
                               }



                              }
                        }

                  }
                  else
                        return false;


                  return false;

             } //End of applicable

/******************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form NOT(A!=B).
*
* Then it changes the post condition to the form A=B
/*******************************************************************
public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){

      exp = (WsExpression) postConditions.get(index);


      if(exp instanceof WsClasses.WsNot)
               {
                         WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                         WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                 if(e1Unary instanceof WsClasses.WsNotEqual){

                         WsBinaryExpression BinaryExp=
(WsBinaryExpression) e1Unary;

                         WsExpression e1Binary=
BinaryExp.getWsBinExpOp1();

                         WsExpression e2Binary=
BinaryExp.getWsBinExpOp2();

                       postConditions.remove(index);


replacePostCondition(postConditions,e1Binary,e2Binary);
```

```
                }


            }

         index++;

            }

return true;
}


      /**************************************************************
*******
      * It creates a new expression of the form A=B
      * using the parameters passed to it and adds it to the post-
condition
      * Vector of the target method.
      **************************************************************
******/
      private void replacePostCondition(Vector
postCond,WsClasses.WsExpression exp1,WsClasses.WsExpression exp2)
      {
            WsClasses.WsEqual newEqual =  new WsEqual(exp1, exp2);

            postCond.add(newEqual);

            WsSubprogram subprog =
targetmethod.getWsMethodSubprogram();

            subprog.setWsPostConditions(postCond);

      }

Transform XNotAbhinav7

/*******************************************************************
******
* Source file: XNotAbhinav7.java
* Purpose:Transforms the post-condition of the form NOT(NOT A) to A
/*******************************************************************
******
/******************************************************************
checks to make sure that there is atleast one post condition of the
form NOT(NOT A)
/******************************************************************

public static boolean applicable(Object target, Object params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();
```

```
                  Vector postConditions =
subprog.getWsPostConditionSet();

                  WsExpression exp;

                  if(postConditions.size() > 0){
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp -> exp)*/
                        for(int i=0; i < postConditions.size();i++){
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof WsClasses.WsNot){

                                    WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                                    WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                        if(e1Unary instanceof WsClasses.WsNot){

                                          return true;
                               }



                              }
                        }

                  }
                  else
                        return false;


                  return false;

             } //End of applicable

/*****************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form  NOT(NOT A)
*
* Then it changes the post condition to the form A
/******************************************************************

public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){
```

```
      exp = (WsExpression) postConditions.get(index);


      if(exp instanceof WsClasses.WsNot)
               {
                        WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                        WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                 if(e1Unary instanceof WsClasses.WsNot){

                        WsUnaryExpression UnaryExp1 =
(WsUnaryExpression) e1Unary;

                        WsExpression e1Unary1=
UnaryExp1.getWsUnaryExpOp();

                       postConditions.remove(index);

                       replacePostCondition(postConditions,e1Unary1);



                }


            }

         index++;

            }

return true;
}


      /***************************************************************
*******
      * It creates a new expression of the form A
      * using the parameters passed to it and adds it to the post-
condition
      * Vector of the target method.
      ***************************************************************
******/
      private void replacePostCondition(Vector
postCond,WsClasses.WsExpression exp1)
      {
            postCond.add(exp1);

            WsSubprogram subprog =
targetmethod.getWsMethodSubprogram();

            subprog.setWsPostConditions(postCond);
      }



Transform XNotAbhinav8

/*******************************************************************
******
* Source file: XNotAbhinav8.java
* Purpose:Transforms the post-condition of the form NOT(FORALL (X:
MYINT) (EXP1)) to EXISTS (X:MYINT) (NOT(EXP1)
/*******************************************************************
******
/*************************************************************
checks to make sure that there is atleast one post condition of the
form NOT(FORALL (X: MYINT) (EXP1)).
/*************************************************************

public static boolean applicable(Object target, Object params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                  Vector postConditions =
subprog.getWsPostConditionSet();

                  WsExpression exp;

                  if(postConditions.size() > 0)
                  {
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp -> exp)*/
                        for(int i=0; i < postConditions.size();i++)
                        {
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof WsClasses.WsNot)
                              {

                                    WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                                    WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                    if ( e1Unary instanceof WsClasses.WsUniversal)
                              {

                             return true;
                              }


                               }
                        }
                     }



                    return false;

               } //End of applicable
/*****************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form NOT(FORALL (X: MYINT) (EXP1)).
*
* Then it changes the post condition to the form...
*     EXISTS (X:MYINT) (NOT(EXP1)
/*****************************************************************

public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExistential eExist=new WsExistential();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){

      exp = (WsExpression) postConditions.get(index);


      if(exp instanceof WsClasses.WsNot)
               {
                        WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                        WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                              if(e1Unary instanceof
WsClasses.WsUniversal)
                                    {

                                          WsQuantifiedExpression
eQuant =(WsQuantifiedExpression) e1Unary;

                                    WsExpression
e1Quant=eQuant.getWsQuantExpConstraint();

                                          Vector
e2Quant=eQuant.getWsQuantExpDeclarations();
eExist.setWsQuantExpDeclarations(e2Quant);
                                    WsNot eNot=new WsNot();
                                    eNot.setWsUnaryExpOp(e1Quant);

eExist.setWsQuantExpConstraint(eNot);

                                        postConditions.remove(index);
                                          postConditions.add(eExist);
                                          subprog =
targetmethod.getWsMethodSubprogram();

subprog.setWsPostConditions(postConditions);




                }


            }
         index++;
            }
return true;
}
```

Class XNotAbhinav9

```
/*******************************************************************
******
* Source file: XNotAbhinav9.java
* Purpose:Transforms the post-condition of the form NOT(EXISTS (X:
MYINT) (EXP1)) to FORALL (X:MYINT) (NOT(EXP1)
/*******************************************************************
******

/******************************************************************
checks to make sure that there is atleast one post condition of theform
NOT(EXISTS (X: MYINT) (EXP1))
/******************************************************************
             public static boolean applicable(Object target, Object
params) {
```

```
                              WsClasses.WsMethod tgtMethod =
(WsClasses.WsMethod) target;
                              WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                              Vector postConditions =
subprog.getWsPostConditionSet();

                              WsExpression exp;

                              if(postConditions.size() > 0)
                              {
                              /* check to make sure that there is
atleast one post condition of the
                              form (exp -> exp)*/
                                    for(int i=0; i <
postConditions.size();i++)
                                    {
                                          exp = (WsExpression)
postConditions.get(i);
                                          if(exp instanceof
WsClasses.WsNot)
                                          {

                                                WsUnaryExpression
UnaryExp = (WsUnaryExpression) exp;

                                                WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                                 if ( e1Unary instanceof
WsClasses.WsExistential)
                                          {

                                          return true;
                                          }


                                           }



                                  }
                               }



                              return false;

                         } //End of applicable

/*****************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form.NOT(EXISTS (X: MYINT) (EXP1))
/*****************************************************************
public boolean execute(Object params) {
      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsUniversal eUniversal=new WsUniversal();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){

      exp = (WsExpression) postConditions.get(index);


      if(exp instanceof WsClasses.WsNot)
               {
                        WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                        WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                              if(e1Unary instanceof
WsClasses.WsExistential)
                                     {

                                          WsQuantifiedExpression
eQuant =(WsQuantifiedExpression) e1Unary;

                                    WsExpression
e1Quant=eQuant.getWsQuantExpConstraint();

                                          Vector
e2Quant=eQuant.getWsQuantExpDeclarations();




eUniversal.setWsQuantExpDeclarations(e2Quant);

                                    WsNot eNot=new WsNot();

                                    eNot.setWsUnaryExpOp(e1Quant);


eUniversal.setWsQuantExpConstraint(eNot);


                                        postConditions.remove(index);


postConditions.add(eUniversal);

                                          subprog =
targetmethod.getWsMethodSubprogram();
```

```
subprog.setWsPostConditions(postConditions);




                }


            }

        index++;

            }

return true;
}


Transform XNotAbhinav10
/*******************************************************************
*****
* Source file: XNotAbhinav10.java
* Purpose:Transforms the post-condition of the form NOT (A AND B) to
NOT A OR NOT B
/*******************************************************************
*****

/*****************************************************************
checks to make sure that there is atleast one post condition of the
form NOT(A AND B)
/*****************************************************************

public static boolean applicable(Object target, Object params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                  Vector postConditions =
subprog.getWsPostConditionSet();

                  WsExpression exp;

                  if(postConditions.size() > 0){
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp -> exp)*/
                        for(int i=0; i < postConditions.size();i++){
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof WsClasses.WsNot){
                                    WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                                    WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                        if(e1Unary instanceof WsClasses.WsAnd){

                                          return true;
                               }



                              }
                        }

                  }
                  else
                        return false;


                  return false;

             } //End of applicable
/*****************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form NOT(A AND B)
*
* Then it changes the post condition to the form...
*     A OR B
/*****************************************************************

public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){

      exp = (WsExpression) postConditions.get(index);


      if(exp instanceof WsClasses.WsNot)
             {
                         WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                         WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                 if(e1Unary instanceof WsClasses.WsAnd){
                         WsBinaryExpression BinaryExp=
(WsBinaryExpression) e1Unary;

                         WsExpression e1Binary=
BinaryExp.getWsBinExpOp1();

                         WsExpression e2Binary=
BinaryExp.getWsBinExpOp2();

                       postConditions.remove(index);


replacePostCondition(postConditions,e1Binary,e2Binary);



                }


            }

         index++;

            }

return true;
}


      /************************************************************
*******
      * It creates a new expression of the form A OR B
      * using the parameters passed to it and adds it to the post-
condition
      * Vector of the target method.
      ************************************************************
******/
      private void replacePostCondition(Vector
postCond,WsClasses.WsExpression exp1,WsClasses.WsExpression exp2)
      {
        WsClasses.WsNot eNot1=new WsNot();

        WsClasses.WsNot eNot2=new WsNot();

        eNot1.setWsUnaryExpOp(exp1);

        eNot2.setWsUnaryExpOp(exp2);

        WsClasses.WsOr newOr =  new WsOr(eNot1, eNot2);

            postCond.add(newOr);

            WsSubprogram subprog =
targetmethod.getWsMethodSubprogram();

            subprog.setWsPostConditions(postCond);
      }

Transform XNotAbhinav11

/**********************************************************************
******
* Source file: XNotAbhinav11.java
* Purpose:Transforms the post-condition of the form NOT (A OR B) to A
AND B
/**********************************************************************
******

/*******************************************************************
checks to make sure that there is atleast one post condition of the
form Converts NOT(A OR B)
/*******************************************************************

public static boolean applicable(Object target, Object params) {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                  Vector postConditions =
subprog.getWsPostConditionSet();

                  WsExpression exp;

                  if(postConditions.size() > 0){
                  /* check to make sure that there is atleast one post
condition of the
                  form (exp -> exp)*/
                        for(int i=0; i < postConditions.size();i++){
                              exp = (WsExpression)
postConditions.get(i);
                              if(exp instanceof WsClasses.WsNot){

                                    WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                                    WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                        if(e1Unary instanceof WsClasses.WsOr){

                                          return true;
                               }



                              }
                        }

                  }
                  else
                        return false;
                  return false;

             } //End of applicable

/*****************************************************************
* Iterates through the post conditions of the method to find post-
* conditions of the form (NOT e1 -> NOT e2).
*
* Then it changes the post condition to the form.A AND B
/******************************************************************

public boolean execute(Object params) {


      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp;
      int index = 0;

      while(index < postConditions.size()){

      exp = (WsExpression) postConditions.get(index);


      if(exp instanceof WsClasses.WsNot)
             {
                         WsUnaryExpression UnaryExp =
(WsUnaryExpression) exp;

                         WsExpression e1Unary=
UnaryExp.getWsUnaryExpOp();

                 if(e1Unary instanceof WsClasses.WsOr){

                         WsBinaryExpression BinaryExp=
(WsBinaryExpression) e1Unary;

                         WsExpression e1Binary=
BinaryExp.getWsBinExpOp1();

                         WsExpression e2Binary=
BinaryExp.getWsBinExpOp2();

                       postConditions.remove(index);


replacePostCondition(postConditions,e1Binary,e2Binary);



                ]
```

```
            }

         index++;

            }

return true;
}


      /*************************************************************
*******
      * It creates a new expression of the form A AND B
      * using the parameters passed to it and adds it to the post-
condition
      * Vector of the target method.
      *************************************************************
******/
      private void replacePostCondition(Vector
postCond,WsClasses.WsExpression exp1,WsClasses.WsExpression exp2)
      {
            WsClasses.WsNot eNot1=new WsNot();

            WsClasses.WsNot eNot2=new WsNot();

            eNot1.setWsUnaryExpOp(exp1);

            eNot2.setWsUnaryExpOp(exp2);

            WsClasses.WsAnd newAnd =  new WsAnd(exp1, exp2);

            postCond.add(newAnd);

            WsSubprogram subprog =
targetmethod.getWsMethodSubprogram();

            subprog.setWsPostConditions(postCond);

      }




Class XAddLocalAbhinav
/**********************************************************************
******
* Source file: XImpliesAbhinav1.java
* Purpose:Adds a Local Variable "_pre" to the Unprimed variable and
*         changes the variable " x' " to x.This Transform also adds
*         an assignment statement at the begining of the method.It also
*         adds the weakest pre-condition to the subprogram.
/**********************************************************************
******

/*******************************************************************
```

```
applicable makes sure that the index is valid.Returns true if there
is a selected post-conditions .
/*******************************************************************

public static boolean applicable(Object target, Object params)
             {

                  WsClasses.WsMethod tgtMethod = (WsClasses.WsMethod)
target;
                  WsSubprogram subprog =
tgtMethod.getWsMethodSubprogram();

                  Vector postConditions =
subprog.getWsPostConditionSet();

            WsExpression exp;

                  if(postConditions.size() > 0)
             {
                 return true;
             }

             else

             {
                   return false;
                 }

         } //End of applicable

/*****************************************************************
* Iterates through the post conditions of the method to change the
* expression of the type x'=x+1 to x = x_pre+1.It adds an assignment
* x_pre=x at the begining of the method.
/*****************************************************************

public boolean execute(Object params)

{

   WsClass myClass=(WsClass)params;

      WsSubprogram subprog = targetmethod.getWsMethodSubprogram();

      Vector postConditions = subprog.getWsPostConditionSet();

      WsExpression exp = null;
      int index = 0;

      Vector primevar=null;
      Vector varprime=null;
      Vector varunprime=null;
      String tempName1 = null;
      String tempName2 = null;
      for(int i=0; i < postConditions.size();i++)
                        {
                              exp = (WsExpression)
postConditions.get(i);
                              WsCopyVisitor vis = new
WsCopyVisitor();
                              WsExpression expNew =
(WsExpression)exp.acceptVisitor(vis,null);

      primevar=XgetVarAbhinav.returnPrimeVar(exp);

      varunprime=XgetVarAbhinav.returnUnPrimeVar(exp,false);
                              Vector vs=new Vector();

      System.out.println(primevar.size());
                              for(int j=0;j<primevar.size();j++)
                              {
                        tempName1=(String) primevar.get(j);
                        System.out.println("insidefor
loop"+tempName1);
                        WsIdentifierRef newVar1 = new
WsIdentifierRef(tempName1);
                    //Adds the _pre tag to the variable
                        tempName2=tempName1+"_pre";
                                    System.out.println("insidefor
loop2"+tempName2);
                      //if conditions checks for the duplicates
paremeters in the method and removes them
if(!XLocalVarAbhinav.getLocalVar(targetmethod,tempName2))
                        {
                                    System.out.println("inside
if");

                        WsIdentifierRef newVar2 = new
WsIdentifierRef(tempName2);

                        String
myType=XTypeNameAbhinav.getTypeName(myClass,tempName1);

                        WsVariable xp=new WsVariable();

                     //sets the Name and Type of the WsVariable to
be added to the AWl file
                        xp.setTypeName(myType);
                        xp.setName(tempName2);
```

```
                              subprog.addWsSubprogLocal(xp);

                              WsAssignment Wexp=new WsAssignment();
                                        Wexp.setWsAssignLHS(newVar2);
                                        Wexp.setWsAssignRHS(newVar1);

                            vs=subprog.getWsSubprogBody();
                            //Adds the assignment to the starting of the
vector
                                        vs.add(0,Wexp);

                                     }



                                  }

                                WsVisitor nv=new
ChangeRefNewVisitor(tempName1,tempName2);
                                exp.acceptVisitor(nv,null);
                                System.out.println(exp);

                                for (int k =0; k<primevar.size();k++)
                                {

System.out.println("primevar"+primevar.get(k));

                               WsIdentifierRef tempname1=new
WsIdentifierRef((String) primevar.get(k));
                               WsVisitor ne=new
ChangeRefNameVisitor(tempname1);

exp.acceptVisitor(ne,null);

                           }


                           //start

                           Vector body=subprog.getWsSubprogBody();
                           WsExpression
pre=XWpAbhinav.WeakestPrecondition(body,exp);
                           System.out.println("Pre is"+pre);
                     System.out.println("\n");

subprog.setWsPostConditions(ToolUtils.toAWLstring(expNew));
                     subprog.addWsPreCondition(pre);



                         }

                           //index++;
      //}


     return true;
```

```
}

Class XWpAbhinav

/*****************************************************************
******
* Source file: XWpAbhinav.java
* Purpose:Determines and adds the weakest pre-condition to the method
/*****************************************************************
******

public static WsExpression WeakestPrecondition(Vector s,WsExpression e)
   {

        WsExpression current=e;
            //WsCopyVisitor vis = new WsCopyVisitor();
            //current = (WsExpression)e.acceptVisitor(vis,null);

        for(int i=s.size()-1;i>=0;i--)
        {

            Object ws=s.get(i);
            if(ws instanceof WsAssignment)

current=WeakestPrecondition((WsAssignment)ws,current);

                  else if(ws instanceof WsSelection)

current=WeakestPrecondition((WsSelection)ws,current);

                  else if(ws instanceof WsProcedureCall)

current=WeakestPrecondition((WsProcedureCall)ws,current);

                  else if(ws instanceof Vector)
                    current=WeakestPrecondition((Vector)ws,current);

            }

        return current;
   }

   public static WsExpression WeakestPrecondition(WsAssignment
wa,WsExpression e)
   {

      //System.out.println("Wp of assignment is called");

      WsName rName=wa.getWsAssignLHS();
      WsExpression eExp=wa.getWsAssignRHS();

      WsIdentifierRef rRef=(WsIdentifierRef)rName;
      String LhsName=rRef.getName();
      //System.out.println("LHS NAME:"+LhsName);

      WsVisitor ne=new FindRefNameVisitor(LhsName,eExp);
        e.acceptVisitor(ne,null);
       return e;
   }

   public static WsExpression WeakestPrecondition(WsSelection
ws,WsExpression e)
   {

            WsExpression e1 = null;
            WsCopyVisitor vis = new WsCopyVisitor();
            e1 = (WsExpression)e.acceptVisitor(vis,null);
          WsExpression condition = (WsExpression)
ws.getWsSelCondition();
          Vector thenParts=ws.getWsSelThenPart();
          Vector elseParts=ws.getWsSelElsePart();

                  System.out.println(thenParts.get(0));
                  //System.out.println(elseParts.get(0));
                  System.out.println(".........."+e);
                  System.out.println(".........."+e1);
          WsImplication implies1 = new WsImplication();
          WsImplication implies2 = new WsImplication();

          implies1.setWsBinExpOp1(condition);
          WsExpression temp =
WeakestPrecondition((WsAssignment)thenParts.get(0), e);
          implies1.setWsBinExpOp2(temp);

          WsNot not= new WsNot();

          not.setWsUnaryExpOp(condition);
          implies2.setWsBinExpOp1(not);

          if(elseParts.size()>0)
          {
                    WsExpression temp2 = null;
                        System.out.println(".............."+e1);
                        if(elseParts.get(0) instanceof WsAssignment)
                        {
                      temp2 =
WeakestPrecondition((WsAssignment)elseParts.get(0), e1);
               }
                     else if(elseParts.get(0) instanceof WsSelection)
                     {
                         temp2 =
WeakestPrecondition((WsSelection)elseParts.get(0), e1);
                  }
                  implies2.setWsBinExpOp2(temp2);
                  }
                  else
                  {
                    implies2.setWsBinExpOp2(e1);
              }

          WsAnd wpand= new WsAnd();

              wpand.setWsBinExpOp1(implies1);
```

```
          wpand.setWsBinExpOp2(implies2);

         return wpand;


   }

    public static WsExpression WeakestPrecondition(WsProcedureCall
wa,WsExpression e)
      {

         //System.out.println("Wp of ProcedureCall is called");
           return e;
      }


}
```

Class ChangeRefNameVisitor

```
/*******************************************************************
******
* Source file: ChangeRefNameVisitor.java
* Purpose:This Visitor traverses through all the hierarchy
* of expression and if it finds the tick varibale it changes toa new
name
* we desired to change in the post-condition.
/*******************************************************************
******

public class ChangeRefNameVisitor extends WsTraverseVisitor

{
            private WsIdentifierRef OldName;
            //private String NewName;

            public ChangeRefNameVisitor(WsIdentifierRef oldn)
            {
                  super();
                  OldName=oldn;
                  //NewName=newn;
                  System.out.println("Oldname1 is"+OldName.getName());
                  //System.out.println("newname is"+NewName);
            }


      public Object visitBinary(WsBinaryExpression node, Object o)
      {


            if (node.getWsBinExpOp1() != null )
            {

                  node.getWsBinExpOp1().acceptVisitor(this, o);

                  if((node.getWsBinExpOp1() instanceof WsTick))
                  {
                        System.out.println("1
"+node.getWsBinExpOp1()+"1
"+((WsTick)node.getWsBinExpOp1()).getWsTickName() );
                  if(((((WsTick)node.getWsBinExpOp1()).getWsTickName()
).toString()).equals(((WsName)OldName).toString()) )
                  {
                        System.out.println("I am in change tick");
                  node.setWsBinExpOp1(OldName);
            }
      }
      }
            if (node.getWsBinExpOp2() != null )
            {

                  if((node.getWsBinExpOp2() instanceof WsTick))
                  {
                        System.out.println("2
"+node.getWsBinExpOp2());
                  if((((WsTick)node.getWsBinExpOp2()).getWsTickName()
).equals(OldName) )
                  {
            node.setWsBinExpOp1(OldName);
            System.out.println("I am in change tick");
                  }
                  }
                  node.getWsBinExpOp2().acceptVisitor(this, o);
            }

            return null;
      }

      public Object visitUnary(WsUnaryExpression node, Object o)
      {

            if (node.getWsUnaryExpOp() != null )
            {

                  node.getWsUnaryExpOp().acceptVisitor(this, o);
                  if((node.getWsUnaryExpOp() instanceof WsTick))
                  {
                        System.out.println("3
"+node.getWsUnaryExpOp());
                  if((((WsTick)node.getWsUnaryExpOp()).getWsTickName()
).equals(OldName) )
                  {
                        System.out.println("I am in change tick");
                  node.setWsUnaryExpOp(OldName);
            }}
      }

                  return null;
      }



}
```

```
Class FindRefnameVisitor

/*********************************************************************
*****
* Source file: FindRefNameVisitor.java
* Purpose: This visitor changes the variable to "_pre" version
/*********************************************************************
******
public Object visitBinary(WsBinaryExpression node, Object o)
{
      if (node.getWsBinExpOp1() != null && !(node.getWsBinExpOp1()
instanceof WsIdentifierRef))
      {
            node.getWsBinExpOp1().acceptVisitor(this, o);

      if((ToolUtils.toAWLstring(node.getWsBinExpOp1())).equals(FName))
            node.setWsBinExpOp1(Eexp);
      }
      if (node.getWsBinExpOp2() != null && !(node.getWsBinExpOp2()
instanceof WsIdentifierRef))
      {
            node.getWsBinExpOp2().acceptVisitor(this, o);

      if((ToolUtils.toAWLstring(node.getWsBinExpOp1())).equals(FName))
            node.setWsBinExpOp1(Eexp);
      }
      return null;
}

      public Object visitUnary(WsUnaryExpression node, Object o)
      {
            if (node.getWsUnaryExpOp() != null &&
(node.getWsUnaryExpOp() instanceof WsIdentifierRef))
            {
                  node.getWsUnaryExpOp().acceptVisitor(this, o);

      if((ToolUtils.toAWLstring(node.getWsUnaryExpOp())).equals(FName))
                  node.setWsUnaryExpOp(Eexp);
            }

                  return null;
      }

Class ChangeRefNewVisitor

/*********************************************************************
*****
* Source file: ChangeRefNewVisitor.java
* Purpose: This Visitor changes the  WsTick variable to AWSOME syntax
              i.e changes the ticked variable to the unticked version
/*********************************************************************
****

public Object visitBinary(WsBinaryExpression node, Object o)
{
      if (node.getWsBinExpOp1() != null && !(node.getWsBinExpOp1()
instanceof WsTick))
            node.getWsBinExpOp1().acceptVisitor(this, o);
      if (node.getWsBinExpOp2() != null && !(node.getWsBinExpOp2()
instanceof WsTick))
            node.getWsBinExpOp2().acceptVisitor(this, o);
      return null;
}

public Object visitUnary(WsUnaryExpression node, Object o)
{
      if (node.getWsUnaryExpOp() != null && !(node.getWsUnaryExpOp()
instanceof WsTick))
            node.getWsUnaryExpOp().acceptVisitor(this, o);

      return null;
}
      public Object visit(WsIdentifierRef node,Object o)
      {
            String myName="";
            myName=node.getName();
            System.out.println(myName);
            if(myName.equals(OldName))
            node.setName(NewName);
            return null;

      }

}
```