Degree Project

A Framework for Call Graph Construction

Seyed AmirHossein Mortazavi Jahromi & Elnaz Honar
2010-06-25
Subject: Computer Science
Level: Master
Course code: 4DV01E
Abstract
In object oriented programming, a Call Graph represents the calling relationships between the program’s methods. To be more precise, a Call Graph is a rooted directed graph where each node of the graph represents a method and each edge \((u, v)\) represents a method call from method \(u\) to method \(v\).

Focus of this thesis is on building a framework for Call Graph construction algorithms which can be used in program analysis. Our framework is able to be initialized by different front-ends and constructs various Call Graph algorithms. Here, we instantiate framework with two bytecode readers (ASM and Soot) as front-ends and implement three Call Graph construction algorithms (CHA, RTA and CTA).

At first, we used two above mentioned bytecode readers to read the bytecode of a specific Java program, then we found reachable methods for each invoked method; meanwhile we kept obtained details on our own data structures. Creating data structures for storing required information about Classes, Methods, Fields and Statements, gives us a great opportunity to implement an independent framework for applying well known Call Graph algorithms. As a result of these data structures, Call Graph construction will not depend on bytecode readers; since, whenever we read the bytecode of a program, we accumulate all necessary points in pre-defined data structures and implement our Call Graphs based on this accumulated data.

Finally, the result is a framework for different Call Graph construction algorithms which is the goal of this thesis. We tested and evaluated the algorithms with a variety of programs as the benchmark and compared the bytecode readers besides the three Call Graph algorithms in different aspects.

Keywords: Call Graph, Program Analysis, Intrprocedural Analysis, Framework, Soot, ASM, Class Hierarchy Analysis, Rapid Type Analysis, Class Type Analysis.
Acknowledgements

First, we would like to express sincere gratitude to our supervisors, Dr. Jonas Lundberg and PhD Student Antonina Khairova, for providing this interesting research topic and guiding us through this thesis.

Many thanks go in particular to our parents for supporting us all along our studies. Finally, we would like to appreciate MSI department in Linnaeus University for the opportunity to study there.
Contents

1 Introduction .............................................................................................................................................. 1
   1.1 Motivation ....................................................................................................................................... 1
   1.2 Outline ............................................................................................................................................. 2
2 The Algorithms ......................................................................................................................................... 3
   2.1 Call Graph Definition ....................................................................................................................... 3
   2.2. Formal Model of Call Graph Algorithms ....................................................................................... 3
      2.2.1 Reachability Analysis (RA) ....................................................................................................... 3
      2.2.2 Class Hierarchy Analysis (CHA) ................................................................................................. 4
      2.2.3 Rapid Type Analysis (RTA) ....................................................................................................... 5
      2.2.4 Class Type Analysis (CTA) ....................................................................................................... 6
   2.3 Implementation Issues ....................................................................................................................... 7
      2.3.1 Class Hierarchy Analysis (CHA) ................................................................................................. 7
      2.3.2 Rapid Type Analysis (RTA) ....................................................................................................... 11
      2.3.3 Class Type Analysis (CTA) ....................................................................................................... 15
3 Framework .............................................................................................................................................. 21
   3.1 Framework Entities ............................................................................................................................ 21
   3.2 Front-end Requirements .................................................................................................................... 26
   3.3 Framework experimental Results ...................................................................................................... 26
4 Call Graph Results ................................................................................................................................... 30
   4.1 Metrics .............................................................................................................................................. 30
   4.2 Results .............................................................................................................................................. 30
      4.2.1 Call Graph Nodes ....................................................................................................................... 30
      4.2.2 Call Graph Edges ....................................................................................................................... 31
      4.2.3 Elapsed Time .............................................................................................................................. 33
      4.2.4 Used Memory ............................................................................................................................ 35
5 Experimental Results .............................................................................................................................. 37
   5.1 Summary .......................................................................................................................................... 37
   5.2 Conclusion ........................................................................................................................................ 37
   5.3 Future Works .................................................................................................................................... 38
References .................................................................................................................................................... 39
List of Figures

1.1 Framework overview ...........................................................................................................2
2.1 Relationship between implemented Call Graph algorithms according to accuracy and cost ....3
2.2 The worklist is based on FIFO data structure .................................................................7
2.3 The initialized worklist ....................................................................................................8
2.4 Worklist approach for Call Graph construction .............................................................8
2.5 Source code and class diagram of the given example ......................................................10
2.6 CHA result for the given sample ....................................................................................11
2.7 A Call Graph sample ......................................................................................................12
2.8 The result of removing non-instantiated targets .............................................................13
2.9 The result of removing head nodes from graph ...............................................................13
2.10 CHA to RTA conversion ...............................................................................................14
2.11 RTA to CTA conversion ...............................................................................................19
2.12 Algorithm evolution for call graph construction ............................................................20
3.1 Framework.......................................................................................................................21
3.2 Framework entities relationship ....................................................................................22
3.3 Framework entities class dependencies ..........................................................................23
3.4 Entities construction phase – Classes dependencies .......................................................24
3.5 Entity builders’ hierarchy ...............................................................................................25
4.1 Class Hierarchy of class A...............................................................................................32
4.2 Possible reachable methods of A.n( ) in CHA Call Graph ................................................32
4.3 Possible reachable methods of A.n( ) in CTA Call Graph ................................................32
List of Tables

3.1 Elapsed Time by using Soot and ASM for building entities of some Java programs ............ 27
3.2 Used Memory by using Soot and ASM for building entities of some Java programs ........... 28
4.1 Metrics .................................................................................................................................. 30
4.2 Number of nodes in CHA, RTA and CTA Call Graphs ...................................................... 31
4.3 Number of edges in CHA, RTA and CTA Call Graphs .......................................................... 31
4.4 Number of nodes and edges in CHA, RTA and CTA Call Graphs ........................................ 33
4.5 Comparing Time Ratio for RTA and CTA ............................................................................ 34
4.6 Comparing Used Memory for ASM and Soot in different Call Graph algorithms ............ 35
5.1 Assessment of CHA, RTA and CTA ..................................................................................... 38
List of Charts

3.1 Comparing Elapsed Time by using Soot and ASM for building entities of Java programs .....27
3.2 Comparing Used Memory by using Soot and ASM for building entities of Java programs .....28
4.1 Comparing Elapsed Time for Call Graph construction in different Call Graph algorithms .....34
4.2 Comparing Used Memory by using ASM for different Call Graph algorithms ..................35
4.3 Comparing Used Memory by using Soot for different Call Graph algorithms ..................36
1 Introduction
Call Graph, as discussed in [1], is a basis task to analyze the calling relationships between the program methods. Call graphs can be dynamic or static by considering dynamic or static behavior of the program. A dynamic Call Graph is constructed by recording all the target methods of one run of the program. However a static Call Graph represents every possible run of the program. Concern of this thesis is about constructing a framework for implementing different static Call Graph algorithms.

In this chapter we motivate the thesis problem in section 1.1 and provide an overview of our work in section 1.2.

1.1 Motivation
One of the extremely popular programming paradigms is Object-Oriented Programming. It has some features such as garbage collection, dynamic binding of function calls and polymorphism, which definitely facilitate programming; however these features could decrease program performance.

In Object-Oriented Languages, method calls are used very frequently and regardless of existence of compiler optimization techniques, there is a possibility of presence of polymorphism calls. Therefore, there is a growing demand for developers to make their codes as precise as possible and overcome to these kinds of issues. Program analysis helps developers to get over these problems and moves towards programming languages optimization.

Analysis of the whole program or Interprocedural Analysis is the analysis of the entire program source code. It is one method to enable an optimizing compiler to more precisely model the effects of noninlined calls, thus enabling it to make less pessimistic assumptions about program behavior and reduce the performance impact of noninlined call sites [8]. Interprocedural Analysis typically produces summaries of the effect of callees at each call site and/or summaries of the effect of callers at each procedure entry [2]. It models the effects of not only individual methods, but also influences of the interactions between methods. In general, a Call Graph is a necessary prerequisite for any Interprocedural Analysis, where information is transported from one method to another. It can eliminate polymorphism; also detect classes and methods which are never called, aliases, strangers and singletons. Moreover, Call Graph can be used in Software engineering tools like Eclipse [3,16] to resolve program source code references or as input to some point-to analysis applications for further analyzing. Points-to analysis is a static program analysis that extracts reference information from a given input program, e.g., possible targets of a call and possible objects referenced by a field [4]. Call Graph also clarifies program control flow which increases human comprehension of the program [5]. Besides, it can facilitate program testing [6] by determining the order of procedures and functions calls.

In this thesis we implement a framework to provide the Call Graph of a given Java program. We initialize the framework with ASM or Soot as a bytecode reader and generate CHA, RTA and CTA Call Graphs based on framework entities. The framework overview is depicted in Figure 1.1. This framework by providing a program’s Call Graph can be useful in any Interprocedural Analysis. Call Graph algorithms are different in complexity and accuracy of result. According to the precision level, time efficiency and memory usage, appropriate algorithm can be chosen. However, the question remains to be answered is whether this precision is worth of this extra cost?
The following tasks need to be accomplished as the primary goals of this thesis:

- A front-end for reading the bytecode of a Java program.
- Extracting all program elements such as classes, methods, fields, and statements which already provided by the bytecode reader and converting them to the framework entities.
- Implementing different Call Graph construction algorithms (CHA-RTA-CTA) based on framework entities.
- Generating the Call Graph.
- Evaluating generated results for different Call Graph algorithms.
- Comparing different bytecode readers when they used as the framework front-ends like Soot and ASM in this thesis.

1.2 Outline
This thesis begins with an introductory chapter that provides thesis overview, motivation and goals. In Chapter 2, we clearly define Call Graph definition and formally explain four well known Call Graph construction algorithms. In the same chapter, we explain our implementations for these algorithms. In addition, we illustrate these algorithms with a simple example and try to demonstrate differences between these algorithms. In Chapter 3, we describe framework entities and front–end requirement. We further compare generated results for our framework construction phase by two different bytecode readers, Soot and ASM. In Chapter 4, we show our Call Graph algorithms results for wide variety of test programs. We also use some metrics to evaluate and assess our results. Finally in Chapter 5, we will summarize the thesis and give a short description of future works that could be done on our framework.
2 The Algorithms
In this chapter, we define Call Graph in section 2.1 and discuss different models of Call Graph algorithms and implementation issues in section 2.2 and 2.3.

2.1 Call Graph Definition
A Call Graph is a rooted directed graph $G= (V, E)$ where each node $V$ represents a method $v$ and each edge $E= (v, u)$ represents a method call from method $v$ to method $u$. It basically shows the calling relationships between the program’s methods. Each method (caller) in the Call Graph might have a set of possible callees. This set could have size 0 when there is no invocation inside the method body.

The more complex Call Graph algorithm gives the more precise and accurate result as illustrated in Figure 2.1.

![Figure 2.1: Relationship between implemented Call Graph algorithms according to accuracy and cost](image)

2.2 Formal Model of Call Graph Algorithms
In this section, at first we explain formally four well-known Call Graph algorithms then go into details about these algorithms implementations. We further provide an example to compare different models of call graphs. Call Graph construction algorithms are conservative which means that they overestimate the possible result set for each call site and the result set is not exact. In general, as we move towards a more precise and complex algorithm, the size of possible result set for each call site decreases. In fact, a conservative Call Graph is a superset of the ideal Call Graph [7]. Ideal Call Graph is the union of the dynamic call graphs over all possible executions of the program.

2.2.1 Reachability Analysis (RA)
The first and simplest Call Graph construction algorithm is Reachability Analysis (RA). Even though, this algorithm is not implemented in our thesis because of its imprecise result, we will give an abstract explanation on it.

For constructing Call Graph, RA just considers name of methods and omits factors like method’s signature. By method signature we means the ordered combination of the method's type and name along with the method’s parameters types and return type. Whereas this algorithm is based on the name resolution, it produces the largest set of reachable methods in comparison with other algorithms.

For RA definition, we consider Reachables as a set of reachable methods that we are about to compute, $b.n( )$ notation as a call site occurring in the method body $A.m( )$ and Edges notation as a set of calling relationships among methods.
RA can be defined as follow:

1. Initialize analysis (Before starting the analysis, main method of the program besides other entry points are considered as Reachables.)

2. \( \forall A.m( ) \) in Reachables and for each call site \( b.n( ) \) where \( b.n( ) \) is an expression of type \( B \) inside method body \( A.m( ) \):
   - if any \( C \) declares \( n( ) \) such that \( \text{name}(B.n( ) )=\text{name}(C.n( ) ) \)
     - \( C.n( ) \in \text{Reachables} \)
     - \( (A.m( ), C.n( ) ) \in \text{Edges} \)

3. Repeat stage 2 until it reaches a fixed-point and no more changes occur in Reachables.

Briefly, Reachability Analysis for each call site \( b.n( ) \) occurring inside the method body \( A.m( ) \), finds all possible methods with the same name \( n( ) \) in the program and adds them to the set of reachable methods.

Note that RA is based on a fixed-point approach which means it keeps track of all reachable methods until there will be no previously added method in Reachables set to be processed and no more changes take place in Reachables.

Since this algorithm does not take method’s parameters and return type besides method’s name into account for acquiring reachable methods, it is very conservative and less precise in proportion to other algorithms; however, regardless of RA’s precision, it can be used by software applications for instance, in removing unreachable methods in link-time.

2.2.2 Class Hierarchy Analysis (CHA)

We can improve Reachability Algorithm (RA) more by considering some other criteria of methods into account to achieve more accurate Call Graph. In this way, Call Graph construction will be based on other factors such as class hierarchy and method’s signature besides method’s name.

Class Hierarchy Analysis (CHA) is a well known algorithm for the purpose of Call Graph construction which can lead to more precise result in comparison with Reachability Analysis. CHA implementation requires class hierarchy information; therefore, class hierarchy of the whole program must be constructed before running the CHA algorithm. The result of CHA is a set of reachable methods, but as it was mentioned before, due to the fact that CHA is more precise than RA, size of this set is smaller than RA result and for each call site in the program, set of reachable methods decreases.

For definition of CHA we use Reachables notation as the set of reachable methods that we are about to compute, \( b.n( ) \) notation as a call site occurring in method body \( A.m( ) \), type\( (b) \) as the static type of \( B \), subtypes\( (type(b)) \) as the set of subtypes(subclasses) of \( type(b) \) including \( B \) itself and Edges notation as a set of calling relationships among methods.

Based on the fixed-point approach discussed in Reachability Analysis, CHA can be defined as follow:
1. Initialize analysis

2. \( \forall \) method \( A.m( ) \) in \( \text{Reachables} \) and for each call site \( b.n( ) \) where \( b.n( ) \) is an expression of type \( B \) inside method body \( A.m( ) \):
   \( \forall \) class \( C \in \text{subtypes( type(b) )} \)
   if any \( C \) declares \( n( ) \) such that \( \text{signature}( B.n( ) ) = \text{signature}( C.n( ) ) \)
   \( \Rightarrow \) \( C.n( ) \in \text{Reachables} \)
   \( \Rightarrow \) \( ( A.m( ), C.n( ) ) \in \text{Edges} \)

3. Repeat stage 2 until it reaches a fixed-point and no more changes occur in \( \text{Reachables} \).

In other words, for each call site \( b.n( ) \) occurring in method \( A.m( ) \), if \( \text{type}(b) = B \), then set of reachable methods for \( b.n( ) \) will contain \( B.n( ) \) and all methods with the same signature as \( B.n( ) \) which were declared in \( B \) and subtypes(subclasses) of \( B \).

2.2.3 Rapid Type Analysis (RTA)

We can even make CHA algorithm more efficient to achieve a better result. Rapid Type Analysis (RTA) considers whole program’s class instantiation details besides class hierarchy information. In fact, RTA is a refinement of CHA which uses one set of instantiated classes for the whole program [1] to limit the set of reachable methods for each call site.

For the definition of RTA we use \( \text{Reachables} \) notation as the set of reachable methods that we are about to compute, \( \text{instantiated classes} \) notation as the set of classes that were instantiated in the program, \( b.n( ) \) notation as a call site occurring in method body \( A.m( ) \), \( \text{Type}(b) \) as the static type of \( B \), \( \text{subtypes}(\text{type}(b)) \) as the set of subtypes (subclasses) of \( \text{type}(b) \) including \( B \) itself and \( \text{Edges} \) notation as a set of calling relationships among methods. Based on the fixed-point approach explained in Reachability Analysis, RTA can be defined as follow:

1. Initialize analysis

2. \( \forall A.m( ) \) in \( \text{Reachables} \) and for each constructor call site \( \text{new } C( ) \) inside method body \( A.m( ) \):
   If \( A.m( ) \in \text{Reachables} \) \( \Rightarrow \) \( C \in \text{instantiated classes} \)

3. \( \forall \) method \( A.m( ) \) in \( \text{Reachables} \) and for each call site \( b.n( ) \) inside method body \( A.m( ) \):
   \( \forall \) class \( C \in \text{subtypes}(\text{type}(b)) \)
   if any \( C \) declares \( n( ) \) such that \( \text{signature}( B.n( ) ) = \text{signature}( C.n( ) ) \)
   \( \land \) \( C \in \text{instantiated classes} \)
   \( \Rightarrow \) \( C.n( ) \in \text{Reachables} \)
   \( \Rightarrow \) \( ( A.m( ), C.n( ) ) \in \text{Edges} \)

4. Repeat stage 2 and 3 until it reaches a fixed-point and no more changes occur in \( \text{Reachables} \).

In short, for each call site \( b.n( ) \) occurring in method \( A.m( ) \), if \( \text{type}(b) = B \), then set of reachable methods for \( b.n( ) \) will contain \( B.n( ) \) and all methods with the same signature as \( B.n( ) \) which were declared in \( B \) and subtypes(subclasses) of \( B \). However, before adding the qualified method to the set of reachable methods, we have to be sure that the
declaring class of this method has been instantiated before; this is the only difference between CHA and RTA.

### 2.2.4 Class Type Analysis (CTA)

Most Call Graph construction algorithms use one set or more to approximate their results. For this purpose, each part of a program such as class or method is associated with a set. This set is used to restrict possible reachable methods for each call site. The accuracy of results will depend on the number of sets that algorithm uses. Intuitively, algorithms that use more sets compute more precise Call Graph, but need more time and space [1]. RA and CHA algorithms do not use any set for computing their results. RTA uses one set of instantiated classes for whole program to restrict reachable methods and create more accurate Call Graph. However, to improve the result and increase precision to a higher level than RTA, we can assume more than one set for entire program.

CTA associates each class A with a set named Contain(A) to restrict the set of possible call targets. This set contains object types that can be found inside A. In more details it consists of class A itself, all its concrete supertypes, types which transported inside by calls such as parameter and return types and types which are created inside.

For definition of CTA we consider Reachables notation as the set of reachable methods, Contain(A) as the contained classes of class A, supertypes(A) as the set of all supertypes(superclasses) of A, F as the static type of field f, subtypes(F) as the set of all subtypes subclasses) of F, ParameterTypes(A.m’’( )) for the set of static types of the parameters of method A.m’’( ) and ReturnType (S.n( )) as the set of static return type of method S.n( ).

CTA can be defined as follow:

1. Initialize analysis

2. \( \forall A.m( ) \in \text{Reachables}:
   \begin{align*}
   & A \in \text{Contain}(A) \\
   & \text{supertypes}(A) \in \text{Contain}(A) \\
   & \forall \text{constructor call site } \text{new } B( ) \text{ inside method body } A.m( ):\n   & \quad B \in \text{Contain}(A) \\
   & \forall A.m'( ) \text{ inside class } A \text{ targeting } S.n( ):\n   & \quad \text{subtypes}(\text{ReturnType}(S.n( ))) \in \text{Contain}(A) \\
   & \forall \text{externally called method } A.m''( ) \text{ in class } A:\n   & \quad \text{subtypes}(\text{ParameterTypes}(A.m''( ))) \in \text{Contain}(A) \\
   & \forall \text{field } f \text{ that is declared in class } A:\n   & \quad \text{subtypes}(F) \in \text{Contain}(A)
   \end{align*}

3. \( \forall \text{method } A.m( ) \text{ in } \text{Reachables} \text{ and for each call site } b.n( ) \text{ inside method body } A.m( ):\n   \begin{align*}
   & \forall \text{class } C \in \text{subtypes}(\text{type}(b)) \\
   & \quad \text{if any } C \text{ declares } n( ) \text{ such that } \text{signature}(B.n( ))=\text{signature}(C.n( )) \land C \in \text{Contain}(A)) \\
   & \quad \Rightarrow C.n( ) \in \text{Reachables} \\
   & \quad \Rightarrow (A.m( ), C.n( )) \in \text{Edges}
   \end{align*}

4. Repeat stage 2 and 3 until it reaches a fixed-point and no more changes occur in \text{Reachables}. 
In fact, CTA has two phases:
1. Data flow phase
2. Call Graph construction phase

In Data flow phase, for each class A, a set is constructed and named $\text{Contain}(A)$. It consists of a group of classes, includes the class A itself, all concrete supertypes of A and All classes allocated within A. Furthermore this set can contain some other classes such as classes that reach class A as the parameter types, return types and all types that can be read from fields (data flow information). In second phase, CTA constructs Call Graph according to the data flow phase which results in restriction of reachable methods and generation of more accurate Call Graph.

### 2.3 Implementation Issues
In this section we explain Call Graph algorithms and give a description of our implementation in details.

In all Call Graph construction algorithms, for keeping reachable methods we used a Queue-based data structure and named it worklist (Figure 2.2). It is a FIFO (First-In-First-Out) data structure which the first element added to the worklist will be the first one to be removed; also worklist has a mechanism to deny identical objects, so each element can be added just once.

![Figure 2.2: The worklist is based on FIFO data structure](image)

#### 2.3.1 Class Hierarchy Analysis (CHA)
For CHA algorithm implementation, at first we initialized worklist by adding entry points which means that we added main method (in most cases) besides a set of methods which are called in every single program such as:
- `java.lang.Object.clinit()`
- `java.lang.System.clinit()`
- `java.lang.System.initializeSystemClass()`

By adding these methods to worklist, we simulate behavior of Java Virtual Machine (JVM) run-time setup [9]. Figure 2.3 shows the initialized worklist.
Figure 2.3: The initialized worklist

The next step is worklist processing and Call Graph construction. As it is shown in Figure 2.4, we removed the first element (method) from worklist and assumed it as a Source node of the Call Graph. Then, we went through all of its statements and for each call site that we had found, we obtained reachable methods. Later on, we made an edge from Source node to all reachable methods and added them to the worklist for further processing. We continued this procedural strategy for all worklist members until there were not any previously added elements and worklist stood empty.

Figure 2.4: Worklist approach for Call Graph construction

In other words, we started from the entry point and went through all the statements of method body. Based on CHA definition, for each call site \( b.n() \) in the method block, if \( \text{static type}(b)=B \), then the targets of call site \( b.n() \) will be all concrete (non-abstract) methods having the same signature as method \( B.n() \) and declared by \( B \) or by any concrete subclasses of \( B \) including \( B \) itself. According to this definition, we added reachable methods of each call site to the worklist then made edges from source to reachable methods.
worklist.initialize() ; // initializing worklist
while (worklist != empty) {
    Method source = worklist.removeNext(); // remove next method from worklist
    for each (stmt in source) { // process statements in source method
        if (stmt is a InvokeStmt) { // if statement is a method/constructor call
            Method n = stmt.getInvokedMethod();
            List reachables = resolveDispatch(n);
            for each (Method target in reachables){
                worklist.add (target);
                callgraph.addEdge (source ➔ target);
            }
        }
    }
} // end of while

(A worklist approach for CHA algorithm)

The method resolveDispatch( ) finds reachable methods of each call site b.n( ) and for computing reachable methods, at first we check if static type(b)= B and if B declares by signature method B.n( ), then reachable methods are method n( ) in class B and all methods having signature B.n( ) defined in concrete subclasses of B, else we have to look at the concrete superclasses of B. At this state, the first concrete superclass of B that declares method B.n( ) by signature and all methods having signature B.n( ) defined in concrete subclasses of this superclass, are reachable; resolveDispatch( ) returns a set of possible methods as the result to the CHA algorithm.

List<Method> resolveDispatch ( Method n ) { // resolveDispatch method
    Class B = m.getOwnerClass( ) ;
    Class declaringClass;
    if (B.declareSignature(n) ) {
        declaringClass = B ;
    } else { // n can be declared by supper classes of B
        for( Class SuperClass in hierarchy.getSuperClasses(self) ) {
            if( SuperClass.declareSignature(n) ) {
                declaringClass = SuperClass;
                break ;
            }
        }
    } // end of else
    reachable.add ( declaringClass.n( ) );
    for each ( Class C in hierarchy.getSubClasses(declaringClass) ) {
        if( C.declareSignature(n) ) {
            reachable.add( C.n( ) ) ;
        }
    } // end of if
} // end of resolveDispatch

(resolveDispatch function for finding reachable methods of a method)
Whereas in our implementation, RTA uses CHA Call Graph as input for further processing and also CTA can use either CHA or RTA Call Graph as input, therefore both RTA and CTA algorithms are based on CHA and we use CHA result directly or indirectly as the input data for implementing those algorithms. Furthermore, as all classes are visited during the CHA construction, by adding some additional features to CHA, we prevent other algorithms from repetitive tasks and make a significant performance improvement in both RTA and CTA. The first additional feature we consider for this purpose is a set of instantiated classes for whole program. To obtain this set whenever we visit an instantiated class, we add it to this set. Later on, we use this instantiated set in RTA. Besides, each time we visit a class, we store all its allocated classes in a separate set, so each class can easily know its allocated classes. We take advantage of these allocated classes in CTA algorithm.

To specify and explain the general idea of Call Graph construction algorithms, we give a simple example and demonstrate results for each algorithm. Figure 2.5 illustrates source code and class diagram of the given example and Figure 2.6 shows the CHA result for this example.

```java
public class Main {
    public static void main(String args[]) {
        A x = new B();
        x.m();
    }
}

interface Interface {
    public static String str = new String("String str");
    public void m();
}

class A implements Interface {
    public void m() {
        System.out.println(str);
    }
}

class B extends A {
    public void m() {
        new C();
    }
}

class C extends A {
    public void m() {
    }
}
```

Figure 2.5: Source code and class diagram of the given example
In the above example, according to CHA algorithm, if we assume `Main.main( )` as a source method for call site `x.m( )` occurring in `main( )`, reachable methods are `A.m( )`, `B.m( )` and `C.m( )`. Whereas variable `x` is of static type `A`, therefore all methods `m( )` in subclasses of `A` with the same signature as `A.m( )` and method `m( )` in class `A`, belong to reachable methods of `x.m( )`.

### 2.3.2 Rapid Type Analysis (RTA)

As we already mentioned, RTA is an improved version of CHA and the only difference between RTA and CHA is elimination of methods which their declaring class is not created in the program.

One approach for implementing RTA is removing extra nodes and edges from previously constructed CHA Call Graph which do not belong to the instantiated classes:

```java
callgraph = cha.getCallGraph();
instantiated classes = CHA.getInstantiatedClasses();
do {
    for each (node in callgraph) {
        if ( node.isHeadnode() && node != entry point ) {
            if ( node.isConstructer() ) // update instantiated classes
                instantiated classes.remove(node.getOwnerClass());
            callgraph.remove(node);
        }
    } // end of for
    if ( node.hasReflexiveEdge() && node.inDegree()==1 && node.outDegree()==1 )
        callgraph.remove(node);
    if ( !node.isStaticMethod() && !node.isConstructer() &&
        node.getOwnerClass() \notin instantiated classes &&
        !node.IsCalledBySubClasses() && !node.IsOverriddenBySubClasses() )
        callgraph.remove(node);
} // end of while (callgraph doesn’t change any more)
```

(RTA algorithm)
boolean IsCalledBySubClasses() {
    for each (Class SubClass in hierarchy.getSubClasses(this.getOwnerclass)) {
        if (SubClass ∈ instantiated classes && this.IsCalledby(SubClass))
            return true;
    }
    return false;
}

(IsCalledBySubClasses function)

boolean IsOverriddenBySubClasses() {
    for each (Class SubClass in hierarchy.getSubClasses(this.getOwnerclass)) {
        if (SubClass ∈ instantiated classes && SubClass.declarebySignature(this.getMethod()))
            return true;
    }
    return false;
}

(IsOverriddenBySubClasses function)

According to this technique, by removing a node from Call Graph, we should also remove its descendent nodes. In other words, after removing a node which does not belong to instantiated classes, if we find a head node in Call Graph (head node is a node that doesn’t have any predecessors) then it means that the only predecessor of this head node was removed. Therefore we have to remove it as well.

For better explanation, we give an example in Figure 2.7. Consider node 1 as entry point and node 5 and node 6 do not belong to instantiated classes set.

![Call Graph sample](image)

Figure 2.7: A Call Graph sample

For converting this CHA Call Graph to RTA, we have to remove node 5 and node 6 since their declaring classes are not instantiated.
Figure 2.8: The result of removing non-instantiated targets

According to Figure 2.8, node 7, node 8 and node 9 do not have any other predecessors except node 5 & node 6 and by removing non-instantiated methods they become head nodes. So, we should remove these heads from graph and this new graph can be considered as RTA result for the given CHA Call Graph (Figure 2.9).

Figure 2.9: The result of removing head nodes from graph

To clarify this approach, we will use the computed CHA Call Graph from the first example (Figure 2.6) and convert it to RTA. Since set of instantiated classes contains Class B & Class C, according to the algorithm, we have to remove node A.m( ). Moreover, if we check Call Graph again, we will find that node Interface.<clinit>( ) has a reflexive edge and it’s indegree=1 . Therefore, this node should be deleted as well. Figure 2.10 illustrates a conversion from CHA to RTA.
Figure 2.10: CHA to RTA conversion
2.3.3 Class Type Analysis (CTA)

CTA’s main idea is narrowing down the set of reachable methods of a call site \( b.n() \) inside method \( A.m() \) by keeping track of "available target types" within class \( A \). Since CTA algorithm is refinement of CHA and RTA, we can reuse CHA or RTA Call Graph result in CTA and decrease the set of reachable methods of a call \( b.n() \) to make it more precise.

CTA algorithm implementation has three phases:

a) Class Graph construction phase

To construct CTA Call Graph, for each class \( C \), we have to find \textit{Contained Classes} of class \( C \) (possible call targets) and \textit{Possible Enter Classes} (possible classes that might reach \( C \)). For this purpose, we constructed a Class Graph which consists of all types (classes) of input Call Graph (either CHA or RTA). After that we associated each node (class) of Class Graph with two sets. One is set of \textit{Contained Classes} and the other one is set of \textit{Possible Enter Classes}. The following code shows Class Graph construction algorithm:

```java
CallGraph callgraph = cha.getCallGraph() or rta.getCallGraph();
ClassGraph classgraph = new ClassGraph();

/* Add all CallGraph method types (all classes used in callgraph) to the classgraph */
for each (CallGraphNode A.a() in callgraph) {
    classgraph.addNode (A);
} // end of for

for each (Edge A.a() \( \rightarrow \) B.b() in callgraph) {
    if (B.b() returns ReferenceValue)
        classgraph.addEdge (B \( \rightarrow \) A);
    if (B.b().has (Reference type parameter))
        classgraph.addEdge (A \( \rightarrow \) B);
} // end of for
```

Computing \textit{Contained Classes} set of a class is tricky and results in an increase in cost of CTA algorithm. This set consists of the class \( C \) itself, all concrete superclasses of \( C \) and all allocated classes within class \( C \). However, there are other classes that might reach \( C \) which we named \textit{Possible Enter Classes} of \( C \) and used them to restrict the set of classes that might reach class \( C \). \textit{Possible Enter Classes} set of the \( C \) contains field types in \( C \) and also types that can enter in class \( C \) as parameter or return type.

For CTA construction, at first we have to associate each class graph node with two sets:

- \textit{Contained Classes}
- \textit{Possible Enter Classes}
Computing Contained Classes set:

for each (ClassGraphNode  C in ClassGraph) {
    // Add the class C itself
    C. addToContainedClasses(C ) ;
    // Add all superclasses of C
    C.addToContainedClasses(C.getSuperClasses( ) ) ;
    // Add allocated classes of C
    C.addToContainedClasses((C.getAllocatedClasses( ) ) );
}

Computing Possible Enter Classes set:

CallGraph  callgraph = rta.getCallGraph() ; (or cha.getCallGraph())
for each (Node  C.m ( ) in callgraph) {
    // For all successor nodes of C.m ( ) in callgraph
    for each (Successor of  C.m ( ) ) { // For all successor nodes of C.m ( ) in callgraph
        if (Successor.hasReturnType()){
            C.addToManyEnterClasses(Successor ReturnType());
            C.addToManyEnterClasses(ReachableSubClasses(Successor ReturnType());
        } // end of if
    } // end of for
    for each (Predecessor of  C.m ( ) ){ //For all predecessor nodes of C.m ( )
        if (Predecessor.hasParameterType()){
            C.addToManyEnterClasses(Predecessor ParameterTypes());
            C.addToManyEnterClasses(ReachableSubClasses(Predecessor ParameterTypes());
        } // end of if
    } // end of for
    for each (FieldEntity Field in C ) {
        C.addToManyEnterClasses(Field getType());
        C.addToManyEnterClasses(ReachableSubClasses(Field getType());
    } // end of for
} // end of for

(Computing Possible Enter Classes for each Class Graph node)

In a brief, the Possible Enter Classes of a class C are:
  a) All possible return types of calls C.m( )  B.n( ) exiting C
  b) All possible parameter types of calls A.n( )  C.m( ) targeting C
  c) All types that can be read from fields from within C.

b) Data flow phase
In previously constructed Class Graph, each node corresponds with two different sets. One is Contained Classes and the other one is Possible Enter Classes. In data flow phase, by using Possible Enter Classes set as a filter, we restricted the set of classes that might reach C. In this way, we modified Contained Classes set of a class and made it more precise. In fact, data flow phase keeps track of Contained Classes from one class to another along the Class Graph edges and apply Possible Enter Classes as filter.
worklist.addAll(ClassGraph.allNodes());

while (worklist != empty) {
    Node C = worklist.removeNext(); // get and remove next class from worklist
    for each (Successor S of C) { // go through all Successors of C
        ContainSet(S) = ContainSet(S) \ (EnterClassesSet(S) \ ContainSet(C));
        if (ContainSet(S) changed) {
            worklist.add(S);
        } // end of if
    } // end of for
} // end of while

(Data flow algorithm)

Data flow phase requires mathematical operations such as union and intersection. Thus, choosing a suitable data structure can make a magnificent improvement in algorithm’s efficiency. For this purpose we used our own bitSet implementation which is more efficient in comparison with other set structures.

c) Call Graph construction phase
Once the data flow phase is completed, ContainSet(C) contains all possible types that might be available within class C and according to this set, we can restrict call targets.

CallGraph CG = rta.getCallGraph(); (or cha.getCallGraph())
CallGraph CTA-CallGraph;
for each (Node C.m( ) in CG) {
    for each (Successor B.n( ) of C.m( )) {
        if (ContainSet(C).contains(B))
            CTA-CallGraph.addEdge(C.m( ) \ B.n( ));
    } // end of for
} // end of for

The previous algorithm shows how we restricted reachable methods by using Contained Classes set to have a more precise Call Graph. However, for getting a better result, we repeated the analysis with the newly constructed Call Graph as our input until no further changes occurred. In the following page, we show our suggested approach:
CallGraph $\text{CallGraph} = \text{rta.getCallGraph()}$ ; (or cha.getCallGraph())

\textbf{do} { 
  /* Storing old Call Graph size */
  $old\text{CallGraphSize} = \text{CallGraph.size()}$;

  /* Making Class Graph from Call Graph */
  $\text{ClassGraph} = \text{new ClassGraph()}$;
  $\text{ClassGraph} = \text{constructClassGraph(}\text{ClassGraph})$;

  /* Computing ContainedClasses set for Class Graph */
  compute$\text{ContainedClasses(}\text{ClassGraph})$;

  /* Computing EnterClasses set for Class Graph */
  compute$\text{EnterClasses(}\text{ClassGraph})$;

  /* Running Data Flow and Construct new Call Graph */
  $\text{CallGraph} = \text{runDataFlow(}\text{ClassGraph})$;

  /* Storing new Call Graph size */
  $new\text{CallGraphSize} = \text{CallGraph.size()}$;
}
\textbf{while} ($old\text{CallGraphSize} != new\text{CallGraphSize}$);

(CTA algorithm)

In the following example we apply CTA algorithm on RTA Call Graph from figure 2.10. There are three edges come out from $\text{Main.main( )}$ which are $B.m( )$, $C.m( )$ and $B.<\text{init}>()$.

After following data flow phase, the Contained Classes set for class $\text{Main}$ will contain class $B$ and class $\text{Main}$ itself. So, in Call Graph construction step, there would be only two valid reachable methods $B.m( )$ & $B.<\text{init}>()$ and the edge between $\text{Main.main( )}$ and $C.m( )$ should not exist anymore, because class $C$ is not presented in Contained Classes set of $\text{Main}$ class. Figure 2.11 shows RTA to CTA conversion for our example and finally, Figure 2.12 depicts algorithm evolution for Call Graph construction purpose.
RTA retrieved call graph

Removing the node that does not belong to Contained set of Main class

Figure 2.11: RTA to CTA conversion
Figure 2.12: Algorithm evolution for call graph construction
3 Framework
In the previous chapter, we presented Call Graph construction algorithms and went into details about definition and differences. In this chapter, we will go through our framework implementation.

3.1 Framework Entities
This thesis has two major phases:
- Framework Entities construction
- Call Graph construction and Analysis

As Figure 3.1 shows, the Framework Entities construction phase starts with providing framework front-end necessities to facilitate a switch between Soot and ASM. To extract information from these bytecode readers we implement two classes as framework adapter. These adapters with the help of Entity Builder extract entities (Class, Method, Field and Statement) from these bytecode readers and build a corresponding framework entity for each extracted entity. Framework entities are used in Call Graph construction phase.

![Figure 3.1: Framework](image)

Figures 3.2 and 3.3 illustrate framework entities relationships and class dependencies. Our framework entities are segregated into:

1. **ClassEntity**
   - Based on the designated bytecode reader, we map each Class (SootClass, in the case Soot is chosen as bytecode reader or ClassNode, in the case ASM is chosen) to a ClassEntity.

2. **MethodEntity**
   - We map each Method (SootMethod, in the case of Soot or MethodNode in the case of ASM) to a MethodEntity.

3. **FieldEntity**
   - We map each SootField (Soot) or FieldNode (ASM) to a FieldEntity.

4. **Statement**
   - For each invoked statement and field reference statement in the method body we build an InvokeStatement and a FieldRefStatement entity, respectively. These entities inherit from Statement entity.
Figure 3.2: Framework entities relationship
Our implementation for entities construction phase consists of four classes. The class hierarchy relation of these classes is depicted in Figure 3.5. After providing framework front-end requirements, we invoked abstract method makeEntities(); this method is implemented by class ReachableEntityBuilder.

Method makeEntities() is responsible for:

1. Building ClassEntity for the entry points of the program. In general, construction of a ClassEntity results in construction of all its MethodEntities, FieldEntities, Superclass, Direct subclasses and ClassEntities correspond to all of its interfaces. These ClassEntities are stored in a repository class named Storage. It is a singleton class which has a data structure for storing all the ClassEntities. In addition, it holds class hierarchy information and adds each constructed ClassEntity to the class hierarchy graph. Consequently, each ClassEntity keeps information about its MethodEntities, FieldEntities, Interfaces, Superclass and direct subclasses.

2. Finding all entities that are reachable from the entry points of the program. In fact it is a CHA-based worklist approach which adds all the reachable methods to a data structure named reachableMethods. It begins by reading all the statements of each entry point and for each invoke and field reference statement it verifies if the declaring class of this method or field target has been built before; method findClassEntity() checks for this existence in Storage class, if it has not been built previously, it invokes method makeClassEntityFor() which
leads to construction of ClassEntity for this class. In addition, based on the
selected bytecode reader, class ASMBuilder and SootBuilder are responsible for
information extraction and building framework entities. Figure 3.4 shows class
dependencies of entities construction phase.

Figure 3.4: Entities construction phase – Classes dependencies
Figure 3.5: Entity builders’ hierarchy
3.2 Front-end Requirements
Our framework takes the bytecode of a Java program as input. In fact, this is the bytecode which users have desire to be analyzed by the framework. Either Soot or ASM as a framework front-end reads this bytecode, Framework Entity Builder extracts the same information to build framework entities. However by the use of an appropriate adaptor, it is possible to adapt other bytecode readers to this framework. For instance, it is feasible to analyze bytecode of any object oriented program such as C# or C++.

3.3 Framework Experimental Results
To extract framework entities, we equipped our framework with two different bytecode readers. For a particular Java program as input, our framework generates same results, no matter whether bytecode reader is Soot or ASM. However, there are differences between their consuming time and memory. The reason we provided our framework with two different bytecode readers is to evaluate and assess these bytecode readers for our usage in Call Graph construction.

First bytecode reader we used is Soot; Soot is a Java optimization framework for analyzing and transforming Java bytecode. Soot can be used as a stand alone tool to optimize or inspect class files, as well as a framework to develop optimizations or transformations on Java bytecode [14]. The second one is ASM; it is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or dynamically generate classes, directly in binary form. ASM contains common transformations and analysis algorithms allowing to easily assembling custom complex transformations and code analysis tools [15].

To evaluate these bytecode readers we used a variety of test programs as benchmark. Following, we give a short description about each of them:

ANTLR [17] is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions. EMMA [18] is an open-source toolkit for measuring and reporting Java code coverage. BLOAT [19] is a Java bytecode optimizer and analysis tool. Javac [20] is the component of the Java Developer's Kit used to transform Java source code files into bytecode. Javadoc [21] is a tool for generating API documentation in HTML format from doc comments in source code. JFreeChart [22] is an open-source Java chart library, supporting a wide range of chart types. Jython [23] is an open source implementation of the high-level, dynamic, Object-Oriented Scripting Language Python seamlessly integrated with the Java platform. Lucene [24] is a high-performance, full-featured text search engine library written entirely in Java. PMD [25] is a Java static code analysis tool could be used to write better and optimized Java codes by finding potential problems. Recoder [26] is a Java framework for source code metaprogramming aimed to deliver a sophisticated infrastructure for many kinds of Java analysis. SableCC [27] is an open source compiler generator (or interpreter generator) in Java.

We measured Elapsed Time and Used Memory of these programs as metrics to assess performance of these bytecode readers. Table 3.1 and 3.2 show measured figures by these two bytecode readers for Elapsed Time and Used Memory, respectively.

From Table 3.1 and Chart 3.1, it is obvious that there is a great difference between ASM and Soot in Elapsed Time. In all cases, ASM is much faster than Soot. The reason for this difference certainly caused by the fact that Soot configuration takes more time than ASM. During configuration phase, Soot builds data structures to represent structures like SootClass and SootMethod. Furthermore, for each structure it stores corresponding information. For instance, for each SootClass it keeps information regarding to class hierarchy such as superclasses or subclasses and so on. In other
words, it converts all Java entities to Soot representation structures and stores information about these entities relationships; it also stores information regarding interprocedural analysis such as points-to information and Call Graphs and sets options concerning program analysis such as application classes and main class [12]. However, the configuration phase for ASM is limited to construction of structures like ClassNode & MethodNode and on the contrary it is not as complete and time consuming as Soot. At the end we would like to mention, in the case of building a Call Graph from scratch, using Soot is a good approach since we can use Soot facilities effortlessly.

<table>
<thead>
<tr>
<th>Java program</th>
<th>Number of Entities</th>
<th>ASM Elapsed Time(Seconds)</th>
<th>Soot Elapsed Time(Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTLR</td>
<td>177</td>
<td>0.094</td>
<td>11.969</td>
</tr>
<tr>
<td>BLOAT</td>
<td>368</td>
<td>2.219</td>
<td>29.016</td>
</tr>
<tr>
<td>EMMA</td>
<td>420</td>
<td>1.656</td>
<td>24.828</td>
</tr>
<tr>
<td>Javac</td>
<td>199</td>
<td>1.172</td>
<td>16.345</td>
</tr>
<tr>
<td>Javadoc</td>
<td>296</td>
<td>1.188</td>
<td>23.063</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>492</td>
<td>2.953</td>
<td>42.984</td>
</tr>
<tr>
<td>Jython</td>
<td>411</td>
<td>3.594</td>
<td>43.960</td>
</tr>
<tr>
<td>Lucene</td>
<td>310</td>
<td>1.485</td>
<td>21.676</td>
</tr>
<tr>
<td>PMD</td>
<td>514</td>
<td>3.469</td>
<td>36.411</td>
</tr>
<tr>
<td>RECODER</td>
<td>575</td>
<td>9.220</td>
<td>39.271</td>
</tr>
<tr>
<td>SableCC</td>
<td>345</td>
<td>2.329</td>
<td>11.969</td>
</tr>
<tr>
<td>AVERAGE</td>
<td></td>
<td>2.670</td>
<td>27.408</td>
</tr>
</tbody>
</table>

Table 3.1: Elapsed Time by using Soot and ASM for building entities of some Java programs

![Building Entities - Elapsed Time](chart.png)

Chart 3.1: Comparing Elapsed Time by using Soot and ASM for building entities of Java programs
As it was expected, Table 3.2 and Chart 3.2 show there is a major difference between Used Memory in ASM and Soot. In the same manner as the Elapsed Time, Soot consumes huge amount of memory in comparison with ASM. Similarly, the reason is due to storing great amount of data structures and setting different analysis options.

<table>
<thead>
<tr>
<th>Java program</th>
<th>Number of Entities</th>
<th>ASM Used Memory (MB)</th>
<th>Soot Used Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTLR</td>
<td>177</td>
<td>5</td>
<td>58</td>
</tr>
<tr>
<td>BLOAT</td>
<td>368</td>
<td>13</td>
<td>87</td>
</tr>
<tr>
<td>EMMA</td>
<td>420</td>
<td>9</td>
<td>76</td>
</tr>
<tr>
<td>Javac</td>
<td>199</td>
<td>8</td>
<td>63</td>
</tr>
<tr>
<td>Javadoc</td>
<td>296</td>
<td>9</td>
<td>52</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>492</td>
<td>19</td>
<td>177</td>
</tr>
<tr>
<td>Jython</td>
<td>411</td>
<td>15</td>
<td>118</td>
</tr>
<tr>
<td>Lucene</td>
<td>310</td>
<td>8</td>
<td>75</td>
</tr>
<tr>
<td>PMD</td>
<td>514</td>
<td>13</td>
<td>98</td>
</tr>
<tr>
<td>RECODER</td>
<td>575</td>
<td>17</td>
<td>109</td>
</tr>
<tr>
<td>SableCC</td>
<td>345</td>
<td>8</td>
<td>73</td>
</tr>
<tr>
<td><strong>AVERAGE</strong></td>
<td><strong>11.2</strong></td>
<td><strong>89.6</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Used Memory by using Soot and ASM for building entities of some Java programs

![Building Entities - Used Memory](chart3.2.png)

Chart 3.2: Comparing Used Memory by using Soot and ASM for building entities of Java programs

All in all, even though for a same test program, independent of chosen bytecode reader, our framework yields equal results and equal number of framework entities, The
experimental results from Table 3.1 and 3.2 show that, ASM is quite faster than Soot and has a great efficiency in consuming time and memory in comparison with Soot. Despite great abilities Soot provides as a Java optimization framework for extracting framework entities, we could not use these facilities due to our limitation for generating same results from Soot and ASM. At the end we would like to mention, even though ASM is more efficient to be used as a front-end in our framework, but it is tough to extract information from ASM. For instance, extracting parameter and return types of a method can be done simply by using the Soot’s internal functions but in ASM, it is complicated and required a bunch of coding.
4 Call Graph Results
In this chapter, we will evaluate our Call Graph construction framework with different test programs as benchmark. In section 4.1 we will present our used metrics, further we will show and discuss our generated Call Graph results in section 4.2.

4.1 Metrics
In Table 4.1, we present our framework metrics which were used for evaluating generated Call Graph results. However, we like to note since constructed Call Graphs in this thesis rely on Interprocedural static analysis, a good approach for comparing and evaluating Call Graphs is comparing them to dynamic Call Graph. A dynamic Call Graph is constructed by recording which call sites are executed and all the target functions that are called from each of them [7]. In other words, static Call Graph approximates the dynamic Call Graph, and is defined by the abstract equivalent [10]. Constructing dynamic Call Graph can be integrated to this framework for the future works.

| Number of Nodes | For a given Java program we associate each reachable method with a Call Graph node. We use Number of Nodes metric as the total number of reachable methods in a program. |
| Number of Edges | Each edge of Call Graph corresponds to a call site and shows the calling relationship between methods in the program. We use Number of Edges metric as the total number of call sites in the whole program. |
| Elapsed Time | After we constructed all framework entities, we measure the time interval between the initiation and the completion of Call Graph construction. We use Elapsed Time as metric for this interval. |
| Used Memory | After framework entities construction, with the use of Java garbage collection, we attempt to free occupied memory by objects that are no longer in use. Then we measure consumed memory for each Call Graph algorithm. We refer to this memory consumption metric with Used Memory. |

Table 4.1: Metrics

4.2 Results
We assess our framework with a range of benchmark programs and for Soot and ASM, we compare computed Call Graph results of CHA, RTA and CTA in terms of defined metrics.

4.2.1 Call Graph Nodes
Statistics in Table 4.2 shows the percentage reduction of reachable methods retrieved from RTA and CTA in comparison with CHA. From this figures we find that:

- RTA reduces reachable methods up to average of 4.27% when minimum is 0.05% and maximum is 12.35%
- CTA reduces reachable methods up to average of 7.57% when minimum is 0.87% and maximum is 15.26%
Table 4.2: Number of nodes in CHA, RTA and CTA Call Graphs

<table>
<thead>
<tr>
<th>benchmark</th>
<th>CHA</th>
<th>RTA</th>
<th>CTA</th>
<th>(CHA-RTA)/CHA</th>
<th>(CHA-CTA)/CHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTLR</td>
<td>736</td>
<td>720</td>
<td>706</td>
<td>2.17 %</td>
<td>4.08 %</td>
</tr>
<tr>
<td>BLOAT</td>
<td>2679</td>
<td>2531</td>
<td>2491</td>
<td>5.52 %</td>
<td>7.02 %</td>
</tr>
<tr>
<td>EMMA</td>
<td>1213</td>
<td>1196</td>
<td>1112</td>
<td>1.40 %</td>
<td>8.33 %</td>
</tr>
<tr>
<td>Java</td>
<td>1492</td>
<td>1486</td>
<td>1479</td>
<td>0.40 %</td>
<td>0.87 %</td>
</tr>
<tr>
<td>Javadoc</td>
<td>890</td>
<td>874</td>
<td>816</td>
<td>1.80 %</td>
<td>8.31 %</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>2137</td>
<td>1873</td>
<td>1811</td>
<td>12.35 %</td>
<td>15.26 %</td>
</tr>
<tr>
<td>Jython</td>
<td>2773</td>
<td>2726</td>
<td>2705</td>
<td>1.69 %</td>
<td>2.45 %</td>
</tr>
<tr>
<td>Lucene</td>
<td>1265</td>
<td>1122</td>
<td>1111</td>
<td>11.30 %</td>
<td>12.17 %</td>
</tr>
<tr>
<td>PMD</td>
<td>2322</td>
<td>2123</td>
<td>2073</td>
<td>8.57 %</td>
<td>10.72 %</td>
</tr>
<tr>
<td>RECODER</td>
<td>5037</td>
<td>4949</td>
<td>4910</td>
<td>1.75 %</td>
<td>2.52 %</td>
</tr>
<tr>
<td>SableCC</td>
<td>1897</td>
<td>1896</td>
<td>1678</td>
<td>0.05 %</td>
<td>11.54 %</td>
</tr>
</tbody>
</table>

AVERAGE:  4.27 %  7.57 %

Table 4.2: Number of nodes in CHA, RTA and CTA Call Graphs

4.2.2 Call Graph Edges

Table 4.3, illustrates the percentage reduction of Call Graph edges generated by RTA and CTA in comparison with CHA. According to the figures, we find that:

- RTA reduces edges up to average of 4.73% when minimum is 0.05% and maximum is 26.73%
- CTA reduces edges up to average of 15.96% when minimum is 2.16% and maximum is 60.06%

<table>
<thead>
<tr>
<th>benchmark</th>
<th>CHA</th>
<th>RTA</th>
<th>CTA</th>
<th>(CHA-RTA)/CHA</th>
<th>(CHA-CTA)/CHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTLR</td>
<td>3018</td>
<td>2991</td>
<td>2916</td>
<td>0.89 %</td>
<td>3.38 %</td>
</tr>
<tr>
<td>BLOAT</td>
<td>21380</td>
<td>21215</td>
<td>20736</td>
<td>0.77 %</td>
<td>3.01 %</td>
</tr>
<tr>
<td>EMMA</td>
<td>4665</td>
<td>4621</td>
<td>3420</td>
<td>0.94 %</td>
<td>26.69 %</td>
</tr>
<tr>
<td>Java</td>
<td>7868</td>
<td>7859</td>
<td>7698</td>
<td>0.11 %</td>
<td>2.16 %</td>
</tr>
<tr>
<td>Javadoc</td>
<td>4283</td>
<td>4254</td>
<td>3008</td>
<td>0.68 %</td>
<td>29.77 %</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>16153</td>
<td>11835</td>
<td>6451</td>
<td>26.73 %</td>
<td>60.06 %</td>
</tr>
<tr>
<td>Jython</td>
<td>14416</td>
<td>14301</td>
<td>12959</td>
<td>0.80 %</td>
<td>10.11 %</td>
</tr>
<tr>
<td>Lucene</td>
<td>3661</td>
<td>3258</td>
<td>3162</td>
<td>11.01 %</td>
<td>13.63 %</td>
</tr>
<tr>
<td>PMD</td>
<td>6917</td>
<td>6334</td>
<td>5850</td>
<td>8.43 %</td>
<td>15.43 %</td>
</tr>
<tr>
<td>RECODER</td>
<td>32008</td>
<td>31500</td>
<td>31010</td>
<td>1.59 %</td>
<td>3.12 %</td>
</tr>
<tr>
<td>SableCC</td>
<td>17021</td>
<td>17012</td>
<td>15626</td>
<td>0.05 %</td>
<td>8.20 %</td>
</tr>
</tbody>
</table>

AVERAGE:  4.73 %  15.96 %

Table 4.3: Number of edges in CHA, RTA and CTA Call Graphs
According to Table 4.2 and 4.3, when Call Graph algorithms get more complex, number of reachable methods and estimated callees per call site decrease because more precise algorithms eliminate overestimated nodes and edges from pre-constructed Call Graph and consequently, generate more accurate results. The decrement percentage of Call Graph nodes and edges for different Call Graph algorithms considerably depends on benchmarks. For instance, this rate in Call Graph edges for Javac is 2.16% but for JFreeChart is 60.06%. A good prove for this significant difference is depicted in Figure 4.1; in CHA algorithm, possible reachable methods for call site super.m( ) occurring in method A.n( ) are G.m( ) and all methods in subtypes of G that have the similar signature as method G.m( ) (Figure 4.2).

![Figure 4.1: Class Hierarchy of class A](image1)

Figure 4.1: Class Hierarchy of class A

However, in CHA to CTA conversion, if Contained Classes set of class A solely contains class A itself and supertypes(A), then the possible reachable methods of A.n( ) will be only G.m( ) (Figure 4.3).

![Figure 4.2: Possible reachable methods of A.n( ) in CHA Call Graph](image2)

Figure 4.2: Possible reachable methods of A.n( ) in CHA Call Graph

However, in CHA to CTA conversion, if Contained Classes set of class A solely contains class A itself and supertypes(A), then the possible reachable methods of A.n( ) will be only G.m( ) (Figure 4.3).

![Figure 4.3: Possible reachable methods of A.n( ) in CTA Call Graph](image3)

Figure 4.3: Possible reachable methods of A.n( ) in CTA Call Graph

This scenario happens for method org.jfree.chart.plot.DefaultDrawingSupplier.clone in JFreeChart. There are 69 possible targets occurring in CHA Call Graph for method clone( ), however this figure declines to 1 by applying CTA algorithm. This leads to the conclusion that results vary from one program to another.
Table 4.4 compares all the measured number of nodes and edges in different Call Graph algorithms.

<table>
<thead>
<tr>
<th>Java program</th>
<th>CHA</th>
<th>RTA</th>
<th>CTA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of nodes</td>
<td>Number of edges</td>
<td>Number of nodes</td>
</tr>
<tr>
<td>ANTLR</td>
<td>736</td>
<td>3018</td>
<td>720</td>
</tr>
<tr>
<td>BLOAT</td>
<td>2679</td>
<td>21380</td>
<td>2531</td>
</tr>
<tr>
<td>EMMA</td>
<td>1213</td>
<td>4665</td>
<td>1196</td>
</tr>
<tr>
<td>Javac</td>
<td>1492</td>
<td>7868</td>
<td>1486</td>
</tr>
<tr>
<td>Javadoc</td>
<td>890</td>
<td>4283</td>
<td>874</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>2137</td>
<td>16153</td>
<td>1873</td>
</tr>
<tr>
<td>Jython</td>
<td>2773</td>
<td>14416</td>
<td>2726</td>
</tr>
<tr>
<td>Lucene</td>
<td>1265</td>
<td>3661</td>
<td>1122</td>
</tr>
<tr>
<td>PMD</td>
<td>2322</td>
<td>6917</td>
<td>2123</td>
</tr>
<tr>
<td>RECODER</td>
<td>5037</td>
<td>32008</td>
<td>4949</td>
</tr>
<tr>
<td>SableCC</td>
<td>1897</td>
<td>17021</td>
<td>1896</td>
</tr>
</tbody>
</table>

Table 4.4: Number of nodes and edges in CHA, RTA and CTA Call Graphs

4.2.3 Elapsed Time
The required time for Call Graph Construction with either ASM or Soot is the same because in our implementation, Call Graph algorithms are based on framework entities not bytecode readers.

Table 4.5 and Chart 4.1 compare Elapsed Time for different Call Graph algorithms for either Soot or ASM. The results can be summarized as follow:

- RTA on average is 1.138 times slower than CHA which means that it roughly consumes a little more time than CHA.

- CTA on average is 5.184 times slower than CHA which shows that it slows down the analysis significantly.
<table>
<thead>
<tr>
<th>benchmark</th>
<th>CHA</th>
<th>RTA</th>
<th>CTA</th>
<th>Time Ratio RTA/CHA</th>
<th>Time Ratio CTA/CHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTLR</td>
<td>0.094</td>
<td>0.110</td>
<td>0.219</td>
<td>1.170</td>
<td>2.330</td>
</tr>
<tr>
<td>BLOAT</td>
<td>0.484</td>
<td>0.530</td>
<td>3.909</td>
<td>1.095</td>
<td>8.076</td>
</tr>
<tr>
<td>EMMA</td>
<td>0.187</td>
<td>0.203</td>
<td>0.584</td>
<td>1.086</td>
<td>3.123</td>
</tr>
<tr>
<td>Javac</td>
<td>0.152</td>
<td>0.183</td>
<td>0.755</td>
<td>1.204</td>
<td>4.967</td>
</tr>
<tr>
<td>Javadoc</td>
<td>0.141</td>
<td>0.157</td>
<td>0.485</td>
<td>1.113</td>
<td>3.440</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>0.576</td>
<td>0.670</td>
<td>1.498</td>
<td>1.163</td>
<td>2.601</td>
</tr>
<tr>
<td>Jython</td>
<td>0.498</td>
<td>0.561</td>
<td>2.623</td>
<td>1.127</td>
<td>5.267</td>
</tr>
<tr>
<td>Lucene</td>
<td>0.141</td>
<td>0.172</td>
<td>0.359</td>
<td>1.220</td>
<td>2.546</td>
</tr>
<tr>
<td>PMD</td>
<td>0.463</td>
<td>0.542</td>
<td>1.645</td>
<td>1.171</td>
<td>3.553</td>
</tr>
<tr>
<td>RECODER</td>
<td>1.204</td>
<td>1.313</td>
<td>7.641</td>
<td>1.091</td>
<td>6.346</td>
</tr>
<tr>
<td>SableCC</td>
<td>0.406</td>
<td>0.437</td>
<td>6.000</td>
<td>1.076</td>
<td>14.778</td>
</tr>
<tr>
<td><strong>AVERAGE</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>1.138</strong></td>
<td><strong>5.184</strong></td>
</tr>
</tbody>
</table>

Table 4.5: Comparing Time Ratio for RTA and CTA

![Elapsed Time Chart](chart.png)

Chart 4.1: Comparing Elapsed Time for Call Graph construction in different Call Graph algorithms

34
4.2.4 Used Memory

Table 4.6 shows Used Memory by using ASM and Soot during Call Graph construction phase for each test program. We compare memory efficiency of these two bytecode readers in Chart 4.2 and 4.3.

<table>
<thead>
<tr>
<th>Java program</th>
<th>ASM Used Memory (MB)</th>
<th>Soot Used Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CHA</td>
<td>RTA</td>
</tr>
<tr>
<td>ALNLR</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>BLOAT</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>EMMA</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Javac</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Javadoc</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>Jython</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>Lucene</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>PMD</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>RECORDER</td>
<td>19</td>
<td>22</td>
</tr>
<tr>
<td>SableCC</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>12.09</td>
<td>13.09</td>
</tr>
</tbody>
</table>

Table 4.6: Comparing Used Memory for ASM and Soot in different Call Graph algorithms

Chart 4.2: Comparing Used Memory by using ASM for different Call Graph algorithms
From Charts 4.2 and 4.3 it can obviously be seen that the memory usage for computing Call Graphs is totally different from one bytecode reader to another. The reason is bytecode reader’s data structures which are accumulated in memory during runtime and cannot be removed by garbage collection; like class hierarchy information in Soot that kept in memory after Soot configuration.
5 Experimental Results
In this chapter, we summarize the previous chapters; then we point to the conclusion and give a brief outline of the future works.

5.1 Summary
In this thesis, we presented a framework for constructing Call Graphs. Our framework equipped with two different bytecode readers (ASM and Soot). Also it is well designed to match up with other bytecode readers by providing entities information like Class Entity, Method Entity, Statements and etc; this feature makes the framework to be more extendable. Since this framework constructs different Call Graphs based on its own entities, Call Graph construction phase is absolutely independent from bytecode readers. The Call Graph algorithms that we considered for this framework are CHA, RTA and CTA.

In chapter 2, we defined Call Graph definition and explained different Call Graph algorithms (RA, CHA, RTA and CTA). Next, we discussed the algorithms implementation issues and showed the major differences in algorithms by providing a comprehensive example. In chapter 3, we explained framework data structures and described how Entity Builders convert the bytecode details of a Java program to the framework entities and store them in the framework storage. At the same Chapter we assessed Elapsed Time and Used Memory for building framework entities by applying both ASM and Soot.

In chapter 4, we compared different constructed Call Graph algorithms with range of benchmarks in terms of Call Graph nodes and edges, Elapsed Time and Used Memory.

5.2 Conclusion
Extracting framework entities from Soot and ASM leads to demonstration of ASM is more appropriate in terms of Elapsed Time and Used Memory in comparison with Soot. According to Table 3.1 and 3.2, benchmarks results show that ASM is in average almost 10 times faster and uses almost 8 times less memory than Soot for constructing framework entities. However, we would like to mention that Soot is a Java optimization framework which it is well equipped with a lot of functions [11]; but due to lack of ASM features like class hierarchy information, we could not use Soot properties entirely. Therefore we used Soot features as the same level as ASM’s and we merely used both as simple bytecode readers. However, in spite of getting fast result by using ASM front-end, extracting information for making entities is tricky and time consuming in comparison with Soot.

According to Table 5.1, comparison between CHA and RTA shows that the average percentage reduction in number of reachable methods for RTA results is 4.27% and in number of edges is 4.73% while it is 1.138 times slower than CHA. In addition, CTA leads to percentage reduction up to average of 7.57% for nodes and 15.96% for edges; when it uses more than 5 times of CHA Elapsed Time to generate its results. These figures entirely fulfill our expectation that as a Call Graph algorithm improves, it generates more precise and costly result as well. However for most benchmarks, this improvement is often slightly; for example in Javac, RTA declines reachable methods only 0.4% in compare to CHA while it is 1.204 times slower; However, by using CTA which is almost 5 times more time consuming, number of reachable methods drops to 0.87% . On the other hand, for few cases such as SableCC, Javadoc and Emma the difference between CTA and RTA generated result is significant. In SableCC, RTA eliminates only 0.05% of reachable methods while 11.54% of possible targets removed by CTA. The level of precision not only depends on the algorithm, but also depends on benchmark. Moreover, we like to mention that in some cases for just a few reductions
in number of reachable methods, algorithm consumes huge amount of memory and time. Therefore, based on end-user preferences, the appropriate algorithm can be chosen.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>Time Ratio RTA/CHA</th>
<th>Reduction Percentage in Nodes (CHA-RTA)/CHA</th>
<th>Reduction Percentage in Edges (CHA-RTA)/CHA</th>
<th>Time Ratio CTA/CHA</th>
<th>Reduction Percentage in Nodes (CHA-CTA)/CHA</th>
<th>Reduction Percentage in Edges (CHA-CTA)/CHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTLR</td>
<td>1.170</td>
<td>2.17 %</td>
<td>0.89 %</td>
<td>2.330</td>
<td>4.08 %</td>
<td>3.38 %</td>
</tr>
<tr>
<td>BLOAT</td>
<td>1.095</td>
<td>5.52 %</td>
<td>0.77 %</td>
<td>8.076</td>
<td>7.02 %</td>
<td>3.01 %</td>
</tr>
<tr>
<td>JQUERY</td>
<td>1.086</td>
<td>1.40 %</td>
<td>0.94 %</td>
<td>3.123</td>
<td>8.33 %</td>
<td>26.69 %</td>
</tr>
<tr>
<td>Javac</td>
<td>1.204</td>
<td>0.40 %</td>
<td>0.11 %</td>
<td>4.967</td>
<td>0.87 %</td>
<td>2.16 %</td>
</tr>
<tr>
<td>Javadoc</td>
<td>1.113</td>
<td>1.80 %</td>
<td>0.68 %</td>
<td>3.440</td>
<td>8.31 %</td>
<td>29.77 %</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>1.163</td>
<td>12.35 %</td>
<td>26.73 %</td>
<td>2.601</td>
<td>15.26 %</td>
<td>60.06 %</td>
</tr>
<tr>
<td>Jython</td>
<td>1.127</td>
<td>1.69 %</td>
<td>0.80 %</td>
<td>5.267</td>
<td>2.45 %</td>
<td>10.11 %</td>
</tr>
<tr>
<td>Lucene</td>
<td>1.220</td>
<td>11.30 %</td>
<td>11.01 %</td>
<td>2.546</td>
<td>12.17 %</td>
<td>13.63 %</td>
</tr>
<tr>
<td>PMD</td>
<td>1.171</td>
<td>8.57 %</td>
<td>8.43 %</td>
<td>3.553</td>
<td>10.72 %</td>
<td>15.43 %</td>
</tr>
<tr>
<td>RECODER</td>
<td>1.091</td>
<td>1.75 %</td>
<td>1.59 %</td>
<td>6.346</td>
<td>2.52 %</td>
<td>3.12 %</td>
</tr>
<tr>
<td>SableCC</td>
<td>1.076</td>
<td>0.05 %</td>
<td>0.05 %</td>
<td>14.778</td>
<td>11.54 %</td>
<td>8.20 %</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>1.138</td>
<td>4.27 %</td>
<td>4.73 %</td>
<td>5.184</td>
<td>7.57 %</td>
<td>15.96 %</td>
</tr>
</tbody>
</table>

Table 5.1: Assessment of CHA, RTA and CTA

5.3 Future Works
Our framework can be extended to support more complex and precise Call Graph construction algorithms like XTA and k-CFA [1]. Furthermore, by defining some metrics for each Call Graph node (each reachable method) such as line numbers, number of variables, number of successors, and adding them to a pre-constructed Call Graph, we have a new graph that each of its nodes corresponds to different kinds of metrics. This new graph can be used as a suitable tool for software quality assessment. Also visualizing this graph and associating each node to different type of geometric shapes based on its metrics values, can increase human understandability of the specified software project. Another feature can be attached to our framework in the future is architecture recovery [13]. It is a reverse engineering technique for indentifying components in Object-Oriented Systems and increasing system understanding.
References


[16] Usman Ismail and David R. Cheriton. Incremental Call Graph Construction for the Eclipse IDE. School of Computer Science University of Waterloo, 2009.


