

**Java Implementation of Scope Graph**

FU Wenjie : wenjie.fu@imt-atlantique.net

Tutor:

Antoine BEUGNARD : antoine.beugnard@imt-atlantique.fr

Julien MALLET : julien.mallet@imt-atlantique.fr

徽标, 公司名称

描述已自动生成

[1. Abstract 3](#_Toc79571755)

[2. Scope Graph 3](#_Toc79571756)

[2.1 Introduction of Scope Graph 3](#_Toc79571757)

[2.2 Definition and Constraints 4](#_Toc79571758)

[3. Realization 4](#_Toc79571759)

[3.1 Scope Graph Modeling 4](#_Toc79571760)

[3.2 Building the Scope Graph 6](#_Toc79571761)

[3.3 Print the Scope Graph 7](#_Toc79571762)

[3.4 Name Resolution 7](#_Toc79571763)

[3.5 Self Replication 9](#_Toc79571764)

[4. Validity 9](#_Toc79571765)

[4.1 Create and Resolve Tests 9](#_Toc79571766)

[4.2 Self replication Tests 10](#_Toc79571767)

[5. Conclusion 11](#_Toc79571768)

[6. References 12](#_Toc79571769)

[7. Appendix 12](#_Toc79571770)

# 1. Abstract

Dependency bug is a common problem in software engineering, especially when we’re building software in multiple languages. Dependency bugs can appear because of modular construction of software, use of multiple languages, and independent evolution of components and languages.[[1]](#footnote-1) For every software engineer, dependency issues are annoying. So, we want to find a unified solution to the dependency problem that can be applied to multiple programming languages.

This project implements Scope Graph and its related algorithms through Java, which is a formalism for describing program binding structure and performing name resolution in an AST-independent way.

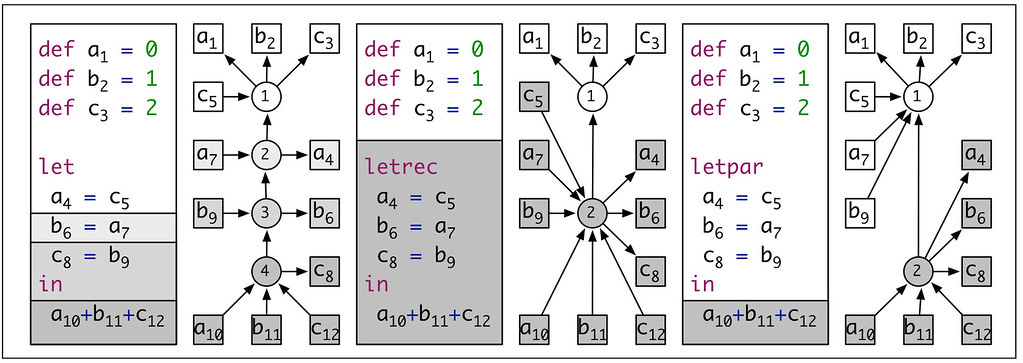
Through this project, you can quickly build a new Scope Graph using the methods provided by the ScopeGraph class. The ScopeGraph class can also automatically generate visual code in Dot language, which can use Graphiz software to draw a Scope Graph

Finally, the ScopeGraph class provides algorithms such as name resolution.

# 2. Scope Graph

Scope graphs provide a new approach to defining the name binding rules of programming languages. In this project, we wanted to use the Scope Graph to achieve a unified solution to the dependency problem

## 2.1 Introduction of Scope Graph



A scope graph represents the name binding facts of a program using the basic concepts of declarations and reference associated with scopes that are connected by edges. Name resolution is defined by searching for paths from references to declarations in a scope graph. Scope graph diagrams provide an illuminating visual notation for explaining the bindings in programs. But scope graphs are more than pretty pictures. The foundational resolution calculus provides the basis for generic, language-independent implementation of a range of tools involving name binding.[[2]](#footnote-2)

## 2.2 Definition and Constraints

In the concept of scope Graph, three types of nodes are defined: **Scope**, **Declaration**, **Reference**

* **Scope**: Scope is denoted by identifiers drawn from an abstract enumerable set. In a scope graph diagram, scopes are represented by circles with numbers representing their identity.
* **Declaration**: a declaration is an occurrence of an identifier that introduces a name.
* **Reference**: a reference is an occurrence of an identifier referring to declaration.

In a Scope Graph, nodes are wired together using multiple constraints.

* **Declaration constraint**: specifies that declaration belongs to scope .
* **Reference constraint**: specifies that reference belongs to scope .
* **Direct edge constraint**: specifies a direct l-labeled edge from scope to scope
* **Association constraints**: specifies as the associated scope of declaration .
* **Nominal edge constraint**: specifies a nominal l-labeled edge from scope to reference.
* **Association constraint**: specifies that a given declarations has a given associated scope.

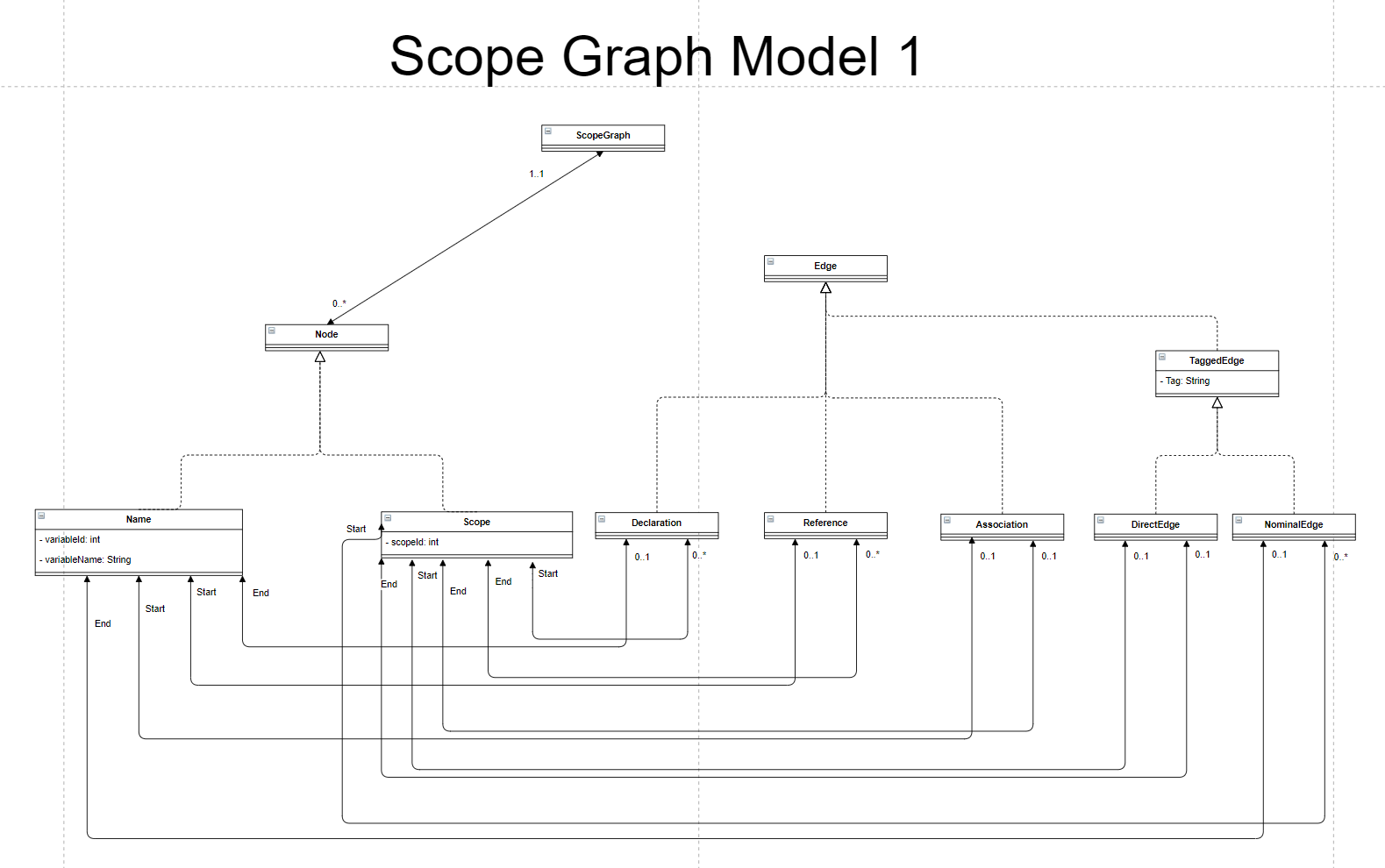
# 3. Realization

This project is developed using the integrated development platform IntelliJ IDEA 2020.2.4 with Java.

## 3.1 Scope Graph Modeling

We abstracted the three definitions from “2.2 Definition and Constraints” into the Node class. The Name and Scope classes inherit from the Node class. Declaration and Reference are represented by the Name class. All constraints in “2.2 Definition and Constraints” are abstracted to the Edge class, and then different types of Edge inherit from the Edge class.

During the construction of this project, two models were tried to build the Scope Graph. Model 1 binds the type of the Edge to the Scope and Name, so that each Scope and Name has a very clear idea of what Edge type is useful to them. Compared to Model1, Model2 is more general, and all the different types of Edge are stored in Node.

Finally, to keep the model scalable, we used Model 2

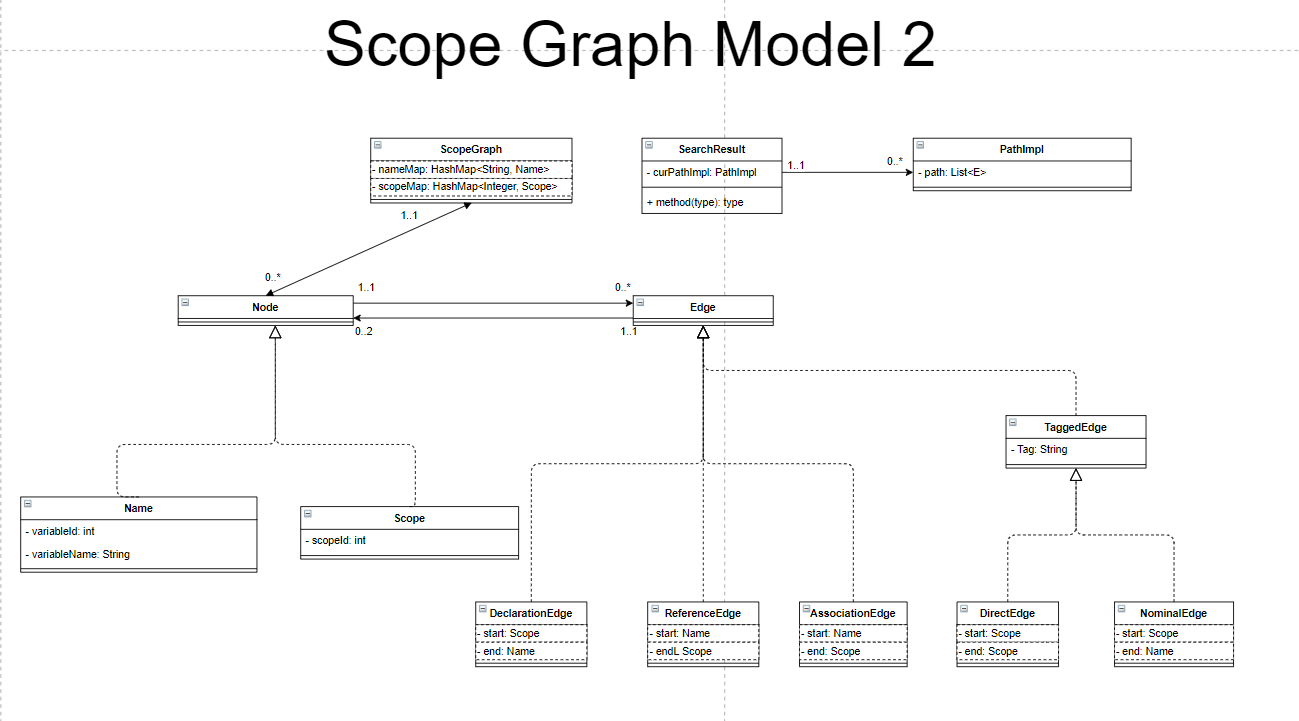
Image 1 Model1

Image 2 Model2

## 3.2 Building the Scope Graph

The ScopeGraph class provides six methods to quickly build a Scope Graph

To build a Scope Graph, first use the generateScope() method to generate all scopes. You can specify the number of scopes directly, and the Scope ID of the Scope is automatically incrementing. Or generate a Scope with the contents of the input array as its serial number

public void generateScope(int amount);

public void generateScope(int[] intArray);

Then concatenates each Scope together using the method used to build the Edge

public Name makeDeclaration(int scopeId, String variableName, int variableId);

public void makeDirectEdge(int childrenScopeId, int parentScopeId);

public Name makeNominalEdge(int scopeId, String variableName, int variableId);

public Name makeAssociation(int declarationScopeId, String variableName, int variableId, int associationScopeId);

public Name makeReference(int scopeId, String variableName, int variableId);

See the ScopeGraphCreater class for details on how to create a Scope. This class creates several Scope Graphs for later testing.

*/\*\*  
 \* The article mentioned below is "A constraint Language for Static Semantic Analysis Based on Scope Graphs"  
 \*/  
  
/\*\*  
 \* The Scope Graph is Figure2 in the article  
 \** ***@return*** *\*/*public static ScopeGraph ScopeGraphFirgure2(){  
 ScopeGraph scopeGraph = new ScopeGraph();  
 scopeGraph.generateScope(1);  
  
 Name referencex4 = scopeGraph.makeReference(1, "x", 4);  
 scopeGraph.makeReference(1, "x", 8);  
  
 scopeGraph.makeDeclaration(1, "x", 1);  
 scopeGraph.makeDeclaration(1, "y", 3);  
  
 return scopeGraph;  
  
}

## 3.3 Print the Scope Graph

We use Graphiz to visualize the Scope Graph. Once we have created a Scope Graph, we can print it out in the DOT language using the method:

public void printDotForScopeGraph();

Put the dot language text printed on the command line into the DOT language compiler to get the corresponding Scope Graph picture. Below is the dot language description of a Scope Graph and the final transformed picture

digraph first{

rankdir=BT

x8 [shape = box]

x1 [shape = box]

y3 [shape = box]

x4 [shape = box]

x4->1

x8->1

1->x1

1->y3

}

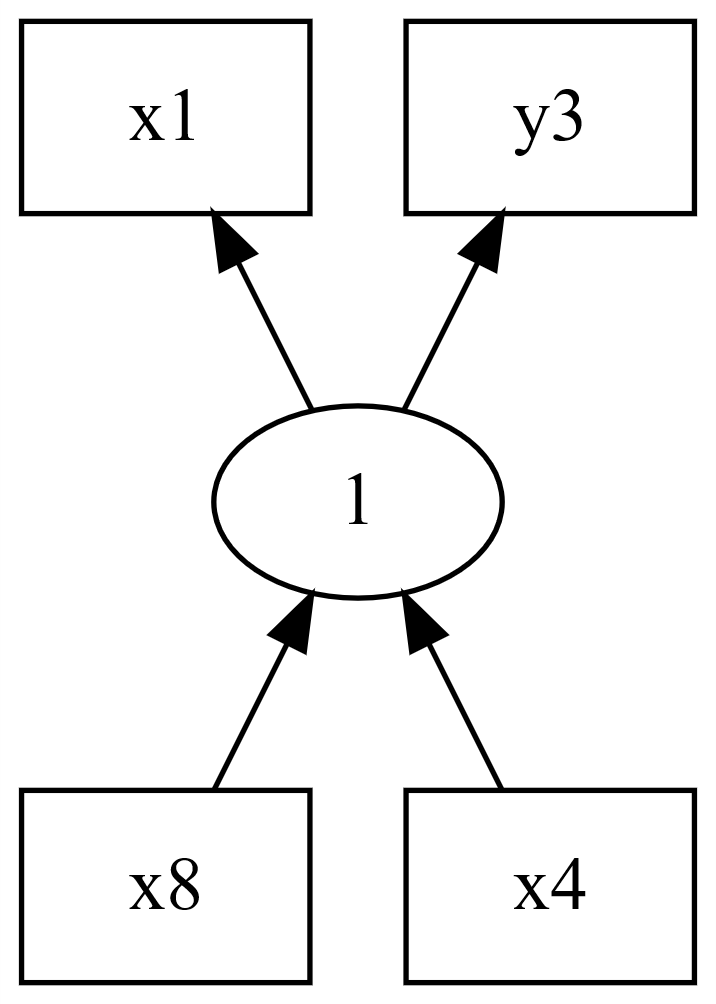


Image 3 Visual Scope Graph

## 3.4 Name Resolution

The ScopeGraph class provides two name resolution methods

public SearchResult checkReference(Name reference);

public SearchResult checkImportModule(Name module);

* The checkReference() method finds the definition corresponding to the input reference
* The checkImportModule() method looks for the definition of the input Module.

The return type of both methods is SearchResult. There is an attribute allPathImpl in the SearchResult class that records all possible search paths. In the test class Test1ScopeGraphCreateAndResolve(), the Test2() method shows a case in which we resolve a reference that has multiple declarations. The Scope Graph in Test6() is shown below.

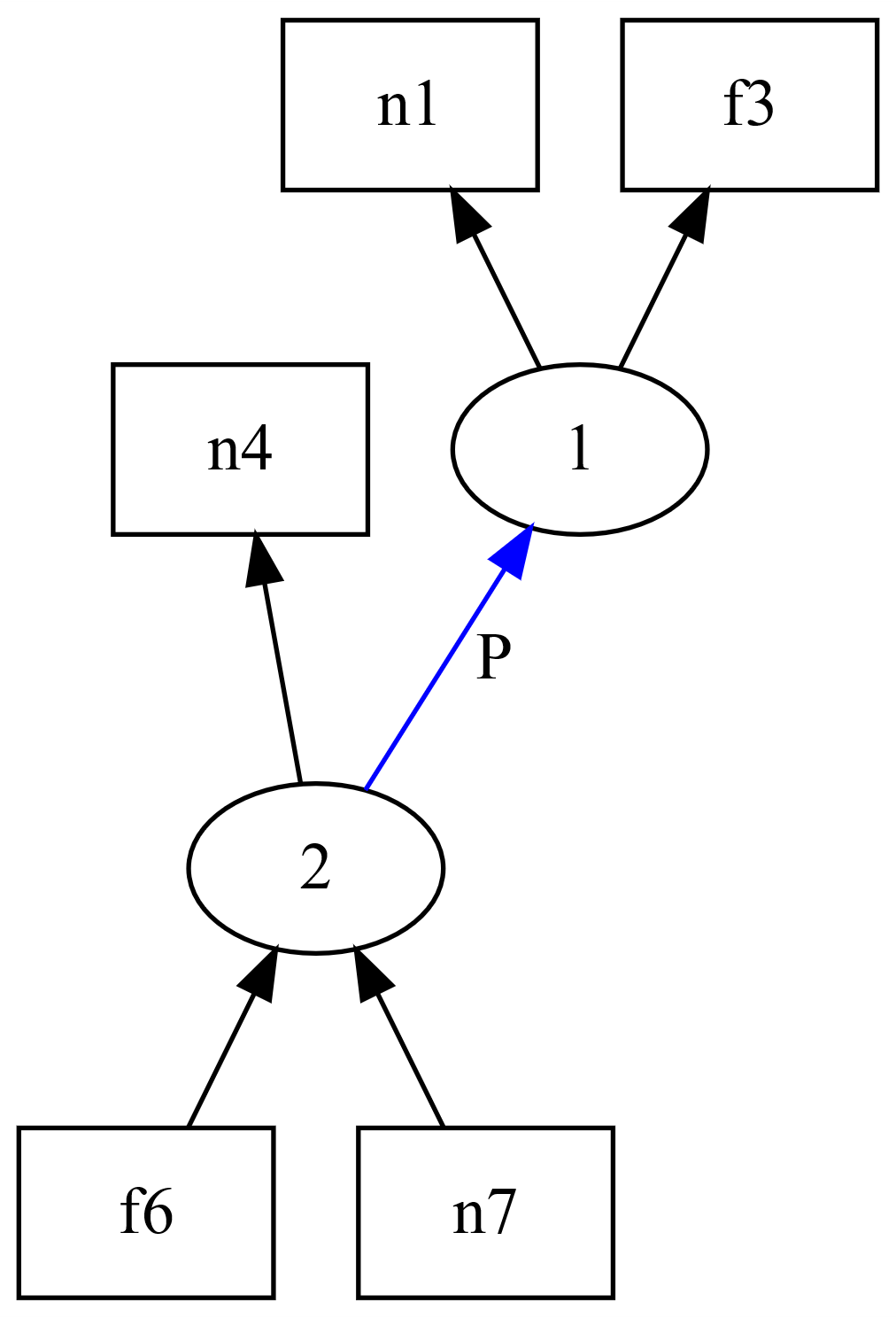


Image 4 Scope Graph in Test6()

As we can see, Reference N7 can correspond to two definitions in total, namely N4 and N1. So if you call checkReference(Name reference) with reference n7 as an argument, we'll find two possible paths.

SearchResult{allPath=[

Path{path=[n7, 2, n4]},

Path{path=[n7, 2, 1, n1]}]}

It should be noted that checkModule(Name Module) is reused in the checkReference(Name Reference) method. Since a reference's definition might be in the module it imports, we should first verify that the module it imports exists by using checkModule(Name Module).

## 3.5 Self Replication

The Scope Graph class provides a selfCopy() method that replicates itself and returns a new Scope Graph that is identical to the original Scope Graph. But they don't share the same data.

public ScopeGraph selfCopy();

The process of copying is to iterate through all the nodes of the old Scope Graph, and then create the nodes of the new Scope Graph as you iterate. To ensure that the new Scope Graph does not share data with the old one, the nodes of the new Scope Graph are created again, and then the corresponding node information of the old Scope Graph is copied to the new Scope Graph.

# 4. Validity

All test examples are in the Test folder. The ScopeGraphCreater contains all Scope graphs that will be used in the next test.

## 4.1 Create and Resolve Tests

First verify that we can implement the Scope Graph creation and the Name Resolutin algorithm correctly.

The test class is Test1ScopeGraphCreateAndResolve. The Scope Graph used in this test is from the paper "A Constraint Language for Static Semantic Analysis Based on Scope Graphs".

In the test class, a total of six examples were created for testing. In each test class, you first create a Scope Graph for the test using the ScopeGraphCreater class. Then we get the Node we need to test, and finally call the Name Resolution methods.

For easier validation of test results, you can call printDotForScopeGraph() to visualize the Scope Graph

*/\*\*  
 \* Test purpose: Find Declaration f3 from Reference f6. Test if the algorithm recurses correctly  
 \*/*@Test  
public void Test1(){  
 ScopeGraph scopeGraph = ScopeGraphCreater.*ScopeGrapFirgure3Left*();  
  
 Name referencef6 = scopeGraph.getName("f6");  
  
 scopeGraph.printDotForScopeGraph();  
  
 System.*out*.println(scopeGraph.checkReference(referencef6));  
}

## 4.2 Self replication Tests

In this test, we verify the Scope Graph's replication capability. The test class is Test2ScopeGraphCopy. In my tests, I copied the Scope Graph that had already been created in the ScopeGraphCreater.

To ensure that the copied Scope Graph does not share data with the original Scope Graph, firstly, the algorithm re-instantiates a new Scope as it traverses all the scopes in the old Scope Graph. For all nodes, we add a constant COPY\_ADD\_NUMBER of 100 to the id of the new Node. Finally, we print out the old and new Scope Graph for comparison.

Let's look at a test example:

@Test  
public void test1(){  
 ScopeGraph scopeGraph = ScopeGraphCreater.*ScopeGraphFirgure2*();  
 scopeGraph.printDotForScopeGraph();  
 ScopeGraph copy = scopeGraph.selfCopy();  
 copy.printDotForScopeGraph();  
  
}

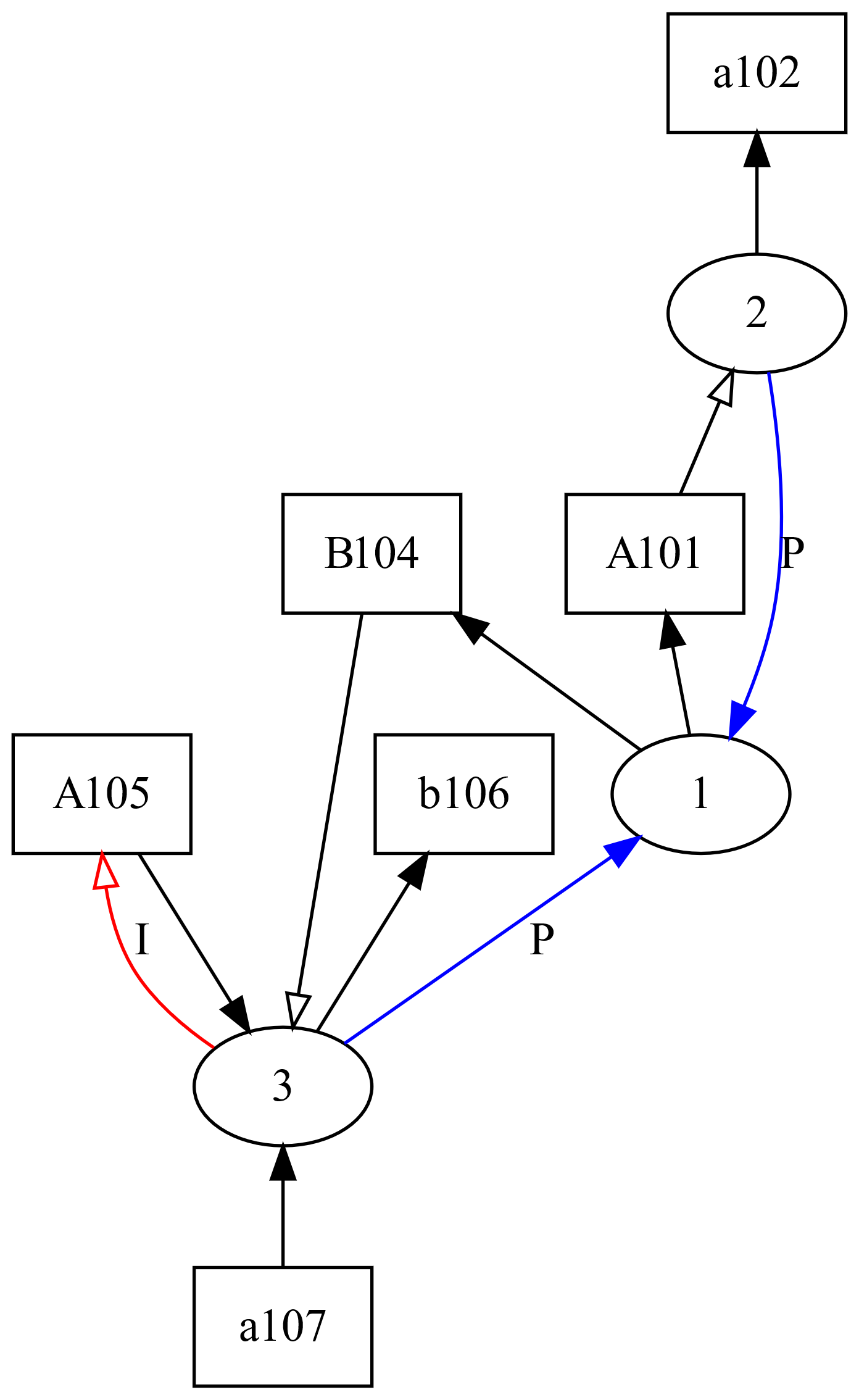
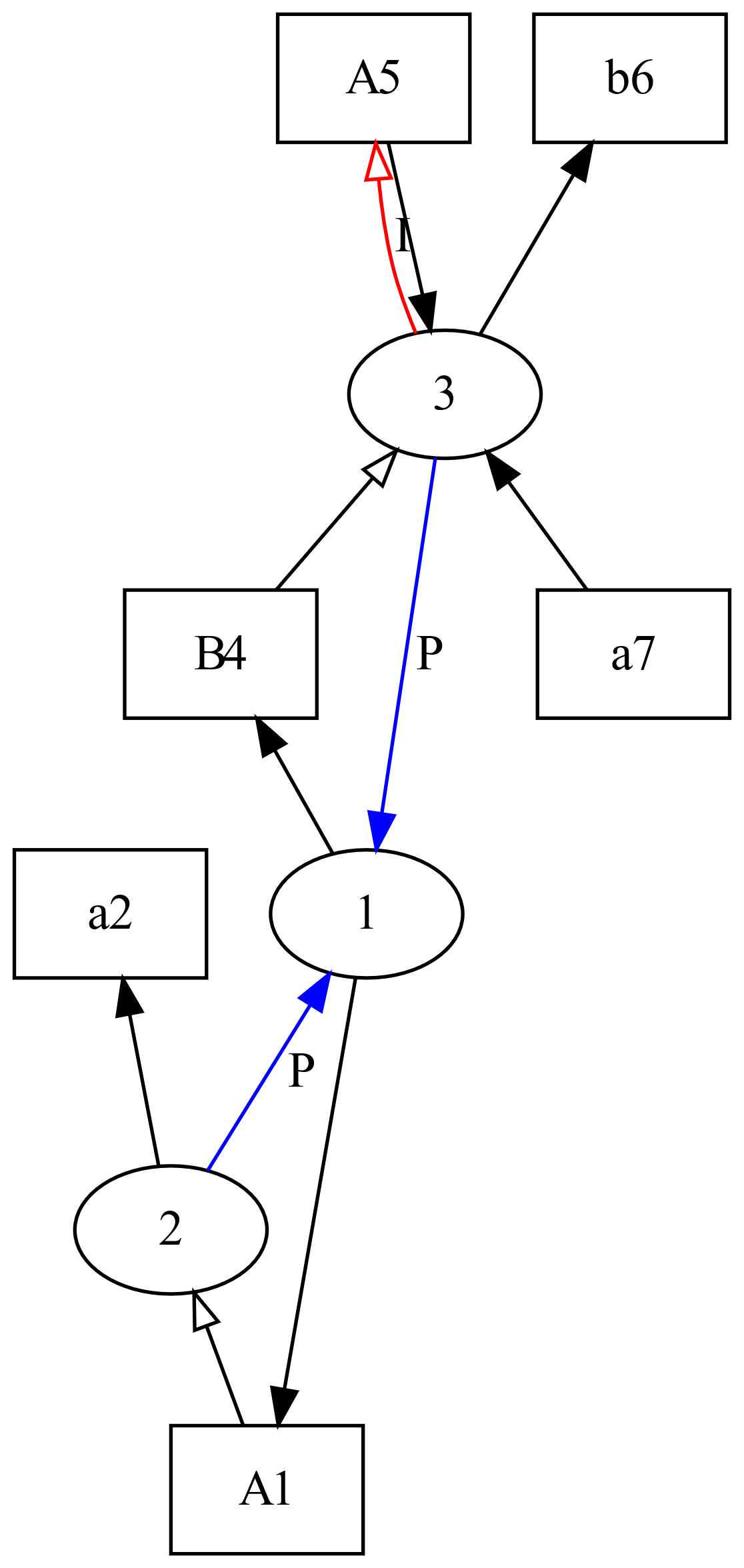


Image 5 The copied Scope Graph

Image 6 The Scope Graph that is copied

# 5. Conclusion

In this project, I implemented the creation of Scope Graph, name resolution, visualization of Scope Graph and self-replication of Scope Graph. On top of the functionality already implemented, we can perform more complex operations on The Scope Graph, such as Scope Graph fusion.

During the implementation of the whole project, I adhered to the object-oriented design principle as much as possible, ensure high cohesion, low coupling design principles. But there are some blocks of code that don't follow this principle very well. For example, in the implementation of Name Resolution, I let the main logic of the algorithm be handled by each Scope, but I did not realize the redistribution of the processing logic in the Scope to each Edge connected with the Scope.

Finally, I would like to thank the two teachers for their patient guidance. Without your help, I could not complete this project!

# 6. References

1. Hendrik van Antwerpen, Pierre Néron, Andrew Tolmach, Eelco Visser, Guido Wachsmuth and Authors Info & Affiliations. A constraint language for static semantic analysis based on scope graphs.

2. Anders Fischer-Nielsen, Zhoulai Fu, Ting Su and Andrzej Wąsowski. The Forgotten Case of the Dependency Bugs. 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)

# 7. Appendix

UML diagram address:

<https://drive.google.com/file/d/17ptOIabCFB--goBjbqUUbOyZTX9YUqBC/view>

Project Github address:

https://github.com/Novmbrain/ScopeGraph/tree/GeneralModel

1. Quote from ‘The forgotten Case of the Dependency Bugs’ [↑](#footnote-ref-1)
2. https://pl.ewi.tudelft.nl/research/projects/scope-graphs/ [↑](#footnote-ref-2)