2008

Transformation of Object-oriented Associations and Embedded References to Them

Sneha Swamy
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Repository Citation
https://corescholar.libraries.wright.edu/etd_all/856

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.
TRANSFORMATION OF OBJECT-ORIENTED ASSOCIATIONS
AND EMBEDDED REFERENCES TO THEM

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

By

Sneha Swamy
Bachelor of Technology in Electronics and Communication Engineering,
Jawaharlal Nehru Technological University, INDIA, 2006

2008
Wright State University
I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Sneha Swamy ENTITLED Transformation of Object-Oriented Associations and Embedded References to Them BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Computer Engineering.

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
Swamy, Sneha. M.S.C.E, Department of Computer Engineering, Wright State University, 2008. Transformation of Object-Oriented Associations and Embedded References to Them.

AFIT Wide Spectrum Object Modeling Environment (AWSOME) automatically generates executable code by applying transforms to the formal specifications developed from the domain knowledge. AWSOME uses the specification language called as AFIT Wide Spectrum Language (AWL). All the transforms developed for AWSOME depend on and manipulate Abstract Syntax Trees (ASTs) created by parsing the formal specifications of a model. The main idea is to speed the process of creating error free code, to aid reusability of code and also to make the maintenance changes to be consistent and correct. The AWSOME metamodel contains the structural (object) model, the functional (operations) model and the dynamic (state based) model. This thesis focuses on designing transforms based on the structural model, with focus on transforming associations, aggregations and associative objects and on transforming the invariants.

Given a formal specification, this thesis transforms associations into one way pointers, two-way pointers or associative object pointers, transforms aggregations into one way pointers and also transforms associative objects into associative object pointers. It also updates the references to the association’s role names which are lost when the associations are transformed and removed from the AST. All together the fifteen transforms developed in this thesis result in an output containing valid expressions which
refer not only to the local attributes or method arguments but also to the transformed association role names.
# TABLE OF CONTENTS

1. INTRODUCTION
   - 1.1 DESIGN OF ASSOCIATIONS ......................................................... 2
   - 1.2 OBJECT CONSTRAINT LANGUAGE .................................................. 4
   - 1.3 PROBLEM STATEMENT .................................................................. 6
   - 1.4 SCOPE ....................................................................................... 6
   - 1.5 RESEARCH APPROACH ................................................................. 7
   - 1.6 ORGANIZATION OF THE THESIS .................................................. 7

2. BACKGROUND
   - 2.1 AST (ABSTRACT SYNTAX TREE) .................................................... 8
   - 2.2 ASSOCIATION IN AWL ................................................................. 9
   - 2.3 ASSOCIATIVE OBJECT IN AWL .................................................... 10
   - 2.4 AGGREGATION IN AWL ............................................................... 11
   - 2.5 ASSOCIATION END IN AWSOME ................................................ 12
   - 2.6 ASSOCIATIONS IN AWSOME ...................................................... 12
   - 2.7 ASSOCIATIVE OBJECT IN AWSOME .......................................... 13
   - 2.8 ASSOCIATIONS IN OCL ............................................................... 13
   - 2.9 AWSOME TRANSFORMS .............................................................. 15
   - 2.10 EXISTING ASSOCIATION TRANSFORMS .................................... 16
     - 2.10.1 ONE-WAY POINTER TRANSFORMS ........................................ 16
     - 2.10.2 TWO-WAY POINTER TRANSFORMS ....................................... 17
     - 2.10.3 TRANSFORM FOR ASSOCIATIVE OBJECTS ............................. 18
     - 2.10.4 AGGREGATION TRANSFORMS ............................................. 19
   - 2.11 OTHER RESEARCH EFFORTS ...................................................... 21

3. REQUIREMENT ANALYSIS
   - 3.1 DETERMINATION OF SYNTAX AND SEMANTICS ............................ 23
     - 3.1.1 DISADVANTAGES OF AWL ASSOCIATION ................................. 23
     - 3.1.2 DISADVANTAGES OF OCL .................................................... 24
     - 3.1.3 SUMMARY OF NEW SYNTAX AND SEMANTICS .......................... 24
   - 3.2 CONVERTING ASSOCIATION REFERENCES .................................. 25
   - 3.3 ONE LINK NAVIGATION .............................................................. 27
     - 3.3.1 ONE-WAY POINTERS ............................................................ 27
     - 3.3.2 TWO-WAY POINTERS ........................................................... 30
   - 3.4 AGGREGATION .......................................................................... 31
   - 3.5 ASSOCIATIVE OBJECTS ............................................................... 31
   - 3.6 MULTIPLE LINKS NAVIGATION .................................................. 32
   - 3.7 ASSOCIATION INVARIANT ........................................................... 34
   - 3.8 SUMMARY ................................................................................. 35

4. DESIGN ......................................................................................... 37
   - 4.1 AWSOME STRUCTURE ............................................................... 37
     - 4.1.1 STRUCTURE OF WSASSOCIATION AND WSASSOCENDS ....... 37
       - 4.1.1.1 WSASSOCIATION ........................................................... 37
       - 4.1.1.2 WSASSOCENDS .............................................................. 38
       - 4.1.1.3 WSASSOCOBJECT ......................................................... 39
       - 4.1.1.4 WSAGGREGATE ............................................................ 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.2</td>
<td>REFERENCES IN EXPRESSIONS</td>
<td>40</td>
</tr>
<tr>
<td>4.1.2.1</td>
<td>SINGLE LINKS</td>
<td>40</td>
</tr>
<tr>
<td>4.1.2.2</td>
<td>MULTIPLE LINKS</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>DESIGN APPROACH</td>
<td>42</td>
</tr>
<tr>
<td>4.2.1</td>
<td>DESIGN DECISIONS TO SUPPORT OCL SEMANTICS</td>
<td>42</td>
</tr>
<tr>
<td>4.2.2</td>
<td>SUPPORTING TRANSFORMS</td>
<td>43</td>
</tr>
<tr>
<td>4.2.2.1</td>
<td>XFORMT3</td>
<td>43</td>
</tr>
<tr>
<td>4.2.2.2</td>
<td>XFORMS3</td>
<td>43</td>
</tr>
<tr>
<td>4.2.2.3</td>
<td>TRANSFORMPREETI5</td>
<td>44</td>
</tr>
<tr>
<td>4.2.2.4</td>
<td>TRANSFORMSNEHA</td>
<td>44</td>
</tr>
<tr>
<td>4.2.2.5</td>
<td>INVARIANTSPLET</td>
<td>44</td>
</tr>
<tr>
<td>4.2.3</td>
<td>TRANSFORMING ASSOCIATIONS</td>
<td>45</td>
</tr>
<tr>
<td>4.2.3.1</td>
<td>ONE-WAY POINTER</td>
<td>46</td>
</tr>
<tr>
<td>4.2.3.2</td>
<td>TWO–WAY POINTER</td>
<td>49</td>
</tr>
<tr>
<td>4.2.4</td>
<td>AGGREGATION</td>
<td>53</td>
</tr>
<tr>
<td>4.2.5</td>
<td>ASSOCIATIVE OBJECT</td>
<td>55</td>
</tr>
<tr>
<td>4.2.6</td>
<td>MOVING INVARIANTS</td>
<td>58</td>
</tr>
<tr>
<td>4.2.6.1</td>
<td>ASSOCIATION WITH MULTIPLICITY MANDATORY</td>
<td>59</td>
</tr>
<tr>
<td>4.2.6.2</td>
<td>ASSOCIATION WITH MULTIPLICITY OPTIONAL</td>
<td>61</td>
</tr>
<tr>
<td>4.2.7</td>
<td>MULTIPLE LINKS</td>
<td>62</td>
</tr>
<tr>
<td>4.2.8</td>
<td>SUMMARY</td>
<td>63</td>
</tr>
<tr>
<td>5</td>
<td>IMPLEMENTATION AND TESTING</td>
<td>64</td>
</tr>
<tr>
<td>5.1</td>
<td>SUPPORTING TRANSFORMS</td>
<td>65</td>
</tr>
<tr>
<td>5.1.1</td>
<td>TRANSFORMSNEHA</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>SUPPORTING UTILITY FUNCTIONS</td>
<td>66</td>
</tr>
<tr>
<td>5.2.1</td>
<td>GETCLASSNAMES(WSASSOCIATION ASSOC)</td>
<td>66</td>
</tr>
<tr>
<td>5.2.2</td>
<td>GETOBJECTCLASSNAMES(WSASSOCOBJECT ASSOC)</td>
<td>67</td>
</tr>
<tr>
<td>5.2.3</td>
<td>CANDO(WSASSOCIATION ASSOC, WSCCLASS FROMCLASS, WSPACKAGE MYAST)</td>
<td>67</td>
</tr>
<tr>
<td>5.2.4</td>
<td>CANDOASSOCOBJECT(WSASSOCOBJECT ASSOC, WSPACKAGE MYAST)</td>
<td>67</td>
</tr>
<tr>
<td>5.2.5</td>
<td>CANDOASSOCTOOBJECT(WSASSOCIATION ASSOC, WSPACKAGE MYAST)</td>
<td>68</td>
</tr>
<tr>
<td>5.2.6</td>
<td>RETURNWCLASS(WSPACKAGE MYAST, STRING CLASSNAME)</td>
<td>68</td>
</tr>
<tr>
<td>5.2.7</td>
<td>GETWSASSOCIATION(WSPACKAGE MYAST, STRING ASSOCNAME)</td>
<td>68</td>
</tr>
<tr>
<td>5.2.8</td>
<td>GETWSASSOCOBJECT(WSPACKAGE MYAST, STRING ASSOCNAME)</td>
<td>68</td>
</tr>
<tr>
<td>5.2.9</td>
<td>RETURNROLENAMES(WSPACKAGE MYAST)</td>
<td>69</td>
</tr>
<tr>
<td>5.2.10</td>
<td>PROMPTANDGETASSOCOBJECT(WSPACKAGE AST, STRING PROMPT)</td>
<td>69</td>
</tr>
<tr>
<td>5.2.11</td>
<td>RETURNCLASSNAMES(WSPACKAGE TOP)</td>
<td>69</td>
</tr>
<tr>
<td>5.2.12</td>
<td>RETURNASSOCNAMES(WSPACKAGE TOP)</td>
<td>69</td>
</tr>
<tr>
<td>5.2.13</td>
<td>RETURNASSOCOBJNAMES(WSPACKAGE TOP)</td>
<td>69</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1.1 An AWL example illustrating an association.........................................................2
Figure 1.2 Possible output of one-way transform for the AWL example shown in fig 1...3
Figure 1.3 Possible output of one-way transform in one-to-many case for the AWL example shown in fig 1.1.................................................................4
Figure 1.4 Possible output of two-way transform in many-to-many case for the AWL example shown in fig 1.1.................................................................4
Figure 1.5 Example of representation of references to association in expressions using OCL semantics.................................................................5
Figure 2.1 AWSOME Visual model......................................................................................8
Figure 2.2 Shows syntax and example of association in AWL........................................9
Figure 2.3 Example of navigating association.....................................................................10
Figure 2.4 Example illustrating an expression navigating multiple associations using AWL syntax and semantics..........................................................10
Figure 2.5 Syntax and example of associative object in AWL........................................11
Figure 2.6 Syntax and example of aggregation in AWL................................................11
Figure 2.7 WsAssocEnd model in AWSOME.................................................................12
Figure 2.8 WsAssociation model in AWSOME...............................................................13
Figure 2.9 WsAssocObject model in AWSOME..............................................................13
Figure 2.10 Example of expression navigating association resulting in an object..........14
Figure 2.11 Example of expression navigating association resulting in a set using OCL..14
Figure 2.12 Example of the output after XformP1 is applied........................................16
Figure 2.13 Example of the output after XformP3 is applied........................................17
Figure 2.14 Example of the output after XformP4 is applied........................................18
Figure 2.15 Example of the output after XformP6 is applied........................................19
Figure 2.16 Input for XformP11.......................................................................................20
Figure 2.17 Example of the output after XformP11 is applied......................................20
Figure 2.18 Input for XformP12.......................................................................................20
Figure 2.19 Example of the output after XformP12 is applied......................................20
Figure 3.1 Example AWL code for School system..........................................................26
Figure 3.2 a) Example of one way pointer in Faculty class for Mentoring association b) Pointer is placed in the Student class.........................................................28
Figure 3.3 One way pointers in One-to-Many case a) The pointer is created in the Faculty class..........................................................................................29
Figure 3.3 b) Pointer is created in the Student class........................................................29
Figure 3.4 One way pointer in many-to-many relations. a) The pointer is created as a set in FACULTY class.................................................................29
Figure 3.4 b) The pointer is placed in the Student class to Faculty class........................29
Figure 3.5 A two way pointer created in one-to-many association................................31
Figure 3.6 An example of associative object created between Faculty and School......32
Figure 3.7 Example of multiple links.............................................................................33
Figure 3.8 Example of pointers created in multiple links..............................................34
Figure 4.1 An example of association and its structure in AWSOME.........................37
Figure 4.2 (A) AWL example of aggregation..................................................38
Figure 4.2 (b) Structure of aggregation in AWSOME........................................38
Figure 4.2 (c) AWL example of associative object.............................................39
Figure 4.2 (d) STRUCTURE of associative object in AWSOME..........................39
Figure 4.3 (a) shows an example of an invariant in class and invariant in an association containing references to association and also attributes..........................................41
Figure 4.3 (b) Shows the structure of invariant in the class Thing..........................41
Figure 4.3 (c) Shows the structure of invariant in the association AssocB in AWSOME.........................................................................................................................41
Figure 4.4 (a)shows an example of a post condition with multiple links containing references to associations.................................................................42
Figure 4.4 (b) Structure of the post-condition shown in the example in figure 4.4(A).........................................................................................................................42
Figure 4.5 Flow diagram for Associative Object..................................................56
Figure 4.6 Dependency Diagram of AssociativeObjectTransform..........................57
Figure 5.1 ToolBox GUI.................................................................................65
Figure 5.2 Checking for the multiplicity.............................................................70
Figure 5.3 Tool model used to test the developed transforms...............................72
Figure 5.4 (a) Test case for one-way pointer, (b) Output AWL code after the NEWTRANSFORMPREETI1 is applied..........................................................73
Figure 5.5 (a) Test case for one-way pointer for set pointer, (b) Output AWL code after the NewTransformPreeti3 is applied..............................................75
Figure 5.6 (a) Test case to test two-way pointer...............................................76
Figure 5.6 (b) Output after the two-way pointer transform, NewTransformPreeti4 is applied.................................................................76
Figure 5.7 AWL test case with invariant and the output after the NewTransformPreeti4 is applied...........................................................................................................77
Figure 5.8 (a) Test case to test the NewTransformPreeti43.................................78
Figure 5.8 (b) Output of the two way pointer after applying NewTransformPreeti43..................................................................................................................78
Figure 5.9 (a) Test case for NewTransformPreeti42............................................79
Figure 5.9 (b) Output AWL code after the transform is applied.........................79
Figure 5.10 (a) Test case to test the aggregation................................................80
Figure 5.10 (b) Output AWL code after the NewTransformPreethi11 is applied.....80
Figure 5.11 (a) Test case to test the aggregation................................................81
Figure 5.11 (b) Output AWL code after the NewTransformPreethi11 is applied.....81
Figure 5.12 (a) Test case to test TransformAssociativeObject2............................83
Figure 5.12 (b) Output AWL code after TransformAssociativeObject2 is applied....83
Figure 5.13 (a) Test case to test TransformAssociativeObject..............................84
Figure 5.13 (b) Output AWL code after TransformAssociativeObject is applied.....84
Figure 5.14 (a) Test case to test MoveInvariantTransform...................................85
Figure 5.14 (b) shows the output AWL code after transform is applied...............85
Figure 5.15 (a) Test case to test MoveInvariantTransform2................................86
Figure 5.15 (b) shows the output AWL code after transform is applied...............86
Figure 5.16 (A) Test case for validation of MoveInvariantTransform3..................88
Figure5.16 (b) Output AWL code after the transform is applied..........................88
Figure 5.17 Output AWL code after.................................................................88
Figure 6.1 The Association tool just after start up………………………………………………..94
Figure 6.2 The association tool after the role names are verified and the association to be transformed is selected………………………………………………………………………….95
LIST OF TABLES

Table 3.1 Summary table of the possible cases to handle………………………………..35
Table 5.1 Describes all the transforms and their dependencies on other transforms…….89
ACKNOWLEDGEMENTS

I would begin by thanking my thesis advisor Dr Thomas C. Hartrum for providing me with the opportunity to work with him. I am grateful to Dr Hartrum for his constant support and guidance throughout the course of this thesis work. His way of explaining things and defining the research problem kept me motivated and focused. I would especially like to thank him for his patience and knowledge in reviewing and editing the contents of this document. I am very thankful to my committee members Dr. Mateen M.Rizki and Dr. Krishnaprasad Thirunarayan for their invaluable feedback on this work.

It is my pleasure to thank my parents who encourage me in achieving what I am today.
1. INTRODUCTION

One of the software ideas was the concept of automated transformation systems, which could save huge amounts of manual labor, time and money. The object oriented models which are renowned for their reusability and platform independency, cause a huge amount of manual labor during maintenance and the enhancing stages as the modifications are made in the source code.

A typical transformation system contains a domain model with domain knowledge. For every specific problem, a formal specification is developed by an application engineer from the domain model. Appropriate decisions and rationale are applied to these specifications by applying transformations to generate the formal design. After the formal design model is developed, it is converted into executable code [2][6].

AWSOME (AFIT Wide Spectrum Object Modeling Environment) is a formal-based software synthesis system with a specification language AWL (AFIT Wide spectrum Language). This transformation system is an object oriented design. This automated tool lets the human designer make some design decisions by defining the specifications formally. In AWSOME, the specifications from the domain model are developed into an AST on which all the transformations are performed to generate code [5].

AWSOME contains a structural model which has the set of attributes, a functional model which has the set of methods, and a dynamic model which defines the states.
Associations describe the relationship among the instances of the classes. Associative objects act like a class which has attributes, methods and can include an association. AWSOME allows collections of more than two association ends, but this thesis deals with only binary associations. An AWL specification of one such association is shown in figure 1.1.

![AWL specification](image)

**Figure 1.1 An example of AWL specification illustrating an association.**

### 1.1 DESIGN OF ASSOCIATIONS

One part of AWSOME deals with converting associations to a form closer to object – oriented code.

An association is converted into a one-way pointer or two-way pointer in the case of one-to-one mapping, or in the case of one-to-many mapping. The class which holds the association, i.e. the class from which it points is called the fromClass and the class to which is pointed is called toClass. A one way relationship is converted into one-way pointer by creating an attribute named by the association name in the fromClass. This
attribute holds the values of type toClass. The transformation of the example in figure 1.1 is illustrated in figure 1.2.

From figure 1.2, we can only know the people who own a particular vehicle. If we create a two way pointer by creating an attribute of type VehicleSet in the Person class also, then different vehicles owned by a person are also known in the case of one-to-many relationship. Prior to this thesis, transforms were developed to convert associations, aggregations and associative objects into attributes.

```
package test is
  type CHAR is abstract;
  type STRING is sequence of CHAR;
  type Optional is range 0 .. 1;
  type ZeroToMany is range 0 .. *;
  class Person is
    public name : STRING;
    end class;
  class Vehicle is
    public number : STRING;
    private Owns : Person;
    end class;
  association Owns is
    role  P: Person multiplicity Optional;
    role  V: Vehicle multiplicity ZeroToMany;
    end association;
  end package;
```

Figure 1.2 Possible output of one-way transform for the AWL example shown in fig 1.1

The figures 1.3 and 1.4 show the transformed code for the example that we have considered before for creating one way pointer in one-to-many relationship and two-way pointers in many-to-many case respectively.
OBJECT CONSTRAINT LANGUAGE

OCL (The Object Constraint Language) is a subset of the Unified Modeling Language. As its name implies, it is the expression language for the UML that allows software developers to specify all types of constraints over the objects in the model. UML (Unified Modeling Language) is a standard developed by the OMG for component and object modeling. It provides a powerful method for describing business processes in a manner helpful to both developers and users [1]. In OCL, the value of an association is the set of objects if multiplicity is more than unity and if the multiplicity of the association-end has a maximum of one (“0..1” or “1”), then the value is an object [4].

Currently the way an association is expressed in AWL is different from the OCL syntax. An association in AWL always denotes a set of objects even when the multiplicity is unity. So, when the transforms are being developed or being applied, the
person has to know by the multiplicity whether he is looking at a set or an object. From the example shown in figure 1.5, the post-condition of method `ownerIs()` is an example of the complexity of the syntax. If the syntax of the post condition is changed to something similar to OCL semantics, the post condition would be similar to one below

$$\text{Owns'} = \text{this.Owns.P.name}$$

The AWL syntax can be changed because currently there are no other tools that use the existing syntax.

When the association `Owns` shown in fig 1.5 is converted into a one way pointer from Vehicle to Person, an attribute `private Owns: Person` is created in the vehicle class.

```awl
class Person is
  public name : STRING;
  invariant forall (v1 : Vehicle, v2 : Vehicle)
    (v1 in this.Owns.V and v2 in this.Owns.V and v1 /= v2
     => v1.number /= v2.number)
end class;

class Vehicle is
  public number : STRING;
  procedure ownerIs(Owns: out STRING)
    guarantees exists (owner: Person) (owner in this.Owns.P and
                            Owns' = owner.name)
  end procedure;
end class;

association Owns is
  role P: Person multiplicity Optional;
  role V: Vehicle multiplicity ZeroToMany;
end association;
end package;
```

Figure 1.5 Example of representation of references to association in expressions using OCL semantics
The expression `this.Owns.V` in the invariant condition is still referencing the association that is now removed. Hence the idea is to find these expressions and change them to `this.Owns.P`.

1.3 PROBLEM STATEMENT

Currently all the associations are transformed into pointers (one way or two way) of an attribute. But the invariants or pre and post conditions which refer to the associations are still referencing those associations which do not exist. Also these transformations were developed assuming that the variable references in the expressions are local attributes or method arguments, but they don’t handle the references to associations.

The problem that is addressed in this thesis is to transform the specifications that are referring to associations to refer to the newly created attributes. AWSOME transformation system contains structural, functional and dynamic models. The problem addressed in this thesis mainly concentrates in areas of structural and functional models of the system.

1.4 SCOPE

This thesis demonstrates that when an association is transformed to pointers, the expressions containing references to association’s role names that are lost can be updated by moving the expressions to another class or to an association. As discussed in sec 1.2, a decision will be made about the AWL syntax and semantics related to association referencing. Though all these expressions have to be transformed to be more meaningful and also to hold their validity, this thesis focuses only on the invariants found in the AST containing references to association role names.
1.5 RESEARCH APPROACH

This thesis follows an object-oriented approach in tackling the problem. The various phases of this approach are analysis of existing AWSOME, analysis of OCL semantics and syntax in the area of associations, requirement specifications for the transforms that will be implemented in this thesis, designing issues, implementation, testing and maintenance of these transforms and methodology in which these transforms can be applied.

1.6 ORGANIZATION OF THESIS

This thesis is an effort to update the expressions containing lost references to the transformed association role names. Chapter 2 describes the needed background to understand the rest of the thesis including a discussion of OCL and AWL syntax and semantics for association navigation, Abstract Syntax Trees of the corresponding structural components and the previously developed association transforms. Chapter 3 is an in-depth explanation of the requirements analysis and the final output desired from this thesis. In Chapter 4 we discuss design issues. This chapter discusses in detail the design approach followed for each transform. Chapter 5 demonstrates how the transforms are implemented and the test results after each transform is executed. It also focuses on test cases and the tools used to execute each transform. Chapter 6 discusses the methodology in which the transforms should be applied to achieve the desired result. Finally we conclude the thesis in Chapter 7 and suggest future work that can be applied to this thesis.
2. BACKGROUND

2.1 AST (ABSTRACT SYNTAX TREE)

The AWSOME system is based on a meta-model represented using an AST. AWL is the formal specification language with which the problem specifications are listed. AST is the interim model between the input specifications and output. Problem specifications are parsed into a specification AST.

Figure 2.1 AWSOME visual model
All the transforms are applied on the AST and finally the transformed tree is converted into an AWL specification or the output code. The AWSOME tools can be applied on the AST at any point to make changes according to the requirements. AWSOME model is illustrated in figure 2.1 [5].

2.2 ASSOCIATION IN AWL

Associations are modeled as a collection of two or more association ends. Each end identifies the role, class and multiplicity of the instances of classes in the association. Associations do not link attributes, methods or anything other than classes. The syntax and an example illustrating the syntax of association in AWL is shown in figure 2.2.

<table>
<thead>
<tr>
<th>Syntax:</th>
</tr>
</thead>
<tbody>
<tr>
<td>association identifier is</td>
</tr>
<tr>
<td>role identifier : class-name multiplicity type-name;</td>
</tr>
<tr>
<td>// Repeat at least once</td>
</tr>
<tr>
<td>invariant condition</td>
</tr>
<tr>
<td>end association</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Optional is range 0..1;</td>
</tr>
<tr>
<td>association Married is</td>
</tr>
<tr>
<td>role husband : Person multiplicity Optional;</td>
</tr>
<tr>
<td>role wife : Person multiplicity Optional;</td>
</tr>
<tr>
<td>invariant husband.Sex = Male and wife.Sex = Female</td>
</tr>
<tr>
<td>end association;</td>
</tr>
</tbody>
</table>

Figure 2.2 Shows syntax and example of association in AWL

Associations are navigated in the following way in AWL

\[ \text{anObj.anAssoc.aRole} \]
The above syntax returns a set of object instances linked to anObj via the anAssoc association in the role aRole. For example, martha.Married.husband is the set containing the husband of Martha. So, the values returned by navigation of an association are treated as sets, even when the multiplicity is one [3].

Example:

```
procedure Marry(man : in Person, woman : in Person)
assume man.Married.wife = {} and woman.Married.husband = {};
gurantees woman in man.Married.wife' and man in woman.Married.husband'
```

Figure 2.3 Example of navigating association

Figure 2.4 shows an example of an expression illustrating navigation through multiple associations, if it is allowed in AWSOME.

```
class City is
    public procedure hasCars(cars : out CARSET)
gurantees forall (ind: Industry) (ind in this.indAt.industries => (ind.carsAt.cars in cars')) and
    forall (c: FreightCar) (c in cars' => (exists (ind: Industry) (ind in this.indAt.industries and c in ind.carsAt.cars))
end class;
association carAt is
    role cars : FreightCar multiplicity ZeroToMany;
    role at: Industry multiplicity Optional;
end association;
association indAt is
    role industries: Industry multiplicity ZeroToMany;
    role at: City multiplicity Mandatory;
end association;
```

Figure 2.4 Example illustrating an expression navigating multiple associations using AWL syntax and semantics

2.3 ASSOCIATIVE OBJECT IN AWL

Associative objects act both like classes and like associations [3]. They have attributes and methods, and participate in associations. The syntax and an example are illustrated in fig 2.5.
2.4 AGGREGATION IN AWL

Aggregation is a special case of association. But for an aggregation exactly two ends must be specified, one of which is the parent end and the other is the child end. The syntax and an example of the aggregation are illustrated in fig 2.6.

Syntax:

```
aggregation identifier is
    parent identifier : class-name multiplicity type-name;
    child identifier : class-name multiplicity type-name;
    invariant condition
end aggregation;
```

Example:

```
Type Optional is range 0..1;
aggregation CarWheel is
    parent car : Person multiplicity Optional;
    child wheel : Person multiplicity ZeroToFour;
end assocobject;
```

Figure 2.6 Syntax and example of aggregation in AWL
2.5 ASSOCIATION END IN AWSOME

WsAssocEnd contains the details of the ends of association, aggregation and associative object. Aggregation is a case of association and they are distinguished by the attribute of the WsAssocEnd i.e. wsAggregate. If wsAggregate is true, then the structure is an aggregation and if false, then the structure will describe an association. WsAssocEnd model is shown in figure 2.7.

![Figure 2.7 wsAssocEnd model in AWSOME](image)

2.6 ASSOCIATION IN AWSOME

WsAssociation has to have a minimum of two association ends, whose information is stored in the WsAssocEnd class. A reference is created for the association itself with the class WsIdentifierRef. The references are created to role name and the instance of the class in association with WsIdentifierRef and the multiplicity is parsed as the WsIntegerType. The AWSOME tree related to WsAssociation is shown figure 2.8. WsAssociation is the aggregation of all the classes that are shown in the figure.
2.7 ASSOCIATIVE OBJECT IN AWSOME

Associative object in AWSOME is an aggregation of WsAssocEnd, WsAttribute and WsSubProgram. The model shown in figure 2.9 illustrates the aggregation diagram of WsAssocObject.

2.8 ASSOCIATIONS IN OCL

Although AWL is based on OCL, there are some differences. Starting from a specific object, we can navigate an association on the class diagram to refer to other objects and
their properties. To do so, we navigate the association by using the opposite association-end:

\[
\text{object.rolename}
\]

The value of this expression is the set of objects on the other side of the role name association. If the multiplicity of the association-end has a maximum of one ("0..1" or "1"), then the value of this expression is an object. If the multiplicity is "0..1" then the result is an NULL (no object), otherwise it is the unique object. An example of this expression is illustrated in fig 2.10.

```
Example:
context Company
inv : self.manager.isUnemployed = false
```

**Figure 2.10 Example of expression navigating association resulting in an object**

A navigation expression generates a collection of objects. By default, navigation will result in a set as shown in fig 2.11. When the association on the class diagram is adorned with {ordered}, the navigation results in a sequence. Collections, like sets, bags and sequences, are predefined types in OCL. They have a large number of predefined operations on them [7].

\[
\text{collection} \rightarrow \text{operation( arguments )}
\]

```
Example:
context Company
inv : self.employee->notEmpty()
```

**Figure 2.11 Example of expression navigating association resulting in a set**
2.9 AWSOME TRANSFORMS

AWSOME has a special transform design upon which all the transforms are developed according to the specific problem. The node transform class is called Transform, which is an abstract class. The methods which are to be implemented by the subclasses are [8].

1. **getName()**: This method is overridden by the concrete subclass to provide the name of the transform.

2. **getDesc()**: This should be overridden to provide the description of the transform.

3. **applicable(Object target, Object params)**: This method is overridden to check the applicability of the transform to the particular target of the given AST. The params would provide the details of comparing or the information of what the applicability has to look for.

4. **explain(Object target, Object params)**: This method explains the functioning of the transform.

5. **execute(Object params)**: This method applies the transform on the AST.

6. **unDo(WsPackage top, Object params)**: This method would undo the transform that was applied to the AST if possible.

7. **replay(WsPackage top)**: Replays the entire transformation that was applied to the AST with the transform by storing the target and params specifications.

8. **toString()**: Converts the transform or any objects that the method is applied to into a string.
2.10 EXISTING ASSOCIATION TRANSFORMS

Preeti Subhedar’s research concentrated on implementing the transforms based on the structural model, with focus on transforming associations, aggregations, associative objects, collection types and abstract types into attributes, pointers, and data types that can be directly converted to source code [9].

2.10.1 ONE-WAY POINTER TRANSFORMS

XformP1 and XformP3 are the transforms developed to create one way pointers to eliminate WsAssociation class. XformP1 is used for one-to-one relationships and XformP3 is applied for one-to-many relations.

```plaintext
package test is
type CHAR is abstract;
type STRING is sequence of CHAR;

class Person is abstract
  public name: STRING;
end class;

class Vehicle is abstract
  private name : STRING;
  private Owns : Person;

  public procedure getName(theValue : out STRING) guarantees theValue’ = name
  public procedure setName(theValue : in STRING) guarantees name’ = theValue
  private procedure getOwns (theValue : out Person) guarantees theValue’ = Owns
  private procedure setOwns(theValue : in Person) guarantees Owns’ = theValue
end class;
end package;
```

Figure 2.12 Example of the output after XformP1 is applied.
The output of XformP1 is shown in fig 2.12 and the output of XformP3 is shown in fig 2.13 for the example input shown in fig 1.1. XformP1 creates an object pointer, where as XformP3 creates a set pointer to the toClass role in the association.

```plaintext
package test is
type CHAR is abstract;
type STRING is sequence of CHAR;

class Person is abstract
  public name: STRING;
  private Owns : VehicleSet;
  public procedure add(V : in Vehicle)
    guarantees V in Owns
  public procedure delete(V : in Vehicle)
    guarantees not (V in Owns')
end class;

class Vehicle is abstract
  private name : STRING;
end class;

type VehicleSet is set of Vehicle;
end package;
```

**Figure 2.13 Example of the output after XformP3 is applied.**

2.10.2 TWO-WAY POINTER TRANSFORMS

The above transforms create one way pointers which allow the search to move in one direction. XformP4 is a transform for the cases where the association requires two way pointers. This transform will also allow adding and removing elements from sets. Figure 2.14 shows the output of XformP4 used to create two-way pointers.
2.10.3 TRANSFORM FOR ASSOCIATIVE OBJECT

This transform converts an association into an associative object pointer. A new class replaces the association. This class contains a pointer to both of the classes that are involved in the association. XformP6 creates a new class and then creates a one-way pointer to the toClass and then another one-way pointer to the fromClass in the new class. Figure 2.15 illustrates the output of XformP6 for an input shown in fig 1.1, where Ownslink is the new class that is created to hold the pointers to the association roles.
2.10.4 AGGREGATION TRANSFORMS

XformP11 transform removes aggregations after converting them into one-way pointers. XformP11 is used to create an object pointer of the child class in parent class. XformP12 transform is used to convert an aggregation in which the child class end multiplicity is more than unity. XformP12 creates a set pointer to the child class from the parent class. An example output of XformP11 is shown in figure 2.17 for an input shown in fig 2.16.
package test is
  type CHAR is abstract;
  type STRING is sequence of CHAR;

  class Car is abstract
    public name : STRING;
  end class;

  class Engine is abstract
    public brandname : STRING;
  end class;

  aggregation hasEngine is
    parent P:Car multiplicity ZeroToMany;
    child C: Engine multiplicity ExactlyOne;
  end aggregation;
end package;

package test is
  type CHAR is abstract;
  type STRING is sequence of CHAR;

  class Bicycle is abstract
    public name : STRING;
  end class;

  class Wheel is abstract
    public brandname : STRING;
  end class;

  type WheelSet is set of Wheel;
end package;

Figure 2.16 Input for XformP11

Figure 2.17 Example of the output after XformP11 is applied.

The output for XformP12 is shown in fig 2.19 for the input AWL specification illustrated in fig 2.18.

package test is
  type CHAR is abstract;
  type STRING is sequence of CHAR;

  class Car is abstract
    public name : STRING;
    private hasEngine : Engine;
    public procedure getName(theValue : out STRING)
      guarantees theValue' = name
    end procedure;
    public procedure setName(theValue : in STRING)
      guarantees name' = theValue
    end procedure;
  end class;

  class Engine is abstract
    public brandname : STRING;
  end class;

  aggregation hasEngine is
    parent P:Car multiplicity ZeroToMany;
    child C: Engine multiplicity ExactlyOne;
  end aggregation;
end package;

package test is
  type CHAR is abstract;
  type STRING is sequence of CHAR;

  class Bicycle is abstract
    public name : STRING;
  end class;

  class Wheel is abstract
    public brandname : STRING;
  end class;

  type WheelSet is set of Wheel;
end package;

Figure 2.18 Input applied to XformP12

Figure 2.19 Example of the output after XformP12 is applied.
2.11 OTHER RESEARCH EFFORTS

One of the works that is aimed at automated code synthesis is cited in this section. ModES (Model-driven Embedded System design) is the implementation which generates code from the formal specifications obtained from UML or Simulink[13]. This design utilizes UML2MOF tool which converts any UML meta-model to meta-model specified by MOF(Meta Object Facility). The MOF Specification is the foundation of OMG's industry-standard environment where models can be exported from one application, imported into another, transported across a network, stored in a repository and then retrieved, rendered into different formats (including XMI, OMG's XML-based standard format for model transmission and storage), transformed, and used to generate application code[11]. The MOF meta-model is then loaded to MDR (Meta Data repository) which is based on the JMI (Java Metadata Interface) technology, which provides a standard mapping from MOF into the Java language[12].

After the specifications are given an interface, the model is translated to an Internal Application Meta Model which captures the modules present and also the actions and then the Internal Platform Meta Model is created which specifies the components available for different platforms. The mapping of IAMM and IPMM into Implementation Meta Model are specified in another meta-model called Mapping Meta Model. The IMM can be described as a list of selected platform components, application components, and associations between them. The generation tool uses the ModES API to obtain information from the Implementation Model and to complete these templates, which are specified using a Java-based template language. The designer must still
implement some methods, since in some cases the information in the metadata repository is not enough to generate all parts of the source code.
3. REQUIREMENT ANALYSIS

3.1 DETERMINATION OF SYNTAX AND SEMANTICS

3.1.1 DISADVANTAGES OF AWL ASSOCIATIONS

There are problems in the way associations are represented in AWL. Because of these problems, the navigability is not so easy with these associations. The disadvantages are

1. In AWL, navigation through an association always results in a set, even when the multiplicity is one. Right now, the navigation of multiple associations is not allowed. But if it were possible, the specification of the pre and post conditions and the invariants would be complicated to represent, as the result of these navigations is a set of sets.

2. Also, Preeti Subedhar has developed the transforms for associations with multiplicity one by assuming the output of that navigation would result in an object. But the result would be a set.

3. AWL semantics limit the navigation to multiple classes through associations.

In OCL, navigating multiple associations is simple as the navigation is done with the role names and not the association name. Also the result of an association with multiplicity one is an object. The semantics of association are fairly simple in OCL.
3.1.2 DISADVANTAGES OF OCL

OCL has its limitations but they are a minor drawback when compared to the limitations caused by AWL. In OCL, an association is navigated by its role name without representing the association name as

Object.rolename

The role name shouldn’t be the same as any attribute name of the class, as there would be a problem of linking the reference to right attribute. This is not a problem in AWL, as the role name is always associated with the association name when used.

3.1.3 SUMMARY OF NEW SYNTAX AND SEMANTICS

The decision of representing the association in OCL or just going along with the current AWL representation has to be made before continuing with requirements analysis, since example test cases need to be represented in the actual AWL syntax to be used, and the corresponding AWSOME AST is dependent on this syntax. Since Preeti already assumed the OCL semantics while creating her transforms, and because the linker is not working for associations so none of the tools depend on the AWL syntax, adapting the OCL syntax seems to be the right choice for the following situations.

- While using links, represent using the role name only

  this.wife

- While using multiple links, use a sequence of role names

  this.industries.cars

- The type of output for each navigation, i.e., the return of an object, set, sequence, or bag, will follow OCL rules.
Requirements for OCL:

1. AWL syntax for references to associations should be changed either manually or with a tool.

2. Preeti’s transforms should be changed such that the role name is used to create attributes instead of the association name, according to the OCL syntax. The remainder of this chapter will assume this has already been done.

3.2 CONVERTING ASSOCIATION REFERENCES

The different situations where the reference to the association is lost, when the association is replaced by a one-way pointer or the two-way pointer, are listed below.

1. When the association is converted into a one-way pointer, the references to the association found in toClass have to be updated.

2. Also the references to the fromClass role name via toClass from another class have to be updated.

3. If the association is converted into an associative object pointer, the references to the association role names have to be updated.

An example of AWL specification is shown in figure 3.1. This example has been used throughout chapter 3 to explain the requirements developed.
package AssociationTest is
    type CHAR is abstract;
    type STRING is sequence of CHAR;
    type POSINT is range 0 .. *
    type FACULTYSET is set of Faculty;
    type STUDENTSET is set of Student
    type ZeroToMany is range 0 .. *
    type Mandatory is range 1 .. 1
    type Optional is range 0 .. 1

class School is
    public name : STRING;
    public studentPopulation : POSINT;
    public FacultyPopulation : POSINT;
    public procedure hasFaculty(professors : out FACULTYSET)
        guarantees professors' = this.employee
    public procedure hasStudents(studs : out STUDENTSET)
        guaranteed studs' = this.stu3
end class;

class Faculty is
    public name : STRING;
    public procedure isMentor(stud1 : out Student)
        guarantees stud1' = this.stu1
    public procedure advises (stud2 : out STUDENTSET)
        guarantees stud2' = this.stu2
    public procedure teaches (stud3 : out STUDENTSET)
        guarantees stud3' = this.stu
end class;

class Student is
    public name : STRING;
    public status : STRING;
    public procedure isMentored(ment : out Faculty)
        guarantees ment' = this.mentor
    public procedure isAdvised(adv : out Faculty)
        guarantees adv' = this.advisor
    public procedure isTaught(teachers : out FACULTYSET)
        guarantees teachers' = this.teacher
end class;

association teach is
    role teacher : Faculty multiplicity ZeroToMany;
    role stu : Student multiplicity ZeroToMany;
end association;

Figure 3.1 Example AWL code for School system
association mentoring is
  role mentor : Faculty multiplicity Mandatory;
  role stu1  : Student multiplicity Optional;
  invariant stu1.status = fullTime
end association;

association advise is
  role advisor : Faculty multiplicity Mandatory;
  role stu2   : Student multiplicity ZeroToMany;
end association;

aggregation employed is
  parent employee : Faculty multiplicity ZeroToMany;
  child organisation : School multiplicity Mandatory;
end aggregation;

aggregation registered is
  parent stu3  : Student multiplicity ZeroToMany;
  child university : School multiplicity Mandatory;
end aggregation;
end package;

3.3 ONE LINK NAVIGATION

3.3.1 ONE-WAY POINTERS

When an association between two classes is replaced by a one way pointer, the reference to this association in post or pre conditions and the invariant is lost. While creating the one way pointer, an attribute with the role name of the toClass is created in the fromClass. The attribute is a Vector when the multiplicity is more than one and is an object when the multiplicity is one. The different cases of the association being referenced are:

1. A one-way pointer is created between the faculty and student for association mentoring, that a professor can mentor only one student. If the fromClass is
Faculty then a pointer is created in the Faculty class to the Student class as shown in figure 3.2a below.

```
class Faculty is
  public name : STRING;
  public stu1 : Student
end class;

class Student is
  public name : STRING;
end class;
```

Figure 3.2 a) Example of one way pointer in Faculty class for Mentoring association
b) Pointer is placed in the Student class

In the same way, if Student is the fromClass, then a pointer to the Faculty class is created in the Student class as shown in figure 3.2b.

The reference to this mentor association can be made from either of the classes, as seen in the post conditions in the procedures isMentored and isMentor in figure 3.1. The reference can be made in the Faculty class with a pointer to the student and also when the pointer is made in the Student from the Student class to the Faculty class.

2. Association mentoring is an example for one-to-one association. Association advise defines a one-to-many relation between Faculty and Student. A student can have only one advisor but a faculty member can advise as many students as possible. There is the same possibility of referencing of the association from the two classes as shown in figure 3.1 in the procedures advise and isAdvised. But the difference is the pointer created in the Faculty class to the Student class is a vector instead of an object. Figure 3.3a shows an example of the vector pointer
created in the fromClass Faculty to the toClass Student. Figure 3.3b shows the object pointer created in the Student class to the Faculty class.

![Figure 3.3 One way pointers in One-to-Many case a) The pointer is created in the Faculty class b) Pointer is created in the Student class](image)

3. A many-to-many relation association can be converted into a one-way pointer. For example, an association of teaching can be formed between Faculty class and Student class. A faculty person can have many students and a student can have many teachers. So, the above first and second case arise in this case also as seen in procedures `teaches` and `isTaught` in figure 3.1. The only difference is, in a many-to-many relationship, the pointer is a vector. Hence a query is performed by searching the set and also an update is made to the set. The structure of pointers is shown below in Figure 3.4.

![Figure 3.4 One way pointer in many-to-many relations a) The pointer is created as a set in Faculty class b) The pointer is placed in the Student class to Faculty class](image)
3.3.2 TWO-WAY POINTERS

When an association between two classes is replaced by a two way pointer, the reference to this association in post or pre conditions and the invariant is lost again. While creating the two-way pointer, an attribute with role name of the toClass is created in the fromClass in both ways, i.e., a pointer is created in Faculty class to the Student class and a pointer is created to Student class in the Faculty class, a combination of two one-way pointers as shown in figure 3.4.

1. The reference to the association can be made from both the classes. For example, consider the procedures isMentor and mentored in Faculty and Student class respectively as shown in figure 3.1. The post condition in the isMentor seeks the student whom the faculty is mentoring. In the same way, the mentor of the student is found from the student class with the procedure mentored. Both the procedures result an object, as the relation between the classes is a one-to-one relation.

2. In many-to-one or one-to-many relationships, the association reference would be the same as the above case. In this case the navigation towards the class with multiplicity more than one would result in a set and the navigation towards the end with multiplicity unity would result in an object. The example below in figure 3.5 shows that the two way pointer is nothing but combination of two one way pointers as shown in fig 3.3.

3. In a many-to-many relationship, the navigation either way would result in a set and the accessibility and reference is the same in both directions between the two classes.
3.4 AGGREGATION

Aggregations are modeled as a special case of association in AWSOME. In aggregation, the ends are defined as parent and child instead of role. The pointers are usually created from the parent class to the child class. Hence, aggregations are converted into one-way pointers. There could be one-to-one, one-to-many or many-to-one and many-to-many as in associations. An example of an aggregation can be seen in Figure 3.1 between School and Faculty and School and Student.

3.5 ASSOCIATIVE OBJECTS

Associative objects act as both classes and associations. Like a class, they have attributes and methods and like associations they have two a more ends. An example of an associative object is shown in the Figure 3.6. It illustrates the association between the Student and Faculty created as an associative object. The class MentorLink is the class that defines the association as a separate class. These associations are stored in a container, another class, Mentor1. The associative objects are added and deleted using the procedure into and from the linkSet respectively as shown in the figure. This is usually linked to a higher class than Faculty and Student, a School class. The School is
an aggregation of the Mentor1 class. Hence, the aggregation can be converted to a one-way pointer as mentioned in section 3.5 and placed in the parent class, i.e., School.

```plaintext
class School is
  public name : STRING;
end class;
class Faculty is
  public name : STRING;
end class;
class Student is
  public name : STRING;
end class;
class MentorLink is
  public mentor: Faculty;
  public stu1: Student;
end class;
class Mentor1 is
  private linkSet: MENTORLINKSET;
  public procedure add(m: in Faculty, s: in Student)
    guarantees exists (men: mentorLink) (men.mentor = m and
    men.stu1 = s and men in MENTORLINKSET)
  public procedure delete(m: in Faculty, s: in Student)
    guarantees (forall (men: mentorLink) (men.mentor /= m and
    men.stu1 /= s and not(men in MENTORLINKSET)) )
end class;
type MENTORLINKSET is set of MentorLink;
aggregation mentoring is
  parent univ: School multiplicity mandatory;
```

Figure 3.6 An example of associative object created between Faculty and School

### 3.6 MULTIPLE NAVIGATION

Sometimes more than one association has to be navigated for information. The different cases that arise while navigating multiple links are as follows.
In the single links case as seen in above section, the context of reference to an association is either in the toClass or the fromClass. In the case of multiple links, the context of reference could be any class. For example, as seen in the figure 3.7, there are three classes and there are associations between Student and Faculty, Student and Course,
and also Faculty and Course. Hence the post condition or invariant could be in any of these three classes and may contain reference to any of the associations.

For example, as seen in figure 3.7, in the procedure courseProf() in Student class, names of the faculty teaching courses can be obtained by first getting the course details with the association register and then the details of the teacher from the takes association. This expression navigates through three associations to obtain the professor’s names. The validity of this expression can still be valid if the associations are properly transformed as seen in fig 3.8.

```plaintext
class Faculty is
    public name: STRING;
end class;
class Student is
    public name: STRING;
    public class1 : COURSESET;
end class;
class Course is
    public name : STRING;
    public number : POSINT;
    public prof : FACULTYSET;
end class;
```

**Figure 3.8 Example of pointers created in multiple links**

### 3.7 ASSOCIATION INVARIANT

Invariants are allowed in association definitions. The invariant of the association should hold for all the members of the set. For example, from Figure 3.1, in the association of mentoring, the student can only have a mentor if the student is a full time student. When the associations are converted into structural forms by converting them to pointers, the invariant in the association is lost when the definition is deleted.
Hence, the invariant should be reconstructed so that the condition must hold for the members of that association. Currently the transforms don’t take this into account.

3.8 SUMMARY

Table 3.1 Summary table of the possible cases to handle

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Context : Faculty</th>
<th>Context : Student</th>
<th>Context : Faculty &amp; Student</th>
<th>Context : Association Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer in Faculty</td>
<td>From class : Reference pointer is in the class itself (A)</td>
<td>To class : Referenced pointer is not available (B)</td>
<td>Half are From class and Half are To class. Pointer is not available for half the transforms (C)</td>
<td>Context is lost. Referenced pointer in the fromClass or the toClass. (F)</td>
</tr>
<tr>
<td>Set pointer in Faculty</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pointer in Student</td>
<td>To class : Referenced pointer is in the class itself (A)</td>
<td>From class : Reference pointer is in the class itself (A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set pointer in Student</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pointer in Faculty and a Pointer</td>
<td>Both the classes have pointers to an object or pointers to a set and the both classes act as toClass and fromClass and the reference from any class to another class is same. The reference could be to a set or to an object depending on the multiplicity (D)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>in Student</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pointer in Faculty and a Set in</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Student</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set in Faculty and a Pointer in</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Student</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Both pointers in link class which</td>
<td>For all these cases, the reference is in the toClass because the reference pointer is in the link class. (E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is in container</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The different transforms that have to be implemented for conversion of association references are defined below.

A. One-way pointer is in the context class. No additional change needed.

B. One-way pointer in the non-context class. Reference must be changed or moved to access the referenced pointer.
C. One-way pointer to either class. Half of the references must be changed or moved to access the referenced pointer.

D. Two-way pointer, no additional change needed.

E. Association Object, references must be changed or moved to access the referenced pointer.

F. One-way pointer or two-way pointer with association invariant, invariant must be moved to access the referenced pointer.

G. Association object with association invariant, invariant must be moved to access the referenced pointer.

Before developing these transforms, there are some changes that are to be made in Preeti’s transforms. The changes are

**General changes:**

The transforms have to be changed such that they use the role names of the association rather than the association name.

**Specific changes:**

Correct the way the link class and container are created when transforming an associative object.
4. DESIGN

4.1 AWSOME STRUCTURE

4.1.1 STRUCTURE OF WsAssociation AND WsAssocEnd

4.1.1.1 WsAssociation

Association defines a relationship between instances of classes. Usually they are binary. The instances of classes involved in an association define WsAssocEnds, which define the role of objects at the ends which are in the association and their multiplicity. Hence, each WsAssociation is made up of more than one WsAssocEnd, depending upon the number of classes in the association. The name of the association and the roles are

```
association mentoring is
    role stu : Student multiplicity Mandatory;
    role fac : Faculty multiplicity Optional;
end association;

  . wsDecls(2):Association
    . . wsDeclName:Identifier (mentoring)
    . . Symbol table present with 2 symbols
    . . wsExternal:false
    . . wsAssocEnd(0):Association End
    . . . wsOrder:false
    . . . wsAggregate:false
    . . . wsAssocEndRole:Identifier (stu)
    . . . . wsAssocEndClass:Identifier Reference (Student) linked
    . . . . wsAssocEndMultiplicity(0):Identifier Reference (Mandatory) linked
    . . . . . wsAssocEnd(1):Association End
    . . . . . . wsOrder:false
    . . . . . . wsAggregate:false
    . . . . . . wsAssocEndRole:Identifier (fac)
    . . . . . . . wsAssocEndClass:Identifier Reference (Faculty) linked
    . . . . . . . wsAssocEndMultiplicity(0):Identifier Reference (Optional)
```

Figure 4.1 An example of association and its structure in AWSOME
defined as **Identifier** (the name of variables) and the multiplicity of each end is defined as an **IdentifierRef** (the name of integer type). The outline of an association in AWSOME is illustrated by the example in fig 4.1.

### 4.1.1.2 WsAssocEnd

Each end of an association is an instance of a class participating in the association. WsAssocEnds define each of these ends, their role names and their multiplicity. WsAssocEnds also define the ends of an aggregation, only these are defined as parent and child. wsAggregate of the parent end is set to true if it’s an aggregation. These are also used in WsAssocObj. Along with all the other fields as in WsAssociation and WsAggregation additional information about wsAssocEndQualifier can be specified. Also, another field wsOrder specifies if there is any ordering, when the WsAssocEnd has more than unit multiplicity. The examples in fig 4.2(a) and 4.2 (b) and fig 4.2(c) and 4.2(d), illustrate the structure of WsAssocEnd in aggregation and associative object respectively. Fig 4.1 shows WsAssocEnd in an association.

<table>
<thead>
<tr>
<th>(a) aggregation HasThing is</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent p : TheTopClass multiplicity Mandatory;</td>
</tr>
<tr>
<td>child c : Thing multiplicity Mandatory;</td>
</tr>
<tr>
<td>end association;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b) The structure of aggregation and assocEnds in it :</th>
</tr>
</thead>
<tbody>
<tr>
<td>. wsDecls(24):Association</td>
</tr>
<tr>
<td>. . wsDeclName:Identifier (HasThing)</td>
</tr>
<tr>
<td>. . Symbol table present with 2 symbols</td>
</tr>
<tr>
<td>. . wsExternal:false</td>
</tr>
<tr>
<td>. . wsAssocEnd(0):Association End</td>
</tr>
<tr>
<td>. . . wsOrder:false</td>
</tr>
<tr>
<td>. . . wsAggregate:true</td>
</tr>
<tr>
<td>. . . wsAssocEndRole:Identifier (p)</td>
</tr>
</tbody>
</table>

**Figure 4.2** (a) AWL example of aggregation (b) Structure of aggregation in AWSOME
An associative object has characteristics of a class; it contains attributes, methods and an association. An associative object is the only way in AWSOME for an association to have link attributes or qualified ends and methods along with the association. Fig 4.2(d) shows the structure of associative object in AWSOME.
4.1.1.4 WsAggregate

Aggregation in AWSOME is represented by WsAssociation itself. If the relationship is an aggregation, the field wsAggregate of the parent WsAssocEnd is marked as true as seen in example illustrated in fig 4.2(b). Other than that aggregation is similar to association in context of their structure.

4.1.2 REFERENCES IN EXPRESSIONS

The main idea is to transform structural components such as associations, aggregations and associative objects of the formal specifications into executable code. This is achieved by converting them to pointers and removing them from the AST. In this process, the references to these structural components are seen in several expressions such as invariants, post-conditions and pre-conditions. When an association, aggregation or associative object is transformed, references to these components are lost. Hence, these expressions have to be transformed to be valid. This thesis only modifies invariants to make sure their validity still holds. The references found in post-conditions and pre-conditions are not handled in this thesis.

Associations, aggregations and associative objects are referenced through a single link or multiple links. A single link expression contains only one rolename or rolenames of only one association and a multiple link expression contains more than one association role name.

4.1.2.1 SINGLE LINKS

Fig 4.3 illustrates an example of single link expressions and their structures. The identifier references in these expressions are defined as a selected component.
Associations are referenced by role names of the association. According to the example in fig 4.3(a), role3 is the association reference and the partNumber or ids are the attribute references of the invariants. The association references are defined as wsSelCompName and the attributes are usually defined by wsSelCompComponent.

(a) class Thing is
   public partNumber : STRING;
   invariant this.partNumber /= role4.id
   end class;
   association AssocB is
   role role3 : Thing multiplicity Mandatory;
   role role4 : Widget multiplicity Mandatory;
   invariant role3.partNumber /= role4.id;
   end association;

(b) wsInvariant(0): Not Equal
   . . . wsBinExpOp1: Selected Component
   . . . . wsSelCompName: This
   . . . . wsSelCompComponent: Identifier Reference (partNumber) not linked
   . . . . wsBinExpOp2: Selected Component
   . . . . wsSelCompName: Identifier Reference (role3) linked
   . . . . wsSelCompComponent: Identifier Reference (id) not linked

(c) wsInvariant(0): Equal
   . . . wsBinExpOp1: Selected Component
   . . . . wsSelCompName: Identifier Reference (role3) linked
   . . . . wsSelCompComponent: Identifier Reference (partNumber) not linked
   . . . . wsBinExpOp2: Selected Component
   . . . . wsSelCompName: Identifier Reference (role3) linked
   . . . . wsSelCompComponent: Identifier Reference (id) not linked

Figure 4.3 (a) shows an example of an invariant in class and invariant in an association containing references to association and also attributes. (b) Shows the structure of invariant in the class Thing. (c) Shows the structure of invariant in the association AssocB in AWSOME

4.1.2.2 MULTIPLE LINKS

It can be seen in fig 4.4 that all the links in the expression followed by wsSelCompName are regarded as wsSelCompComponents.
4.2 DESIGN APPROACH

4.2.1 DESIGN DECISIONS TO SUPPORT OCL SEMANTICS

In support of section 3.1.3 and as seen in fig 3.2 and fig 3.3, changes have to be made to handle the references to the association. Changes have to be made in Preeti’s transform such that while creating the object pointer or set pointer, transforms use the role name of the toClass instead of the association name. The goal here would be to make the transforms XformS3 use the role names for the attribute names instead of the association name.
4.2.2 SUPPORTING TRANSFORMS

To obtain the desired AST as seen in fig 3.2 and fig 3.3, the transforms are built using smaller transforms for both structural transformation and expression transformation. In the following sections, we discuss these existing transforms for better understanding of association transforms.

4.2.2.1 XformT3

This transform adds a set type to the AST. The target of this transform is the AST being manipulated and the parameters would be the name and data type of the new set attribute that is to be added to the AST. According to the AWSOME transform design, applicable() checks to see if a set type of the same name already exists in the AST and if it does, applicable() returns false, stopping from creating a set of a similar name. If it is applicable, then the transform method execute() creates a set type by creating an instance of the WsSetType class and adds it to the AST.

4.2.2.2 XformS3

This transform adds an attribute of a given name and type to the given class in the AST. applicable() of this transforms checks if there is any other attribute of the same name in the class and also checks to see if the said type exists, and returns false if any of them are wrong. The execute() method adds an attribute of the given name and type to the target class.
4.2.2.3 TransformPreeti5

This transform adds a new class to the AST with the name given to it as a parameter. The transform adds a string ‘Link’ to the name given for the name of the class. For example, “mentor” becomes “mentorLink”. Before applying this transform applicable() checks for a similar class name in the AST and if the class with the exact name exists, the transform is not applied. Hence, the target to this method is the AST and param is a vector containing the name to be used to name the class.

4.2.2.4 TransformSneha

This new transform should add a new aggregation to the AST. For now it is assumed that the multiplicity is always unity in the aggregation for both parent and child. The parameters to the transform should contain the name of the class as specified, which will be the parent. The parent and child role names are specified and the name of the association or associative object that is to be transformed. The main goal in developing this transform is to be able to use it while transforming an association into an associative object. The target should be the AST, so that the transforms can add the aggregation and also for the transform to check for the same aggregation which might already exist in the AST through applicable().

4.2.2.5 InvariantSplit

This transform takes an expression as the target. The method applicable() checks if the expression contains more than one expression, i.e. expressions connected with ‘and’
to become a single expression. The execute() method splits these expressions by taking
the operands of the ‘and’ expression and adding them to a vector.

4.2.3 TRANSFORMING ASSOCIATION

Associations in the package are removed by transforming them to one-way pointers,
two-way pointers or associative objects, so that information of the association can still be
retained. The assumptions made about these associations are that

➢ They are binary associations, i.e., there are exactly two classes participating in the
  association.

➢ For a given class, all role names directly visible from that class must be unique.
  This is because the linker doesn’t work so it’s not possible to differentiate which
  role name belongs to which association after transforming it to an attribute of a
  class.

➢ It’s also assumed that the references to associations will be made by the role
  names of the association as mentioned in section 4.2.1.

Also, the general factors that the applicable() method of almost all the transforms
should check for are

➢ Whether the size of the association is two, i.e. if the association is binary.

➢ Checks for an invariant in the association, because the association is removed
  after creating a pointer and in that case the invariant would also be lost.

➢ If the role names clash with any attribute in the class it is associated with.

➢ If the classes in the association exist in the AST.
Along with these, applicable() checks for more factors for different types of association i.e., depending upon their multiplicity and the type of pointers to be created.

4.2.3.1 ONE-WAY POINTER

As described in section 3.3.1, there are two cases for which two different types of pointers have to be created. Hence, it would be feasible to create two different transforms to handle these two cases. The first case is when the multiplicity is unity and the second case would be if the multiplicity is more than unity. In the first case an object pointer should be created and for the second case a set pointer should be created.

I. MULTIPLICITY IS UNITY

The goal here is to create an object pointer of the toClass in the chosen fromClass and in order to do this applicable() should make sure that

1. The multiplicity of the toClass end of an association is unit multiplicity.
2. There is no reference to the fromClass in any of the classes in the AST as that pointer would be lost when the association is transformed to a one-way pointer.
3. Invariant in the association contains role names of only one association.

In order to create this pointer, the transform can use XformS3 to create the pointer to the toClass in the fromClass. The example shown in fig 3.2(a) is the transformed AWL code of the association mentoring shown in fig 3.1. The fromClass would be Faculty and the toClass would be Student in this association. Hence, a pointer of stu1 is created in the Faculty class and to do this the params have to specify Faculty as the fromClass.
Hence, the target to the transform should be the package, so that applicable() can check for the references in the AST. The parameters should contain the fromClass and the association that has to be transformed. The execute() method can do this by

1. Finding the toClass for the given fromClass in the AST.
2. Finding the role name of the association end related to the toClass.
3. Call transform XformS3 to add the attribute by passing the fromClass pointer to it as a target and the toClass role name for the name and the toClass class name for the type as parameters.
4. Remove the association from the AST.

If the applicable() of this transform does not meet the requirements as to multiplicity, then the following transform should be chosen to transform the association into a one-way pointer.

II. MULTIPLICITY MORE THAN UNITY

The goal here is to create a set pointer of the toClass in the chosen fromClass and in order to do this applicable() should make sure that

1. The multiplicity of the toClass end of an association is more than unity.
2. There is no reference to the fromClass in any of the classes in the AST as that pointer would be lost when the association is transformed to one-way pointer as in the case of unit multiplicity.
3. Invariant in the association contains role names of only one association.

This transform can use the same XformS3 to create a set pointer to the toClass in the fromClass. The example shown in fig 3.3 (a) is the transformed AWL code of the
association advises as shown in fig 3.1. The fromClass would be Faculty and the toClass would be Student in this association. Hence, a set pointer of stu2 is created in the Faculty class and to do this the user has to select Faculty as fromClass as before.

Hence, the target to the transform should be the package, so that applicable can check for the references in the AST. The parameters should contain the fromClass and the association that has to be transformed. The execute() method can do this by

1. Finding the toClass for the given fromClass in the AST.
2. Call transform XformT3 to create a set type of class toClass. Hence, a pointer to the toClass is sent as param and a pointer to the AST is sent as the target to the transform, so that the new set type can be added to the AST.
3. In order to create a set pointer, find the role name of the end associated with the toClass for the name of the attribute and add a string “Set” at the end of the toClass name for the type of the attribute.
4. Call transform XformS3 to add the attribute by passing the fromClass pointer to it as a target and the toClass role name for the name and the toClass class name+“Set” for the type as parameters.
5. Remove the association that is transformed from the AST.

If the applicable() of this transform does not meet the requirements as to multiplicity, then the transform of unit multiplicity should be chosen to transform the association into a one-way pointer.

For both these cases, if applicable() found an invariant in the association, then it should return false and the following can be done

1. Invariant can be moved to the fromClass where the pointer is created.
2. Invariant can be moved to any superior class, if it contains a pointer to the fromClass.

3. Stop the user from creating a one-way pointer and the association can be transformed to an associative object and the invariant can be moved into the link class.

Also, if applicable() finds a reference of the fromClass rolename of an association in an invariant, the following things can be done

1. If the invariant is in the toClass, it can be moved to the fromClass.

2. If the invariant contains more than one association reference, stop the user from transforming the association.

3. The invariant can be moved to the association before transforming it. The expression containing the reference to the fromClass should only be moved if there is more than one invariant.

4.2.3.2 TWO-WAY POINTER

As explained in section 3.3.2, while transforming an association into two-way pointers, an attribute of one end is placed in the class of other end. After the pointers are created and the association is removed from the AST, the references to the association are lost. As in a one-way pointer, the pointers should be created using the role names of the associated end class as seen in fig 3.5.

Three transforms have to be created to handle all the combinations of multiplicity in associations. These transforms continue to make assumptions that are given in sec 4.2.3 and also the applicable() method checks for the characteristics that are mentioned in sec
4.2.3. Designs of these three transforms to handle the conversion of association to two-way pointers are discussed below.

I. ONE-TO-ONE MULTIPLICITY

When an association with unit multiplicity at both ends (as the association \textbf{mentor} shown in fig 3.1) is converted into a two-way pointers, an object pointer of the opposite end is placed in each class. Hence, the aim is to create an object pointer using the role names of the association ends and this will handle all the references made to this association. Before handling the pointers, the transform has to check for the things listed below in the method applicable().

1. Checks to see if the both the ends have unit multiplicity.

2. Invariant in the association contains role names of only one association.

Target should be the AST and the params should contain the association to be transformed. If applicable() returns true, then the method execute() should do the below to transform the association

1. Get the role names of both the ends of the association.

2. Call transform XformS3 to add the attributes by passing the class1 pointer to it as a target and the vector containing class2 role name for the name and the class1 class name for the type as parameters. In the same way create an attribute of class1 in class2 using the same XformS3 transform.

3. If an invariant is found in the association, two things can be done
   i. The invariant is moved into either of the classes in the association as per the user’s choice.
ii. The transform can be stopped, then transform the association to an associative object and place the invariant in the class holding both the pointers.

4. Remove the association from the AST.

II. ONE-TO-MANY OR MANY-TO-ONE MULTIPLICITY

In an association with one-to-many multiplicity, a set pointer is created in the class opposite to the end with multiplicity more than one and an object pointer in the class opposite to the end with multiplicity unity. The case of many-to-one multiplicity is the same. An example of an association with one-to-many multiplicity is the association advisor shown in the fig 3.1. Before transforming the association and deleting it, the applicable() has to check for the below.

1. Checks to see that one of the ends is of unit multiplicity and the other end has a multiplicity more than unity.

2. Invariant in the association contains role names of only one association.

Target should be the AST and the params should contain the association to be transformed. If the applicable() returns true, execute() should do the following to complete the transformation.

1. Obtain the role names and the class names of both the classes of the association and also the multiplicities of both the ends.

2. Determine which end has multiplicity more than unity and obtain the name and pointer to that class. Call transform XformT3 to create a set type of the obtained class by sending the transform the class name and the pointer to that class, so that the new set type can be added to the AST.
3. Find the role name of the association end with multiplicity more than 1 and also related class pointer and add a String “Set” to the class name for the type of the attribute, as the pointer to be created is a set.

4. Call transform XformS3 to add the attribute by passing the class pointer of the end with multiplicity one to it as a target and the other class role name for the name and above obtained type as parameters.

5. Call transform XformS3 again to add the attribute of the end with unit multiplicity in the class with multiplicity more than one as above by passing the target class and the role name and the type of the attribute.

6. If an invariant is present in the association, place the invariant in the class with multiplicity more than one as this class contains an object pointer but not the set pointer.

7. Remove the association from the AST after the pointers are created in the class of their opposite ends.

### III. MANY-TO-MANY MULTIPLICITY

If this kind of link is seen in the association, then set pointers have to be created in both the classes. In order to create set pointers, the transform must be sure of two things via applicable().

1. It should check to make sure that both the ends of the association should have multiplicity more than unity.

2. It should also check to see that there is no invariant in the association.
Target should be the AST and the params should contain the association to be transformed. execute() should be called only if applicable() returns true. If it returns true, execute() should do the following.

1. Find the role names and class names of both the ends in the association.
2. Call XformT3 to create set types for both the classes in the association and to add them to the AST. Hence the names of the classes and the pointer to the AST should be sent to the transform.
3. Call XformS3 twice to create the set pointers of the class1 in the class2 and also class2 in the class1 by sending the role names for the attribute names and created set names for the type of the attribute and also the pointer to the class in which the attribute is supposed to be added.
4. After adding the attributes to both the classes, the association can be removed from the AST.

In this case, if an invariant is found in the association, then the transform should stop from applying the transformation and the only thing that can be done to prohibit losing the invariant is to convert the association into associative object.

4.2.4 AGGREGATION

As discussed in section 3.4, aggregations are a special case of associations. Only one-way pointers are created in the case of aggregations as only the parent class should have the pointer to the child class for making any modifications or for any queries. A child’s reference to the parent class is prohibited. Hence, two transforms have to be designed as seen in one-way transforms of association discussed above.
As for associations, applicable should check to see that

1. Aggregations cannot contain more than two ends.

2. Both the classes should exist in the AST.

3. The child role name should not clash with the existing attributes in the parent class.

Parameters needed for the method execute() are AST as the target and the params which should include the aggregation that is to be transformed. Aggregation can also be transformed using either of the two transforms depending on the child’s multiplicity as discussed below.

I. AGGREGATION WITH MULTIPLICITY ONE

1. As in association with one to one multiplicity, obtain the role names of both the ends of the aggregation and also their class names.

2. Figure out the end with WsAggregation set as true, this is considered the parent class. After obtaining the association end, find the pointer to that class.

3. Call XformS3 to add an object pointer to the obtained class pointer in step 2. The other end role name is considered as the attribute name and the class name as the type. Hence, the target is the parent’s class and params should contain the child role name and class name.

4. Remove the aggregation from the AST after creating the pointer.
II. AGGREGATION WITH MULTIPLICITY MORE THAN ONE

1. As in the association with one to many multiplicity, obtain the role names of both the ends of the aggregation and also their class names.

2. Figure out the end with WsAggregation set as true, this is considered the parent class. After obtaining the association end, find the pointer to that class.

3. Call XformT3 to create a set of type child’s class and add it to the AST. Hence, the target is AST and the params are the child’s class name appended with string ‘set’ and the type as class name appended with set.

4. Call XformS3 to add a set pointer to the obtained class pointer in step 2. The other end role name is considered as attribute name and new set name as the type. Hence, the target is the parent’s class and params should contain the child role name and class name.

5. Remove the aggregation from the AST after creating the pointer.

4.2.5 ASSOCIATIVE OBJECT

A WsAssociation can be converted into an associative object and all the WsAssocObjects have to be converted into associative objects. Fig 3.6 illustrates an example of how an associative object looks. An associative object contains a class made with both the links in the association and a container class which contains a link to the set of instances of new link classes.

For each association an object of the link class can be created and added to the set. This pointer to the container class is created in a chosen class by creating an aggregation between the parent class and the container class using the transform described in sec 4.2.
The multiplicity is assumed as mandatory in this case. Fig 4.5 shows the structure of the associative object.

![Figure 4.5 Structure diagram for Associative Object](image)

Transform dependency is shown in fig 4.6. AssociativeObjectTransform depends upon XformS5 and also TransformSneha along with XformT3 and XformS3 as mentioned in the above classes. XformS5 adds the link class and the container class to the AST and TransformSneha creates an aggregation between the class linking with the association and the container class.

WsAssociation and WsAssocObject structure are almost the same, hence the transforms for both these choices could be the same, but since both of them are different types and their attributes are obtained using different methods, it is best to use different transforms. But the functionality would still be the same.
As mentioned in sec 3.5, after creating an associative object all the references to the association are lost, hence applicable() should check for any references to the association and if there are any, the transform should not allow the user to convert the relation into an associative object. Hence applicable() should check for the below before applying the transform.

1. Checks for any references to the association or associative object in any class of the AST.

2. Invariant in the association or associative object contains role names of only one association.

If there are no references, then the WsAssociation or WsAssocObject can be converted into associative object as mentioned above by the following mentioned steps.

1. Obtain the role names and the class names as before for the attribute names and their type.

2. Call XformS5 to create the new container class, link class. For example as seen in fig 3.6, the link class is MentorLink and the container class is MentorContainerlink.

3. Call XformT3 to create a set of the new link class.
4. Call XfromS3 to create a pointer in the new link class by passing the role names of the association/associative object for attribute names and the end class names as their type and the pointer to the new link class.

5. Call XformS3 to create a set pointer to the link class in the container class that was created in step 2. The name of the pointer is the association name appended with a string ‘link’ and the type would be the links set.

6. Call TransformSneha to create an aggregation between the container class and the class linked with the association or the associative object that the user chooses.

The only thing that differs between transforming an association and associative object to an associative object is the way the association ends are obtained from them.

ASSUMPTIONS

When a WsAssocObject is converted into associative object, a link class is created with object pointers to both the classes in the association. In this process, the expressions found in the classes other than the link class would become invalid as the rolenames are not visible to these classes. Until these expressions are changed or moved associative objects will not be converted. Also the expressions in the related to associations or associative objects with multiplicity more than unity are not handled as the expressions deal with sets.

4.2.6 MOVING INVARIANTS

As mentioned in section 3.8 case B, when the association is transformed into a one-way pointer, the references to the association in the toClass are lost and the one-way
design checks for those kind of references and wouldn’t apply the transform until those references are taken care as mentioned in section 4.2.3.1. Two things can be done to take care of the references in invariants in the toClass.

1. The invariant can be moved to the fromClass from the toClass by making some modifications to the expression, such that the condition is still valid.

2. Move the invariant to the association that has to be transformed, so that the one-way pointer transform can take care of the invariant. But as in case 1, some modifications have to be made to the expression.

This thesis scope transforms move only the expressions that are referencing associations with multiplicity either unity or null, i.e. Mandatory or optional. Hence, the applicable() of these transforms check to see that the association doesn’t contain an end with multiplicity more than unity. Also applicable() checks for the following

1. If the classes in the association exist in the AST.

2. If there is an invariant in the toClass.

3. If the invariant contains fromClass reference.

4. If the invariant references only one association.

Target to these transforms will be the AST and the params contain the association to be transformed and the fromClass name.

4.2.6.1 ASSOCIATION WITH MULTIPLICITY MANDATORY

The invariant in the toClass can either be moved to the fromClass or to the association that will be transformed to a one-way pointer.
The invariants referencing an association with unit multiplicity are generally in this form,

In Faculty class - this.lastname /= stu.lastname (4.1)

In Student class – fac.lastname /= this.lastname (4.2)

This expression would look like below if it was in the association

fac.lastname/=stu.lastname (4.3)

The two designs for unit multiplicity are discussed in detail below.

I. MOVING INVARIANT TO THE FROMCLASS

If the invariant is moved to the fromClass from the toClass then the invariant is transformed from expression (4.1) to expression (4.2) if the fromClass is Student. The design of this transform is as below

1. Obtain the role names and respective class names.
2. Figure out the toClass name using the fromClass name provided.
3. Split the invariant if the invariant contains more than one expression.
4. Obtain the invariant with the fromClass role reference.
5. Convert the expression such that it references the toClass role from the fromClass, i.e. find ‘this’ and replace it with the toClass role name and find the fromClass role name and replace it with ‘this’.
6. Add the invariant to the fromClass.
7. Change the invariant in the toClass by removing the moved expression.
II. MOVING INVARIANT TO THE ASSOCIATION

If the invariant is moved from the toClass to the association, then it has to be transformed to the expression (4.3).

1. Steps 1 to 4 as above.
2. Convert the expression by finding ‘this’ and replacing it with the toClass role name.
3. Add the invariant to the association.
4. Remove the invariant from the toClass.

4.2.6.2 ASSOCIATION WITH MULTIPLICITY OPTIONAL

The invariant referencing an association in the toClass, fromClass and association are as below.

In fromClass Student – (fac/=null)=>(fac.lastName/=this.lastName) (4.4)

In toClass Faculty – (stu/=null)=>(stu.lastName/=this.lastName) (4.5)

In association – fac.lastName/=stu.lastName (4.6)

I. MOVING INVARIANT TO THE FROMCLASS

If the invariant is moved to the fromClass from the toClass then the invariant is transformed from expression (4.5) to expression (4.4) assuming fromClass as Student.

The design of this transform is as below

1. Obtain the role names and respective class names.
2. Figure out the toClass name using the fromClass name provided.
3. Split the invariant if the invariant contains more than one expression.
4. Obtain the invariant with the fromClass role reference.

5. Convert the expression such that it references the toClass role from the fromClass, i.e. find ‘this’ and replace it with the toClass role name and find the fromClass role name and replace it with ‘this’ and also the fromClass role name/!=null is changed to toClass role name/!=null and interchange the operands of the expression.

6. Add the invariant to the fromClass.

7. Change the invariant in the toClass by removing the moved expression.

II. MOVING INVARIANT TO THE ASSOCIATION

If the invariant is moved from the toClass to the association then it has to be transformed to the expression (4.6).

1. Steps 1 to 4 as above.

2. Convert the expression by finding ‘this’ and replacing it with the toClass role name and also by removing the null expression from the invariant as seen in exp (4.2.6).

3. Add the invariant to the association and remove the invariant from the toClass.

4.2.7 MULTIPLE LINKS

All the multiple references are automatically taken care of when the associations are properly transformed into pointers. If an association is transformed into a one-way pointer, navigation is possible only in one direction and any expression in the third class trying to navigate in the other direction will become invalid. As mentioned in section
4.2.6 these expressions cannot be handled by moving them to a different class or to an association as they involve references to rolenames of different associations. In order to still hold their validity, transformation of these expressions are not allowed and so the applicable() method of transforms designed to move invariants should check that the invariant contains only references to rolenames of a single association.

If the invariant is not moved, the applicable() method in the transforms of a one-way pointer finds the unhandled references in a third class and stops the transform from converting the association into a one-way pointer.

4.3 SUMMARY

All the different cases that have to be handled are mentioned in section 3.8. According to that section,

1. Case A is handled by the transforms in section 4.2.3.1.
2. Case B is handled by the transforms in section 4.2.3.1 and also section 4.2.6.
3. Case C is handled by any of the above 1 or 2 or section 4.2.7.
4. Case D is handled by transforms in section 4.2.3.2.
5. Case E is handled by transform in section 4.2.5.
6. Association invariants are handled by all the transforms which create pointers in the classes i.e. one-way pointer transforms, two-way pointer transforms and associative object transforms.
7. Associative object invariants are handled by the transform of associative object.
5. IMPLEMENTATION AND TESTING

In this chapter, the implementation of each design mentioned in chapter 4 is described. The functionality of each design is discussed and also how it achieved what we are looking for as a result. Each transform is designed based on the AWSOME Transform discussed in chapter 2. Also, we will discuss the validity of each transform by testing it with a test case.

In order to test these transforms, small individual tools are designed and added to the toolbox which is a GUI which already exists in the system. The other tools which will help verify the transforms other than the tool developed for each individual transform are ToolLoader, TooltoOutline, and TooltoAWL.

**ToolLoader:** Creates an AST by taking the given input AWL file and loads it for the individual tools to manipulate the AST.

**TooltoOutline:** Prints out the tree structure of the AST to a text file.

**TooltoAWL:** Will write the AWL code to a file for the AST that has been loaded into the toolbox for manipulations.

**STRUCTURE OF AWSOME TOOLBOX**

All the tools in AWSOME are placed in a package named toolbox. These tools are all based on a common tool interface Tool0.java, which allows a common trait in all the tools. ToolBox.java is a driver tool which allows us to add and delete tools for user
access shown in fig 5.1. Hence, all the tools that are developed to test these transforms are created based on Tool0.java [10].

To support Tool0.java, three methods have to be implemented.

1. **Void setAST(WsPackage ast)** – sets the AST for the tool to work on.
2. **Void run()** – executes the tool.
3. **WsPackage getAST()** – returns the passed AST.

![Figure 5.1 ToolBox GUI](image)

**Figure 5.1 ToolBox GUI**

### 5.1 SUPPORTING TRANFORMS

In section 4.2.2, the designs of several supporting transforms that will be used are described. This section will provide implementation and validation details for those designs. XformT3, XformS3 and TransformPreeti5 are already developed and they can be used as they are. The other transform is discussed below.

#### 5.1.1 TransformSneha

This transform was mainly developed to create an aggregation and add it to the AST when transforming an association or associative object to an associative object pointer.
The transform creates a new aggregation using the association/associative object name. Both the ends are set using the names sent as parameters and this formed aggregation is added to the AST. The parent is the class in which the association link has to be placed, chosen by the user. applicable() checks for an aggregation with the same name and this has been tested for validation. The container class should exist for it to create the aggregation. This transform is tested by getting information of the parent class name, parent name, child name and the association name or associative object name and sending them as a vector to the execute() and applicable() via params. An example of the output of this transform is shown later in Fig 5.4.

5.2 SUPPORTING UTILITY FUNCTIONS

According to the designs, there are several functions that are used commonly in all the transforms. Hence, along with the supporting transforms, some library functions are developed for an incremental form of designing. These library functions are placed in ToolUtils and XformUtils, and are described below.

5.2.1 getClassNames(WsAssociation assoc)

Returns a vector of strings made of the association name, and the names of the classes involved in the association. All the transforms need the name of the association for all the prompts and to verify the classes involved in the association.
5.2.2 **getObjectClassNames(WsAssocObject assoc)**

Returns a vector of strings made of the associative object name, and the names of the classes in the associative object. All the transforms need the name of the associative object for all the prompts and to verify the classes in the associative object.

5.2.3 **canDo(WsAssociation assoc, WsClass fromClass, WsPackage myAST)**

After an association is transformed into pointers, it is removed from the AST. When the association is removed, the expressions containing references to the role names that are removed will be lost. Hence, an association cannot be transformed until all the expressions containing references to an association role name are transformed or moved to a class in which pointers to those role names will be created or to the association itself. In order to avoid transforming these kinds of associations a library function has been developed which will find the references which will be lost and if it does, it returns false and if the method doesn’t find any references which have to be deleted, it returns true.

5.2.4 **canDoAssocObject(WsAssocObject assoc, WsPackage myAST)**

After an associative object is transformed into a class holding the pointers of the role names of an associative object, it is removed from the AST. When the associative object is removed, the expressions containing references to the role names that are moved to a different class will be lost. Hence, an association cannot be transformed until all the expressions containing references to an associative object role name are transformed or moved to a class in which pointers to those role names will be created or to the associative object itself. In order to avoid transforming these kinds of associative objects
a library function has been developed which will find the references which will be lost and if it does, it returns false and if the method doesn’t find any references which have been moved, it returns true.

5.2.5 canDoAssocToObject(WsAssociation assoc, WsPackage myAST)

Returns a boolean value about the possibility of transforming an association to an associative object by verifying the references to the roles of an association in the package. The method returns false when there is even one reference to the association in any class other than the link class and true otherwise. This method also will be used in the transform used to create an associative object.

5.2.6 returnWsClass(WsPackage myAST, String className)

When a class name is given to the method, it returns the WsClass by searching the entire AST. Most of the times, the class name is obtained from the tool as a string and to do any kind of a manipulation, WsClass has to be known from the AST.

5.2.7 getWsAssociation(WsPackage myAST, String assocName)

This method returns the WsAssociation when the association name is given. The method searches throughout the AST and returns the WsAssociation.

5.2.8 getWsAssocObject(WsPackage myAST, String assocName)

This method returns the WsAssociativeObject when the association name is given. The method searches throughout the AST and return the WsAssociativeObject.
5.2.9 returnRoleNames(WsPackage myAST)

This method gathers all the role names found in the AST and returns a vector containing those role names.

5.2.10 promptAndGetAssocObject(WsPackage ast, String prompt)

This method prompts the user to give the name of an associative object and obtain the WsAssocObject from the AST.

5.2.11 returnClassNames(WsPackage top)

This method returns a vector of names of all the classes present in the AST.

5.2.12 returnAssocNames(WsPackage top)

This method returns a vector of names of all the associations present in the AST.

5.2.13 returnAssocObjNames(WsPackage top)

This method returns a vector of names of all the associative objects present in the AST.

5.3 CREATING ONE-WAY POINTER

5.3.1 NEWTRANSFORMPREETII(UNIT MULTIPLICITY):

According to the design discussed in sec 4.2.3.1 I this transform requires the association name and the fromClass name to check the applicability of the transform and also to actually transform the association to a one-way pointer. Hence, the tool designed
to test this transform will prompt the user to enter the association to be transformed and also the fromClass name from the two classes participating in the association. An association can contain more than two ends, but this thesis allows only binary associations.

NewTransformPreeti1 has to create an object pointer of type toClass in the fromClass, using the role name of the toClass end. Preeti developed the one-way pointer transform for unit multiplicity, but in her TransformPreeti1 implementation she used the association name as the attribute name. According to the decision that was made during the requirement chapter about the semantics, the attribute was supposed to be created using the role names. Hence, Preeti’s transform has been modified to use the role name of the toClass end.

The other modifications that were made on Preeti’s transform TransformPreeti1 are:

1. TransformPreeti1 applicable() doesn’t check whether the multiplicity of the end which is to be created as an attribute has a multiplicity of unity. Hence, a check for multiplicity is created in NewTransformPreeti1 to make sure that only an end with multiplicity one is created as an object pointer. A snippet of checking multiplicity is shown in fig 5.2.

```java
while (decs.hasMoreElements()) {
    decl = (WsClasses.WsDeclaration) decs.nextElement();
    if (((String) decl.getName()).equals(multiplicity1)) {
        if (decl instanceof WsClasses.WsIntegerType) {
            if (!(ToolUtils.toAWLstring(((WsIntegerType) decl).getWsIntUpperBound()).equals("1"))) {
                System.out.println("Class doesn't have one-to-one multiplicity");
                return false;
            }
        }
    }
} //End of Checking multiplicity
```

Figure 5.2 Checking for the multiplicity
2. Also, as mentioned in sec 4.2.3.1(I), it was important to check whether there are references to the fromClass in any other class including the toClass. Hence, a library function mentioned in sec 5.2.3 was developed and added to ToolUtils, `canDo(association, fromClass, AST)`, which uses a visitor called ReferenceVisitor to check all the expressions in the AST for the forClass role name and saves them if found. `canDo()` is a static method which returns false if the visitor finds a reference in the AST.

3. TransformPreeti1 transform uses the fromClass as the target, because several properties of the AST have been obtained at the tool level and sent to the transform. Ideally, the tool should be as much as possible independent of the AWSOME system. Hence, the number of properties sent as params to the transforms is decreased and only the association and the fromClass name are sent as params for the NewTransformPreeti1.

4. `execute()` uses the params that have been sent from the tool and figures out the toClass name and the role names from the WsAssociation. The outline of the WsAssociation and their ends are illustrated in fig 4.1. A vector containing the toClass role name, and the toClass name is sent to XformS3 to add an attribute in the target fromClass.

5. `execute()` also adds the invariant to the fromClass, if there is any in the association.

6. After the attribute is added, there is no need for the association in the model any more hence the association is removed from the AST.
A tool called NewToolXformP1 is developed and added to toolbox to test the transform. On executing the tool, the tool prompts for the association name and the fromClass name as seen in fig 5.3.

```java
package toolbox;
public class NewToolXformP1 extends Tool0 {
    private Transform myXform = null;

    public void run() {
        assoc = ToolUtils.promptAndGetAssoc(myAST, "Association to transform.");
        if (assoc == null) {
            JOptionPane.showMessageDialog(null, "Aborting, no association.");
            System.exit(0);
        }
        names = ToolUtils.getClassNames(assoc);
        aName = (String) names.elementAt(0);
        class1 = (String) names.elementAt(1);
        class2 = (String) names.elementAt(2);
        fromName = JOptionPane.showInputDialog(null, "Association " + aName + " between " + class1 + " and " + class2 + "\n" + "Enter the name of the FROM class.", "FROM Name", JOptionPane.PLAIN_MESSAGE);
        if (fromName.equals(class1)) {
            toName = class2;
        } else {
            toName = class1;
        }
        JOptionPane.showMessageDialog(null, "Implementing association " + aName + ", " + "1-way, as one-way pointer from " + fromName + " to " + toName + ";");
        params = new Vector();
        params.add(assoc);
        params.add(fromName);
        if (NewTransformPreeti1.applicable(myAST, params)) {
            myXform = new NewTransformPreeti1 (myAST);
            myXform.execute(params);
            JOptionPane.showMessageDialog(null, "Association " + aName + " has been implemented");
        } else {
            JOptionPane.showMessageDialog(null, "Transform Not applicable");
        }
    }
}
```

**Figure 5.3 Tool model used to test the developed transforms.**
Before running this tool, ToolLoader is executed to load the AST in which the association is present. The test case shown in fig 5.4(a) is loaded and the transform is applied to the AST using the tool shown in fig 5.3. The AWL output is shown in fig 5.4(b). If any of the constraints in applicable() are false, then the user is prompted with the respective error messages and the transform is not applied. Error messages include "Class doesn't have one-to-one multiplicity", or "It has more than one association reference in invariant" if there are more than one association references in the invariant found in the association.

Here the fromClass is given as Faculty, hence a pointer is created in Faculty. If the fromClass was given as Student, the transform would give an error message as there is a reference to stu in the post condition in toClass Faculty.
5.3.2 NEWTRANSFORMPREETI3(MANY MULTIPLICITY)

The only difference in this case as opposed to unit multiplicity is a set pointer is created instead of an object pointer. Preeti developed a transform for this called TransformPreeti3, whose behavior is the same as discussed earlier. Hence, all the above mentioned changes have to be made in this case as well.

1. applicable() in this case checks for multiplicity to be more than unity with the same fragment of code seen before in fig 5.2 except it checks for the upper end to be more than one.

2. Changes showed in steps 2 to 4 in case I are also applied to this transform.

3. If applicable() finds an invariant is the association then the transform is not allowed.

A separate tool, NewToolXformP3 similar to the one shown in fig 5.3 is developed to test this transform. The test case used to test this transform is shown in fig 5.5(a) and its output AWL code is shown in fig 5.5(b).

In order to test all the invalid cases, a test case is written including the StudentSet pointer, so the applicable() of XformT3 should show an error message saying “StudentSet already exists”. Also, if the fromClass is chosen as Student in the above test case, the toClass Faculty has a multiplicity Mandatory, hence the transform applicable() will display an error message saying “Multiplicity is not Many” and the tool will display “Transform cannot be applied” assuring that applicable() works properly. Also, if there is an invariant in the association and the fromClass is chosen as Faculty, then the transform would show an error message “Found an invariant, Hence cannot be applicable”.

74
5.4 CREATING TWO-WAY POINTER

According to the section 4.2.3.2I, in order to convert an association into two-way pointers we needed three transforms because of different multiplicity relations between the two ends. The transforms described below are developed based on TransformPreeti4.

5.4.1 NEWTRANSFORMPREETI4 (ONE-TO-ONE MULTIPLICITY)

The tool used to test the transform NewTransformPreeti4, is similar to the one seen in fig 5.3. According to the example shown in fig 5.6(a), a student has to be mentored by only one faculty member and a faculty member can either mentor only one student or none. This transform calls XformS3 as the relationship doesn’t include more than one instance of either class. Hence, an object pointer is created in both the classes. As both
the classes act as fromClass and toClass, the user doesn’t have to give the fromClass name. applicable() checks everything exactly like the one-way pointer transform except for the references of fromRoleName in the AST.

![Figure 5.6(a) Test case to test two-way pointer. (b) Output after the two-way pointer transform, NewTransformPreeti4 is applied.](image)

The output for this transform is shown in fig 5.6(b). There are no problems of checking for references in any class to any of the association ends as both the pointers are created. If there is an invariant in the association, it can be moved to any of the classes in the association. Hence, the transform asks the user the name of the class in which they would like to place the invariant. Fig 5.7 illustrates a test case and its output after the transform is applied.
5.4.2 NEWTRANSFORMPREETI43 (ONE-TO-MANY OR MANY-TO-ONE MULTIPLICITY)

This transform combines both the transforms of the one-way pointer, a transform which creates the object pointer and the one which creates a set pointer. XformT3 is used in the transform to create a new set type to the AST and XformS3 creates the pointers in the classes of the association. In this transform, if an invariant is found in the association, then the invariant is moved to the class in which object pointer is created instead of asking the user about the class in which the invariant is placed. The test case in fig 5.8(a) is used to test this transform NewTransformPreeti43 and the tool NewToolXform43 as
seen in fig 5.3. The output is shown in the fig 5.8(b) below. applicable() is same as before.

5.4.3 NEWTRANSFORMPREETI42 (MANY-TO-MANY MULTIPLICITY)

NewTransformPreeti42 uses XformT3 to create sets of both classes in the association and uses XformS3 to create set pointers in both classes. applicable() for this case is the same as the previous two-way transforms except that in this case, applicable() would check for an invariant in the association and if there was one, the transform would be stopped from being applied. The test case used to test this transform is shown in fig 5.9(a) and the output of this test case in fig 5.9(b). Association teach defines the relation of student and faculty that a student can have more than one faculty (and none, too) and a teacher can have zero or more students.
This transform’s applicable() as mentioned in the design chapter checks to see if there is an invariant in the association and if there is one, it will inform the user “Found Invariant in association” and does not let the tool apply the transform. Except for this, applicable() is similar in functionality to in NewTransformPreeti4.

5.5 TRANSFORMING AGGREGATION

Aggregation can only be transformed into a one-way pointer. The transform uses the same design as the association to one-way transform. In the one-way pointer transform,
NewTransformPreeti1 needs to know the fromClass to place the pointer, but in this case the fromClass doesn’t have to be provided. The fromClass is always the parent class. Hence, applicable() checks for references to the parent class in the AST and if it finds any, applicable() reports the references. There will be two transforms, one for multiplicity unity and the other for an aggregation with multiplicity more than unity.

5.5.1 NEWTRANSFORMPREETHI11 (AGGREGATION WITH MULTIPLICITY ONE)

```plaintext
package test is
type CHAR is abstract;
type STRING is sequence of CHAR;
type Mandatory is range 1 .. 1;
type Optional is range 0 .. 1;
class School is
    public name : STRING;
    public studentPopulation : POSINT;
    public FacultyPopulation : POSINT;
    end class;
class Faculty is
    public name : STRING;
    public procedure isMentor(stud : out STRING)
        guarantees stud' = this.stu.name
    end class;
aggregation teach is
    parent P : School multiplicity Mandatory;
    child C : Faculty multiplicity Mandatory;
end aggregation;
end package;
```

```plaintext
package test is
type CHAR is abstract;
type STRING is sequence of CHAR;
type Mandatory is range 1 .. 1;
type Optional is range 0 .. 1;
class School is
    public name : STRING;
    public studentPopulation : POSINT;
    public FacultyPopulation : POSINT;
    end class;
class Faculty is
    public name : STRING;
    public procedure isMentor(stud : out STRING)
        guarantees stud' = this.stu.name
    end class;
end package;
```

Figure 5.10(a) Test case to test the aggregation. (b) Output AWL code after the NewTransformPreethi11 is applied.

Figure 5.10 (a) and fig 5.10 (b) shows the test case to test the aggregation and the output after the transform is applied to the test case.
5.5.2 NEWTRANSFORMPREETHI12(AGGREGATION WITH MULTIPLICITY MORE THAN UNITY)

The test case for this transform is shown in fig 5.11(a) and the output in fig 5.11(b). The test case shows an aggregation hasTeacher, which represents an aggregation of school and Faculty. School contains more than one teacher and Faculty may belong to only one School. The transform creates the child’s set in the parent class.

The transform is same as the association transform of a one-way pointer. Hence, applicable() checks for all the things mentioned in section 5.3.2.

```plaintext
package test is
type CHAR is abstract;
type STRING is sequence of CHAR;
type Mandatory is range 1 .. 1;
type Optional is range 0 .. 1;
type ZeroToMany is range 0 .. *;
class School is
  public name : STRING;
  public studentPopulation : POSINT;
  public FacultyPopulation : POSINT;
end class;
class Faculty is
  public name : STRING;
  public procedure isMentor(stud : out STRING)
  guarantees stud' = this.stu.name
end class;
aggregation hasTeacher is
  parent P : School multiplicity Mandatory;
  child C : Faculty multiplicity ZeroToMany;
end aggregation;
end package;

package test is
  type CHAR is abstract;
type STRING is sequence of CHAR;
type Mandatory is range 1 .. 1;
type Optional is range 0 .. 1;
type ZeroToMany is range 0 .. *;
class School is
  public name : STRING;
  public studentPopulation : POSINT;
  public FacultyPopulation : POSINT;
  public C : FacultySet;
end class;
class Faculty is
  public name : STRING;
  public procedure isMentor(stud : out STRING)
  guarantees stud' = this.stu.name
end class;
type FacultySet is set of Faculty;
end package;
```

Figure 5.11(a) Test case to test the aggregation. (b) Output AWL code after the NewTransformPreethi11 is applied.

The transform is same as the association transform of a one-way pointer. Hence, applicable() checks for all the things mentioned in section 5.3.2.
5.6 ASSOCIATIVE OBJECT

Two transforms are developed for creating an associative object pointer; one of them converts an association into an associative object pointer and the other transform converts an associative object into an associative object pointer. Both the transforms have the same design; the only difference is the target that the transform is working on. Fig 5.12(a) shows the test case for transforming an associative object into associative object pointer. The test case also shows that the associative object mentor contains an invariant; hence the transform moves the invariant into the link class.

As mentioned in section 4.2.5, no matter what the multiplicity of the ends are in the association, only object pointers are created in the link class. TransformSneha1 is used to create the aggregation of the container class and the class in which the pointer is to be placed in the case of transforming an associative object, and TransformSneha is used in the case of transforming an association. TransformPreeti5 is used to create the new link class and container class. TransformPreeti5, TransformSneha and TransformSneha1 are described in sec 4.2.2.

The parent name and the child name in the aggregation to be formed are obtained from the user and also the name of the class in which the association pointer is to be placed. Fig 5.13 shows a test case to validate TransformAssociativeObject transform, which converts an association into an associative object pointer.
Figure 5.12(a) Test case to test TransformAssociativeObject2. (b) Output AWL code after TransformAssociativeObject2 is applied.
Two transforms are developed for this case, one to move the invariant to the fromClass and the other moves it to the association. The design discussed in sec 4.2.6.1 is implemented using a visitor. The main aim was to find ‘this’ in the expression and replace it with the toClass role name and to change the fromClass role name to ‘this’.

Figure 5.13(a) Test case to test TransformAssociativeObject. (b) Output AWL code after TransformAssociativeObject is applied.

5.7 MOVING INVARIANTS

5.7.1 ASSOCIATION WITH MULTIPLICITY MANDATORY

(MOVEINVARIANTTRANSFORM & MOVEINVARIANTTRANSFORM2)

Two transforms are developed for this case, one to move the invariant to the fromClass and the other moves it to the association. The design discussed in sec 4.2.6.1 is implemented using a visitor. The main aim was to find ‘this’ in the expression and replace it with the toClass role name and to change the fromClass role name to ‘this’.

```awkern
package test is
    type CHAR is abstract;
    type STRING is sequence of CHAR;
    type Mandatory is range 1 .. 1;
    type Optional is range 0 .. 1;

class Student is
    public name : STRING;
    public lastName : STRING;
end class;

class Faculty is
    public name : STRING;
    public lastName : STRING;
end class;

class mentorlink is
    public fac : Faculty;
    public stu : Student;
invariant stu.lastName /= fac.lastName
end class;

class mentorContainerlink is
    public mentor1link : mentorlinkSet;
end class;

type mentorlinkSet is set of mentorlink;
aggregation newmentor is
    parent p : Student multiplicity Mandatory;
    child c : mentorContainerlink multiplicity Mandatory;
end aggregation;
end package;
```
Hence, the visitor searches the expression, i.e., the invariant, and finds ‘this’ first and then the required role name. Also, there was a case where there could be more than one invariant for the class, hence a transform was developed to split the invariants into single expressions and add them to an array individually. Each expression is then visited to check for the fromClass role name and only if found is the visitor is used on that expression. Fig 5.14 illustrates the test case which is used to validate the transform moving the invariant to the fromClass and also the result after MoveInvariantTransform is applied.

Fig 5.14(a) Test case to test MoveInvariantTransform (b) shows the output AWL code after transform is applied.

Fig 5.15 shows the output when MoveInvariantTransform2 is applied to the testcase to move the invariant to the association. In the test case, the fromClass is chosen as Student; hence the invariants found in the Faculty class with references to student are
moved. The applicable() in both the transforms mainly check for the multiplicities of both the ends to be one, not even zero or one and after testing it with a test case with other multiplicities, applicable() performed correctly by informing the user about improper multiplicity.

```awl
package test is
    type CHAR is abstract;
    type STRING is sequence of CHAR;
    type Mandatory is range 1 .. 1;
    type Optional is range 0 .. 1;

    class Student is
        public name : STRING;
        public lastName : STRING;
    end class;

    class Faculty is
        public name : STRING;
        public lastName : STRING;
        invariant stu.lastName /= fac.lastName
    end class;

    association mentor is
        role stu : Student multiplicity Mandatory;
        role fac : Faculty multiplicity Mandatory;
    end association;
end package;
```

```awl
package test is
    type CHAR is abstract;
    type STRING is sequence of CHAR;
    type Mandatory is range 1 .. 1;
    type Optional is range 0 .. 1;

    class Student is
        public name : STRING;
        public lastName : STRING;
    end class;

    class Faculty is
        public name : STRING;
        public lastName : STRING;
        invariant stu.lastName /= this.lastName
    end class;

    association mentor is
        role stu : Student multiplicity Mandatory;
        role fac : Faculty multiplicity Mandatory;
        invariant stu.lastName /= fac.lastName
    end association;
end package;
```

Figure 5.15 (a) Test case to test MoveInvariantTransform2 (b) shows the output AWL code after transform is applied.
5.7.2 ASSOCIATION WITH MULTIPlicity OPTIONAL

(MoveInvariantTransform3 & MoveInvariantTransform)

The expressions seen in this case are shown in sec 4.2.6.2 with these expressions, when moved to the fromClass where a pointer is created using MoveInvariantTransform3, three places are be modified. ‘this’ is replaced by the toClass end role name, the fromClass role name by ‘this’, and the expression with null will now check for the fromClass to be null. A visitor is used to replace toClass role name to fromClass role name in the implies expression. The same visitor is used to do the other two modifications, too. The test case to validate this transform is shown in fig 5.16 and also its output. The invariant can also be moved to the association itself and this is done using MoveInvariantTransform4. The implication expression is removed and ‘this’ is replaced with the toClass role name in the expression found in the fromClass. The output for this is shown in fig 5.17.

The test case shows an invariant in the toClass Faculty. The association stuFacResearch states that a student can be involved with only one professor in research and vice versa. If the fromClass is chosen as Student then the invariant has to be moved to the fromClass or association itself.
Figure 5.16 Test case for validation of MoveInvariantTransform3 (b) Output AWL code after the transform is applied

```
package test is
  type CHAR is abstract;
  type STRING is sequence of CHAR;
  type Mandatory is range 1 .. 1;
  type Optional is range 0 .. 1;
  class Student is
    public name : STRING;
    public lastName : STRING;
  end class;
  class Faculty is
    public name : STRING;
    public lastName : STRING;
    invariant ()
  end class;
  association stuFacResearch is
    role stu : Student multiplicity Optional;
    role fac : Faculty multiplicity Optional;
  end association;
end package;
```

Figure 5.17 Output AWL code after MoveInvariantTransform4 is applied

```
package test is
  type CHAR is abstract;
  type STRING is sequence of CHAR;
  type Mandatory is range 1 .. 1;
  type Optional is range 0 .. 1;
  class Student is
    public name : STRING;
    public lastName : STRING;
    invariant (fac /= null) => (this.lastName /= fac.lastName)
  end class;
  class Faculty is
    public name : STRING;
    public lastName : STRING;
    invariant()
  end class;
  association stuFacResearch is
    role stu : Student multiplicity Optional;
    role fac : Faculty multiplicity Optional;
  end association;
end package;
```
5.8 SUMMARY

This chapter describes the implementation of each transform to achieve the goal according to the design decisions made in chapter 4. Several tools were built to test these transforms with many test cases to validate the transform functionality. The applicable() method of all the transforms were also verified by developing specific test cases for each condition. This was done in an incremental fashion by first testing all the small tool methods, visitors and the small transforms and then the final versions. Table 5.1 gives description of each transform and the smaller transforms it is dependent on.

<table>
<thead>
<tr>
<th>Transform Name</th>
<th>Description</th>
<th>Dependencies (Other transforms used)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewTransformPreeti1</td>
<td>This transform converts into a one-way pointer depending upon the provided fromClass name. This transform is applicable on an association whose toClass role has unit multiplicity.</td>
<td>Depends on XformS3</td>
</tr>
<tr>
<td>NewTransformPreeti3</td>
<td>This transform creates a set pointer while converting the association into a one-way pointer. This is applicable when the multiplicity of the toClass role is more than unity.</td>
<td>Depends on XformS3 and XformT3</td>
</tr>
<tr>
<td>NewTransformPreeti4</td>
<td>Converts an association into two-way pointers by creating object pointers to both the classes. This is applicable only if the multiplicity of both the ends is unity.</td>
<td>Depends on XformS3</td>
</tr>
<tr>
<td>Transformer</td>
<td>Description</td>
<td>Dependencies</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>NewTransformPreeti42</td>
<td>This transform also converts an association into a two-way pointer, but this is applicable to an association with a many-to-many relationship. Hence, it creates set pointers to both the classes.</td>
<td>Depends on XformS3 and XformT3</td>
</tr>
<tr>
<td>NewTransformPreeti43</td>
<td>Converts an association into a two-way pointer but is applicable to associations with one-to-many or many-to-one multiplicity.</td>
<td>Depends on XformS3 and XformT3</td>
</tr>
<tr>
<td>TransformAssociativeObject</td>
<td>Converts a WsAssociation into an associative object by creating a link class containing both the ends as pointers and also a container class.</td>
<td>Depends on XformS3, XformT3, TransformPreeti5, and TransformSneha.</td>
</tr>
<tr>
<td>TransformAssociativeObject2</td>
<td>Converts a WsAssocObject into an associative object by creating a link class containing both the ends as pointers and also a container class</td>
<td>Depends on XformS3, XformT3, TransformPreeti5, and TransformSneha1.</td>
</tr>
<tr>
<td>NewTransformPreethi11</td>
<td>Converts an aggregation into a one-way pointer by creating an object pointer. This is applicable only when the multiplicity of the child end is unity.</td>
<td>Depends on XformS3</td>
</tr>
<tr>
<td>NewTransformPreethi12</td>
<td>Converts an aggregation into a one-way pointer by creating a set pointer. This is applicable only when the multiplicity of a child end is more than unity.</td>
<td>Depends on XformS3 and XformT3</td>
</tr>
<tr>
<td>MoveInvariantTransform</td>
<td>Moves the invariant from the toClass to the fromClass when the</td>
<td>Depends on InvariantSplit</td>
</tr>
<tr>
<td>Transform Name</td>
<td>Description</td>
<td>Depends On</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>MoveInvariantTransform2</td>
<td>Moves the invariant from the toClass to the association when the multiplicity of both the ends is exactly unity.</td>
<td>InvariantSplit</td>
</tr>
<tr>
<td>MoveInvariantTransform3</td>
<td>Moves the invariant from the toClass to the fromClass when the multiplicity of both the ends is either null or unity.</td>
<td>InvariantSplit</td>
</tr>
<tr>
<td>MoveInvariantTransform4</td>
<td>Moves the invariant from the toClass to the fromClass when the multiplicity of both the ends is either null or unity.</td>
<td>InvariantSplit</td>
</tr>
</tbody>
</table>
6. METHODOLOGY

Several transforms have been developed or modified to achieve one goal, i.e. to convert associations into meaningful pointers. But using these transforms by someone requires complete understanding of each transform and also remembering which one does what. Hence, a methodology is developed to make the application of these transforms less complicated. The main idea is to make the choice of transforms clear and simple to the user. Also, when these transforms were developed, they were checked using individual tools; a second goal then becomes the creation of a simple single tool which integrates all the transforms.

There are 13 main transforms.

1. 2 transforms for creation of a one way pointer for different multiplicities.

2. 3 transforms for creating two-way pointers.

3. 2 transforms for creating an associative object one for an association and the other for an associative object.

4. 2 transforms to transform aggregations.

5. 4 transforms for moving invariants to different places in the AST.

The idea is to be able to create one-way pointers, two-way pointers and associative objects without dealing with the different cases for each structure.
6.1 ASSOCIATION TOOL

The ToolAssociation.java inherits Tool0.java as do all the tools, so that it can be easily added to the toolbox for usage. ToolLoader.java loads an AWL file and creates an AST for all the other tools to work on. Inheriting Tool0, ToolAssociation will be able to obtain the AST to work on.

Based on the idea that an association can be converted into 3 kinds of pointers, three buttons are created for these 3 cases in the tool.

1. One-way pointer.
2. Two-way pointer.
3. Associative object

The third button is also used for creating an associative object pointer by transforming an associative object. The one-way pointer button can be used for transforming aggregation. There are buttons for moving invariants, one for each case. Hence, in total there are 5 buttons for 13 transforms.

According to the design chapter, before applying any transform it is important to check if all the attributes are unique, and that they don’t match the role names of the associations. As this check is common to all the transforms and no transform can be applied to the AST until this is completely true, another button is implemented to check for the clashes. Also, each transform needs to know the association name, aggregation name or associative object which it will be transforming. Hence, there is a combo box to select the structure that the user is looking to transform between association, aggregation and association.
There is another combo box which is filled with the results obtained for the selection of a value in first combo box, i.e. if a user selects an association, the second combo box is filled with all the associations that are present in the AST. No transform will work until the values are selected in the combo box and values will appear in the combo box only after the role names are verified. Hence, after the checkRoleNames button is clicked and everything is all right, values will be filled in the combo boxes. Figure 6.1 shows the tool after an AWL file test2 is loaded and before checkRoleNames.

![Figure 6.1 The Association tool just after start up.](image)

Figure 6.2 shows the tool after the values are selected in the combo boxes. After the values are selected for both the combo boxes, the tool will display both the outline and the AWL specification of the entire model in the text areas in the tool. At this point, the user has an idea about the associations and the class as displayed in the text area. The
user can choose to transform an association, aggregation or associative object by clicking the buttons.

Figure 6.2 The association tool after the role names are verified and the association to be transformed is selected.

6.2 DESCRIPTION OF BUTTONS

6.2.1 ONEWAYPOINTER

This button is used to create a one-way pointer. An association can be converted into a one way pointer and aggregations can only be converted into one-way pointers. Four transforms are integrated in one method which is invoked by clicking this button. Depending upon the choice, i.e. association or aggregation and depending upon the applicability, the transforms are applied. If none of the transforms are applicable, then the related problems are listed in standard out. In this thesis, the main problem that is seen in this one-way pointer case is
• Containing references in the classes where the pointer is not created.

This can be avoided by trying to move the invariant from the toClass to the fromClass by clicking one of the move invariant buttons.

6.2.2 TWOWAYPOINTER

This button is used to create two-way pointers. The three transforms are integrated into one method and are called when the button is clicked. The applicability of each transform is checked and only when the transform is applicable is it applied to the association.

6.2.3 ASSOCIATIVEOBJECT

This button creates an associative object pointer. The transforms are chosen based on the selection of transforming an association or associative object.

6.2.4 MOVEINV

This button moves an invariant from the toClass to either the fromClass or the association, if the invariant has references to the fromClass. This could be mainly used if the association is to be converted into a one-way pointer and there is a reference to the fromClass from the toClass. In this case the reference is caught by applicable() and cannot be transformed; hence, the invariant has to be moved.

There are 2 buttons which are used to display or just clear the values in the combo boxes. They are display and clear respectively.
6.3 SUGGESTED METHODOLOGY

Before applying the transforms to convert associations into pointers, the first thing that has to be done is to check if any of the role names clash with the names of the attributes in that class. A visitor called assocVisitor1 checks and reports the results. Hence the visitor should be applied before transforming any of the structures using the checkRoleName button.

If the association or aggregation should be converted into a one-way pointer, an appropriate transform should be chosen from the transforms NewTransformPreeti1, NewTransformPreeti3, NewTransformPreethi11 and NewTransformPreethi12 depending upon their functionality and this is performed by the button OneWayPointer. If these transforms will not apply because of the references to fromClass then the references have to be moved or transformed. This thesis allows transforming and moving of invariants. Hence a transform from the transforms MoveInvariantTransform, MoveInvariantTransform2, MoveInvariantTransform3, and MoveInvariantTransform4 should be chosen and applied appropriately and this functionality is performed by the button MoveInv. After the references are taken care of, the association can be converted into a one-way pointer.

If the association should be converted into a two-way pointer, an appropriate transform should be chosen from NewTransformPreeti41, NewTransformPreeti42, and NewTransformPreeti43 depending upon their multiplicity. An appropriate transform is chosen by using the button TwoWayPointer.

If an association should be converted into an associative object, the transform TransformAssociativeObject should be applied to the AST. If a WsAssocObject should
be converted into associative object, transform TransformAssociativeObject2 should be applied to the AST. TransformAssociativeObject is used for these transformations.

6.4 SUMMARY

This chapter gives a detailed description of the tool developed which will make it easier to apply the transforms in a certain pattern for correctness. It will help in avoiding the application of transforms which are not to be applied and are not applicable. The methodology of the application of transforms becomes easy with this tool.
7. CONCLUSION AND RECOMMENDATIONS

This thesis began with a focus on transforming association-based expressions. The object oriented approach was followed while developing transforms to approach the problem. Each stage of the software engineering life cycle was thoroughly developed to reach the goal. The goal was to transform the structural model in such a way that all the expressions containing the references to associations, aggregations and associative objects are updated to be consistent with the new structure. An incremental approach was followed by designing simple transforms first and then moving towards complex ones. Despite what was achieved in this thesis, there is a lot of future work that could be done to enhance this research.

7.1 CONCLUSION

Preeti Subedhar’s transforms for associations are able to convert associations, aggregations and associative objects into one-way pointers, two-way pointers, and associative object types. Her transforms only change the structure of the model, they don’t handle the expressions which are based on original associations, aggregations and associative objects.

First the semantics of association reference in AWL were studied and it was decided to change the AWL semantics of association reference to be closer to OCL style. Then the possible cases where the references can be seen are developed in the requirements
analysis stage. The chapter 3 summary section illustrates all the possible cases that have to be handled. Based on the developed specifications, designs were developed to approach each specification and are described in chapter 4. Chapter 5 describes the implementation and test results of each transform and also the supporting transforms and utility methods. Walk through examples are provided in each chapter to help the reader understand the design, implementation and testing strategies. Chapter 6 describes the methodology of applying the transforms to achieve the required goal.

In the process of achieving the main goal, along with the new transforms developed, some of Preeti’s transforms were also modified. The following 14 transforms are responsible to achieve the final goal.

1. **ONE-WAY POINTER TRANSFORMS**

   (i) **NewTransformPreeti1**: Transforms an association into a one-way object pointer when the multiplicity is unity.

   (ii) **NewTransformPreeti3**: Transforms an association into a one-way set pointer when the multiplicity is more than unity.

   (iii) **NewTransformPreethi11**: Transforms an aggregation into a one-way object pointer when the multiplicity is unity.

   (iv) **NewTransformPreethi12**: Transforms an aggregation into a one-way set pointer when the multiplicity is more than unity.
2. TWO-WAY POINTER TRANSFORMS

(i) **NewTransformPreeti4**: Creates two-way pointers when the association has a one-to-one relation.

(ii) **NewTransformPreeti42**: Creates two-way pointers when the association has many-to-many relation.

(iii) **NewTransformPreeti43**: Creates two-way pointers when the association has one-to-many or many-to-one relation.

3. ASSOCIATIVE OBJECT TYPE TRANSFORMS

(i) **TransformAssociativeObject**: Transforms an association into an associative object pointer.

(ii) **TransformAssociativeObject**: Transforms a WsAssocObj into an associative object pointer.

(iii) **TransformSneha**: Creates and adds an aggregation to the AST with multiplicity of both the ends as mandatory. The role names are obtained from the user and the classes in aggregation are the container class and the class where user wants the pointer to be placed.

4. MOVING INVARIANTS TRANSFORMS

(i) **MoveInvariantTransform**: Moves the invariant from the toClass to the fromClass when the multiplicity of both the ends is exactly unity.

(ii) **MoveInvariantTransform2**: Moves the invariant from the toClass to the association when the multiplicity of both the ends is exactly unity.
(iii) **MoveInvariantTransform3**: Moves the invariant from the toClass to the fromClass when the multiplicity of both the ends is either null or unity.

(iv) **MoveInvariantTransform4**: Moves the invariant from the toClass to the association when the multiplicity of both the ends is either null or unity.

These transforms together convert the structural components such as associations, aggregations and associative objects into executable code and also the references to these transformed associations are updated so that they reference the new attributes instead of the association role names.

### 7.2 RECOMMENDATIONS FOR FUTURE RESEARCH

This thesis has achieved the output that was desired while developing the requirements. Along the way of designing and implementing the transforms, several cases have been identified that have to be tackled.

1. Technically, the role names of a certain class participating in different associations can be the same and we should be able to distinguish the references. But, since the linker doesn’t make this distinction and as all the references to role names of same name are considered the same, several limitations were implied. Developing the linker to make this distinction will omit some of the constraints in developing the expressions and associations.

2. Most of the methods in the transforms have been implemented except undo(), explain() and replay().
3. This thesis concentrated on transforming invariants. In the future, transforms should be developed to handle the post and pre conditions referencing associations.

4. The expressions referencing associations with a many-to-many and one-to-many or many-to-one relationships should be transformed or updated in the one-way pointer case.

5. In the one-way pointer case, expressions referencing an associative object have to be updated.

7.3 SUMMARY

Most of these cases are easily fixed once the linker can make the distinction between the role names with same name in different associations. This thesis deals with updating expressions referencing associations, aggregations and associative objects which are transformed into pointers. Combining the work that was done on transforming associations in the past and the work that was done on updating some of the expressions referencing those associations and also the future work to fix other cases that were shown above will create a transformation system that can develop meaningful expressions in the process of converting the specification to executable code.
APPENDIX

JAVA CODE

The following section includes the important methods, i.e., the applicable and the execute methods of the Transforms and leaves out the rest of the source code. The structure of the complete source code is discussed in design chapter.

Transform 1 : NewTransformPreeti1.java

/*************************************************************************
 * Source file: NewTransformPreeti1.java
 * Purpose: Creates a Pointer to the toClass in the fromClass. The classes toClass and the
 *          fromClass are the classes in the association.
 **************************************************************************/

/******
*******
**********
*/

public boolean execute(Object params) {
    System.out.println("In Execute");
    Vector v = (Vector) params;
    //Executes only if applicable
    if (!applicable(target, params)) {
        return false;
    }
    //Variable Declarations
    Vector names = new Vector();
    //The first parameter of the Vector params - association
    WsAssociation assoc = (WsAssociation) v.get(0);
    //The second parameter of the Vector params - from class
    String fclass = (String) v.get(1);
    String toClass = null;
    String toRoleName = null;
    String fromRoleName = null;
    String attype = null;
    String atname = null;
    Vector references = null;
    Vector assocEnds = null;
    WsClass storeclass = null;
    WsExpression invariant = null;
    int option = 0;
    //Checking to see if there is an invariant in the association and
    // making sure that the user wants to continue the transformation
    invariant = assoc.getWsInvariant();
    if (invariant!=null){

while(option<1 || option>2)
  option = Integer.parseInt(JOptionPane.showInputDialog( "There is an invariant in " +
    "the association. You can do 2 things.
    1. Move invariant to the class.
    2. Transform association to associative object.
    If you want to move the invariant to from class enter " +
    "1 to continue else 2 to quit"));
if(option==2){
  return false;
}

// Obtaining the pointer to the WsClass using the name of the from class
storeclass = ToolUtils.returnWsClass(target, fclass);
names = ToolUtils.getClassNames(assoc);
assocEnds = assoc.getWsAssociationEnds();
// Sneha : To find the toClass, in order to create a pointer of type
// toClass and also finding the toRoleName to set the attribute name.
if (((String) names.get(1)).equals(fclass)) {
  toClass = (String) names.get(2);
  toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).
  getWsAssocEndRole());
  fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.
  get(0)).getWsAssocEndRole());
} else {
  toClass = (String) names.get(1);
  toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).
  getWsAssocEndRole());
  fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.
  get(1)).getWsAssocEndRole());
}

// Sneha : to check XformS3.applicable().
ArrayList typename = new ArrayList();
attype = (String) toClass;
typename.add(attype);
atname = (String) toRoleName;
typename.add(atname);
typename.add(target);

if ((XformS3.applicable((WsClass) storeclass, typename))) {
  System.out.println("It is applicable");
  JOptionPane.showMessageDialog(null,
    "Attribute of role name " + toClass +
    " added to AST");
  XformS3 myXform = new XformS3((WsClasses.WsClass) storeclass);
  // pass attrib name that is assoc name and attrib type that is toclass name
  myXform.execute(typename);
} else {
  JOptionPane.showMessageDialog(null,
    "Attribute of role name " + toClass +
    " already exists!");
}

// If there is an invariant in the association, it is moved to the
// class in which the pointer is created.
if (invariant != null) {
    OperandsVisitor vis = new OperandsVisitor();
    invariant.acceptVisitor(vis, null);
    references = vis.getReferences();
    for (Object ref : references) {
        if (((WsIdentifierRef) ref).toString().equals(fromRoleName)) {
            ChangeVariableVisitor newvis = new ChangeVariableVisitor(fromRoleName,"this");
            invariant.acceptVisitor(newvis, null);
        }
    }
    storeclass.addInvariant(invariant);
}

//Removing the association after the pointer is created
target.removeWsDecl(assoc.getName());
return (true);

public static boolean applicable(Object tgt, Object params) {
    Vector v = (Vector) params;
    WsTools.ToolUtils util = new WsTools.ToolUtils();
    String fclassNames = null;
    String toclass = null;
    String multiplicity1 = null;
    String roleName1 = null;
    String roleName2 = null;
    int size = 0;
    int count = 0;
    Vector assocEnds = null;
    Vector names = null;
    Vector references = null;
    Vector roleNames = null;
    Vector assocClassNames = null;
    Enumeration decs = null;
    WsPackage myAST = null;
    WsClasses.WsDeclaration decl = null;
    WsExpression invariant = null;
    WsAssociation assoc = (WsAssociation) v.elementAt(0);
    assocEnds = assoc.getWsAssociationEnds();
    size = assocEnds.size();
    fclassNames = (String) v.elementAt(1);
    myAST = (WsPackage) tgt;
    decs = myAST.getWsDecl();
    invariant = assoc.getWsInvariant();
//returnRoleNames() is a util method which return a vector of all the
//role names of all the associations of the target AST
roleNames = ToolUtils.returnRoleNames(myAST);
//getClassName is a util method which returns the association name
//and the names of two classes participating in the association
assocClassNames = ToolUtils.getClassName(assoc);
//Obtaining the role names of the ends of the association to be transformed
roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).
getWsAssocEndRole());
roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).
getWsAssocEndRole());
//Obtaining the to class name using the given from class name and the
//association
if ((assocClassNames.get(1).toString()).equals(fclassNames)) {
toclass = assocClassNames.get(2).toString();
} else {
toclass = assocClassNames.get(1).toString();
}
// Checking if size of any association is less than 2 or if it is more than 2 returns false.
if (size > 2 || size < 2) {
    System.out.println("The size of the association is more than 2");
    return false;
} //End of Checking for size

//Obtaining the class names in the given AST using the util method to
//check if the classes in the association of interest exist in the
//given AST
names = ToolUtils.returnClassNames(myAST);
count = 0;
for (int k = 0; k < names.size(); k++)
{
    if ((fclassNames.equals((names.elementAt(k)).toString())) {
        count++;
    }
    if ((toclass.equals((names.elementAt(k)).toString()))) {
        count++;
    }
} //for
if (count != 2) {
    System.out.println("The class doesn't exist.");
    return false;
} // End of check for fromClass name

//checking for the multiplicity******Sneha
//Obtaining the multiplicity if the ends of the association and
//trying to keep track of the multiplicity of the fromClass and the
//toClass, in order to check the multiplicity of only the toClass
// of which the pointer is created.
if (toclass.equals(assocClassNames.get(1).toString())) {
    multiplicity1 = (((WsAssocEnd) assocEnds.get(0)).
    getWsAssocEndMultiplicity().elementAt(0).toString());
} else {
    multiplicity1 = (((WsAssocEnd) assocEnds.get(1)).
    getWsAssocEndMultiplicity().elementAt(0).toString());
}
count = 0;
decls = myAST.getWsDecls().elements();

// Checking if the upper bound is not more than one as this transform can only create object.
// Returns false if it finds multiplicity greater than unity
while (decs.hasMoreElements()) {
  decl = (WsClasses.WsDeclaration) decs.nextElement();
  if (((String) decl.getName()).equals(multiplicity1)) {
    if (decl instanceof WsClasses.WsIntegerType) {
      if (!(ToolUtils.toAWLstring(((WsIntegerType) decl).getWsIntUpperBound()).equals("1"))) {
        System.out.println("Class doesn't have one-to-one "+multiplicity");
        return false;
      } //End if
    } //End if
  } //End if
} //End of Checking multiplicity

// Checking to see if there are any references to the fromClass role
// name in any expression in the AST as the association will be
// removed and only to role name exists.
if (!((ToolUtils.canDo(assoc, ToolUtils.returnWsClass(myAST, fclassNames), myAST)))){
  JOptionPane.showMessageDialog(null, "See the report for the list " +"of references found to the from class.\n Transform cannot be" +"applied until the references are moved ");
  return false;
} //End if

// To see if there are more than one association reference in the
// invariant of the Association, if there is one. If there are
// more than one association reference, returns false
if (invariant != null) {
  OperandsVisitor vis = new OperandsVisitor();
  invariant.acceptVisitor(vis, null);
  references = vis.getReferences();
  for (Object ref : references) {
    for (Object roleN : roleNames) {
      if (((WsIdentifierRef) ref).toString().equals((String) roleN)) {
        if (!((((WsIdentifierRef) ref).toString()).equals(roleName1) ||
          (((WsIdentifierRef) ref).toString()).equals(roleName2))) {
          count++;
        } //End if
      } //End if
    } //End for
  } //End for

  if (count > 0) {
    System.out.println("It has more than one association " +"reference in invariant");
    return false;
  } //End if
Transform 2 : NewTransformPreeti3.java

/***************************************************************************
* Source file: NewTransformPreeti3.java
* Purpose: Creates a set pointer of the toClass in the fromClass, 
* toClass and the fromClass are the classes in an association.
***************************************************************************/
/***************************************************************************
* execute() transforms the association into one-way pointer by creating a set pointer of the toClass in the 
* chosen fromClass. The name of the attribute is the name of the toClass role name in the association.
* It also removes the association after creating the pointer
* @param params contains 1. Association to be transformed
* 2. fromClass name chosen by the user
***************************************************************************/

public boolean execute(Object params) {
    System.out.println("Now in Execute");
    Vector v = (Vector) params;
    //Executes only if applicable
    if (!applicable(target, params)) {
        return (false);
    }
    String fclass = (String) v.get(1);
    String toClass = null;
    String toRoleName = null;
    String attype = null;
    String atname = null;
    Vector names = null;
    Vector assocEnds = null;
    WsAssociation assoc = (WsAssociation) v.get(0);
    WsClasses.WsClass storeclass = null;
    //Obtaining the pointer to the WsClass using the name of the fromClass
    storeclass = ToolUtils.returnWsClass(target, fclass);
    //Calling the static method to get the names of the assoc and assoc ends
    names = ToolUtils.getClassNames(assoc);
    assocEnds = assoc.getWsAssociationEnds();
    //Sneha : To find the toClass, in order to create a pointer of type
    // toClass and also finding the toRoleName to set the attribute name.
    if (((String) names.get(1)).equals(fclass)) {
        toClass = (String) names.get(2);
        toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.
            get(1)).getWsAssocEndRole());
    } else {
        toClass = (String) names.get(1);
        toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
    }
    // call the set transform passing the toClass name
    ArrayList v1 = new ArrayList();
    v1.add(toClass + "Set"); // name of the set
    v1.add(toClass); // type of the set
if ((XformT3.applicable(target, v1))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null, "Set Data type for class " + toClass + " added to AST");
    XformT3 myXform = new XformT3(target);
    myXform.execute(v1);
} else {
    JOptionPane.showMessageDialog(null, "Set Data type for class " + toClass + " already exists!");
}

// add toClass set in fromClass and pass the name and type of the toClass set
ArrayList typename = new ArrayList();
attype = (String) toClass + "Set";
typename.add(attype);
atname = (String) toRoleName;
typename.add(atname);
typename.add(target);

//Checking for applicability before applying the transform XformS3
if ((XformS3.applicable((WsClass) storeclass, typename))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null, "Attribute of role name " + toClass + " added to AST");
    XformS3 myXform = new XformS3((WsClass) storeclass); //pass the class
    // pass attrib n
    atname = (String) assocName and attrib type that is toclass name
    myXform.execute(typename);
} else {
    JOptionPane.showMessageDialog(null, "Attribute of role name " + toClass + " already exists!");
}

//Removing the association after the pointer is created
target.removeWsDecl(assoc.getName());
return (true);
} //End of execute

/******************************************************************************
* applicable() is almost same as NewTransformPreeti1 except the
* multiplicity check.
*******************************************************************************/
//Checking if the upper bound is more than one as this transform can
//only create set pointer. Returns false if it finds multiplicity of
//unity
count = 0;
decs = myAST.getWsDecls().elements();
while (decs.hasMoreElements()) {
    decl = (WsClasses.WsDeclaration) decs.nextElement();
    if (((String) decl.getName()).equals(multiplicity1)) {
        if (decl instanceof WsClasses.WsIntegerType) {
            if (ToolUtils.toAWLstring(((WsIntegerType) decl).getWsIntUpperBound()).equals("1")) {

```
System.out.println("toClass doesn't have many" +
" multiplicity");
return false;
}
}
}   //End of Checking multiplicity

Transform 3 : NewTransformPreeti4.java

/******************************************************************************
* Source file: NewTransformPreeti4.java
* Purpose: Creates a two way Pointer between the two classes participating in the association.
******************************************************************************
******************************************************************************
* execute() transforms the association into two-way pointer by creating an object of the opposite ends in
* their respective classes. It also removes the association after creating the pointers.
* @param params contains 1. Association to be transformed
******************************************************************************
******************************************************************************
public boolean execute(Object params) {
    System.out.println("In Execute");
    Vector v = (Vector) params;
    //Checking for the applicability of this transform before executing
    if (!applicable(target, params)) {
        return (false);
    }
    Vector assocEnds = null;
    Vector assocClassNames = null;
    String aName = null;
    WsAssociation assoc = (WsAssociation) v.get(0);
    WsClasses.WsDeclaration decl = null;
    WsClasses.WsClass storeclass1 = null;
    WsClasses.WsClass storeclass2 = null;
    WsClass myClass = null;
    WsExpression invariant = null;
    String class1 = null;
    String class2 = null;
    String attype;
    String atname;
    String roleName1 = null;
    String roleName2 = null;
    String roleName = null;
    int option = 0;

    //Sneha : Calling the static method to get the names of the assoc and assoc ends
    assocClassNames = ToolUtils.getClassNames(assoc);
    aName = assocClassNames.get(0).toString();
    class1 = (String) assocClassNames.get(1);
    class2 = (String) assocClassNames.get(2);
    assocEnds = assoc.getWsAssociationEnds();
    storeclass1 = ToolUtils.returnWsClass(target, class1);
    storeclass2 = ToolUtils.returnWsClass(target, class2);

    //Setting the rolenames to create pointers
    if (class1.equals(storeclass1.toString())) {
        roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
    }
roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
} else {
    roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
    roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
}

//Checking to see if there is an invariant in the association and making sure
//that the user wants to continue the transformation
invariant = assoc.getWsInvariant();
if (invariant != null) {
    while (option < 1 || option > 2) {
        option = Integer.parseInt(JOptionPane.showInputDialog("There is an invariant in the association. You can do 2 things.\n1. Move invariant to one of the classes.\n2. Transform association to associative object.\nIf you want to move the invariant to from class enter 1 to continue else 2 to quit");
    }
    if (option == 2) {
        return false;
    }

    //Obtaining the pointer to the class in which the user chooses to place
    //the possible invariant in the association.
    String response = JOptionPane.showInputDialog("In which class do you want to place the invariant?\nclass1 or class2");
    if (class1.equals(response)) {
        roleName = roleName2;
        myClass = storeclass1;
    } else if (class2.equals(response)) {
        myClass = storeclass2;
        roleName = roleName1;
    }

    ChangeVariableVisitor newvis =
        new ChangeVariableVisitor(roleName, "this");
    ((WsExpression) invariant).acceptVisitor(newvis, null);
    System.out.println(invariant);
    myClass.addInvariant(invariant);
}

// add to class set in the from class and pass the name and type of the attribute
ArrayList typenamename = new ArrayList(); //attribute types and names to add in the from class
attype = class2;
typename.add(attype);
atname = (String) roleName1;
typename.add(atname);
typename.add(target);

// add from class set in the to class and pass the name and type of the attribute
ArrayList typename1 = new ArrayList(); //attribute types and names to add in the from class
attype = class1;
typename1.add(attype);
atname = (String) roleName2;
typename1.add(atname);
typename1.add(target);

// Checking for the applicability of the transform before applying
if ( (XformS3.applicable((WsClass)storeclass1,typename))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null, "Attribute of role name " + storeclass1 + 
    " added to AST");
    XformS3 myXform = new XformS3( (WsClass) storeclass1); //pass the class
    myXform.execute(typename);
} else {
    JOptionPane.showMessageDialog(null, "Attribute of role name " + storeclass1 + 
    " already exists!");
}

// Checking for the applicability of the transform before applying
if ( (XformS3.applicable((WsClass)storeclass2,typename1))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null, "Attribute of role name " + storeclass2 + 
    " added to AST");
    XformS3 myXform1 = new XformS3( (WsClass) storeclass2); //pass the class
    myXform1.execute(typename1);
} else {
    JOptionPane.showMessageDialog(null, "Attribute of role name " + storeclass2 + 
    " already exists!");
}

//Removes the association which has been converted to a pointer
target.removeWsDecl(aName);

return(true);
} //End of execute

***********************************************************************
* applicable() checks to see if this transform is applicable
* It checks for the size of the association to be greater than 1. For the classes in the association to exist
* in the AST. Checks for both classes multiplicity to be mandatory or optional. Also, if there is an
* invariant, checks to see if the expression references only one association.
* @param tgt is the pointer to the AST in which the association is present.
* @param params 1. The association which is to be transformed.
***********************************************************************

public static boolean applicable(Object tgt, Object params) {
    System.out.println("I am in applicable");
    Vector v = (Vector) params;
    WsClasses.WsAssociation assoc = (WsClasses.WsAssociation) v.get(0);
    WsClasses.WsDeclaration decl = null;
    WsClasses.WsDeclaration decl1 = null;
    WsClasses.WsDeclaration decl2 = null;
    WsPackage myAST = (WsPackage) tgt;
    WsExpression invariant = null;
    Enumeration decs = null;
    Vector names = null;
    Vector assocEnds = null;
    Vector roleNames = null;
    Vector references = null;
}
Vector assocClassNames = null;
int count = 0;
int size = 0;
String class1 = null;
String class2 = null;
String multiplicity1 = null;
String multiplicity2 = null;
String roleName1 = null;
String roleName2 = null;
assocEnds = assoc.getWsAssociationEnds();
size = assocEnds.size();
invariant = assoc.getWsInvariant();
roleNames = ToolUtils.returnRoleNames(myAST);
assocClassNames = ToolUtils.getClassNames(assoc);
class1 = assocClassNames.get(1).toString();
class2 = assocClassNames.get(2).toString();
roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0))
    .getWsAssocEndRole());
roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1))
    .getWsAssocEndRole());

// Checking if size of any association is less than 2 if it is returns false.
if (size > 2 || size < 2) {
    System.out.println("The size of the association is more than 2");
    return false;
} //End of Checking for size

//Obtaining the class names in the given AST using the util method to check
//if the classes in the association of interest exist in the given AST
names = ToolUtils.returnClassNames(myAST);
count = 0;
for (int k = 0; k < names.size(); k++) {
    if (class1.equals((names.elementAt(k)).toString())) {
        count++;
    }
    if (class2.equals((names.elementAt(k)).toString())) {
        count++;
    }
}
if (count != 2) {
    System.out.println("The class doesn't exist.");//TCH
    return false;
} //End of check for from class name
//checking for the multiplicity*****Sneha
//Checking for both the classes in association to have a multiplicity of
//one to one.
multiplicity1 = (((WsAssocEnd) assocEnds.get(0)).
    getWsAssocEndMultiplicity().elementAt(0).toString());
multiplicity2 = (((WsAssocEnd) assocEnds.get(1)).
    getWsAssocEndMultiplicity().elementAt(0).toString());
count = 0;
decs = myAST.getWsDecs().elements();

while (decs.hasMoreElements()) {
    decl = (WsClasses.WsDeclaration) decs.nextElement();
if (((String) decl.getName()).equals(multiplicity1)) {
    decl1 = decl;
} else {
    if (((String) decl.getName()).equals(multiplicity2)) {
        decl2 = decl;
    } else {
        //End of Checking multiplicity

        //To see if there are more than one association reference in the invariant of the Association, if there is
        //one. If there are more than one association reference, returns false
        if (invariant != null) {
            OperandsVisitor vis = new OperandsVisitor();
            invariant.acceptVisitor(vis, null);
            references = vis.getReferences();
            for (Object ref : references) {
                for (Object roleN : roleNames) {
                    if (((WsIdentifierRef) ref).toString().equals(String roleN)) {
                        if (! (((WsIdentifierRef) ref).toString().equals(roleName1)) ||
                            (((WsIdentifierRef) ref).toString().equals(roleName2))) {
                            count++;
                        }
                    }
                }
            }
        }
    }
}
//End if
//End if

return true;
} //End of applicable
/**
 * execute() transforms the association into two-way pointer by creating an object of the opposite ends in their respective classes. It also removes the association after creating the set pointers.
 * @param params contains 1. Association to be transformed
 **/

public boolean execute(Object params) {
    System.out.println("In Execute");
    Vector v = (Vector) params;
    if (!applicable(target, params)) {
        return (false);
    }
    Vector assocEnds = null;
    Vector assocClassNames = null;
    Enumeration decs = (Enumeration) target.getWsDecls().elements();
    WsAssociation assoc = (WsAssociation) v.get(0);
    WsClasses.WsDeclaration decl = null;
    WsClasses.WsClass storeclass1 = null;
    WsClasses.WsClass storeclass2 = null;
    String aName = null;
    String class1 = null;
    String class2 = null;
    String attype;
    String atname;
    String roleName1 = null;
    String roleName2 = null;

    // Calling the static method to get the names of the assoc and assoc ends
    assocClassNames = ToolUtils.getClassNames(assoc);
aName = assocClassNames.get(0).toString();
    class1 = (String) assocClassNames.get(1);
    class2 = (String) assocClassNames.get(2);
    assocEnds = assoc.getWsAssociationEnds();
    storeclass1 = ToolUtils.returnWsClass(target, class1);
    storeclass2 = ToolUtils.returnWsClass(target, class2);

    // Setting the rolenames to create pointers
    if (class1.equals(storeclass1.toString())) {
        roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
        roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
    } else {
        roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
        roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
    }

    // call the set transform passing the toClass name
    ArrayList v1 = new ArrayList(); // Creating the toClass set
    v1.add(class1 + "Set"); // name of the set
    v1.add(class1); // type of the set

    ArrayList v2 = new ArrayList(); // Creating the fromClass set
    v2.add(class2 + "Set"); // name of the set
    v2.add(class2); // type of the set

    // checking for applicable for toClass and creating the set
    if ((XformT3.applicable(target, v1))) {
        System.out.println("XformT3 is applicable");
    }
JOptionPane.showMessageDialog(null, "Set Data type for class " + storeclass1 +" added to AST");
XformT3 myXform = new XformT3(target);
myXform.execute(v1);
} else {
    JOptionPane.showMessageDialog(null, "Set Data type for class " + storeclass1 +" already exists!");
}

// checking for aplicable for fromClass and creating the set
if ((XformT3.applicable(target, v2))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null, "Set Data type for class " + storeclass2 +" added to AST");
    XformT3 myXform = new XformT3(target);
    myXform.execute(v2);
} else {
    JOptionPane.showMessageDialog(null, "Set Data type for class " + storeclass2 +" already exists!");
}

// add toClass set in the from class and pass the name and type of the attribute
ArrayList typename = new ArrayList(); //attribute types and names to add in the fromClass
attype = class2 + "Set";
typename.add(attype);
atname = (String) roleName1;
typename.add(atname);
typename.add(target);

// add fromClass set in the toClass and pass the name and type of the attribute
ArrayList typename1 = new ArrayList(); //attribute types and names to add in the fromClass
attype = class1 + "Set";
typename1.add(attype);
atname = (String) roleName2;
typename1.add(atname);
typename1.add(target);

//Checking the applicability of the Transform which creates the attribute
if ((XformS3.applicable((WsClass) storeclass1, typename))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null, "Attribute of role name " + storeclass1 +" added to AST");
    XformS3 myXform = new XformS3((WsClass) storeclass1); //pass the class
    myXform.execute(typename);
} else {
    JOptionPane.showMessageDialog(null, "Attribute of role name " + storeclass1 +" already exists!");
}

//Checking the applicability of the Transform which creates the attribute
if ((XformS3.applicable((WsClass) storeclass2, typename1))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null, "Attribute of role name " + storeclass2 +" added to AST");
    XformS3 myXform1 = new XformS3((WsClass) storeclass2); //pass the class
    myXform1.execute(typename1);
} else {
    JOptionPane.showMessageDialog(null, 
    "Attribute of role name " + storeclass2 +" already exists!");
}

//removing the association that is transformed
target.removeWsDecl(aName);
return true;
} //End of execute

/***********************************************************************
* applicable() checks to see if this transform is applicable
* It checks for the size of the association to be greater than 1.
* For the classes in the association to exist in the AST.
* Checks for both classes multiplicity to be more than unity.
* Also, if there is an invariant, the transform is not applicable.
* @param tgt is the pointer to the AST in which the association is present.
* @param params 1. The association which is to be transformed.
***********************************************************************/
public static boolean applicable(Object tgt, Object params) {
    System.out.println("In applicable");
    Vector v = (Vector) params;
    WsClasses.WsAssociation assoc = (WsClasses.WsAssociation) v.get(0);
    WsClasses.WsDeclaration decl = null;
    WsClasses.WsDeclaration decl1 = null;
    WsClasses.WsDeclaration decl2 = null;
    WsPackage myAST = (WsPackage) tgt;
    WsExpression invariant = null;
    Enumeration decs = null;
    Vector names = null;
    Vector assocEnds = null;
    Vector assocClassNames = null;
    int count = 0;
    int size = 0;
    String class1 = null;
    String class2 = null;
    String multiplicity1 = null;
    String multiplicity2 = null;

    //Using static methods to obtain association ends and the class names
    //of the association
    assocEnds = assoc.getWsAssociationEnds();
    size = assocEnds.size();
    assocClassNames = ToolUtils.getClassNames(assoc);
    class1 = assocClassNames.get(1).toString();
    class2 = assocClassNames.get(2).toString();

    // Checking if size of any association is less than 2 if it is returns false.
    if (size > 2 || size < 2) {
        System.out.println("The size of the association is more than 2");
        return false;
    }
    //End of Checking for size

    // checking the fromClass exists in the tree
    names = ToolUtils.returnClassNames(myAST);
count = 0;
for (int k = 0; k < names.size(); k++) {
    if (class1.equals((names.elementAt(k)).toString())) {
        count++;
    }
    if (class2.equals((names.elementAt(k)).toString())) {
        count++;
    }
}
if (count != 2) {
    System.out.println("The class doesn't exist.");
    return false;
}
// End of check for fromClass name

//checking for the multiplicity*****Sneha
//Checking for both the classes in association to have a multiplicity of
//many to many.
multiplicity1 = (((WsAssocEnd) assocEnds.get(0)).getWsAssocEndMultiplicity().elementAt(0).toString());
multiplicity2 = (((WsAssocEnd) assocEnds.get(1)).getWsAssocEndMultiplicity().elementAt(0).toString());
count = 0;
decs = myAST.getWsDecls().elements();

while (decs.hasMoreElements()) {
    decl = (WsClasses.WsDeclaration) decs.nextElement();
    if (((String) decl.getName()).equals(multiplicity1)) {
        decl1 = decl;
    }
    if (((String) decl.getName()).equals(multiplicity2)) {
        decl2 = decl;
    }
}
if (decl1 instanceof WsClasses.WsIntegerType) {
    if ((ToolUtils.toAWLstring(((WsIntegerType) decl1).getWsIntUpperBound()).equals("1"))) {
        System.out.println("Class doesn't have many-to-many multiplicity");
        return false;
    }
}
if (decl2 instanceof WsClasses.WsIntegerType) {
    if ((ToolUtils.toAWLstring(((WsIntegerType) decl2).getWsIntUpperBound()).equals("1"))) {
        System.out.println("Class doesn't have many-to-many multiplicity");
        return false;
    }
}
//End of Checking multiplicity

//Checking for the invariant
invariant = assoc.getWsInvariant();
if (invariant != null) {
    JOptionPane.showMessageDialog(null, "There is an invariant in association");
}
return true;
} //End of applicable
public boolean execute(Object params) {
    System.out.println("In Execute");
    Vector v = (Vector) params;
    if (!applicable(target, params)) {
        return (false);
    } Vector assocEnds = null;
    Vector assocClassNames = null;
    Enumeration decs = (Enumeration) target.getWsDecls().elements();
    WsAssociation assoc = (WsAssociation) v.get(0); WsClasses.WsDeclaration decl = null;
    WsClasses.WsDeclaration decl1 = null;
    WsClasses.WsDeclaration decl2 = null;
    WsClasses.WsClass storeclass1 = null;
    WsClasses.WsClass storeclass2 = null;
    WsExpression invariant = null;
    String aName = null;
    String class1 = null;
    String class2 = null;
    String attype = null;
    String atname = null;
    String multiplicity1 = null;
    String multiplicity2 = null;
    String roleName1 = null;
    String roleName2 = null;
    int option = 0;

    //Sneha : Calling the static method to get the names of the assoc and assoc ends
    assocClassNames = ToolUtils.getClassNames(assoc);
    aName = assocClassNames.get(0).toString();
    class1 = (String) assocClassNames.get(1);
    class2 = (String) assocClassNames.get(2);
    assocEnds = assoc.getWsAssociationEnds();
    storeclass1 = ToolUtils.returnWsClass(target, class1);
    storeclass2 = ToolUtils.returnWsClass(target, class2);

    //Setting the role names which are used to create pointers
    if (class1.equals(storeclass1.toString())) {
        roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
        roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
    } else {
        roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
        roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
    }
// Checking to see if there is an invariant in the association and making sure
// that the user wants to continue the transformation
invariant = assoc.getWsInvariant();
if (invariant!=null){
    while (option<1 || option>2)
        option = Integer.parseInt(JOptionPane.showInputDialog("There is an invariant in " +
                "the association. You can do 2 things.\n" +
                "1. Move invariant to the class.\n" +
                "2. Transform association to associative object.\n" +
                "If you want to move the invariant to fromClass enter " +
                "1 to continue else 2 to quit");
        if (option==2){
            return false;
        }
}
// Multiplicities of the two ends.
multiplicity1 = (((WsAssocEnd) assocEnds.get(0)).getWsAssocEndMultiplicity().elementAt(0).toString());
multiplicity2 = (((WsAssocEnd) assocEnds.get(1)).getWsAssocEndMultiplicity().elementAt(0).toString());
decs = target.getWsDecls().elements();
while (decs.hasMoreElements()) {
    decl = (WsClasses.WsDeclaration) decs.nextElement();
    if ((((String) decl.getName()).equals(multiplicity1)) { 
        decl1 = decl;
    }
    if ((((String) decl.getName()).equals(multiplicity2)) {
        decl2 = decl;
    }
}
// call the set transform passing the class name
// Applies only when the multiplicity is more than 1
if (!(ToolUtils.toAWLstring(((WsIntegerType) decl1).getWsIntUpperBound()).equals("1"))) {
    ArrayList v1 = new ArrayList(); // Creating the toClass set
    v1.add(class1 + "Set"); // name of the set
    v1.add(class1); // type of the set
    if (XformT3.applicable(target, v1)) {
        System.out.println("XformT3 is applicable");
        JOptionPane.showMessageDialog(null,
                "Set Data type for class " + storeclass1 +
                " added to AST");
        XformT3 myXform = new XformT3(target);
        myXform.execute(v1);
        if (invariant!=null){
            ChangeVariableVisitor newvis =
                    new ChangeVariableVisitor(roleName2, "this");
            ((WsExpression) invariant).acceptVisitor(newvis, null);
            storeclass1.addInvariant(invariant);
        }
    }
} else {
JOptionPane.showMessageDialog(null, 
    "Set Data type for class " + storeclass1 + 
    " already exists!");

// Applies only when the multiplicity is more than 1
if (!ToolUtils.toAWLstring(((WsIntegerType) decl2).getWsIntUpperBound()).equals("1")) {
    ArrayList v2 = new ArrayList(); // Creating the fromClass set
    v2.add(class2 + "Set"); // name of the set
    v2.add(class2); // type of the set

    // checking for applicable for fromClass and creating the set
    if (XformT3.applicable(target, v2)) {
        System.out.println("It is applicable");
        JOptionPane.showMessageDialog(null, 
            "Set Data type for class " + storeclass2 + 
            " added to AST");
        XformT3 myXform = new XformT3(target);
        myXform.execute(v2);
        if (invariant!=null){
            ChangeVariableVisitor newvis = 
                new ChangeVariableVisitor(roleName1, "this");
            ((WsExpression) invariant).acceptVisitor(newvis, null);
            storeclass2.addInvariant(invariant);
        }
    }
    else {
        JOptionPane.showMessageDialog(null, 
            "Set Data type for class " + storeclass2 + 
            " already exists!");
    }
}

// add class set in the from class and pass the name and type of the attribute
ArrayList typename = new ArrayList(); //attribute types and names to add in the fromClass
if (!ToolUtils.toAWLstring(((WsIntegerType) decl2).getWsIntUpperBound()).equals("1")) {
    attype = class2 + "Set";
} else {
    attype = class2;
}
typename.add(attype);
atname = (String) roleName1;
typename.add(atname);
typename.add(target);

// add class set in the toClass and pass the name and type of the attribute
ArrayList typename1 = new ArrayList(); //attribute types and names to add in the fromClass
if (!ToolUtils.toAWLstring(((WsIntegerType) decl1).getWsIntUpperBound()).equals("1")) {
    attype = class1 + "Set";
} else {
    attype = class1;
}
typename1.add(attype);
atname = (String) roleName2;
typename1.add(atname);
typename1.add(target);
if (XformS3.applicable((WsClass) storeclass1, typename)) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null,
        "Attribute of role name " + storeclass1 +
        " added to AST");
    XformS3 myXform = new XformS3((WsClass) storeclass1); //pass the class
    myXform.execute(typename);
} else {
    JOptionPane.showMessageDialog(null,
        "Attribute of role name " + storeclass1 +
        " already exists!");
}

//Checks for the applicability before applying it
if (XformS3.applicable((WsClass) storeclass2, typename1)) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null,
        "Attribute of role name " + storeclass2 +
        " added to AST");
    XformS3 myXform1 = new XformS3((WsClass) storeclass2); //pass the class
    myXform1.execute(typename1);
} else {
    JOptionPane.showMessageDialog(null,
        "Attribute of role name " + storeclass2 +
        " already exists!");
}

//Removes the association which has to be converted to a pointer
target.removeWsDecl(aName);

return (true);
} //End of execute

/***************/
/*public static boolean applicable(Object tgt, Object params)
 * Argument "params" contains 1.Association
 * It checks to see that 1.The size of the association is less than 2
 * 2.The fromClass and the toClass exists.
 * 3.Checks the multiplicity of the ends of the association.
 **********************************************/
public static boolean applicable(Object tgt, Object params) {
    System.out.println("In applicable");
    Vector v = (Vector) params;
    int size = 0;
    int count = 0;
    WsClasses.WsAssociation assoc = (WsClasses.WsAssociation) v.get(0);
    WsClasses.WsDeclaration decl = null;
    WsClasses.WsDeclaration decl1 = null;
    WsClasses.WsDeclaration decl2 = null;
    WsPackage myAST = (WsPackage) tgt;
    WsExpression invariant = null;
    Enumeration decs = null;
    Vector names = null;
Vector assocEnds = null;
Vector roleNames = null;
Vector references = null;
Vector assocClassNames = null;
String class1 = null;
String class2 = null;
String multiplicity1 = null;
String multiplicity2 = null;
String roleName1 = null;
String roleName2 = null;
assocEnds = assoc.getWsAssociationEnds();
size = assocEnds.size();
invariant = assoc.getWsInvariant();
roleNames = ToolUtils.returnRoleNames(myAST);
assocClassNames = ToolUtils.getClassNames(assoc);
class1 = assocClassNames.get(1).toString();
class2 = assocClassNames.get(2).toString();
roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());

// Checking if size of any association is less than 2 if it is returns false.
if (size > 2 || size < 2) {
    System.out.println("The size of the association is more than 2");
    return false;
}

// checking the fromClass exists in the tree
names = ToolUtils.returnClassNames(myAST);
count = 0;
for (int k = 0; k < names.size(); k++) {
    if (class1.equals((names.elementAt(k)).toString()))
        count++;
    if (class2.equals((names.elementAt(k)).toString()))
        count++;
}
if (count != 2) {
    System.out.println("The class doesn't exist.");
    return false;
}

//checking for the multiplicity*****Sneha
//Checking for both the classes in association to have a multiplicity of
//one to many or many to one.
multiplicity1 = (((WsAssocEnd)
assocEnds.get(0)).getWsAssocEndMultiplicity().elementAt(0).toString());
multiplicity2 = (((WsAssocEnd)
assocEnds.get(1)).getWsAssocEndMultiplicity().elementAt(0).toString());
count = 0;
decs = myAST.getWsDecs().elements();
while (decs.hasMoreElements()) {
decl = (WsClasses.WsDeclaration) decs.nextElement();
if (((String) decl.getName()).equals(multiplicity1)) {
    decl1 = decl;
} else if (((String) decl.getName()).equals(multiplicity2)) {
    decl2 = decl;
}
if (decl1 instanceof WsClasses.WsIntegerType && decl2 instanceof WsClasses.WsIntegerType) {
    if (ToolUtils.toAWLstring(((WsIntegerType) decl1).getWsIntUpperBound()).equals("1")) {
        if (ToolUtils.toAWLstring(((WsIntegerType) decl2).getWsIntUpperBound()).equals("1")) {
            System.out.println("Association doesn't have atleast one role with many multiplicity");
            return false;
        }
    } else {
        if (!(ToolUtils.toAWLstring(((WsIntegerType) decl1).getWsIntUpperBound()).equals("1"))) {
            if (!(ToolUtils.toAWLstring(((WsIntegerType) decl2).getWsIntUpperBound()).equals("1"))) {
                System.out.println("Association doesn't have atleast one role with unit multiplicity");
                return false;
            }
        }
    }
} //End of Checking multiplicity

//To see if there are more than one association reference in the invariant of the Association, if there is
//one. If there are more than one association reference, returns false
if (invariant != null) {
    OperandsVisitor vis = new OperandsVisitor();
    invariant.acceptVisitor(vis, null);
    references = vis.getReferences();
    for (Object ref : references) {
        for (Object roleN : roleNames) {
            if (((WsIdentifierRef) ref).toString().equals(String.valueOf(roleN))) {
                if (!(((WsIdentifierRef) ref).toString()).equals(roleName1) ||
                    (((WsIdentifierRef) ref).toString()).equals(roleName2))) {
                    count++;
                }
            }
        }
    }
} //End for
} //End if
if (count > 0) {
    System.out.println("It has more than one association reference in invariant");
    return false;
} //End if
return true;
} //End of applicable
Transform 6 : NewTransformPreeti5.java

/************************************************************************
* Source file: NewTransformPreeti5.java
* Creates a new class and adds it to the AST
**************************************************************************/

public boolean execute(Object params) {
    Vector v = new Vector();
    int dtcount = 0;
    v = (Vector) params;
    String dtname = (String) v.get(0) + "link";
    System.out.println("dtname : " + dtname);
    String dname = (String) v.get(0) + "1";
    System.out.println("dname : " + dname);
    WsClass t = new WsClass();
    WsClass t1 = new WsClass();
    t.setWsDeclName(new WsIdentifier(dtname)); // set the name of the class
    t1.setWsDeclName(new WsIdentifier(dname)); // Sneha: set the name of the container class
    targetObject.addWsDecl(t); // adding the class to the tree
    targetObject.addWsDecl(t1); // Sneha: adding the class to the tree
    Vector decs = targetObject.getWsDecls();
    for (int i = 0; i < decs.size(); i++) {
        WsDeclaration d = (WsDeclaration) decs.get(i);
        if (d instanceof WsClass) {
            dtcount = dtcount + 1;
        } //End of if
    } //Inner for
    if (dtcount == 0)
        System.out.println("There are no classes to display");
    return(true);//TCH
} //End of execute

/************************************************************************
* Method applicable() appends the String "link" to the name of the class
* checks for the duplication of the data type and returns false if the data type already exists.
* @param tgt is a package, AST
* @param params takes a vector that has a string as the name of the class.
**************************************************************************/

public static boolean applicable(Object tgt, Object params) {
    Vector v = new Vector();
    v = (Vector) params;
    for (int j = 0; j < v.size(); j++) {
        v.get(j);
        String set = (String) v.get(j) + "link";
        Vector decs = ((WsClasses.WsPackage) tgt).getWsDecls();
        for (int i = 0; i < decs.size(); i++) {
            WsDeclaration d = (WsDeclaration) decs.get(i);
            if (d instanceof WsClass && d.getName().equals(set)) {
                System.out.println("There are no classes to display");
                return(false);//TCH
            } //End of if
        } //Inner for
    } //End of for
    return(true);//TCH
} //End of applicable

Transform 7 : TransformAssociativeObject.java

/***********************************************************************************
* Source file: TransformAssociativeObject.java
* Purpose: calls the transform to create a new class and
* Creates a two way Pointer between the two classes of an association in that new class and also
* creates a container class holding the reference to the associations developed. Also creates an aggregation
* between the container and the class in which pointer is to be created.
************************************************************************************/

/***********************************************************************************
* execute() transforms the association into associative object
* Also, it moves the invariant from the association to the new class with the class links.
* @param params contains 1.Association 2. child class name 3. parent name 4. child name
************************************************************************************/

public boolean execute(Object params) {
    System.out.println("In Execute");
    Vector v = (Vector) params;
    if (!applicable(target, params)) {
        return (false);
    }
    Vector assocEnds = null;
    Vector assocClassNames = null;
    WsAssociation assoc = (WsAssociation) v.get(0);
    WsClass storeclass1 = null;
    WsClass storeclass2 = null;
    WsClass storefclass = null;
    WsClass newClass = null;
    WsExpression invariant = null;
    String aName = null;
    String class1 = null;
    String class2 = null;
    String attype;
    String atname;
    String roleName1 = null;
    String roleName2 = null;

    //Sneha : Calling the static method to get the names of the assoc and assoc ends
    assocClassNames = ToolUtils.getClassNames(assoc);
    aName = assocClassNames.get(0).toString();
    class1 = (String) assocClassNames.get(1);
    class2 = (String) assocClassNames.get(2);
    assocEnds = assoc.getWsAssociationEnds();
    invariant = assoc.getWsInvariant();
    storeclass1 = ToolUtils.returnWsClass(target, class1);
    storeclass2 = ToolUtils.returnWsClass(target, class2);

    //Setting the rolenames to create pointers
    roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).
getWsAssocEndRole();
roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).
getWsAssocEndRole());

// adding the new class to the tree
Vector va = new Vector(); // Creating the toClass set
va.add(aName); // name of the set
if (((TransformPreeti5.applicable(target, va))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null,
            "class " + aName + "link" +" added to AST");
    TransformPreeti5 myXform = new TransformPreeti5(target);
    myXform.execute(va);
} else {
    JOptionPane.showMessageDialog(null,
            "class " + aName + "link" +" already exists!");
}

va = new Vector(); // Creating the toClass set
va.add(aName + "Container"); // name of the set
if (((TransformPreeti5.applicable(target, va))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null,
            "class " + aName + "Containerlink" +" added to AST");
    TransformPreeti5 myXform = new TransformPreeti5(target);
    myXform.execute(va);
} else {
    JOptionPane.showMessageDialog(null,
            "class " + aName + "Containerlink" +" already exists!");
}

storefclass = ToolUtils.returnWsClass(target, (aName + "link");
newClass = ToolUtils.returnWsClass(target, (aName + "Container" +"link");

// call Bhoomas transform to add a pointer to the toClass in the new class
ArrayList typename = new ArrayList(); //attribute types and names to add in the fromClass
attype = (String) class2;
typename.add(attype);
atname = (String) roleName2;
typename.add(atname);
typename.add(target);

if ((XformS3.applicable((WsClass) storefclass, typename))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null,
            "Attribute of role name " + storeclass1 +" added to AST");
    XformS3 myXform = new XformS3((WsClass) storefclass); //pass the class
    myXform.execute(typename);
} else {
    JOptionPane.showMessageDialog(null,
            "Attribute of role name " + storeclass1 +" already exists!");
}

// add fromClass set in the toClass and pass the name and type of the attribute
ArrayList typename1 = new ArrayList(); //attribute types and names to add in the fromClass
attype = (String) class1;
typename1.add(attype);
atname = (String) roleName1;
typename1.add(atname);
typename1.add(target);

if ((XformS3.applicable((WsClass) storefclass, typename1))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null,
        "Attribute of role name " + storeclass2 +" added to AST");
    XformS3 myXform1 = new XformS3((WsClass) storefclass); //pass the class
    myXform1.execute(typename1);
} else {
    JOptionPane.showMessageDialog(null,
        "Attribute of role name " + storeclass2 +" already exists!");
}

//create a set of the new class that is added to the tree
ArrayList v3 = new ArrayList(); // Creating the fromClass set
v3.add(aName + "linkSet"); // name of the set
v3.add(aName + "link"); // type of the set

// checking for applicable for new class and creating the set
if ((XformT3.applicable(target, v3))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null,
        "Set Data type for class " + aName + "link" + " added to AST");
    XformT3 myXform5 = new XformT3(target);
    myXform5.execute(v3);
} else {
    JOptionPane.showMessageDialog(null,
        "Set Data type for class " + aName + "link" + " already exists!");
}

ArrayList typename2 = new ArrayList(); //attribute types and names to add in the fromClass
attype = (String) aName + "link" + "Set";
typename2.add(attype);
atname = (String) (aName + "1link");
typename2.add(atname);
typename2.add(target);

if ((XformS3.applicable((WsClass) newClass, typename2))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null,
        "Attribute of role name " + newClass +" added to AST");
    XformS3 myXform2 = new XformS3((WsClasses.WsClass) newClass); //pass the name of the class
    myXform2.execute(typename2); // pass attrib name that is assoc name and attrib type that is toclass name
} else {
    JOptionPane.showMessageDialog(null,
        "Attribute of role name " + newClass +" already exists!");
}

//Calling the transform to create aggregation of the container class and any existing class
Vector agg = new Vector();
agg.add((String) v.get(1));
agg.add((String) v.get(2));
agg.add((String) v.get(3));
agg.add(assoc.getName());

if (TransformSneha.applicable(target, agg)) {
    TransformSneha snehaForm = new TransformSneha(target);
    snehaForm.execute(agg);
} else {
    JOptionPane.showMessageDialog(null, "This transform is not applicable-Sneha");
}

if (invariant != null) {
    storefclass.addInvariant(invariant);
} target.removeWsDecl(aName);
return (true);
} //End of execute

/*******************************************************************************/
*************
* applicable() checks 1.The size of the association is less than 2, 2.The fromClass and toClass exists. 
* 3.Checks if there are any references to the associative object in any class. 
* 4.checks if the invariant in association references only one association. 
* @param tgt contains the target package in which association is present. 
* @param params contains 1.Association 
*******************************************************************************/

public static boolean applicable(Object tgt, Object params) {
    System.out.println("In applicable");
    Vector v = (Vector) params;
    int size = 0;
    WsClasses.WsAssociation assoc = (WsClasses.WsAssociation) v.get(0);
    WsExpression invariant = null;
    WsPackage myAST = (WsPackage) tgt;
    Vector names = new Vector();
    Vector assocEnds = null;
    Vector references = null;
    Vector roleNames = null;
    Vector invariants = new Vector();
    Vector assocClassNames = null;
    int count = 0;
    String class1 = null;
    String class2 = null;
    String roleName1 = null;
    String roleName2 = null;

    //Using static methods to obtain the role names of all the associations
    //in the package and also the class names of the ends in association.
    assocEnds = assoc.getWsAssociationEnds();
    size = assocEnds.size();
    roleNames = ToolUtils.returnRoleNames(myAST);
    assocClassNames = ToolUtils.getClassNames(assoc);
    class1 = assocClassNames.get(1).toString();
    class2 = assocClassNames.get(2).toString();

    //Obtaining the invariants of the classes in association.
    invariants.add((ToolUtils.returnWsClass(myAST, class1)).getWsInvariant());
invariants.add((ToolUtils.returnWsClass(myAST, class2)).getWsInvariant());

// Obtaining the rolenames of the classes in association.
roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).
    getWsAssocEndRole());
roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).
    getWsAssocEndRole());
invariant = assoc.getWsInvariant();

// Checking if size of any association is less than 2 if it is returns false.
if (size > 2 || size < 2) {
    System.out.println("The size of the association is more than 2");
    return false;
}
// End of Checking for size

if (!ToolUtils.canDoAssocToObject(assoc, myAST))
{
    System.out.println("Found references in the classes to the role names");
    return false;
} // checking for the classes to exist in the tree.
names = ToolUtils.returnClassNames(myAST);
count = 0;
for (int k = 0; k < names.size(); k++) {
    if (class1.equals((names.elementAt(k)).toString())) {
        count++;
    }
    if (class2.equals((names.elementAt(k)).toString())) {
        count++;
    }
}
if (count != 2) {
    System.out.println("The class doesn't exist.");
    return false;
} // End of check for fromClass name

// Checking the invariant of the association if it references more than
// one association.
count = 0;
if (invariant != null) {
    OperandsVisitor vis = new OperandsVisitor();
    invariant.acceptVisitor(vis, null);
    references = vis.getReferences();
    for (Object ref : references) {
        for (Object roleN : roleNames) {
            if (!((WsIdentifierRef) ref).toString().
                equals((String) roleN)) {
                if (((WsIdentifierRef) ref).toString().
                    equals(roleName1) ||
                    ((WsIdentifierRef) ref).toString().
                        equals(roleName2))) {
                    count++;
                }
            }
        }
    }
}
System.out.println("count is "+count);
if (count > 0) {
    System.out.println("It has more than one association " +
    "reference in invariant");
    return false;
}
//End if
//End if
//End of checking for only one association reference
return true;
//End of applicable

Transform 8: TransformAssociativeObject2.java

/***************************************************************
* Source file: TransformAssociativeObject2.java
* Purpose: calls the transform to create a new class and Creates a two way Pointer between the two
* classes of an associative object in that new class.
***************************************************************
/***************************************************************
* execute() transforms the associative object into associative object
* Also, it moves the invariant from the associative object to the new class with the class links.
* @param params contains 1. Associative Object 2. child class name 3. parent name 4. child name
***************************************************************
public boolean execute(Object params) {
    System.out.println("In Execute");
    Vector v = (Vector) params;
    if (!applicable(target, params)) {
        return false;
    }
    WsAssocObject assoc = (WsAssocObject) v.get(0);
    Vector assocEnds = null;
    Vector assocClassNames = null;
    WsClass storeclass1 = null;
    WsClass storeclass2 = null;
    WsClass storefclass = null;
    WsClass newClass = null;
    WsExpression invariant = null;
    String aName = null;
    String class1 = null;
    String class2 = null;
    String attype = null;
    String atname = null;
    String roleName1 = null;
    String roleName2 = null;
    //Sneha : Calling the static method to get the names of the assocObj and assocObj ends
    assocClassNames = ToolUtils.getObjectClassNames(assoc);
aName = assocClassNames.get(0).toString();
class1 = (String) assocClassNames.get(1);
class2 = (String) assocClassNames.get(2);
assocEnds = assoc.getWsAssocObjectEnds();
invariant = assoc.getWsInvariant();
storeclass1 = ToolUtils.returnWsClass(target, class1);
storeclass2 = ToolUtils.returnWsClass(target, class2);

//Setting the rolenames to create pointers
if (class1.equals(storeclass1.toString())) {
    roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).
    getWsAssocEndRole());
    roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).
    getWsAssocEndRole());
} else {
    roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).
    getWsAssocEndRole());
    roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).
    getWsAssocEndRole());
}

// adding the new class to the tree
Vector va = new Vector(); // Creating the toClass set
va.add(aName); // name of the set
if ((TransformPreeti5.applicable(target, va))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null, 
    "class " + aName + " link" + " added to AST");
    TransformPreeti5 myXform = new TransformPreeti5(target);
    myXform.execute(va);
} else {
    JOptionPane.showMessageDialog(null, 
    "class " + aName + "Containerlink" + " already exists!");
}

va = new Vector(); // Creating the toClass set
va.add(aName + "Container"); // name of the set
if ((TransformPreeti5.applicable(target, va))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null, 
    "class " + aName + "Containerlink" + " added to AST");
    TransformPreeti5 myXform = new TransformPreeti5(target);
    myXform.execute(va);
} else {
    JOptionPane.showMessageDialog(null, 
    "class " + aName + "Containerlink" + " already exists!");
}

storeclass = ToolUtils.returnWsClass(target, (aName + "link"));
newClass = ToolUtils.returnWsClass(target, (aName + "Container" + "link"));

// call Bhooma's transform to add a pointer to the toClass in the new class
ArrayList typename = new ArrayList(); //attribute types and names to add in the fromClass
attype = (String) class2;
typename.add(attype);
atname = (String) roleName2;
typename.add(atname);
typename.add(target);

if ((XformS3.applicable((WsClass) storefclass, typename))) {
    System.out.println("It is applicable");
JOptionPane.showMessageDialog(null, "Attribute of role name "+ storeclass1 + " added to AST");

XformS3 myXform = new XformS3((WsClass) storefclass); //pass the class
myXform.execute(typename);
}
else {
    JOptionPane.showMessageDialog(null, "Attribute of role name "+ storeclass1 + " already exists!");
}

// add fromClass set in the toClass and pass the name and type of the attribute
ArrayList typename1 = new ArrayList(); //attribute types and names to add in the fromClass
attype = (String) class1;
typename1.add(attype);
atname = (String) roleName1;
typename1.add(atname);
typename1.add(target);
if ((XformS3.applicable((WsClass) storefclass, typename1))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null, "Attribute of role name "+ storeclass2 + " added to AST");
    XformS3 myXform1 = new XformS3((WsClass) storefclass); //pass the class
    myXform1.execute(typename1);
} else {
    JOptionPane.showMessageDialog(null, "Attribute of role name "+ storeclass2 + " already exists!");
}

//create a set of the new class that is added to the tree
//add an attribute of this set to the systemclass using bhoomas transform
ArrayList v3 = new ArrayList(); // Creating the fromClass set
v3.add(aName + "linkSet"); // name of the set
v3.add(aName + "link"); // type of the set
if ((XformT3.applicable(target, v3))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null, "Set Data type for class " + aName + "link" + " added to AST");
    XformT3 myXform5 = new XformT3(target);
    myXform5.execute(v3);
} else {
    JOptionPane.showMessageDialog(null, "Set Data type for class " + aName + "link" + " already exists!");
}

ArrayList typename2 = new ArrayList(); //attribute types and names to add in the fromClass
attype = (String) aName + "link" + "Set";
typename2.add(attype);
atname = (String) (aName + "1link");
typename2.add(atname);
typename2.add(target);
if ((XformS3.applicable((WsClass) newClass, typename2))) {

}
System.out.println("It is applicable");
JOptionPane.showMessageDialog(null,
   "Attribute of role name " + newClass + " added to AST");
XformS3 myXform2 = new XformS3((WsClasses.WsClass) newClass); //pass the name of the class
myXform2.execute(typename2); // pass attrib name that is assoc name and attrib type that is toclass
else {
    JOptionPane.showMessageDialog(null,
    "Attribute of role name " + newClass + " already exists!");
}

//Calling the transform to create aggregation of the container class and any existing class
Vector agg = new Vector();
agg.add((String) v.get(1));
agg.add((String) v.get(2));
agg.add((String) v.get(3));
agg.add(assoc.getName());
if (TransformSneha.applicable(target, agg)) {
    TransformSneha snehaForm = new TransformSneha(target);
snehaForm.execute(agg);
} else {
    JOptionPane.showMessageDialog(null, "This transform is not applicable-Sneha");
}

if (invariant != null) {
    storefclass.addInvariant(invariant);
}

target.removeWsDecl(aName);
return (true);
} //End of execute

*******************************************************************************/
* public static boolean applicable(Object tgt, Object params)*
* Argument "params" contains 1.Associative Object*
* It checks to see that*
* 1.The size of the associative Object is less than 2*
* 2.The fromClass and toClass exists.
* 3.Checks if there are any references to association in any class*
*******************************************************************************/
public static boolean applicable(Object tgt, Object params) {
    System.out.println("In applicable");
    Vector v = (Vector) params;
    int size = 0;
    int count = 0;
    WsAssocObject assoc = (WsAssocObject) v.get(0);
    WsExpression invariant = null;
    WsPackage myAST = (WsPackage) tgt;
    Vector names = null;
    Vector assocEnds = null;
    Vector references = null;
    Vector roleNames = null;
    Vector invariants = new Vector();
    Vector assocClassNames = null;
    String class1 = null;
String class2 = null;
String roleName1 = null;
String roleName2 = null;

assocEnds = assoc.getWsAssocObjectEnds();
size = assocEnds.size();
roleNames = ToolUtils.returnRoleNames(myAST);
assocClassNames = ToolUtils.getObjectClassNames(assoc);
class1 = assocClassNames.get(1).toString();
class2 = assocClassNames.get(2).toString();
invariants.add((ToolUtils.returnWsClass(myAST, class1)).getWsInvariant());
invariants.add((ToolUtils.returnWsClass(myAST, class2)).getWsInvariant());
roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
invariant = assoc.getWsInvariant();

// Checking if size of any associative Object is less than 2 if it is returns false.
if (size > 2 || size < 2) {
    System.out.println("The size of the association is more than 2");
    return false;
}

// checking the fromClass exists in the tree
names = ToolUtils.returnClassNames(myAST);
count = 0;
for (int k = 0; k < names.size(); k++) {
    if (class1.equals((names.elementAt(k)).toString())) {
        count++;
    }
    if (class2.equals((names.elementAt(k)).toString())) {
        count++;
    }
}
if (count != 2) {
    System.out.println("The class doesn't exist.");//TCH
    return false;
}

// End of check for fromClass name

if (!ToolUtils.canDoAssocObject(assoc, myAST)) {
    System.out.println("Found references in the classes to the role names");
    return false;
}

//Checking the invariant of the associative Object if it references
//more than one associative Object.
count = 0;
if (invariant != null) {
    OperandsVisitor vis = new OperandsVisitor();
invariant.acceptVisitor(vis, null);
    references = vis.getReferences();
    for (Object ref : references) {

for (Object roleN : roleNames) {
    if (!(ref instanceof WsThis)) {
        if (ToolUtils.toAWLstring((WsIdentifierRef) ref)).equals((String) roleN) {
            if (!(((ToolUtils.toAWLstring((WsIdentifierRef) ref)).equals(roleName1)) ||
                    ((ToolUtils.toAWLstring((WsIdentifierRef) ref)).equals(roleName2)))) {
                count++;
            }
        }
    }
} //End for
} //End for

if (count > 0) {
    System.out.println("It has more than one associative Object " +
            "reference in invariant");
    return false;
} //End if
} //End if
//End of checking for only one association reference

return true;
} //End of applicable

Transform 9 : NewTransformPreeti11.java

/******************************************************************************
* Source file: NewTransformPreeti11.java
* Purpose: Transforms the aggregation into a pointer in parent class. This transformation places a pointer
* of child class in the parent class of the aggregation.
*******************************************************************************/
public boolean execute(Object params) {
    System.out.println("In Execute");
    Vector v = (Vector) params;
    if (!applicable(target, params)) {
        return false;
    }
    Vector names = null;
    Vector references = null;
    Vector assocEnds = null;
    WsAssociation assoc = (WsAssociation) v.get(0);
    WsClass storeclass = null;
    WsExpression invariant = null;
    String toRoleName = null;
    String fromRoleName = null;
    String class1 = null;
    String class2 = null;
String attype;
String atname;

//Sneha : Calling the static method to get the names of the agg and agg ends
names = ToolUtils.getClassNames(assoc);
class1 = (String) names.get(1);
class2 = (String) names.get(2);
assocEnds = assoc.getWsAssociationEnds();
invariant = assoc.getWsInvariant();
storeclass = ToolUtils.returnWsClass(target, class1);

//Sneha: Setting the attribute name
toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());

// Sneha : to check XformS3.applicable().
ArrayList typename = new ArrayList(); //attribute types and names to add in the fromClass
attype = (String) class2;
typename.add(attype);
atname = (String) toRoleName;
typename.add(atname);
typename.add(target);
if (XformS3.applicable((WsClass) storeclass, typename)) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null, "Attribute of role name " + class2 + " added to AST");
    XformS3 myXform = new XformS3((WsClasses.WsClass) storeclass); //pass the name of the class
    // pass attrib name that is agg name and attrib type that is toClass name
    myXform.execute(typename);
} else {
    JOptionPane.showMessageDialog(null, "Attribute of role name " + class2 + " already exists!");
}

//If there is an invariant in the aggregation, it is moved to the class
//in which the pointer is created.
if (invariant != null) {
    OperandsVisitor vis = new OperandsVisitor();
invariant.acceptVisitor(vis, null);
references = vis.getReferences();
for (Object ref : references) {
    if (((WsIdentifierRef) ref).toString().equals(fromRoleName)) {
        ChangeVariableVisitor newvis = new ChangeVariableVisitor(fromRoleName,"this");
invariant.acceptVisitor(newvis, null);
    }
}
storeclass.addInvARIANT(invariant);
}
//Removing the aggregation after the pointer is created
target.removeWsDecl(assoc.getName());
return (true);
applicable() checks to see if this transform is applicable
* It checks for the size of the aggregation to be greater than 1.
* For the classes in the aggregation to exist in the AST.
* Checks whether the multiplicity is either mandatory or optional.
* Checks of there are any references pointing to the toClass
* Also, if there is an invariant, checks to see if the expression
* references only one aggregation.
* @param tgt is the pointer to the AST in which the aggregation is present.
* @param params 1. The aggregation which is to be transformed.

```java
public static boolean applicable(Object tgt, Object params) {
    Vector v = (Vector) params;
    WsAssociation assoc = (WsAssociation) v.elementAt(0);
    int count = 0;
    int size = 0;
    WsPackage myAST = null;
    WsClasses.WsDeclaration decl = null;
    WsExpression invariant = null;
    Enumeration decs = null;
    Vector names = null;
    Vector references = null;
    Vector roleNames = null;
    Vector assocEnds = null;
    Vector assocClassNames = null;
    String class1 = null;
    String class2 = null;
    String multiplicity1 = null;
    String roleName1 = null;
    String roleName2 = null;

    assocEnds = assoc.getWsAssociationEnds();
    size = assocEnds.size();
    myAST = (WsPackage) tgt;
    decs = myAST.getWsDecls().elements();
    invariant = assoc.getWsInvariant();
    roleNames = ToolUtils.returnRoleNames(myAST);

    //obtaining the assoc class names using the tool method developed.
    assocClassNames = ToolUtils.getClassNames(assoc);
    class1 = assocClassNames.get(1).toString();
    class2 = assocClassNames.get(2).toString();
    roleName1 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).
        getWsAssocEndRole());
    roleName2 = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).
        getWsAssocEndRole());

    // Checking if size of any association is less than 2 or if it is more than 2 returns false.
    if (size > 2 || size < 2) {
        System.out.println("The size of the association is more than 2");
        return false;
    }
    //End of Checking for size
```
names = ToolUtils:returnClassNames(myAST);
count = 0;
for (int k = 0; k < names.size(); k++)
{
  if ((class1.equals((names.elementAt(k)).toString()))
    { count++; }
  if ((class2.equals((names.elementAt(k)).toString())))
    { count++; }
}
//for
if (count != 2) {
  System.out.println("The class doesn't exist.");//TCH
  return false;
}
// End of check for fromClass name

//checking for the multiplicity*****Sneha
multiplicity1 = (((WsAssocEnd) assocEnds.get(1)).
  getWsAssocEndMultiplicity().elementAt(0).toString());
count = 0;
decs = myAST.getWsDecls().elements();
while (decs.hasMoreElements())
{
  decl = (WsClasses.WsDeclaration) decs.nextElement();
  if (((String) decl.getName()).equals(multiplicity1))
  { if (decl
        instanceof WsClasses.WsIntegerType) {
      if (!(!ToolUtils.toAWLstring(((WsIntegerType) decl).
          getWsIntUpperBound()).equals("1")))
      { System.out.println("Class doesn't have one-to-one " +
          "multiplicity");
        return false;
      }
    }
  }
}
//End of Checking multiplicity

//To see if there are more than one aggregation reference in the
//invariant of the Aggregation, if there is one. If there are
//more than one aggregation reference, returns false
if (invariant != null) {
  OperandsVisitor vis = new OperandsVisitor();
invariant.acceptVisitor(vis, null);
  references = vis.getReferences();
  for (Object ref : references) {
    for (Object roleN : roleNames) {
      if (((WsIdentifierRef) ref).toString()).
        equals((String) roleN))
        { if (!(((WsIdentifierRef) ref).toString()).
            equals(roleName1)) ||
          (((WsIdentifierRef) ref).toString()).
            equals(roleName2)))
          { count++;
          }
        }
    }
  } //End for
} //End for
Transform 10 : NewTransformPreeti12.java

/**********************************************************
* Source file: NewTransformPreeti12.java
* Purpose:Transforms the aggregation into a pointer in
* parent class. This transformation places a pointer of
* child class in the parent class of the aggregation of
* multiplicity more than unity.
**********************************************************/

public boolean execute(Object params) {
    System.out.println("Now in Execute");
    Vector v = (Vector) params;
    if (!applicable(target, params)) {
        return (false);
    }
    Vector names = new Vector();
    WsAssociation assoc = (WsAssociation) v.get(0);
    Vector assocEnds = null;
    WsClasses.WsClass storeclass = null;
    String toRoleName = null;
    String attype;
    String atname;
    String class1;
    String class2;
    //Calling the static method to get the names of the agg and agg ends
    names = ToolUtils.getClassNames(assoc);
    class1 = (String) names.get(1);
    class2 = (String) names.get(2);
    assocEnds = assoc.getWsAssociationEnds();
    storeclass = ToolUtils.returnWsClass(target, class1);
    //Setting the attribute name
    toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
// call the set transform passing the toClass name
ArrayList v1 = new ArrayList();
v1.add(class2 + "Set"); // name of the set
v1.add(class2); // type of the set
if ((XformT3.applicable(target, v1))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null,
        "Set Data type for class " + class2 + 
        " added to AST");
    XformT3 myXform = new XformT3(target);
    myXform.execute(v1);
} else {
    JOptionPane.showMessageDialog(null,
        "Set Data type for class " + class2 + 
        " already exists!");
}

// add toClass set in fromClass and pass the name and type of the toClass set
ArrayList typename = new ArrayList();
attype = (String) class2 + "Set";
typename.add(attype);
atname = (String) toRoleName;
typename.add(atname);
typename.add(target);

// Checking for applicability before applying the transform XformS3
if ((XformS3.applicable((WsClass) storeclass, typename))) {
    System.out.println("It is applicable");
    JOptionPane.showMessageDialog(null,
        "Attribute of role name " + class2 + 
        " added to AST");
    XformS3 myXform = new XformS3((WsClass) storeclass); // pass the class
    // pass attrib name that is assoc name and attrib type that is toclass name
    myXform.execute(typename);
} else {
    JOptionPane.showMessageDialog(null,
        "Attribute of role name " + class2 + 
        " already exists!");
}

target.removeWsDecl(assoc.getName());

    return true;
} //End of execute

/* *****************************************/
* applicable() checks to see if this transform is applicable
* It checks for the size of the aggregation to be greater than 1.
* For the classes in the aggregation to exist in the AST.
* Checks whether the multiplicity is more than unity.
* Checks of there are any references pointing to the toClass
* Also, checks if there is an invariant
* @param tgt is the pointer to the AST in which the aggregation is present.
* @param params 1. The aggregation which is to be transformed.
* *****************************************/
public static boolean applicable(Object tgt, Object params) {
Vector v = (Vector) params;
WsAssociation assoc = null;
int size = 0;
WsClasses.WsDeclaration decl = null;
WsClasses.WsPackage myAST = null;
Enumeration decs = null;
Vector names = null;
WsExpression invariant = null;
int count = 0;
String class1 = null;
String class2 = null;
String multiplicity1 = null;
Vector assocClassNames = null;
Vector assocEnds = null;
assoc = (WsAssociation) v.elementAt(0);
assocEnds = assoc.getWsAssociationEnds();
size = assocEnds.size();
myAST = (WsPackage) tgt;
invariant = assoc.getWsInvariant();
assocClassNames = ToolUtils.getClassNames(assoc);
class1 = assocClassNames.get(1).toString();
class2 = assocClassNames.get(2).toString();

// Checking if size of any association is less than 2 if it is returns false.
if (size > 2 || size < 2) {
    System.out.println("The size of the association is more than 2");
    return false;
}
// End of Checking for size

names = ToolUtils.returnClassNames(myAST);
// checking the fromClass exists in the tree
count = 0;
for (int k = 0; k < names.size(); k++) {
    if (((class1.equals((names.elementAt(k)).toString()))) {
        count++;
    }
    if (((class2.equals((names.elementAt(k)).toString()))) {
        count++;
    }
}
if (count != 2) {
    System.out.println("The class doesn't exist.");
    return false;
}
// End of check for fromClass name

//checking for the multiplicity*****Sneha
multiplicity1 = (((WsAssocEnd) assocEnds.get(1)).
    getWsAssocEndMultiplicity().elementAt(0).toString());
count = 0;
decs = myAST.getWsDecls().elements();
while (decs.hasMoreElements()) {
    decl = (WsClasses.WsDeclaration) decs.nextElement();
    if (((String) decl.getName()).equals(multiplicity1)) {
        if (decl instanceof WsClasses.WsIntegerType) {
if (ToolUtils.toAWLstring(((WsIntegerType) decl).getWsIntUpperBound()).equals("1")) {
    System.out.println("toClass doesn't have many" + " multiplicity");
    return false;
}
} //End of Checking multiplicity

if (invariant!=null) {
    return false;
} //End of checking for invariant

return true;
} //End of applicable

Transform 11 : MoveInvariantTransform.java

Transient 11 : MoveInvariantTransform.java

/* Source file: MoveInvariantTransform
 * Purpose: Moves the invariant from the toClass to
 * the fromClass if it contains any references of
 * the fromClass role name of the association.
 */
public boolean execute(Object params) {
    System.out.println("In Execute");
    if (!applicable(myAST,params))
        return false;
    Vector v = (Vector) params;
    Vector expressions = null;
    Vector names = null;
    Vector newExp = null;
    Vector unwantedExp = null;
    Vector assocEnds = null;
    WsAssociation assoc = (WsAssociation) v.get(0);
    WsExpression invariant = null;
    WsClass fromClass = null;
    WsClass toClass = null;
    String fromName = (String) v.get(1);
    String toRoleName = null;
    String fromRoleName = null;
    String toName = null;
    boolean valid = false;

    //returns the WsClass for the given class name
    fromClass = ToolUtils.returnWsClass(myAST, fromName);
    //obtaining the names of the classes in association
    names = ToolUtils.getClassNames(assoc);
    assocEnds = assoc.getWsAssociationEnds();
// Obtaining the role names and also the toClass names with the given
// fromClass name.
if (((String) names.get(1)).equals(fromName)) {
    toName = (String) names.get(2);
    toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.
        get(1)).getWsAssocEndRole());
    fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.
        get(0)).getWsAssocEndRole());
} else {
    toName = (String) names.get(1);
    toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.
        get(0)).getWsAssocEndRole());
    fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.
        get(1)).getWsAssocEndRole());
}

toClass = ToolUtils.returnWsClass(myAST, toName);
invariant = toClass.getWsInvariant();

// Splitting the invariant if there are more than one
if (InvariantSplit.applicable(invariant, null)) {
    InvariantSplit inv = new InvariantSplit(invariant);
    inv.execute(null);
    expressions = inv.getExps();
}

// Checking for the expression with the fromRoleName reference
// Vector obtained if there is only one expression then the else part is
// executed.
if (expressions != null) {
    newExp = new Vector();
    unwantedExp = new Vector();
    valid = false;
    for (Object exp : expressions) {
        valid = this.hasFromRole(fromRoleName, (WsExpression) exp);
        if (valid) {
            ChangeVariableVisitor newvis1 =
                new ChangeVariableVisitor("this", toRoleName);
            ((WsExpression) exp).acceptVisitor(newvis1, null);
            ChangeVariableVisitor newvis =
                new ChangeVariableVisitor(fromRoleName, "this");
            ((WsExpression) exp).acceptVisitor(newvis, null);
            newExp.add((WsExpression) exp);
        } else {
            unwantedExp.add((WsExpression) exp);
        }
    }
} else {
    valid = this.hasFromRole(fromRoleName, invariant);
    if (valid) {
        ChangeVariableVisitor newvis1 =
            new ChangeVariableVisitor("this", toRoleName);
        invariant.acceptVisitor(newvis1, null);
        ChangeVariableVisitor newvis =
            new ChangeVariableVisitor(fromRoleName, "this");
        invariant.acceptVisitor(newvis, null);
    }
if(newExp!="null"){
    if (newExp.size() > 1) {
        for (Object exp : newExp) {
            fromClass.addInvariant((WsExpression) exp);
        }
        toClass.setWsInvariant(unwantedExp);
    } else if (newExp.size() == 1) {
        fromClass.addInvariant((WsExpression) newExp.get(0));
        toClass.setWsInvariant(unwantedExp);
    } else {
        fromClass.addInvariant(invariant);
        String newInv = null;
        toClass.setWsInvariant(newInv);
    }
    return true;
}
**************************************************************************
* applicable() checks 1. if both the classes in association exist.
* 2. Checks for the multiplicity of both the ends to be mandatory.
* 3. Checks if the association has only one assoc reference.
* 4. Checks if the toClass has forRoleName as reference in invariant.
* 5. Also if there is any invariant in the toClass.
* @param target is the AST containing the association.
* @param params contains 1. association.
* 2.fromClass name.
**************************************************************************
public static boolean applicable(Object target, Object params) {
    System.out.println("In applicable");
    Vector v = (Vector) params;
    WsPackage myAST = (WsPackage) target;
    WsAssociation assoc = (WsAssociation) v.get(0);
    WsDeclaration decl = null;
    WsDeclaration decl1 = null;
    WsDeclaration decl2 = null;
    WsExpression invariant = null;
    WsClass toClass = null;
    String fromClassName = (String) v.get(1);
    Vector classNames = null;
    Vector assocClassNames = null;
    Vector assocEnds = null;
    Vector references = null;
    Vector roleNames = null;
    Enumeration decs = null;
    int count = 0;
    int count1 = 0;
    String class2 = null;
    String multiplicity1 = null;
    String multiplicity2 = null;
    String toRoleName = null;
    String fromRoleName = null;
    String toName = null;
assocClassNames = ToolUtils.getClassNames(assoc);
class2 = assocClassNames.get(2).toString();
classNames = ToolUtils.returnClassNames(myAST);
assocEnds = assoc.getWsAssociationEnds();

if (((String) assocClassNames.get(1)).equals(fromClassName)) {

toName = (String) assocClassNames.get(2);
toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
} else {

toName = (String) assocClassNames.get(1);
toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
}
toClass = ToolUtils.returnWsClass(myAST, toName);
invariant = toClass.getWsInvariant();
count = 0;

//check if the class exists
for (int k = 0; k < classNames.size(); k++) {
    if (fromClassName.equals((classNames.elementAt(k)).toString())) {
        count++;
    }
    if (class2.equals((classNames.elementAt(k)).toString())) {
        count++;
    }
}
if (count != 2) {
    System.out.println("The class doesn't exist.");
    return false;
}

//check multiplicity of both the ends to be mandatory
multiplicity1 = (((WsAssocEnd) assocEnds.get(0)).getWsAssocEndMultiplicity().elementAt(0).toString());
multiplicity2 = (((WsAssocEnd) assocEnds.get(1)).getWsAssocEndMultiplicity().elementAt(0).toString());
decs = myAST.getWsDecls().elements();

while (decs.hasMoreElements()) {
    decl = (WsClasses.WsDeclaration) decs.nextElement();
    if (((String) decl.getName()).equals(multiplicity1)) {
        decl1 = decl;
    }
    if (((String) decl.getName()).equals(multiplicity2)) {
        decl2 = decl;
    }
}
count = 0;
if (decl1 instanceof WsClasses.WsIntegerType) {
    if (((ToolUtils.toAWLstring(((WsIntegerType) decl1).toString()))).equals(multiplicity1)) {
        // Further checks...
    }
}
getWsIntUpperBound()).equals("1")&&
(ToolUtils.toAWLstring(((WsIntegerType) decl1).getWsIntLowerBound()).
equals("1")))

    count++;
}]
if (decl2 instanceof WsClasses.WsIntegerType) {
    if ((ToolUtils.toAWLstring(((WsIntegerType) decl2).getWsIntUpperBound()).equals("1"))&&
(ToolUtils.toAWLstring(((WsIntegerType) decl2).getWsIntLowerBound()).equals("1")))
    count++;
}
if (count<2) {
    System.out.println("The multiplicity is not mandatory for both ends");
    return false;
} //End of Checking multiplicity

//Checking the invariant for only one association reference
count = 0;
roleNames = ToolUtils.returnRoleNames(myAST);

if (invariant!=null) {
    OperandsVisitor vis = new OperandsVisitor();
invariant.acceptVisitor(vis, null);
    references = vis.getReferences();
    for (Object ref : references){
        for (Object roleN : roleNames){
            if (!((WsThis)ref).equals((String)roleN)){
                if (!((WsIdentifierRef)ref).toString().equals(toRoleName)||
(ToolUtils.toAWLstring((WsIdentifierRef)ref).toString()).equals(fromRoleName)))
                    count++;
            }
        }
    }
}
if (count>0) {
    System.out.println("It has more than one association "+
        "reference in invariant");
    return false;
} //End if
if (count1<1) {
    System.out.println("There are no references to the fromClass" +
        "from the to class in the invariant ");
    return false;


Transform 12 : MoveInvariantTransform2.java

/**************************
 * Source file: MoveInvariantTransform2
 * Purpose: Moves the invariant from the toClass to
 * the association itself if it contains any references of
 * the fromClass role name of the association.
 *************************/

/****************************
 * execute() moves the invariant with the from rolenames to the association
 * as the pointer will be placed in that class.
 * @param params contain a vector of 1. Association
 * 2. From class name
 ****************************/

public boolean execute(Object params) {
    System.out.println("In Execute");
    if (!applicable(myAST, params)) {
        return false;
    }
    Vector v = (Vector) params;
    Vector expressions = null;
    Vector names = null;
    Vector newExp = null;
    Vector unwantedExp = null;
    Vector assocEnds = null;
    WsAssociation assoc = (WsAssociation) v.get(0);
    WsExpression invariant = null;
    WsClass fromClass = null;
    WsClass toClass = null;
    String fromName = (String) v.get(1);
    String toRoleName = null;
    String fromRoleName = null;
    boolean valid = false;
    //returns the WsClass for the given class name
    fromClass = ToolUtils.returnWsClass(myAST, fromName);
    //obtaining the names of the classes in association
    names = ToolUtils.getClassNames(assoc);
    assocEnds = assoc.getWsAssociationEnds();

    //Obtaining the role names and also the toClass names with the given
    //from class name.
    if (((String) names.get(1)).equals(fromName)) {
        toName = (String) names.get(2);
        toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).
                                           getWsAssocEndRole());
    }
fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
} else {

toName = (String) names.get(1);
toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
}

toClass = ToolUtils.returnWsClass(myAST, toName);
invariant = toClass.getWsInvariant();

//Splitting the invariant if there are more than one
if (InvariantSplit.applicable(invariant, null)) {
    InvariantSplit inv = new InvariantSplit(invariant);
    inv.execute(null);
    expressions = inv.getExps();
}

//Checking for the expression with the fromRoleName reference
//Vector obtained if there is only one expression then the else part is executed.
if (expressions != null) {
    newExp = new Vector();
    unwantedExp = new Vector();
    valid = false;
    for (Object exp : expressions) {
        valid = this.hasFromRole(fromRoleName, (WsExpression) exp);
        if (valid) {
            ChangeVariableVisitor newvis1 =
                new ChangeVariableVisitor("this", toRoleName);
            ((WsExpression) exp).acceptVisitor(newvis1, null);
            newExp.add((WsExpression) exp);
        } else {
            unwantedExp.add((WsExpression) exp);
        }
    }
} else {
    valid = this.hasFromRole(fromRoleName, invariant);
    if (valid) {
        ChangeVariableVisitor newvis1 =
            new ChangeVariableVisitor("this", toRoleName);
        invariant.acceptVisitor(newvis1, null);
    }
}

//now moving only those expression with fromRoleName references
if (newExp != null) {
    if (newExp.size() >= 1) {
        for (Object exp : newExp) {
            assoc.addInvariant((WsExpression) exp);
        }
    }
    toClass.setWsInvariant(unwantedExp);
} else {
    assoc.addInvariant(invariant);
}
String newInv = null;
toClass.setWsInvariant(newInv);
}

return true;
}

/***************************************************************************
* applicable() is same as MoveInvariantTransform
***************************************************************************/

Transform 13 : MoveInvariantTransform3.java

/***************************************************************************
* Source file: MoveInvariantTransform3
* Purpose: Moves the invariant from the toClass to
* the fromClass if it contains any references of
* the fromClass role name of the association. This is the
* case when the association multiplicity is optional.
***************************************************************************/

/**
* execute() moves the invariant
* with the from rolenames to the fromClass
* as the pointer will be placed in that class.
* @param params contain a vector of 1. Association
* 2. fromClass name
*/

public boolean execute(Object params) {
    System.out.println("In Execute");
    if (!applicable(myAST,params))
        return false;
    Vector v = (Vector) params;
    Vector expressions = null;
    Vector names = null;
    Vector newExp = null;
    Vector unwantedExp = null;
    WsAssociation assoc = (WsAssociation) v.get(0);
    WsExpression invariant = null;
    WsTraverseVisitor newvis = null;
    WsClass fromClass = null;
    WsClass toClass = null;
    String fromName = (String) v.get(1);
    String toRoleName = null;
    String fromRoleName = null;
    String toName = null;
    boolean valid = false;
    Vector assocEnds = null;

    //returns the WsClass for the given class name
    fromClass = ToolUtils.returnWsClass(myAST, fromName);
    //obtaining the names of the classes in association
    names = ToolUtils.getClassNames(assoc);
    assocEnds = assoc.getWsAssociationEnds();

    //Obtaining the role names and also the toClass names with the given
    //fromClass name.
    if (((String) names.get(1)).equals(fromName)) {

toName = (String) names.get(2);
toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.
    get(1)).getWsAssocEndRole());

fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.
    get(0)).getWsAssocEndRole());

} else {
toName = (String) names.get(1);
toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.
    get(0)).getWsAssocEndRole());

fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.
    get(1)).getWsAssocEndRole());
}
toClass = ToolUtils.returnWsClass(myAST, toName);
invariant = toClass.getWsInvariant();

//Splitting the invariant if there are more than one
if (InvariantSplit.applicable(invariant, null)) {
    InvariantSplit inv = new InvariantSplit(invariant);
    inv.execute(null);
    expressions = inv.getExps();
}

//Checking for the expression with the fromRoleName reference int
//Vector obtained if there is only one expression then the else part is
//executed.
if (expressions != null) {
    newExp = new Vector();
    unwantedExp = new Vector();
    valid = false;
    for (Object exp : expressions) {
        valid = this.hasFromRole(fromRoleName, (WsExpression) exp);
        if (valid) {
            if ((WsExpression)exp instanceof WsImplication)
                newvis = new InvariantVisitor2(toRoleName);
            ((WsExpression) exp).acceptVisitor(newvis, null);
        }
    }
} else {
    valid = this.hasFromRole(fromRoleName, invariant);
    if (valid) {
        if (invariant instanceof WsImplication)
            newvis = new InvariantVisitor2(toRoleName);
    }
    newExp.add((WsExpression) exp);
    }
invariant.acceptVisitor(newvis, null);
    }
    newvis = new ChangeVariableVisitor("this", toRoleName);
    invariant.acceptVisitor(newvis, null);
    newvis = new ChangeVariableVisitor(fromRoleName, "this");
    invariant.acceptVisitor(newvis, null);
    }
    }

    //now moving only those expression with fromRoleName references
    if (newExp != null) {
        if (newExp.size() > 1) {
            for (Object exp : newExp) {
                fromClass.addInvariant((WsExpression) exp);
            }
            toClass.setWsInvariant(unwantedExp);
        } else if (newExp.size() == 1) {
            fromClass.addInvariant((WsExpression) newExp.get(0));
            toClass.setWsInvariant(unwantedExp);
        } else {
            fromClass.addInvariant(invariant);
            String newInv = null;
            toClass.setWsInvariant(newInv);
        }
    } else {
        fromClass.addInvariant(invariant);
        String newInv = null;
        toClass.setWsInvariant(newInv);
    }
    return true;
    }
    }

/***************************************************************************/
* applicable() checks 1. if both the classes in association exist.    
* 2. Checks for the multiplicity of both the ends to be optional.         
* 3. Checks if the association has only one assoc reference.            
* 4. Checks if the toClass has forRoleName as reference in invariant.    
* 5. Also if there is any invariant in the toClass.                     
* @param target is the AST containing the association.                 
* @param params contains 1. association. 2.fromClass name.             
/***************************************************************************/
public static boolean applicable(Object target, Object params) {
    System.out.println("In applicable");
    Vector v = (Vector) params;
    WsPackage myAST = (WsPackage) target;
    WsAssociation assoc = (WsAssociation) v.get(0);
    WsDeclaration decl = null;
    WsDeclaration decl1 = null;
    WsDeclaration decl2 = null;
    WsExpression invariant = null;
    WsClass toClass = null;
    String fromClassName = (String) v.get(1);
    Vector classNames = null;
    Vector assocClassNames = null;
    Vector assocEnds = null;
    Vector references = null;
    Vector roleNames = null;
    Enumeration decs = null;
    int count = 0;
    int count1 = 0;
    return true;
    }
    }

153
String class2 = null;
String multiplicity1 = null;
String multiplicity2 = null;
String toRoleName = null;
String fromRoleName = null;
String toName = null;

assocClassNames = ToolUtils.getClassNames(assoc);
class2 = assocClassNames.get(2).toString();
classNames = ToolUtils.returnClassNames(myAST);
assocEnds = assoc.getWsAssociationEnds();
System.out.println("fromClassName, "+fromClassName+" , "+(String) classNames.get(1));
if (((String) assocClassNames.get(1)).equals(fromClassName)) {
    toName = (String) assocClassNames.get(2);
    toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
    fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
} else {
    toName = (String) assocClassNames.get(1);
    toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0)).getWsAssocEndRole());
    fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1)).getWsAssocEndRole());
}

toClass = ToolUtils.returnWsClass(myAST, toName);
invariant = toClass.getWsInvariant();
//check if the class exists
for (int k = 0; k < classNames.size(); k++) {
    if (fromClassName.equals((classNames.elementAt(k)).toString())) {
        count++;
    }
    if (class2.equals((classNames.elementAt(k)).toString())) {
        count++;
    }
}
if (count != 2) {
    System.out.println("The class doesn't exist.");
    return false;
}
//check multiplicity of both the ends to be optional
multiplicity1 = (((WsAssocEnd) assocEnds.get(0)).getWsAssocEndMultiplicity().elementAt(0).toString());
multiplicity2 = (((WsAssocEnd) assocEnds.get(1)).getWsAssocEndMultiplicity().elementAt(0).toString());
count = 0;
decs = myAST.getWsDecls().elements();
while (decs.hasMoreElements()) {
    decl = (WsClasses.WsDeclaration) decs.nextElement();
    if (((String) decl.getName()).equals(multiplicity1)) {
        decl1 = decl;
    }
}
if (((String) decl.getName()).equals(multiplicity2)) {
    decl2 = decl;
}
}

count = 0;
if (decl1 instanceof WsClasses.WsIntegerType) {
    if ((ToolUtils.toAWLString(((WsIntegerType) decl1).getWsIntUpperBound()).equals("1") &&
        (ToolUtils.toAWLString(((WsIntegerType) decl1).getWsIntLowerBound()).equals("0")))
        count++;}
if (decl2 instanceof WsClasses.WsIntegerType) {
    if ((ToolUtils.toAWLString(((WsIntegerType) decl2).getWsIntUpperBound()).equals("1") &&
        (ToolUtils.toAWLString(((WsIntegerType) decl2).getWsIntLowerBound()).equals("0")))
        count++;}
if (count<2)
{
    System.out.println("The multiplicity is not mandatory for both ends");
    return false;
}
//End of Checking multiplicity

count = 0;
roleNames = ToolUtils.returnRoleNames(myAST);
//Checking the invariant for only one association reference
if (invariant!=null)
{
    OperandsVisitor vis = new OperandsVisitor();
    invariant.acceptVisitor(vis, null);
    references = vis.getReferences();
    for (Object ref : references){
        for (Object roleN : roleNames){
            if (!((WsThis))
                if ((ToolUtils.toAWLString(((WsIdentifierRef)ref).toString()).equals((String)roleN)) \n                    if (((WsIdentifierRef)ref).toString()).equals(toRoleName) ||
                        !(((WsIdentifierRef)ref).toString()).equals(fromRoleName))
                        count++;
                //End of checking for only one association reference
                //check if the toClass has a reference to forClass
                if (((WsIdentifierRef)ref).toString()).equals(fromRoleName))
                    count1++;}}
    //End for
    }//End for
} //End for
if (count>0) {
    System.out.println("It has more than one association " +
    "reference in invariant");
    return false;
} //End if

if (count<1) {
    System.out.println("There are no references to the fromClass" +
    "from the toClass in the invariant ");
    return false;
} //End of checking for the fromClass reference

else {
    System.out.println("There is no invariant in the toClass");
    return false;
}

return true;

Transform 14 : MoveInvariantTransform4

/*******************************************************************************
* Source file: MoveInvariantTransform4
* Purpose: Moves the invariant from the toClass to the association itself if it contains any references of
* the fromClass role name of the association in the case of multiplicity being optional.
***********************************************************************/

public boolean execute(Object params) {
    System.out.println("In Execute");
    if (!applicable(myAST, params)) {
        return false;
    }
    Vector v = (Vector) params;
    Vector expressions = null;
    Vector names = null;
    Vector newExp = null;
    Vector unwantedExp = null;
    WsAssociation assoc = (WsAssociation) v.get(0);
    WsTraverseVisitor newvis = null;
    WsExpression invariant = null;
    WsExpression temp = null;
    WsClass fromClass = null;
    WsClass toClass = null;
    String fromName = (String) v.get(1);
    String toRoleName = null;
    String fromRoleName = null;
    String toName = null;
    boolean valid = false;
    Vector assocEnds = null;
//returns the WsClass for the given class name
fromClass = ToolUtils.returnWsClass(myAST, fromName);
//obtaining the names of the classes in association
names = ToolUtils.getClassNames(assoc);
assocEnds = assoc.getWsAssociationEnds();

//Obtaining the role names and also the toClass names with the given
//fromClass name.
if ((((String) names.get(1)).equals(fromName)) {  
toName = (String) names.get(2);
toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1))
.flow.getWsAssocEndRole());
fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0))
.flow.getWsAssocEndRole());
} else {  
toName = (String) names.get(1);
toRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(0))
.flow.getWsAssocEndRole());
fromRoleName = ToolUtils.toAWLstring(((WsAssocEnd) assocEnds.get(1))
.flow.getWsAssocEndRole());
}

toClass = ToolUtils.returnWsClass(myAST, toName);
invariant = toClass.getWsInvariant();

//Splitting the invariant if there are more than one
if (InvariantSplit.applicable(invariant, null)) {
    InvariantSplit inv = new InvariantSplit(invariant);
    inv.execute(null);
    expressions = inv.getExps();
}

//Checking for the expression with the fromRoleName reference int
//Vector obtained if there is only one expression then the else part is
//executed.
if (expressions != null) {
    newExp = new Vector();
    unwantedExp = new Vector();
    valid = false;
    for (Object exp : expressions) {
        valid = this.hasFromRole(fromRoleName, (WsExpression) exp);
        if (valid) {
            if ((WsExpression)exp instanceof WsImplication)
            {
                temp = ((WsBinaryExpression)exp).getWsBinExpOp2();
                exp = temp;
            }
            newvis = new ChangeVariableVisitor("this", toRoleName);
            ((WsExpression) exp).acceptVisitor(newvis, null);
            newExp.add((WsExpression) exp);
        } else {
            unwantedExp.add((WsExpression) exp);
        }
    }
} else {
    valid = this.hasFromRole(fromRoleName, invariant);
if (valid) {
    if (invariant instanceof WsImplication)
    {
        invariant = ((WsBinaryExpression)invariant).getWsBinExpOp2();
        newvis = new InvariantVisitor2(toRoleName); 
        invariant.acceptVisitor(newvis, null);
    }
    newvis = new ChangeVariableVisitor("this", toRoleName);
    invariant.acceptVisitor(newvis, null);
}
//now moving only those expression with fromRoleName references
if(newExp != null){
    if (newExp.size() >= 1) {
        for (Object exp : newExp) {
            assoc.addInvariant((WsExpression) exp);
        }
        toClass.setWsInvariant(unwantedExp);
    } else {
        assoc.addInvariant(invariant);
        String newInv = null;
        toClass.setWsInvariant(newInv);
    }
    return true;
}

/******************************
* applicable() is same as MoveInvariantTransform3
**********************************/
REFERENCES


