The Effect of Call Graph Construction Algorithms for Object-Oriented Programs on Automatic Clustering

Derek Rayside[†], Steve Reuss[‡], Erik Hedges[†], and Kostas Kontogiannis[†]
[†]Electrical & Computer Engineering and [‡]Mechanical Engineering
University of Waterloo
Waterloo, Canada
{drayside, ehedges, kostas}@swen.uwaterloo.ca
sreuss@sunwise.uwaterloo.ca

Abstract

Call graphs are commonly used as input for automatic clustering algorithms, the goal of which is to extract the high level structure of the program under study. Determining the call graph for a procedural program is fairly simple. However, this is not the case for programs written in object-oriented languages, due to polymorphism. A number of algorithms for the static construction of an object-oriented program's call graph have been developed in the compiler optimization literature in recent years. In this study we investigate the effect of three such algorithms on the automatic clustering of the Java Expert System Shell (JESS).

Object-oriented programs have an inherently richer structure than those written in procedural languages, and so even medium sized programs such as JESS produce large graphs. Existing tools that we are aware of are not able to process such graphs. Consequently, we have developed our own algorithm for automatic clustering that is scalable to large graphs. This algorithm also supports user specified constraints through the use of 'weighted' arcs.

1 Introduction

The majority 'legacy' applications have been written in procedural programming languages, and consequently these languages have received the most attention from the reverse engineering and program understanding communities. However, the increasing popularity of object-oriented programming languages such as Java means that we will soon be facing a new 'generation' of 'legacy' code in these languages. Therefore, we have turned our attention to the analysis of applications written in object-oriented languages. Specifically, in this paper, we investigate the application of a new automatic clustering algorithm to the well known Java expert system shell JESS.

Automatic clustering algorithms have a rich history in the artificial intelligence literature, and in recent years have been applied to understanding programs written in procedural languages (e.g. [15]). The purpose of an automatic clustering algorithm in artificial intelligence is to group together *similar* entities. Automatic clustering algorithms are used within the context of program understanding to discover the structure (architecture) of the program under study.

We note here that the term *similar* is inappropriate and misleading in the context of software clustering, as we wish to cluster software entities based on their *unity of purpose* rather than their *unity of form*.[1] To illustrate: it is not meaningful to cluster all four letter variables together, even though they are similar. We mention this as the term 'similar' persists in some of the software clustering literature.

In software, we attempt to infer the unity of purpose of entities based on their *relationships*, commonly represented in such abstractions as data dependency graphs and call graphs. These abstractions can be generated for programs written in procedural languages by simply parsing the source code. However, this is not the case for programs written in object-oriented languages, due to *polymorphism*. The three most common algorithms for the static construction of an object-oriented programs call graph are, in order of increasing accuracy: *naive* , *Class Hierarchy Analysis* (CHA), and *Rapid Type Analysis* (RTA).

Our hypothesis is that the accuracy of the call graph construction algorithm used will affect the accuracy of the automatic clustering algorithm. In this preliminary investi-

[1]This, incidentally, is what distinguishes Aristotle's work from Plato's. Furthermore, we posit that what makes clustering 'library' code difficult is that while it has *common* purpose it has no *unity* of purpose. We also note that this whole investigation rather begs the question of whether purpose can be inferred from an algorithmic examination of form. However, we assume that some headway can be made in this direction, as we are working within the limitations of computer science.

gation we have only been able to verify that the choice of call graph construction algorithm does indeed affect the automatic clustering process. However, we are not yet able to state conclusively that the more accurate call graph construction algorithms will produce a more accurate clustering, as we have not yet determined how to objectively assess the accuracy of a clustering.[2] It seems that most of the existing literature relies on subjective assessments of clustering algorithms (and we suspect this indicates a fundamental difficulty).

Contributions The major contributions of this paper are:

- To the best of our knowledge, this is the first application of call graph construction algorithms to program understanding. These algorithms are usually developed for compiler optimizations, although mention is made of their potential for program understanding.
- A new, scalable, user-guided, clustering algorithm. Programs written in object-oriented languages produce much larger graphs than those written in procedural languages, and so a scalable algorithm is required. Also, our algorithm allows the analyst to force entities to be clustered together or apart.
- A discussion of the meaning of clustering within the context of object-oriented programming. Object-oriented languages are different than procedural languages, and so clustering means something slightly different in this context.

The minor contributions of this paper are discussions of the following: the importance of reflection in understanding object-oriented programs; the structure of programs written with file based compilers vs those written in integrated development environments (IDE); and the use of XML technology for managing program understanding data.

Case Study Finally, the case study used for this investigation is the well known Java expert system shell JESS, and so we contribute to the body of knowledge on this program. JESS is an interesting program to study because it has almost identical functionality to the well studied CLIPS expert system shell, which was written in C. Internally, both programs are based on the Rete pattern matching algorithm.

Outline of the text First we give a discussion of the purpose of clustering an object-oriented program (§2). Then we explain the three call graph construction algorithms used in this study (§3). Section §4 reviews the domain model and tools we used. Section §5 presents our clustering algorithm. Experimental results are reported in section §6. Section §7 concludes and gives directions for future work.

[2] *Objective* is not a synonym for *algorithmic*, *automated*, or *quantified*.

2 Understanding Clustering for Java

The purpose of clustering software is to discover the *implicit* order of the program under study, as opposed to the *explicit* order, which is expressed, as such, by the programmer. The analyst does not need tools developed with reference to the artificial intelligence literature to discover the explicit organization of a program. Before we can consider the implicit order of a program, we must first know what its constituent entities are and how they may be explicitly ordered.

2.1 Entities

Program entities may be identified according to one of two criteria: either by *programming language constructs*, or by *program representation*. It is often, but not always, the case that these two things coincide.

The programming language constructs that identify entities in Java are, in order of increasing size: *field*, *method*, *class*, and *package*.[3] We will use the term class to mean any of *regular class*, *inner class*, or *interface*; we will use these particular terms when appropriate.[4] A package corresponds, roughly, with a directory in the source tree. However, since there are scoping rules associated with packages, they should be considered entities according to language constructs and not according to program representation.

The only entity in Java that can be identified on the basis of program representation is *file*. Some languages associate scoping rules with files, and in those languages a file may be considered as an entity on the basis of the programming language. However, in Java, as in Smalltalk, files are simply an artifact of the program representation. For example, the IBM VisualAge for Java and IBM VisualAge for Smalltalk integrated development environments are repository based and *do not use files* to represent programs.

While the algorithm used in an automatic clustering methodology for software does not need to be sensitive to the programming language under consideration, the identification of entities to be clustered does. Note that there are no entities named *module* or *sub-system* in Java.

2.2 Explicit Order

The entities that are expressible in a language can be ordered according to one of three bases: by *declaration containment*, by *scoping rules*, or by *program representation*. It is often, but not always, the case that two or more of these coincide. Considering each basis in turn:

[3] Entities at the sub-method level, such as local variables, are too fine grained to be germane to this work.

[4] The Java notion of an interface is a special kind of abstract class that has different inheritance rules associated with it than other classes do.

Declaration Containment:

- packages contain regular classes and interfaces
- classes contain fields and methods
- classes and methods contain inner classes

Scoping Rules:

- within a class
- within a package
- within an inheritance (`extends`) hierarchy
- within a method (e.g. inner classes)

Program Representation (HFS):

- directories correspond to packages
- directories contain files
- a file contains one or more classes

Note again that these last groupings are only possible if the program is represented in a hierarchical file system (HFS), which is not necessarily the case.

2.3 Looking for Implicit Order

Now that we have identified the entities and their potential explicit orderings, we can begin to consider the potential implicit orderings. The implicit order that we are looking for depends on what we take to be the basic unit of clustering: methods (and fields), classes, files, or packages. The clustering that we have done on JESS has taken classes as the basic unit, but here we consider the meaning of all of the possibilities, in order of increasing size.

Fields and Methods We can see two reasons why it might be worthwhile to take fields and methods as the basic units for clustering. The first is to migrate procedural programs to object-oriented languages (one of the authors has investigated this previously [13]). We speculate that a second reason may be to migrate object-oriented programs to either aspect-oriented (AOP) or subject-oriented (SOP) paradigms. The reason is that both AOP and SOP are concerned with features that 'cross-cut' the class hierarchy.

Classes It is clear from the discussion above that JESS could benefit from clustering classes within a package, and this is what we have attempted in this study. We refer to this as intra-package class clustering. It may also be useful to cluster classes without regard for the package structure, just as the inheritance structure is orthogonal to the package structure. File inclusion may (or may not) be taken into account when clustering classes.

Packages For larger (greater than 10 KLOC) programs consisting of many packages, it would be useful to take packages as the basic unit of clustering, at least for an initial high level view of the program.

Summary The meaning of a clustering is dependent on what entities are taken as the basic units, and if those entities are considered in-line with or orthogonal to the explicit order of the program. Most medium and large size Java programs would benefit from clustering with classes as the basic unit. Large size Java programs could benefit from clustering with packages as the basic unit. We speculate that clustering with fields and methods as the basic units may be useful for migrating object-oriented programs to either the aspect-oriented or subject-oriented paradigms. In this study we have used classes as the basic entity for clustering.

3 Call Graph Construction Algorithms

Devising cost-efficient algorithms for constructing an object-oriented program's call graph from a static analysis of the source code has been an active area of research for the last few years. This research is usually carried out in the context of compiler optimization, as many conventional optimizations such as in-lining cannot be performed without a call graph. A good discussion of the problem is given by Grove et al in [7]. In this section we will explain three of the most common approaches to solving this problem by constructing the call graph for `foo()`:[5]

```
static void foo(Shape s) {
   s.draw();
}
```

The target of the invocation `s.draw()` depends on the actual type of the object that is bound to the formal parameter `s` each time `foo()` is executed. The *declared* type is `Shape`, but the *actual* type may be any sub-type of `Shape`. Furthermore, the actual type may 'inherit'[6] the implementation from the *implementing* type, which may be any super-type of the actual type (including super-types of the declared type). Suppose `foo()` is written with reference to the following code:

```
abstract class Shape {
   abstract void draw();
}
class Circle extends Shape {
   void draw() {printf("circle");}
}
```

[5]All code examples are written in a Java-like syntax.

[6]The term 'inheritance' as it is used in computer science is metaphorical, and hence equivocal [18]. Likewise for the terms 'parent' and 'child'. We make an effort to avoid such terms, and anthropomorphism in general.

```
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).

used this research tool in previous studies on impact analysis [19] and library subset extraction [20].

Rapid Type Analysis is currently considered to be the best practical algorithm for call graph construction in object-oriented languages because it produces good results very inexpensively. There are a number of research groups looking for algorithms that produce better results than RTA for similar cost (e.g. [23]).

4 Domain Model and Tools

We use the domain model for Java that is summarized in [19, 20], and presented fully in [17]. This domain model needs to be transformed somewhat for use in clustering, which is described in [8]. In brief, we represent programs with a typed (nodes and arcs), directed, multi-graph.

4.1 Arc Identity

It is interesting and insightful to compare how various tools consider arc identity. This is an indication of how accurate the tool is at representing the source code.

Obviously the goal of automatic clustering is to represent the program at a higher level of abstraction, and this necessarily requires removing some detail. However, we take the position that as much detail should be kept as long as possible during computation, and then removed as a final step before presentation to the user. By analogy to numerical computation, rounding is not performed at every step along the way, but only at the very end; otherwise, the result may be significantly in error.

Besides the source node and target node of the arc, we note three other characteristics that may be used to identify it: direction, type, and source code line number. An 'x' in Table 1 below indicates that the particular tool uses the particular characteristic for identifying arcs.

Tool	Direction	Type	Line No.
javac	x	x	x
jport	x	x	
Rigi	x		
Bunch	x		
Hawa*	x	w	

Table 1. Arc Identity

Obviously the compiler (javac) identifies arcs at the finest level of granularity, including direction, type and source code line number. jport does not use line number for arc identity. In other words, if a method fetches some field on two different lines, jport will only extract one fetch arc from the method to the field.

We feel that it is not necessary to include line number information in an arcs identity for most reverse engineering tasks, although it may be useful to record such information separately. It may also be useful to record a count of how many lines an arc occurs on. However, Rigi Standard Form (RSF) does not allow such information (attributes on arcs) to be recorded easily.

It is interesting to note that while Rigi records arc type information, it is not used for arc identity. For example, if there are two arcs of different types from node A to node B, Rigi will consider the second arc a duplicate and will discard it. This is particularly annoying when viewing the graphs produced by our fact extractor, which contain many such 'duplicates'. The Bunch clustering tool also does not use arc type as a characteristic to identify arcs.

The clustering tool that we have developed (referred to as 'Hawa' above, for hierarchical, agglomerative, weighted, algorithm) retains arc direction, but replaces arc type with arc weight. The weights we used are described next.

4.2 Arc Type and Weight

We have translated arc type into arc weight for use with our clustering tool according to Table 2. We note that these weights are based on our judgment, which is described in this section. Obviously the weights have a significant effect on the clustering process, and this is an area for future study.

Arc Type	Weight
Inheritance	low
Inner Class Decl.	high
Type dependence	low
Exceptions	low
Instantiation	high
Array Creation	medium
Field Read	medium
Static Field Read	low
Field Write	high
Static Field Write	high
Invocation	medium

Table 2. Arc Weights

The 'inheritance' arcs include `extends`, `implements`, and an arc to represent 'inherited' method implementations. The previous example would contain one of these latter arcs, from `Square` to `Rectangle.draw()`. All of these arcs are produced by Class Hierarchy Analysis. The reason we give 'inheritance' related arcs a low weighting is that we do not want the result of the clustering to simply reflect the class hierarchy: this can be extracted with significantly less effort. However, each sub-class will still have a fairly strong relationship with its super-class, due to the 'inherited' method implementations, as well as constructor invocations (every sub-class constructor must invoke a super-class constructor [6]).

The type dependence arcs include method parameter and return types, field types, as well as casting and type tests. These are given a low weight because they are common, and do not really distinguish the use of a type in the same way as instantiation does, for example.

The invocation arcs include constructors, private methods, static methods, as well as the three types of polymorphic invocation in Java: virtual, interface, and super. For a more detailed discussion of these see [6, 14, 17].

4.3 Tools and XML Technology

As mentioned previously, we have used the Rigi command language to execute the results of our clustering algorithm, which produces a tree over the input data. We have found, however, that Rigi is not the best tool for viewing the results of the clustering, for the reason that Rigi's graphical representation of a tree does not leave room on the screen to show the names of the leaves (classes). It has become very apparent that one of the main ways in which we judge the results of the clustering algorithm is by how the names of the classes are organized.

Therefore, we have written another RCL script to transform the clustering results into XML, and have used the Xeena XML browser/editor from IBM Research (available on alphaWorks) for analysis. XML files are tree structured by nature, so they are a very good fit for the results of a hierarchical clustering algorithm. The point of this, of course, being that any XML browser will show the tree in a text oriented way such that the names of the nodes are readable.

A further benefit of encoding the clustering results in XML is that there are tools available for comparing XML trees to each other, such as XML Tree Diff (available on alphaWorks). This provides the opportunity to compare our three clustering results with each other, as well as comparing them with the package and 'inheritance' hierarchies.

5 Our Clustering Algorithm

The clustering algorithm that we have designed works by collapsing (agglomerating) lower level nodes into higher level nodes. This process builds a tree structure over the input nodes, and the process terminates when the root of the tree is reached. Lower level nodes are selected to be clustered together based on their 'proximity', which is determined by summing the weights of the edges between them. The user can guide the results of the algorithm by setting a very large weight between two nodes: a positive weight will force them to be clustered together, and a negative weight will force them apart. This section discusses the details of our algorithm and its implementation.

Main Steps The main steps in the algorithm are:

1. select active node
2. select agglomerate node
3. combine active and agglomerate nodes
4. repeat until only one node remains (the root)

Active Node Selection There are a number of possible criteria for selecting the active node, such as the following:

- external dependencies (coupling) normalized by the number of nodes in the cluster
- ratio of external dependencies to internal dependencies (coupling / cohesion)
- difference between internal and external dependencies (coupling − cohesion)

Our implementation uses the first approach because of its simplicity and because it allows us to incorporate arc weights into the calculation efficiently (see data structure discussion below). The Bunch clustering tool uses the third criterion for selecting the active node [15].

Agglomerate Node Selection There are also a number of possible criteria for selecting the agglomerate node:

- resulting external dependencies (coupling)
- ratio of external to internal dependencies
- internal dependencies produced (cohesion)

We have opted for the first criterion: the active node is agglomerated with the agglomeration node that yields the lowest number of external dependencies for the resulting higher level node. The objective is to reduce the number of external arcs by collapsing them inside the higher level node. This approach gives priority to low coupling, but also considers cohesion.

Figure 1 illustrates the agglomerate node selection: the black node is the active node, and the circled node has been selected as the agglomerate node, since it minimizes the external dependencies of the resulting node.

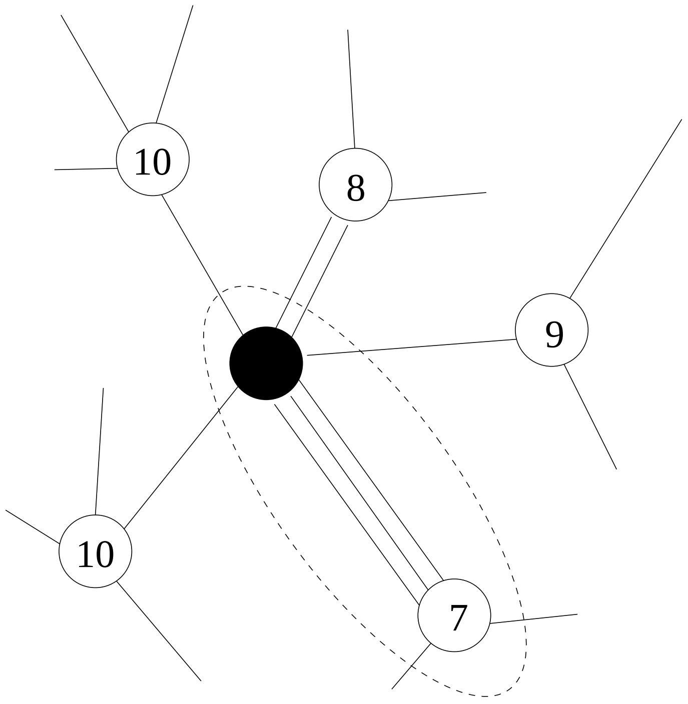

Figure 1. Agglomerate Node Selection

Combining Nodes The reader may note that always combining two lower level nodes into one higher level node will produce a binary tree over the input nodes. To reduce the height of this tree we distinguish between input level nodes and higher level nodes (i.e. those produced by the algorithm). The expected binary tree structure is created when combining two input level nodes, and when combining two higher level nodes. However, when combining an input level node with a higher level node, we simply add the input level node to the existing higher level node: a new higher level node is not created.

This strategy makes the resulting tree structure shallower and hence easier for the analyst to work with. Nevertheless, larger graphs still produce trees around ten levels deep. These trees are manageable, but we feel that applying a second algorithm to reduce the height of the tree would be worth investigating. It is worth noting that some other clustering approaches cannot produce a result for graphs of this size: a tall tree is better than nothing. A tall tree also may be of some benefit when analyzing a large program, as it allows the analyst to look at the clustering results at different levels of granularity.

In the Context of AI There are many different types of clustering algorithms that have been developed in the artificial intelligence literature, and so a standard terminology has also been developed to describe these algorithms. In this terminology, our algorithm may be described as hierarchical, exclusive, intrinsic, agglomerative, serial, and monothetic [3, 9, 21, 22].

5.1 Data Structure

We use a fairly standard adjacency matrix to represent the graph: one dimension represents the source node, and the other represents the target node. We also use a vector to hold the names of the individual nodes.

Each off-diagonal cell represents a potential arc between two nodes. A *positive* value signifies that there is an arc between the two nodes; the magnitude of the value indicates the 'weight' of the arc. A *zero* value indicates that there is no arc between the two nodes, which is the most common, and so a sparse matrix is used in practice. A *negative* value indicates that the two nodes should not be clustered together: this allows the analyst to place constraints on the outcome of the algorithm.

On-diagonal cells are used to record information about each node. We use these cells to record the number of nodes in the cluster.

5.2 Formulae

The adjacency matrix representation reduces all of the steps in the algorithm to simple matrix operations. These are given by the following formulae:

- M adjacency matrix.
- $W_{i,j} = M[i,j]$ weight from node i to node j.
  (off-diagonal values)
- $C_i = M[i,i]$ count of nodes clustered in node i.
  (on-diagonal values)
- $E_i = \sum_{\forall j}(W_{i,j} + W_{j,i}) - 2 \times W_{i,i}$
  (Sum of the weighted external arcs connected to node i = row sum + column sum $-$ on diagonal value.)
- $Active_i = E_i \times E_i \div C_i$
  (active node objective function for node i)
- $Agglomerate_i = E_i \times E_i$
  (agglomerate node objective function for node i)

The function to combine two lower level nodes x and y into one higher level node z is:

$$W'_{r,z} = W_{r,x} + W_{r,y} \forall r$$

$$W'_{z,c} = W_{x,c} + W_{y,c} \forall c$$

$$C'_z = C_x + C_y$$

In practice, the new higher level node z takes the row and column formerly occupied by x, and the row and column formerly occupied by y are deleted from the matrix.

5.3 Implementation

We have implemented our hierarchical agglomeration algorithm as matrix operations in MatLab. The algorithm generates a Rigi Command Language (RCL) script that instructs Rigi to cluster the graph according to the findings of the algorithm. There are two benefits to this approach: it reduces the memory required for the clustering algorithm, because it does not need to remember what is in each cluster, only the number of things in the cluster; and, Rigi is a well known and well tested tool for graph visualization and manipulation.

5.4 Testing

We have tested our algorithm with the simple file system given in [15] and the CLIPS expert system shell. Our algorithm produced the same result as the Bunch clustering tool [15] on the simple file system example, and produced reasonable results for CLIPS [8].

The file system example was chosen as a simple comparison of our tool with a known tool (Bunch). CLIPS was chosen for three reasons: one, it has an architecture manual; two, it is reasonably well studied in the literature; and three, it provides an opportunity to compare a procedural program with its object-oriented equivalent.

Our algorithm is able to cluster very large graphs, with thousands of nodes and tens of thousands of arcs efficiently. In fact, our algorithm is able to handle graphs larger than Rigi can: Rigi can load the graphs, but it runs out of memory before it can complete execution of the clustering script.

The running time of our algorithm is proportional to the number of nodes in the graph, as all of the arcs are reduced to their weight and then dealt with via matrix operations. The number of nodes determines the size of the matrix. For JESS, it was necessary to apply a heuristic to the graph to reduce it from about 2700 nodes to about 240 nodes so that Rigi could execute the clustering script. Our algorithm took about 10 minutes and 30mb of RAM to cluster these simplified graphs.

6 Experimental Analysis of JESS

We have chosen to use the Java Expert System Shell (JESS) version 5.0b1 as the main case study for this paper. JESS provides almost identical functionality to the well studied CLIPS expert system shell: in fact, JESS can read and execute CLIPS rule sets. Internally, both JESS and CLIPS are based on the Rete pattern-matching algorithm. JESS is also able to perform backward-chaining, and will also be able to do 'fuzzy-logic' in an upcoming release. JESS has been written such that it is fairly easy to integrate into any Java program, and can work directly with Java objects from that program.

JESS was developed with the file based Jikes compiler from IBM Research, and the author has used files to organize JESS. From Table 3 below, it can be seen that there are 91 source files and 238 binary files in JESS. 24 of these binary files represent inner classes, so there are 123 regular classes that are tertiary classes in a file in which some other class is the primary class. These classes represent almost half of the classes in JESS. JESS is organized into two packages: `jess` for most of the code, and `jess.awt` for the GUI code (which is a small fraction of the total).

Size Metric	**jess**	**./awt**	**Total**
No. of .java files	76	15	91
No. of .class files	223	15	238
No. of methods	1227	56	1283
No. of fields	525	11	536
No. of source lines	19746	807	20553
No. of `;` source lines	5596	143	5739

Table 3. JESS Size Metrics

6.1 Three JESS Clusters

The naive call graph for JESS has 2048 invocation arcs; while the Class Hierarchy Analysis graph has 3872 arcs; Rapid Type Analysis prunes 343 arcs from the CHA graph. RTA resolves 291 call groups to a single target within JESS, while 35 call groups still have more than one target within JESS. This shows that RTA can produce fairly accurate call graphs: in this example almost 90% of call groups are resolved to a single target. RTA also identifies 604 entities in JESS that are not strictly required for the program to execute (note that these may provide extra functionality for Java programs that incorporate JESS).

We have used the XML Tree Diff tool, to compare the clustering results with each other and the source file organization. Unfortunately, it seems that this tool has had some difficulty with these problems, and was not always able to compute a result. The only problems that it was able to compute successfully were transforming the CHA clustering to the RTA clustering, and transforming the RTA clustering to the source file organization. CHA to RTA requires the following tree operations: 6 adds, 1 graft, 18 moves, 125 prunes, and 52 removes. RTA to source organization requires the following tree operations: 294 adds, 0 grafts, 322 moves, 2 prunes, and 61 removes.

So, as mentioned previously, we have only been able to partially verify our hypothesis at this time: each call graph does indeed produce a different clustering, but we have not yet developed an objective basis upon which to assess them.

6.2 Considering Reflection

In our investigation of JESS, we feel that we have learned more about how it is organized from examining its use of reflection than from our attempts to cluster it. Recall that it is necessary to examine how a Java program uses reflection in order to compute Rapid Type Analysis accurately. This finding agrees with our past experience analyzing Java programs much larger than JESS, and so here we consider why this may be the case.

Java reflection is one of two options that the programmer has to deal with the 'make isn't generic' problem [11]. The other way to deal with this problem is through some programming idiom, such as delegation or the creational design patterns (e.g. abstract factory). Dynamically bound class level method can also be used to address this problem in some circumstances, but these are not available in Java (although they are in some other object-oriented languages). While the creational design patterns make instantiation more generic through the use of polymorphism, the actual instantiation is still bound statically at compile time. Reflection is the only option the Java programmer has for instantiation to be determined dynamically at run time, and therefore represents a major abstraction boundary.

Furthermore, reflection is not implemented as part of the Java language *per se*, but as a part of the virtual machine and class libraries. This makes Java reflection rather cumbersome to use, in comparison to languages that incorporate reflection more directly, such as Smalltalk, or languages that support a meta-object protocol [12].

So, because reflection addresses the 'make isn't generic' problem at the deepest level possible in Java, it tends to serve an important role when it is used. Also, because reflection is rather cumbersome to use, and because the 'make isn't generic' problem usually only needs to be addressed for a few key classes, reflection tends to be used sparingly. Therefore, the use of reflection in Java programs can usually be analyzed fairly quickly, and often gives good insight into the relation of the parts to the whole.

6.3 Compilers and Program Organization

There is a wide variation in the style in which Java programs are organized by Java programmers. JESS and our fact extractor (an experimental version of jport) serve as distinctive examples. The contrast between the package structure of these two similarly sized programs is striking: JESS consists of 238 classes in 91 files, divided into two packages. jport consists of 147 classes divided into 17 packages. JESS has an average of 119 classes (45 files) per package, whereas jport has an average of 8.6 classes per package. *This is an order of magnitude difference.*

However, there are some strong similarities between JESS and jport: both are of around 5 KLOC of uncommented code; both were written by a single author; and both are written in an advanced object-oriented style that displays good understanding of Java and design patterns.

JESS and jport differ in the development tools that were used to create them: JESS was written with a file based compiler (Jikes, from IBM Research), whereas jport was written with a repository based integrated development environment (IBM's VisualAge for Java).

While the organization of a program is certainly subject to the peculiarities of the programmer who wrote it, we feel that the choice of development environment has a significant impact. We suspect that using a repository based IDE leads to a more organized package structure for three reasons: one, the package structure is always visible to the programmer, and so is likely to get more attention than it might otherwise; two, there is no opportunity to group classes in files, and so the programmer must focus on the packages; and three, it is easier to organize classes into packages within a repository based IDE than within the file system.

7 Conclusions

We conclude along the lines of our contributions, outlined in the introduction:

- Call graph construction algorithms do indeed have an affect on clustering. Further study is required to state conclusively that the more accurate call graphs result in more accurate clusterings. Specifically, an objective basis for determining the accuracy of a clustering must be developed. This will depend on what the clustering is supposed to mean, and we have explored potential solutions to that problem here.
- We have developed a clustering algorithm that produces reasonable results on large graphs. This is important for clustering object-oriented programs, as even medium sized programs produce large graphs.
- Clustering object-oriented programs is a worthwhile area of research, as current programming languages still only allow the programmer to express limited structure (even though this has improved tremendously over the years).
- Programs developed with file based compilers and those developed with integrated development environments can display very different organizational characteristics. The influence of compiler technology of program organization is an area deserving further study.
- XML technology can be useful for managing program understanding data, particularly if that data is tree structured.

Finally, we feel that call graph construction algorithms are essential to understanding object-oriented programs. As with procedural programs, almost all important analyses require a call graph. This is even more important in object-oriented programs, as methods tend to be shorter than functions and composed of more calls.

We note that call graphs generated from actual program execution traces may also be used, as well as those constructed from a static analysis. We intend to continue investigating the application of call graph construction algorithms to the understanding of object-oriented programs.

In the future, as our clustering algorithm represents arc type with a numerical value ('weight'), we are considering 'training' the algorithm against well organized programs to determine optimal values for each arc type.

Acknowledgments

This work was funded by the IBM Canada Ltd Centre for Advanced Studies and the National Research Council of Canada. We thank the anonymous referees for their considered criticism.

References

[1] D. F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California at Berkeley, December 1997. UCB/CSD-98-1017.

[2] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of OOPSLA*, pages 324 – 341, 1996.

[3] H. Clifford and W. Stephenson. *An Introduction to Numerical Classification*. Academic Press, Inc., New York, 1975.

[4] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of ECOOP*, August 1995.

[5] A. Diwan, J. E. B. Moss, and K. S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of OOPSLA*, pages 292–305, 1996.

[6] J. Gosling, B. Joy, and G. Steele Jr. *The Java Language Specification*. Addison Wesley, 1996.

[7] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings of OOPSLA'97*, 1997.

[8] E. Hedges, D. Rayside, and S. Reuss. E&CE 756 project report. Technical report, University of Waterloo, December 1999.

[9] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[10] M. Karasick. The architecture of montana: An open and extensible programming environment with an incremental c++ compiler. In *Proceedings of sixth ACM/SIGSOFT International Symposium on the Foundations of Software Engineering*, pages pp. 131 – 142, Orlando, Florida, November 1998.

[11] G. Kiczales. Traces (a cut at the 'make isn't generic' problem). In *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS)*, 1993. LNCS 742.

[12] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[13] K. Kontogiannis and P. Patil. Evidence driven object identification in procedural systems. In *Proc. of IEEE-STEP*, 1999.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[15] S. Mancoridis, B.S.Mitchell, C. Rorres, Y.Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of IEEE 6th International Workshop on Program Understanding*, Ischia, Italy, June 1998.

[16] S. Porat, B. Mendelson, and I. Shapira. Sharpening global static analysis to cope with java. In *Proceedings of CASCON '98*, Toronto, December 1998.

[17] D. Rayside. Towards a declarative formalism for worklist style algorithms. Master's thesis, University of Waterloo, 2000. *To appear (any day now ...)*.

[18] D. Rayside and G. T. Campbell. Aristotle and object-oriented programming: Why modern students need traditional logic. In *Proceedings of the 31st ACM/SIGCSE Technical Symposium on Computer Science Education*, Austin, Texas, March 2000. The first author acknowledges that he has made some errors in this paper. These will be corrected in a subsequent publication.

[19] D. Rayside, S. Kerr, and K. Kontogiannis. Change and adaptive maintenance detection in Java software systems. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE)*, Honolulu, October 1998.

[20] D. Rayside and K. Kontogiannis. Extracting Java library subsets for deployment on embedded systems. In *Proceedings of the 3rd Conference on Software Maintenance and Re-engineering (CSMR)*, Amsterdam, March 1999.

[21] R. Shepard and J. Carrol. Additive clustering: Representation of similarities as combinations of discrete overlapping properties. *Psychological Review*, 86:pp. 87–123, 1979.

[22] P. Sneath and R. Sokal. *Numerical Taxonomy*. W.H. Freeman and Co., San Francisco, 1973.

[23] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. Technical Report 1999-4, McGill University, November 1999.

[24] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *Proceedings of OOPSLA*, November 1999.