```
class Triangle extends Shape {
   void draw() {printf("triangle");}
}

class Rectangle extends Shape {
   void draw() {printf("rectangle");}
}

class Square extends Rectangle {}
```

3.1 Naive

A simple and inaccurate solution to this problem is to assume that the actual and implementing types are the same as the declared type. In the terms of this example, to assume that because s is declared to be a Shape, that s.draw() always resolves to Shape.draw(). This is the result recorded in the bytecode by every Java compiler, and the one used in the static analysis of regular function calls in procedural languages.

The benefits of this solution are that it requires no extra analysis, is sufficient for the purposes of a non-optimizing compiler, and is very simple. However, its accuracy leaves something to be desired. In the given example Shape.draw() is abstract, and so the s.draw() invocation could not actually branch there: there is no code to branch to. This is a somewhat less than desirable solution for re-engineering tasks that require a reasonably accurate call graph, such as automatic clustering.

3.2 Class Hierarchy Analysis

Class Hierarchy Analysis (CHA) [4, 5] is a *whole program analysis* that determines the actual and implementing types for each method invocation based on the type structure of the program. The whole program is not always available for analysis, due to features such as reflection and remote method invocation. However, for many practical reverse engineering tasks it is sufficient to analyze the code that is available for analysis (this may not be conservative enough for the purposes of compiler optimization).

In the above example, Class Hierarchy Analysis would construct *three* invocation arcs from s.draw(), to Circle.draw(), Triangle.draw(), and Rectangle.draw(). CHA would not produce an invocation arc to Shape.draw(), as it is abstract. This result is a significant improvement over the naive approach, which produced only one arc that could not possibly be traversed during execution.

Class Hierarchy Analysis is flow and context insensitive, and consequently is efficient in both time and space.

3.3 Rapid Type Analysis

Rapid Type Analysis (RTA) [1, 2] uses extra information from the program to eliminate spurious invocation arcs from the graph produced by CHA. This extra information is the set of instantiated (used) types: clearly Triangle.draw() can never be invoked if Triangle is never used in the program. This analysis is particularly effective when a program is only using a small portion of a large library, which is often the case in Java.

RTA begins at all program entry points and traverses over the program, building the call graph and the set of instantiated types as it goes. Consider the following main() as an entry point for the example program:

```
static void main(String[] args) {
   foo( new Square() );
}
```

Now it can be seen that the only sub-type of Shape instantiated in the program is Square, and so this must be the *actual* type of s in foo(). Note, however, that the *implementing* type is Rectangle: that is, Square 'inherits' the implementation of draw() from Rectangle.

Like CHA, RTA is flow and context insensitive, and consequently is efficient in both space and time. Again like CHA, RTA also requires the whole program for analysis. However, RTA is more sensitive to the use of reflection: the analyst must inform the algorithm if reflection is used to instantiate any class, otherwise the algorithm may incorrectly eliminate some arcs from the call graph. CHA is not as sensitive to the use of reflection, as long as the whole program is available for analysis.

In summary, for this example, the naive approach produces a single impossible arc, CHA produces three possible arcs, and RTA narrows these three down to a single target. Most studies have shown that RTA is a significant improvement over CHA, often resolving more than 80% of the polymorphic invocations to a single target [2, 16, 17]. Furthermore, RTA is an extremely fast analysis: in our experience it can usually be computed in a matter of seconds, even for very large programs [20]. RTA does require the results of CHA, which can usually be computed in a minute or two. Both of these analyses combined take less than 10% of the time required to parse the program.[7]

RTA is implemented in the Jax [24] and Toad [16] tools from IBM Research, both available on the alphaWorks website, as well as the front end of the IBM VisualAge C++ compiler [10]. For this study we used a research version of the jport tool, which was originally developed as a part of IBM VisualAge for Java, Enterprise Toolkit 390. We have

[7] Our implementation is written in Java and uses bytecode as input. It can typically parse about 1mb of bytecode per minute on a relatively modest personal computer (with a good JIT, of course).