

Mining for Program Structure*

Renée J. Miller
Dept. of Computer Science
University of Toronto
Toronto, ON M5S 3G4
miller@cs.toronto.edu

Ashish Gujarathi
Dept. of Computer & Info. Science
Ohio State University
Columbus, OH 43210
gujarath@cis.ohio-state.edu

Abstract

We report the first in a series of experiments from the Assay project that is designed to tap the wealth of information hidden in existing program analysis data. We make use of statement definition-use data along with variable typing information to learn about the structure of legacy code. Using concept analysis, we develop an interactive knowledge discovery framework into which different types of analysis data can be integrated. We present some initial case studies conducted using a tool built on top of a large warehouse of program analysis data.

1 Introduction

Recent research has investigated the use of techniques that analyze code to provide information to help software engineers perform specific program maintenance and understanding tasks. These program-analysis-based techniques can fully or partially automate maintenance tasks; however, these techniques are rarely used in practice, in part because their efficiency and effectiveness on large-scale systems have not been demonstrated empirically. In addition, many techniques require the often expensive gathering of task specific information about a program. If program analysis information can be reused for numerous tasks, the amortized cost of gathering this information is reduced.

The Assay project, sponsored by the National Science Foundation's Experimental Software Systems (ESS) Initiative, is designed to empirically investigate the applicability of program-analysis-based maintenance techniques to large-scale systems. As part of this effort, we have constructed a data warehouse of program analysis data describing a large collection of software artifacts. The warehouse has allowed us to quickly prototype and use database algorithms for performing complex mining or knowledge discovery operations on large, non-memory-resident structured data [HMPR97].

In this paper, we report the first in a series of experiments designed to tap the wealth of information hidden in existing program analysis data. Specifically, we make use of statement definition-use data along with variable typing information to learn about the structure of legacy code. The problem of inferring

*This research is being supported by the University of Toronto and an Experimental Software Systems grant under NSF Award Number 9707792.

modular structure from legacy code has been studied extensively in the literature. Reverse engineering techniques make use of various types of program analysis data to deduce structure. Often these techniques attempt to group or classify procedures according to properties of the procedures. We revisit this problem with an eye toward applying knowledge discovery techniques to produce a reverse engineering technique with the following properties.

- **Scalable** Many reverse engineering techniques use algorithms that are exponential in the number of procedures or code segments, or exponential in the number of types or variables. Such techniques have been proven effective on small programs, but do not scale to massive or even moderately large program systems with even hundreds of types or procedures. We present a technique that builds on existing reverse engineering principles and adapts them to be scalable while maintaining or improving their effectiveness.
- **Measurable** The ultimate goal of the reverse engineering process is to enhance the maintainability of a software system. To do this, a redesign or reorganization of the code is sought that is less prone to faults and easier to modify. Object-oriented design metrics can be used to guide the design or redesign process in order to minimize the impact of code changes and to reduce the risk of introducing faults as code evolves [CK94, LH93, BBM96]. Some metrics have been empirically validated as being good measures of the susceptibility of a software system to faults [BBM96]. We use these metrics to inform the knowledge discovery process and to evaluate its results.
- **Interactive** We recognize the importance of the semantic component of reverse engineering. At each stage of the knowledge discovery process, a designer may refine, and if necessary correct, the discovered results. Such a process has been referred to as data archaeology in the mining literature [BST⁺93], in recognition of the fact that the goal of discovery is to manage and sift through large volumes of complex information while permitting a user to give detailed guidance when valuable nuggets of knowledge are uncovered.

2 Architecture

This study is one component of the Assay project, which brings together experts from the fields of program analysis, experimental software engineering and database systems [HMPR97]. To enable reuse of program analysis information collected by a wide assortment of tools, we have developed a data warehouse for long-term storage and efficient integrated manipulation of this information. The warehouse provides powerful aggregation and summarization facilities to permit the extraction and analysis of relevant information. The

warehouse resides in an Informix Universal Server Object-Relational Database Management System.¹ The sophisticated data model, query language, and indexing mechanism of an object-relational system facilitate the storage of complex analysis data. While the warehouse stores information gathered from a variety of tools, in this study, we make use of information gathered by the Aristotle [HR95] and PAF [PAF98] tool suites.

In keeping with our emphasis on reuse, we have not constructed any new front-end tools for analyzing programs. Rather, we aim to discover new knowledge from information already collected (and proven useful) for other tasks. An important tenant of the Assay project is to study ways of adding value to existing program analysis techniques without increasing the time required to gather the analysis data.

3 Background

Since our techniques are founded upon a large body of work on reverse engineering, we begin with a discussion of these techniques as a means of setting the stage for our improvements. At the highest level, reverse engineering techniques can be grouped into two broad classes. The first attempts to group procedures or existing, well-defined, code segments along with variables or types into modules. The modules correspond to classes with attributes and methods [LS97, SR97, and others]. The second approach attempts to identify semantically meaningful code segments using techniques such as program slicing [GL91, and others].

In this study, we focus on the first type of reverse engineering. By grouping existing procedures or code segments automatically, a tool can distinguish between segments that exhibit good modularity and those that do not. This allows the software engineer to focus her attention on the most tangled and least modular code segments. Code segments are typically grouped according to their common features or characteristics.

Several types of characteristics have been used to group program segments. These include the usage (or definition) of global variables [LS97, SMLD97], usage of user-defined types [GK97, SR97], the types of arguments and return types of functions or procedures [CCTM94, SR97], procedure calls [Sch91, CV95], and the frequency of procedure calls or variable usage [Sch91].

To perform the grouping, different methods have been employed. One technique has emerged as being particularly effective in this respect. Concept analysis provides a formal mechanism to group objects that are maximally similar based on a description of the objects by a set of attributes [Wil81]. The method builds a lattice of concepts where each concept contains the largest set of objects sharing a common set of attributes (or equivalently, the largest set of attributes shared by a set of objects). With appropriately chosen attributes, the concept lattice has been shown to provide valuable insight into the structure of software systems [ST98, LS97, SR97, Sne96, Sne98].

¹The database management system is provided through an Informix Engines for Innovation Research Grant.

We begin by introducing some basic notation and definitions from concept analysis, then illustrate, through an example, how concept analysis has been applied to the problem of inferring program structure.

3.1 Concept Analysis

Concepts are constructed on *contexts* which are triples, $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$, containing a set of objects \mathcal{O} , a set of attributes \mathcal{A} describing the objects, and a binary relation \mathcal{R} between \mathcal{O} and \mathcal{A} specifying which attributes hold for which objects. Consider the C code of Figure 1. Let \mathcal{O} be the set of procedures of this program (Table 1) and let \mathcal{A} be the set of attributes listed in Table 2. Table 3 gives the relation \mathcal{R} (in the form of a relation matrix) on the code of Figure 1. In Table 3, an X in row P_1 , column A_1 indicates that the procedure `initStack` has a return type of `stack`.

P_1	<code>initStack</code>
P_2	<code>initQ</code>
P_3	<code>isEmptyStack</code>
P_4	<code>isEmptyQ</code>
P_5	<code>push</code>
P_6	<code>enq</code>
P_7	<code>pop</code>
P_8	<code>deq</code>

Table 1: Program objects for code of Figure 1.

A_1	Return type is struct <code>stack</code> *
A_2	Return type is struct <code>queue</code> *
A_3	Has Argument of type struct <code>stack</code> *
A_4	Has Argument of type struct <code>queue</code> *
A_5	Uses or defines fields of struct <code>stack</code>
A_6	Uses or defines fields of struct <code>queue</code>

Table 2: Attributes of program objects for code of Figure 1.

	A_1	A_2	A_3	A_4	A_5	A_6
P_1	X				X	
P_2		X				X
P_3			X		X	
P_4				X		X
P_5			X		X	
P_6				X		X
P_7			X		X	
P_8				X		X

Table 3: Relation matrix for the code of Figure 1.

```

#define QUEUE_SIZE 10
struct stack { int *base;
               int *sp;
               int size};
struct queue {struct stack *front, *back};

struct stack* initStack(int sz)
{ struct stack *s = (struct stack *)
  malloc(sizeof(struct stack));
  s->base = s->sp =
    (int *) malloc(sz *(sizeof(int)));
  s->size = sz;
  return s;}

struct queue *initQ()
{ struct queue *q = (struct queue *)
  malloc(sizeof(struct queue));
  q->front = initStack(QUEUE_SIZE);
  q->back = initStack(QUEUE_SIZE);
  return q;}

int isEmptyStack(struct stack *s)
{ return (s->sp == s->base);}

int isEmptyQ(struct queue *q)
{ return (isEmptyStack(q->front) &&
  isEmptyStack(q->back));}

void push(struct stack *s,int i)
{ *(s->sp) = i;
  s->sp++;}

void enq(struct queue *q ,int i)
{ push(q->front,i);}

int pop(struct stack *s)
{ if(isEmptyStack(s))
  return -1;
  s->sp--;
  return(*(s->sp));}

int deq(struct queue *q)
{ if (isEmptyQ(q))
  return -1;
  if (isEmptyStack(q->back))
  while(!isEmptyStack(q->front))
  push(q->back,pop(q->front));
  return(pop(q->back));}

```

Figure 1: Example program using the user defined types stack and queue.

For a set of objects $X \subseteq \mathcal{O}$, the attribute mapping σ gives the attributes common to all objects in X , $\sigma(X) = \{a | \forall o \in X, (o, a) \in \mathcal{R}\}$. For a set of attributes $Y \subseteq \mathcal{A}$, the object mapping τ gives the objects possessing all attributes in Y , $\tau(Y) = \{o | \forall a \in Y, (o, a) \in \mathcal{R}\}$. A *concept* is a set of objects X and attributes Y such that $\sigma(X) = Y$ and $X = \tau(Y)$. So a concept (X, Y) is a set of objects and attributes such that Y is the largest set of attributes shared by all objects X and no other objects have all these attributes. For a concept (X, Y) , the set of objects X is referred to as the *extent* of the concept and the set of attributes Y as the *intent* of the concept. Table 4 shows the concepts for our example context.

\top	$(\{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8\}, \emptyset)$	
C_6	$(\{P_2, P_4, P_6, P_8\}, \{A_6\})$	Queue concept
C_5	$(\{P_1, P_3, P_5, P_7\}, \{A_5\})$	Stack concept
C_4	$(\{P_4, P_6, P_8\}, \{A_4, A_6\})$	isEmptyQ, enq, deq
C_3	$(\{P_3, P_5, P_7\}, \{A_3, A_5\})$	isEmptyStack, push, pop
C_2	$(\{P_2\}, \{A_2, A_6\})$	initQ
C_1	$(\{P_1\}, \{A_1, A_5\})$	initStack
\perp	$(\emptyset, \{A_1, A_2, A_3, A_4, A_5, A_6\})$	

Table 4: Concepts for code of Figure 1.

The set of concepts over a given context can be ordered by the *subconcept relation* $(X_1, Y_1) \subseteq (X_2, Y_2)$ if $X_1 \subseteq X_2$, or, equivalently, $Y_2 \subseteq Y_1$. The subconcept relation forms the *concept lattice*. Figure 2 shows the concept lattice for the concepts of Table 4. The top element (\top) corresponds to the concept formed by all objects. The intent of \top is often empty. The bottom element (\perp) corresponds to the concept formed by all attributes. The extent of \perp is often empty. We will omit concepts with empty intents or extents from our tables throughout the paper.

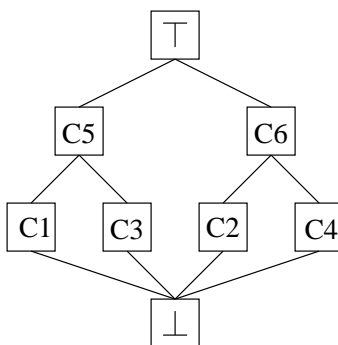


Figure 2: Concept lattice for code of Figure 1.

Note that the concepts C_5 and C_6 correspond to natural groupings of procedures into those that manipulate each of the two user defined types. By grouping the objects and attributes of these two concepts together, we can define natural program classes, the queue class and the stack class. Once these groupings have been determined, a tool can easily restructure the code into the classes of Figure 3. Notice that using

```

const int QUEUE_SIZE = 10;
class stack {
private:
    int *base;
    int *sp;
    int *size;
public:
    stack (int sz){
        base = sp = new int[sz];
        size = sz;}
    int isEmpty(){
        return (sp == base)}
    int pop {
        if(isEmpty())
            return -1;

        sp--;
        return (*sp); }
    void push(int i){
        *sp = i;
        sp++;}
}

class queue{
private:
    stack * front, *back;
public:
    queue(){
        front = new stack(QUEUE_SIZE);
        back = new stack(QUEUE_SIZE);}
    int isEmpty() {
        return (front->isEmpty() && back->isEmpty());}
    int deq() {
        if(isEmpty())
            return -1;
        if(back->isEmpty())
            while (!front->isEmpty())
                back->push(front->pop());
        return back->pop();}
    void enq(int i){
        front->push(i);}
}

```

Figure 3: Code of Figure 1 reverse engineered using concept analysis into C++ code.

type or variable usage information, concept analysis automatically generates cohesive classes [HS96]. In a concept containing the attributes *uses type x* and *uses type y*, all procedures, by definition, use both type x and y. This property has been an important motivation for much of the work on using concept analysis to reveal program structure.

To compute the concept lattice, a simple bottom-up algorithm can be used [SR97]. First, all objects that share all attributes in the context are computed. This set of objects forms the extent of the bottom element of the lattice. Then, for each object o , the smallest concept containing o is computed. Such concepts are referred to as the *atomic* concepts and can be computed as $(\tau(\sigma(o)), \sigma(o))$. The atomic concepts are the immediate suprema of the bottom concept in the concept lattice. The suprema of the atomic concepts can be computed by taking each pair of concepts and intersecting their intents and joining their extents. For a pair of concepts, A and B the immediate supremum would be the concept $(\tau(\text{intent}(A) \cap \text{intent}(B)), \text{intent}(A) \cup \text{intent}(B))$.

There are two primary issues that must be addressed to realize a reverse engineering tool based on this approach. The first is the selection of attributes. The second is the selection of a subset of concepts from the potentially large set of possible concepts. We address each of these issues in turn and report on the approaches that have been proposed to date.

3.2 Choosing Attributes

One of the critical problems in concept analysis is deciding which attributes to use. Lindig and Snelting have suggested the use of *names* of global variables in FORTRAN programs [LS97]. Hence, a procedure or program segment has an attribute for each global variable it uses. For C programs, Siff and Reps have suggested using attributes related to user defined data types [SR97]. For modular code, they suggest that attributes corresponding to the usage of user defined types may be sufficient to group procedures. For less modular code, they include separate attributes based on the way a variable of a user defined type is used. For each user defined type x , there is an attribute indicating an object *uses fields of type x*, *has an argument of type x*, and *has return type of x*. They argue that this choice of attributes works well for C programs which are already modularized to a certain extent. This argument is supported by our example which uses the attributes proposed by Siff and Reps. The queue and stack structures correspond to the concepts C_5 and C_6 identified by this approach.

However, on less modular code, these attributes may not work as well. Consider the stack and queue example with slight modifications to the procedures `isEmptyQ` and `enq` such that they use the fields of the stack type directly, instead of calling `isEmptyStack` and `push` (Figure 4). Here, the procedures that should be associated with the stack module are not all of those that use fields of stack (which would include both `isEmptyQ` and `enq`). Rather, what distinguishes the procedures of the stack module is that they use fields

```

void enq(struct queue *q ,int i){
  *(q->front->sp)= i;
  q->front->sp++;}

int isEmptyQ(struct queue *q){
  return
  (q->front->sp == q->front->base
   && q->back->sp == q->back->base);}

```

Figure 4: Modification to two procedures of the code of Figure 1.

of stack, but do not use fields of queue. To handle such code, Siff and Reps suggest the use of negative attributes (*does not use fields of queue*) to identify modules [SR97].

For the modified code of Figure 4, using the negative attribute *does not use fields of queue* (denoted $!A_6$) in addition to the positive attributes defined in Table 2, yields the relation matrix of Table 5. The resulting concepts are shown in Table 6. The matrix shows concepts C_6 and C_7 , which correspond to the stack and queue concepts respectively.

	A_1	A_2	A_3	A_4	A_5	A_6	$!A_6$
P_1	X				X		X
P_2		X				X	
P_3			X		X		X
P_4				X	X	X	
P_5			X		X		X
P_6				X	X	X	
P_7			X		X		X
P_8				X		X	

Table 5: Relation matrix for the modified code of Figure 4 using one negative attribute.

C_8	$(\{P_1, P_3, P_4, P_5, P_6, P_7\}, \{A_5\})$	
C_7	$(\{P_2, P_4, P_6, P_8\}, \{A_6\})$	Queue concept
C_6	$(\{P_1, P_3, P_5, P_7\}, \{A_5, !A_6\})$	Stack concept
C_5	$(\{P_4, P_6, P_8\}, \{A_4, A_6\})$	isEmptyQ, enq, deq
C_4	$(\{P_4, P_6\}, \{A_4, A_5, A_6\})$	isEmptyQ, enq
C_3	$(\{P_3, P_5, P_7\}, \{A_3, A_5, !A_6\})$	isEmptyStack, push, pop
C_2	$(\{P_2\}, \{A_2, A_6\})$	initQ
C_1	$(\{P_1\}, \{A_1, A_5, !A_6\})$	initStack

Table 6: Concepts for modified code of Figure 4.

The use of a single (well chosen) negative attribute in this example produces meaningful concepts. However, to be useful in practice, a software engineer must be able to discern which negative attributes to use in order to yield the best concepts. Such discernment requires intimate knowledge of the legacy code along with in depth knowledge of concept analysis. An inappropriate choice can explode the number of concepts,

hiding or omitting relevant groupings of procedures.

The application of concept analysis is not limited to type usage information. Information about procedure calls (or their frequency), type definitions, or variable usage, may be used to define object features. Indeed, research has shown how each of these different types of information can be useful in inferring structure from legacy code. However, as our examples show, the indiscriminate application of concept analysis to a large set of attributes will likely yield unwieldy, tangled concept lattices. Before considering the addition of new types of attributes, it is important to consider how concepts are selected and used in the reverse engineering process.

3.3 Selection of Concepts

Once a set of attributes has been selected, the next step is to determine a set of concepts that best reflects the program structure. The number of concepts can be exponential in the number of attributes (or objects) so it is important that a subset of the concepts can be quickly determined as relevant to the program structuring task. Using attributes such as variable or type usage, the selected concepts can be used to define abstract data types or classes by grouping the procedures of a concept together with the variables or types of the concept's attributes [LS97, SR97]. When additional attributes are used, such as argument, return type or negative attributes, the mapping from concepts to classes is less clear. Table 7 is the relation matrix for the (slightly tangled) code of Figure 4 using the two negative attributes *does not use fields of stack* ($\neg A_5$) and *does not use fields of queue* ($\neg A_6$). The resulting concepts are shown in Table 8. To map concepts to classes, one might single out the Concept C_6 to identify the stack concept as it has attributes *uses fields of stack* and *does not use fields of queue*. Similar reasoning would identify Concept C_9 as the queue concept since it has attributes *uses fields of queue* and *does not use fields of stack*. Thus the procedures P_1, P_3, P_5 and P_7 can be made the methods of the class stack and the procedures P_2 and P_8 , the methods of the class queue. The concepts C_7 and C_8 are ambiguous since the absence of negative attributes in their intents, indicates that some of procedures in the extents of these concepts use fields of both stack and queue. C_4 is the only concept with procedures P_4 and P_6 , and examining their attributes, it is not possible to make these procedures as methods of either class stack or queue. For such cases, it would be possible to create a new class which will just have as methods P_4 and P_6 and declare this class as a friend of both the class stack and the class queue.

The process of assigning objects to classes has not been fully formalized in the presence of return type, argument type and negative attributes. One heuristic (suggested in the paragraph above) might favor the use of concepts containing either a negative or positive usage attribute for every type. To guide the process of assigning objects to classes, Siff and Reps suggest the use of a concept partition, that is a set of concepts that partitions the set of objects [SR97]. Each partition corresponds to a complete classification

	A_1	A_2	A_3	A_4	A_5	A_6	$!A_5$	$!A_6$
P_1	X				X			X
P_2		X				X	X	
P_3			X		X			X
P_4				X	X	X		
P_5			X		X			X
P_6				X	X	X		
P_7			X		X			X
P_8				X		X	X	

Table 7: Relation matrix for code of Figure 4 using two negative attributes.

C_9	$(\{P_2, P_8\}, \{A_6, !A_5\})$	Possible Queue Concept
C_8	$(\{P_2, P_4, P_6, P_8\}, \{A_6\})$	Possible Queue Concept
C_7	$(\{P_1, P_3, P_4, P_5, P_6, P_7\}, \{A_5\})$??????
C_6	$(\{P_1, P_3, P_5, P_7\}, \{A_5, !A_6\})$	Stack Concept
C_5	$(\{P_8\}, \{A_4, A_6, !A_5\})$	deq
C_4	$(\{P_4, P_6\}, \{A_4, A_5, A_6\})$	isEmptyQ, enq
C_3	$(\{P_3, P_5, P_7\}, \{A_3, A_5, !A_6\})$	isEmpty Stack, push, pop
C_2	$(\{P_2\}, \{A_2, A_6, !A_5\})$	initQ
C_1	$(\{P_1\}, \{A_1, A_5, !A_6\})$	initStack

Table 8: Concepts for relation matrix of Table 7.

of objects and can be used as the basis for deriving a modularization. Selection among partitions is done using heuristic reasoning about the attributes of concepts. In our first example of Figure 1 and Table 4, the sets $\{C_1, C_2, C_3, C_4\}$, $\{C_1, C_3, C_6\}$, $\{C_2, C_4, C_5\}$ and $\{C_5, C_6\}$ each forms a partition and therefore a potential modularization. For the modified code of Figure 4 using two negative attributes, the resulting concepts (Table 8) can be grouped into seven (non-trivial) concept partitions ($\{C_5, C_7\}$, $\{C_1, C_3, C_4, C_9\}$, etc.) Even for this toy example with only eight procedures and eight attributes, adding in more attributes will greatly increase the number of both concepts and concept partitions. These partitions must be analyzed by a software engineer.

Note that the problem does not rest exclusively with the use of negative attributes. Even the addition of the return type and argument type attributes can result in extraneous concepts if type usage information is sufficient to derive interesting concepts. Using multiple types of attributes can result in an overly complex lattice, particularly if the attributes are not independent. For typing information, a procedure cannot have a return or argument type of x unless a variable of type x is used. Hence, a concept containing an argument or return type attribute, must necessarily contain the corresponding usage attribute. Such dependence between attributes are known *a priori*. We would like to simplify the concept lattice to include only dependencies discovered through concept analysis. By doing so, we can refine the knowledge discovery process to focus on discovered dependencies, rather than those dictated by a programming language or programming convention.

4 Algorithm

While research has shown that relying on type and variable usage information is not sufficient to reverse engineer legacy code, it is not clear how to add additional information into the knowledge discovery process in a systematic, clean way. The direct use of a large set of attributes would lead to an explosion in the number of concepts and the complexity of the concept lattice limiting the usefulness of the approach for large programs. In order to integrate additional information in the knowledge discovery process, we make use of an incremental, interactive approach. We use type usage information to classify procedures. For some portions of a legacy system, this information may be sufficient to deduce structure. For more tangled portions of the code, we re-apply concept analysis using finer grain attributes including return type, argument type, and procedure call attributes.

4.1 Degree of Association Between a Type and a Procedure

For portions of the code that are modularized to a certain extent, some of the procedures will use fields of only one type. Hence, some of the concepts produced will have just one positive type usage attribute. The remaining usage attributes will be negative. Thus the procedures in the extents of such concepts can be associated with the only type being used and a corresponding class can be created. To find such procedures, we can apply concept analysis using only positive and negative usage attributes.

Ideally every concept would map to one class (this would be true if every procedure used fields of just one user defined type and no other). Clearly, this will not be the case for legacy code. Rather, most of the concepts that are obtained will use multiple user defined types. For every concept that has procedures which use more than one type, we need to determine to which data type the procedures are most closely associated. To do this, we re-apply concept analysis treating every concept containing multiple types as a context by itself. That is, the set of procedures in this concept is considered as the object set. The relation matrix is created using a different attribute. Two types are placed in the same class only if there are procedures that are associated with both types to the same degree. Provided below is the list of attributes that are used to measure the degree of association between a procedure and a type.

1. **Uses type x .** If a procedure uses fields of a single type and no other, it is classified as a method of that type. If a procedure uses fields of multiple types (i.e., is in a concept with multiple positive usage attributes), then concept analysis is re-applied using the attributes of Level 2.

Let $\mathcal{A}_1 = \{ \text{uses } x | x \text{ is a user defined type} \} \cup \{ \text{does not use } x | x \text{ is a user defined type} \}$.

2. **Defines type x .** If a procedure uses two or more types, but defines fields of just one type x , then this procedure is more closely associated with type x . This criteria is similar to the receiver based object

identification method suggested by Livadas and Roy [LR92].

Let $\mathcal{A}_2 = \{ \text{defines } x | x \text{ is a user defined type} \} \cup \{ \text{does not define } x | x \text{ is a user defined type} \}$.

3. **Return type x .** If a procedure uses and defines multiple types, but has a single return type of x , then it is very likely that it constructs an instance of type x and initializes its fields. Thus this procedure can be thought of as a constructor of x and is associated with the class for x .

Let $\mathcal{A}_3 = \{ \text{has return type } x | x \text{ is a user defined type} \} \cup \{ \text{does not have return type } x | x \text{ is a user defined type} \}$.

4. **Argument type x .** If a procedure has an argument type of x and if it also defines fields of this type, then it can be thought of as a method of this class.

Let $\mathcal{A}_4 = \{ \text{has argument of type } x | x \text{ is a user defined type} \} \cup \{ \text{does not have argument of type } x | x \text{ is a user defined type} \}$.

5. **Frequency of uses of x .** If a procedure cannot be associated with a single type based on the way it uses types, then the procedure can be associated with the type it uses most. This criteria has been suggested by others [GK97].

Let $\mathcal{A}_5 = \{ \text{uses } x \text{ more than any other type } | x \text{ is a user defined type} \} \cup \{ \text{does not use } x \text{ the most } | x \text{ is a user defined type} \}$.

4.2 Algorithm

Let $C = (\mathcal{O}, \mathcal{A}, \mathcal{R})$ be the context on which concept analysis will be applied. Initialize \mathcal{O} to be the set of procedures of the program and let $\mathcal{A} = \mathcal{A}_1$. Let $R = \{(o, a) \mid \text{procedure } o \text{ satisfies attribute } a\}$. Let the level $l = 1$.

1. Find the atomic concepts on $(\mathcal{O}, \mathcal{A}, \mathcal{R})$. Since \mathcal{A} includes all negative attributes, the atomic concepts will form a partition, which will be used for creating the classes [SR97].
2. For every concept that has just one positive attribute for type x associate all procedures in the extent of the concept with the class which has as its data members the fields of x .
3. For every Concept C that has more than one positive attribute, do the following.
 - (a) If level $l = 1$ and none of the types referred to by positive attributes of C are present in the intent of any other concept at level 1, then create a class, which has as data members the fields of all the types of the concept and as methods, all the procedures in the extent of the concept.

- (b) Otherwise, let $l = l + 1$ and create a new context, where $\mathcal{O} = \text{extent (objects) of } C$ and $\mathcal{A} = \mathcal{A}_l$.
Let $R = \{(o, a) \mid \text{procedure } o \text{ satisfies attribute } a\}$. Goto Step 1.

After each iteration, a software engineer may review and modify the assignment of procedures to classes. We have found that such an interactive approach is particularly effective in allowing a software engineer to quickly identify tangled portions of code that may not be amenable to reverse engineering and may require rewriting.

This approach has numerous advantages over existing methods.

- Separation and prioritization of concerns. If a set of procedures only uses a single data type, there is no need to analyze and subdivide the procedures based on shared argument or return type information. In the stack and queue code, the four procedures `initStack`, `isEmptyStack`, `push` and `pop` can be immediately classified as part of stack without the processing required to form the concepts C_1 and C_3 of Table 8.
- Reduced computation. By using all negative attributes and applying concept analysis incrementally, an atomic partition will always exist that can be used to group procedures. We use a bottom up algorithm to compute the atomic partition without computing the full (potentially massive) concept lattice. The atomic concepts can be found efficiently using simple database queries over a database with appropriate indexes constructed.
- Automatic selection of attributes. We do not require users to understand code to be able to determine which subset of attributes to use. However, a software engineer can control which type of information is used (for example, type usage or frequency information) and tailor these to the programming language or programming convention of the legacy system. The control can be directed by the results of each mining stage. As we will discuss in Section 5, our experiments show that this feature can be particularly useful.

5 Experiments

We present a few representative case studies in which we evaluate the results of our algorithm using some standard metrics for evaluating object-oriented designs [BBM96, CK94, HS96, LH93]. Specifically, we report the following metrics.

1. **NLM** The number of local methods for a class.
2. **NRM** The number of (distinct) remote methods called within the methods of a class.

3. **RFC** The response for a class which is the sum of the number of local and remote methods ($RFC = NLM + NRM$).
4. **MPC** The message passing coupling which is the number of remote method calls. If the same remote method is called by two local methods, $MPC = 2$ whereas $NRM = 1$.
5. **NRCM** The number of (distinct) remote classes whose methods are called within the methods of a class.
6. **NRCV** The number of (distinct) remote classes whose instance variables are used within the methods of a class. Note that if instance variables are private members and a class uses instance variables of another class, then the former must be declared as a friend of the latter.

The first four metrics measure the complexity of a class. The more complex a class, the more fault-prone and harder to modify it may be [BBM96]. The last two metrics measure the coupling between classes. Coupling indicates dependence between classes that can also make the classes harder to maintain [BBM96].

5.1 Case Study 1

Our first case study is a relatively small C program called *DejaVu* which implements a regression test selection algorithm [HR97]. The program has 91 procedures and 23 user-defined types. Initially, concept analysis was applied using only type usage attributes (positive and negative) for all 23 types. This produced 36 atomic concepts forming the atomic partition. Of the 36 concepts, 13 concepts had just one positive attribute in their intents. For each of the corresponding 13 types, a class was created and the procedures in the extents of the concepts were associated with the respective classes. For the procedures in the remaining concepts, concept analysis was re-applied using the definition attributes (Level 2). This resulted in 18 procedures being associated with some type. Successive application of concept analysis based on the return type and argument type attributes resulted in only one and two procedures being associated with some type respectively. Finally, each of the remaining 18 procedures were associated with some type based on the maximum usage frequency attributes. Eight procedures could not be associated with any type, as they did not use fields of any type.

Table 9 shows the metrics computed for the discovered classes. Three types were not associated with any procedures and are not reported in the table. For the remaining classes, the metrics indicate that the classes are relatively maintainable and uncoupled. Notice in particular the last two classes. The metrics indicate that these classes, which use some instance variables of other classes but no methods of other classes, are well-defined modules.

There are a few classes that are less modular and more tightly coupled with other classes. The 18 procedures that were classified based on the type they used the most (level 5), were assigned to these classes

Class Name	NLM	NRM	RFC	MPC	NRCM	NRCV
CALL_LIST	1	0	1	0	0	0
CSTACK_TYPE	2	0	2	0	0	2
DBH_CF_EDGE_INFO	1	0	1	0	0	0
DBH_CF_INFO	1	3	4	5	2	4
DBH_MAP_INFO	5	1	6	2	1	4
DBH_PROC_NAMES	4	2	6	8	1	1
DBH_SRC_INFO	3	0	3	0	0	1
DBH_SYM_TABLE	4	3	7	10	2	3
DBH_TH_INFO	11	1	12	3	1	2
DB_FILE	18	1	19	3	0	9
E_LIST	2	0	2	0	0	3
FILE	4	29	33	60	10	6
NAME_LIST_NODE_TYPE	2	0	2	0	0	2
NODE_LIST	1	0	1	0	0	0
N_LIST	6	8	14	15	6	0
UNREACHABLE	1	0	1	0	0	0
VIS_ARRAY_TYPE	2	0	2	0	0	2
VIS_LIST_TYPE	2	0	2	0	0	2
list_element	7	0	7	0	0	2
list_header	6	0	6	0	0	4

Table 9: Metrics for the reverse engineered classes of the DejaVu program.

indicating that the frequency criteria may not be a good indicator of structure for this code. The class FILE in particular has very high values for NRM (29), RFC (33), MPC (60) and for the number of remote classes accessed (10). Examining the methods of FILE, we found that the main() procedure of the DejaVu program had been associated with the FILE type. Removing the main procedure from the class FILE, resulted in lower values of the metrics as expected, NRM = 6, RFC = 9, MPC = 20 and NRCM = 2.

Examining the code also reveals that the discovered classes form an excellent starting point for reverse engineering. Modular portions of the code are separated out allowing a software engineer to focus on the more tangled portions.

5.2 Case Study 2

Our next case study is a C program called Space, which is a parser for antenna-array description language, provided by Alberto Pasquini. The program has 137 procedures and 14 types. Table 10 shows the number of procedures that were associated with some class when concept analysis was applied using different attributes. Twenty-eight procedures could not be associated with any type as they did not use any user defined types.

Table 11 shows the metrics for the discovered classes. Two types were not associated with any procedures and are not reported in the table. The classes are significantly more complex than those of the DejaVu program, however, the class coupling is still relatively low for many classes suggesting the discovered groupings

Attribute Used	Number of Procedures
Uses type	74
Defines type	5
Return type	0
Argument type	4
Usage frequency	26

Table 10: Breakdown of information used to classify the procedures of the Space program.

Class Name	NLM	NRM	RFC	MPC	NRCM	NRCV
AddRem	3	8	11	34	2	3
Elem	2	6	8	9	2	4
FILE	1	1	2	1	0	1
Geomnode	13	7	20	22	1	3
Geomport	4	3	7	4	1	2
GrAmpExc	2	6	8	15	1	3
GrPhaExc	2	8	10	25	1	4
Grid	3	16	19	32	2	2
Group	5	23	28	30	7	5
Node	5	5	10	29	1	1
Port	3	11	14	20	1	3
charac	66	16	82	17	8	8

Table 11: Metrics for the reverse engineered classes of the Space program.

may be a good starting point for reverse engineering. Note that the class `charac` has 66 local methods and 60 of them use no other type. The remaining six methods were assigned to the class at the later levels of the knowledge discovery process. A software engineer may chose to separate these methods from the class. Doing so would significantly reduce the complexity of the class by dropping the NRM, NRCM, and NRCV values to zero.

5.3 Case Study 3

Our third case study is a C program called Empire, which is an Internet game which has 693 procedures and 69 types. Applying concept analysis using type usage attributes produced 265 concepts, which is quite high, indicating that the code is not very modular. Table 12 shows the number of procedures that were associated with some type, when concept analysis was applied using different attributes. One hundred and forty six procedures did not use any user defined types.

Sixty-nine classes were created, of which approximately 50 classes had very low values for the complexity and coupling metrics. The remaining classes had high values for the above metrics, suggesting that this part of the program requires additional manual attention from a software engineer.

Attribute Used	Number of Procedures
Uses type	187
Defines type	106
Return type	4
Argument type	60
Usage frequency	190

Table 12: Breakdown of information used to classify the procedures of the Empire program.

6 Conclusion

We have described one study conducted as part of the Assay project. In keeping with the philosophy of the overall project, we have made use of an existing warehouse of software analysis data and demonstrated how knowledge discovery techniques can be employed to discover structural information about legacy programs. Our techniques make use of standard definition-use information which may already be gathered to support numerous other software testing and maintenance techniques [HR95, PAF98]. To infer modular structure, we make use of concept analysis, a technique employed in several reverse engineering studies [Sne96, Sne98, SR97, LS97]. We improve upon these results by proposing an interactive framework for knowledge discovery into which different program analysis data can be integrated. We have presented preliminary experiments on only moderately large programs (up to 42,000 lines of code) using the framework with an array of typing information. We are extending these experiments to large programs. Interestingly, the current stumbling block for this work is getting larger programs through the PAF and Aristotle front-ends which gather the program analysis data. The algorithms we propose will easily scale to thousands or even millions of program objects given the low complexity (which is effectively linear when appropriate indices are used). We also plan to experiment on using our framework with other forms of analysis data including procedure call information.

References

- [BBM96] V. Basili, L. Briand, and W. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.
- [BST⁺93] R. J. Brachman, P. G. Selfridge, L. G. Terveen, B. Altman, A. Borgida, F. Halper, T. Kirk, A. Lazar, D. L. McGuinness, and L. A. Resnick. Integrated Support for Data Archaeology. *Int'l Journal of Intelligent and Cooperative Information Systems*, 2(2):159–185, 1993.
- [CCTM94] G. Canfora, A. Cimitile, M. Tortorella, and M. Munro. A Precise Method for Identifying Reusable Abstract Data Types in Code. In *International Conference on Software Maintenance*, pages 404–413, September 1994.

- [CK94] S. Chidamber and C. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
- [CV95] A. Cimitile and G. Visaggio. Software Salvaging and the Call Dominance Tree. *Journal of Systems and Software*, 28(2):117–127, February 1995.
- [GK97] J. Girard and R. Koschke. Finding Components in a Hierarchy of Modules - a Step towards Architectural Understanding. In *International Conference on Software Maintenance*, pages 58–65, 1997.
- [GL91] Keith Brian Gallagher and James R. Lyle. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.
- [HMPR97] M. J. Harrold, R. J. Miller, A. Porter, and G. Rothermel. A Collaborative Investigation of Program-Analysis-Based Testing and Maintenance. In *International Workshop on Experimental Studies of Software Maintenance*, pages 51–56, Bari, Italy, October 1997.
- [HR95] M. J. Harrold and G. Rothermel. Aristotle: A System for Research on and Development of Program Analysis Based Tools. Technical Report OSU-CISRC-3/97-TR17, Ohio State University, Department of Computer Science, 1995.
- [HR97] M. J. Harrold and G. Rothermel. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.
- [HS96] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.
- [LH93] W. Li and S. Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 23:111–122, November 1993.
- [LR92] P. Livadas and P. Roy. Program Dependence Analysis. In *International Conference on Software Maintenance*, pages 356–365, 1992.
- [LS97] C. Lindig and G. Snelting. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. In *International Conference on Software Engineering*, pages 349–359, 1997.
- [PAF98] PAF: PROLANGS Analysis Framework. Programming Languages Research Group. Rutgers University. <http://www.prolangs.rutgers.edu/public.html>, 1998.
- [Sch91] R. W. Schwanke. An Intelligent Tool for Re-engineering Software Modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.

- [SMLD97] H.A. Sahraoui, W. Melo, H. Lounis, and F. Dumont. Applying Concept Formation Methods to Object Identification in Procedural Code. In *Proc. of IEEE Automated Software Engineering Conference*, pages 210–218, November 1997.
- [Sne96] Gregor Snelting. Reengineering of Configurations Based on Mathematical Concept Analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–89, April 1996.
- [Sne98] Gregor Snelting. Concept Analysis — A New Framework for Program Understanding (Invited Paper). *ACM SIGPLAN Notices*, 33(7):1–10, July 1998.
- [SR97] M. Siff and T. Reps. Identifying Modules Via Concept Analysis. In *International Conference on Software Engineering*, pages 170–179, October 1997.
- [ST98] G. Snelting and F. Tip. Reengineering Class Hierarchies Using Concept Analysis. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, November 1998.
- [Wil81] R. Wille. Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts. In Ivan Rival, editor, *Ordered Sets*, pages 445–470. NATO Advanced Study Institute, September 1981.