Identifying Modules Via Concept Analysis*

Michael Siff and Thomas Reps
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706
{siff, reps}@cs.wisc.edu

Abstract

We describe a general technique for identifying modules in legacy code. The method is based on concept analysis*—a branch of lattice theory that can be used to identify similarities among a set of* objects *based on their* attributes*. We discuss how concept analysis can identify potential modules using both "positive" and "negative" information. We present an algorithmic framework to construct a lattice of concepts from a program, where each concept represents a potential module. We define the notion of a concept partition, present an algorithm for discovering all concept partitions of a given concept lattice, and prove the algorithm correct.*

1 Introduction

Many existing software systems were developed using programming languages and paradigms that do not incorporate object-oriented features and design principles. In particular, these systems often lack a modular style, making maintenance and further enhancement an arduous task. The software engineer's job would be less difficult if there were tools that could transform code that does not make explicit use of modules into functionally equivalent object-oriented code that does make use of modules (or classes). Given a tool to (partially) automate such a transformation, legacy systems could be modernized, making them easier to maintain. The modularization of programs offers the added benefit of increased opportunity for code reuse.

A major difficulty with software modularization is the accurate identification of potential modules and classes. This paper describes how a technique known

*A preliminary version of this paper appeared in the Proceedings of the International Conference on Software Maintenance [20]. Part of this research was done while Siff was a summer intern at Lucent Technologies. This work was supported in part by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937), by the National Science Foundation under grant CCR-9625667, and by a Vilas Associate Award from the University of Wisconsin.

as *concept analysis* can help automate modularization. The main contributions of this paper are:

- We show how to apply concept analysis to the modularization problem. We focus on one variant of the modularization problem—the conversion of a C program to a C++ program, where the C program's `struct` types are the starting point for the C++ program's classes.

- Previous work on the modularization problem has made use only of "positive" information: Modules are identified based on properties such as "function `f` uses variable `x`" or "`f` has an argument of type `t`". It is sometimes the case that a module can be identified by what values or types it does *not* depend upon—for example, "function `f` uses the fields of `struct queue`, but not the fields of `struct stack`". Concept analysis allows both positive and negative information to be incorporated into a modularization criterion. (See Section 3.2.)

- Unlike several previously proposed techniques, the concept-analysis approach offers the ability to "stay within the system" (as opposed to applying ad hoc methods) when the first suggested modularization is judged to be inadequate:
  - If the proposed modularization is on too fine a scale, the user can "move up" the partition lattice. (See Section 4.)
  - If the proposed modularization is too coarse, the user can add additional attributes to identify finer-granularity concepts. (See Section 3.)

- We demonstrate how concept analysis can be used to identify distinct modules amid "tangled" code. (See Section 3.2.)

- We define the notion of a *concept partition* and present an algorithm to generate partitions from a concept lattice. (See Section 4.)

- We have implemented a prototype tool that uses concept analysis to find potential modularizations of C programs. The implementation has been tested on several medium-to-large-sized examples. (See Section 5.)

As an example, consider the C implementation of stacks and queues shown in Figure 1. Queues are represented by two stacks, one for the front and one for the back; information is shifted from the front stack to the back stack when the back stack is empty. The queue functions only make use of the stack fields indirectly—by calling the stack functions. Although the stack and queue functions are written in an interleaved order, we would like to be able to tease the two components apart and make them separate classes, one a client of the other, as in the C++ code given in Figure 2.

This paper discusses a technique by which modules (in this case C++ classes) can be identified in legacy C code. The resulting information can then be

		attributes				
		four-legged	hair-covered	intelligent	marine	thumbed
objects	cats	$\surd$	$\surd$			
	dogs	$\surd$	$\surd$			
	dolphins			$\surd$	$\surd$	
	gibbons		$\surd$	$\surd$		$\surd$
	humans			$\surd$		$\surd$
	whales			$\surd$	$\surd$	

Table 1: A crude characterization of mammals.

supplied to a suitable transformation tool that maps C code to C++ code, as in the aforementioned example. Although other modularization algorithms are able to identify the same decomposition [3, 24], they are unable to handle a variant of this example in which stacks and queues are more tightly intertwined (see Section 3.2). In Section 3.2, we show that concept analysis *is* able to group the code from the latter example into separate queue and stack modules.

The remainder of this paper is structured as follows: Section 2 introduces contexts and concept analysis, and an algorithm for building concept lattices from contexts. Section 3 discusses a process for identifying modules in C programs based on concept analysis. Section 4 defines the notion of a concept partition and presents an algorithm for finding the partitions of a concept lattice. Section 5 discusses the implementation. Section 6 concerns related work.

2 A Concept-Analysis Primer

Concept analysis provides a way to identify sensible groupings of *objects* that have common *attributes* [23].

To illustrate concept analysis, we consider the example of a crude classification of a group of mammals: cats, gibbons, dogs, dolphins, humans, and whales. Suppose we consider five attributes: four-legged, hair-covered, intelligent, marine, and thumbed. Table 1 shows which animals are considered to have which attributes. We can interpret this data in a variety of ways. For example, we might observe that whales, dolphins, humans, and gibbons are all intelligent. On the other hand, gibbons, dogs, and cats are all hair-covered, but only the latter two are four-legged.

In order to understand the basics of concept analysis, a few definitions are required. A *context* is a triple $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$, where $\mathcal{O}$ and $\mathcal{A}$ are finite sets (the objects and attributes, respectively), and $\mathcal{R}$ is a binary relation between $\mathcal{O}$ and $\mathcal{A}$. In the mammal example, the objects are the different kinds of mammals, the attributes are the characteristics four-legged, hair-covered, etc. The binary relation $\mathcal{R}$ is given in Table 1. For example, the tuple (whales, marine) is in $\mathcal{R}$, but (cats, intelligent) is not.

Let $X \subseteq \mathcal{O}$ and $Y \subseteq \mathcal{A}$. The mappings $\sigma(X) = \{a \in \mathcal{A} | \forall o \in X : (o, a) \in \mathcal{R}\}$

```c
#define QUEUE_SIZE 10
struct stack { int *base, *sp, 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 isEmptyS(struct stack* s)
{ return (s->sp == s->base); }

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

void push(struct stack* s, int i)
{ *(s->sp) = i; s->sp++; }    /* no overflow check */

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

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

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

Figure 1: A queue using two stacks in C.

```
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) {
    // no overflow check
    *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 2: Queue and stack classes in C++.

(the *common attributes* of X) and $\tau(Y) = \{o \in \mathcal{O} | \forall a \in Y : (o, a) \in \mathcal{R}\}$ (the *common objects* of Y) form a *Galois connection*. That is, the mappings are *antimonotone*:

$$X_1 \subseteq X_2 \Rightarrow \sigma(X_2) \subseteq \sigma(X_1)$$
$$Y_1 \subseteq Y_2 \Rightarrow \tau(Y_2) \subseteq \tau(Y_1)$$

and *extensive*:

$$X \subseteq \tau(\sigma(X)) \quad \text{and} \quad Y \subseteq \sigma(\tau(Y)).$$

In the mammal example, $\sigma(\{\text{cats}, \text{gibbons}\}) = \{\text{hair-covered}\}$ and $\tau(\{\text{marine}\}) = \{\text{dolphins}, \text{whales}\}$.

A *concept* is a pair of sets—a set of objects (the *extent*) and a set of attributes (the *intent*) (X, Y)—such that $Y = \sigma(X)$ and $X = \tau(Y)$. That is, a concept is a maximal collection of objects sharing common attributes. In the example, ({cats, dogs}, {four-legged, hair-covered}) is a concept, whereas ({cats, gibbons}, {hair-covered}) is not a concept. A concept (X_0, Y_0) is a *subconcept* of concept (X_1, Y_1), denoted by $(X_0, Y_0) \leq (X_1, Y_1)$, if $X_0 \subseteq X_1$ (or, equivalently, $Y_1 \subseteq Y_0$). For instance, ({dolphins, whales}, {intelligent, marine}) is a subconcept of ({gibbons, dolphins, humans, whales}, {intelligent}). The subconcept relation forms a complete partial order (the *concept lattice*) over the set of concepts. The concept lattice for the mammal example is shown in Figure 3. Each node in the lattice represents a concept. A key indicating the extent and intent of each concept is shown in Table 2.

The fundamental theorem for concept lattices [23] relates subconcepts and superconcepts as follows:

$$\bigsqcup_{i \in I} (X_i, Y_i) = \left(\tau \left(\bigcap_{i \in I} Y_i \right), \bigcap_{i \in I} Y_i \right).$$

The significance of the theorem is that the least common superconcept of a set of concepts can be computed by intersecting their intents, and by finding the common objects of the resulting intersection. An example of the application of the fundamental theorem is shown in Figure 4.

There are several algorithms for computing the concept lattice for a given context [9, 21]. We describe a simple bottom-up algorithm here.

An important fact about concepts and contexts used in the algorithm is that, given a set of objects X, the smallest concept with extent containing X is $(\tau(\sigma(X)), \sigma(X))$. Thus, the bottom element of the concept lattice is $(\tau(\sigma(\emptyset)), \sigma(\emptyset))$—the concept consisting of all objects (often the empty set, as in our example) that have all the attributes in the context relation.

The initial step of the algorithm is to compute the bottom element of the concept lattice. The next step is to compute *atomic* concepts—smallest concepts with extent containing each of the objects treated as a singleton set[1]. Computation of the atomic concepts for the mammal example is shown in Figure 5.

[1]Atomic concepts should be distinguished from the *atoms* of a lattice—the nodes reachable

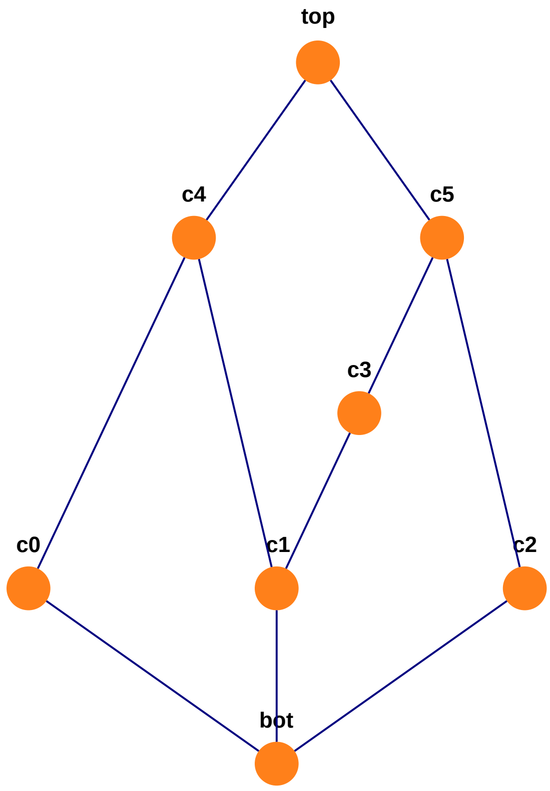

Figure 3: The concept lattice for the mammal example.

top	$(\{\text{cats}, \text{gibbons}, \text{dogs}, \text{dolphins}, \text{humans}, \text{whales}\}, \emptyset)$
c_5	$(\{\text{gibbons}, \text{dolphins}, \text{humans}, \text{whales}\}, \{\text{intelligent}\})$
c_4	$(\{\text{cats}, \text{gibbons}, \text{dogs}\}, \{\text{hair-covered}\})$
c_3	$(\{\text{gibbons}, \text{humans}\}, \{\text{intelligent}, \text{thumbed}\})$
c_2	$(\{\text{dolphins, whales}\}, \{\text{intelligent}, \text{marine}\})$
c_1	$(\{\text{gibbons}\}, \{\text{hair-covered}, \text{intelligent}, \text{thumbed}\})$
c_0	$(\{\text{cats}, \text{dogs}\}, \{\text{hair-covered}, \text{four-legged}\})$
bot	$(\emptyset, \{\text{four-legged}, \text{hair-covered}, \text{intelligent}, \text{marine}, \text{thumbed}\})$

Table 2: The extent and intent of the concepts for the mammal example.

$$
\begin{aligned}
c_1 \sqcup c_2 &= \\
(\{\text{gibbons}\}, \{\text{hair}, \text{intell.}, \text{thumbed}\}) &\sqcup (\{\text{dolphins}, \text{whales}\}, \{\text{intell.}, \text{marine}\}) \\
&= (\tau(\{\text{intelligent}\}), \{\text{intelligent}\}) \\
&= (\{\text{gibb.}, \text{humans}, \text{dolph.}, \text{whales}\}, \{\text{intell.}\}) \\
&= c_5
\end{aligned}
$$

Figure 4: An example use of the fundamental theorem of concept lattices: This computation corresponds to the fact that $c_1 \sqcup c_2 = c_5$ in the lattice shown in Figure 3.

$$
\begin{aligned}
\tau(\sigma(\{\text{cats}\})) &= \tau(\{\text{four-legged}, \text{hair-covered}\}) \\
&= \{\text{cats}, \text{dogs}\} \\
\tau(\sigma(\{\text{dogs}\})) &= \tau(\{\text{four-legged}, \text{hair-covered}\}) \\
&= \{\text{cats}, \text{dogs}\} \\
\tau(\sigma(\{\text{dolphins}\})) &= \tau(\{\text{intelligent}, \text{marine}\}) \\
&= \{\text{dolphins, whales}\} \\
\tau(\sigma(\{\text{gibbons}\})) &= \tau(\{\text{hair-covered}, \text{intelligent}, \text{thumbed}\}) \\
&= \{\text{gibbons}\} \\
\tau(\sigma(\{\text{humans}\})) &= \tau(\{\text{intelligent}, \text{thumbed}\}) \\
&= \{\text{humans, gibbons}\} \\
\tau(\sigma(\{\text{whales}\})) &= \tau(\{\text{intelligent}, \text{marine}\}) \\
&= \{\text{dolphins, whales}\}
\end{aligned}
$$

Figure 5: Computing atomic concepts in the mammal example.

The algorithm then closes the set of atomic concepts under join: Initially, a worklist is formed containing all pairs of atomic concepts (c', c) such that $c \not\leq c'$ and $c' \not\leq c$. While the worklist is not empty, remove an element of the worklist (c_0, c_1) and compute $c'' = c_0 \sqcup c_1$. If c'' is a concept that is yet to be discovered then add all pairs of concepts (c'', c) such that $c \not\leq c''$ and $c'' \not\leq c$ to the worklist. The process is repeated until the worklist is empty.

The iterative step of the concept-building algorithm is illustrated in Figure 6.

3 Using Concept Analysis to Identify Modules

The main idea of this paper is to apply concept analysis to the problem of identifying potential modules in legacy code. An outline of the process is as follows:

1. Build a context, where objects are functions defined in the input program and attributes are properties of those functions. The attributes could be any of several properties relating the functions to data structures. Attributes are discussed in more detail below.
2. Construct the concept lattice from the context, as described in Section 2.
3. Identify *concept partitions*—collections of concepts whose extents partition the set of objects. Each concept partition corresponds to a possible modularization of the input program. Concept partitions are discussed in Section 4.

3.1 Concept analysis and the stack and queue example

Consider the stack and queue example from the introduction. In this section, we will demonstrate how concept analysis can be used to identify the module partition indicated by the C++ code in Figure 2 on page 5.

First, we define a context as shown in Table 3. The next step is to build the concept lattice from the context, as described in Section 2. The concept lattice for the stack and queue example is shown in Figure 7. Table 4 shows the extent and intent of the concepts corresponding to the nodes in the lattice.

One of the advantages of using concept analysis is that multiple possibilities for modularization are offered. In addition, the relationships among concepts in the concept lattice also offers insight into the structure within proposed modules. For example, at the atomic level, initialization functions (concepts c_0 and c_1) are distinct concepts from other functions (concepts c_2 and c_3). The former two concepts correspond to constructors and the latter two to sets of member functions. Concept c_4 corresponds to a stack module and c_5 corresponds to a queue module. The subconcept relationships $c_0 \subseteq c_4$ and $c_2 \subseteq c_4$ indicate

from the bottom element in one step. In Section 4, we define the notion of a well-formed context—a context in which the atomic concepts correspond to the atoms of the concept lattice.

$$
\begin{array}{rcl}
c_0 & = & (\{\text{cats}, \text{dogs}\}, \{\text{hair-covered}, \text{four-legged}\}) \\
c_1 & = & (\{\text{gibbons}\}, \{\text{hair-covered}, \text{intelligent}, \text{thumbed}\}) \\
c_2 & = & (\{\text{dolphins}, \text{ whales}\}, \{\text{intelligent}, \text{marine}\}) \\
c_3 & = & (\{\text{gibbons}, \text{humans}\}, \{\text{intelligent}, \text{thumbed}\}) \\
Worklist & = & [(c_0, c_1), (c_0, c_2), (c_0, c_3), (c_1, c_2), (c_2, c_3)] \\
c_4 & = & c_0 \sqcup c_1 = (\{\text{cats}, \text{gibbons}, \text{dogs}\}, \{\text{hair-covered}\}) \\
Worklist & = & [(c_0, c_2), (c_0, c_3), (c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_0 \sqcup c_2 & = & \top = (\{\text{cats}, \text{gibbons}, \text{dogs}, \text{dolphins}, \text{humans}, \text{whales}\}, \emptyset) \\
Worklist & = & [(c_0, c_3), (c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_0 \sqcup c_3 & = & \top \\
Worklist & = & [(c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_5 & = & c_1 \sqcup c_2 = (\{\text{gibbons}, \text{dolphins}, \text{humans}, \text{whales}\}, \{\text{intelligent}\}) \\
Worklist & = & [(c_2, c_3), (c_2, c_4), (c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_2 \sqcup c_3 & = & c_5 \\
Worklist & = & [(c_2, c_4), (c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_2 \sqcup c_4 & = & \top \\
Worklist & = & [(c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_3 \sqcup c_4 & = & \top \\
Worklist & = & [(c_0, c_5), (c_4, c_5)] \\
c_0 \sqcup c_5 & = & \top \\
Worklist & = & [(c_4, c_5)] \\
c_4 \sqcup c_5 & = & \top \\
Worklist & = & \emptyset
\end{array}
$$

Figure 6: Bottom-up computation of concepts for the mammals example.

	returns `stack`	*returns* `queue`	*has* `stack` *arg.*	*has* `queue` *arg.*	*uses* `stack` *fields*	*uses* `queue` *fields*
`initStack`	$\surd$				$\surd$	
`initQ`		$\surd$				$\surd$
`isEmptyS`			$\surd$		$\surd$	
`isEmptyQ`				$\surd$		$\surd$
`push`			$\surd$		$\surd$	
`enq`				$\surd$		$\surd$
`pop`			$\surd$		$\surd$	
`deq`				$\surd$		$\surd$

Table 3: The context for the stack and queue example.

top	(*all objects*, $\emptyset$)
c_5	({`initQ`, `isEmptyQ`, `enq`, `deq`}, {uses `queue` fields})
c_4	({`initStack`, `isEmptyS`, `push`, `pop`}, {uses `stack` fields})
c_3	({`isEmptyQ`, `enq`, `deq`}, {has `queue` argument, uses `queue` fields})
c_2	({`isEmptyS`, `push`, `pop`}, {has `stack` argument, uses `stack fields`})
c_1	({`initQ`}, {returns `queue`})
c_0	({`initStack`}, {returns `stack`})
bot	($\emptyset$, *all attributes*)

Table 4: The extent and intent of the concepts for the stack/queue example.

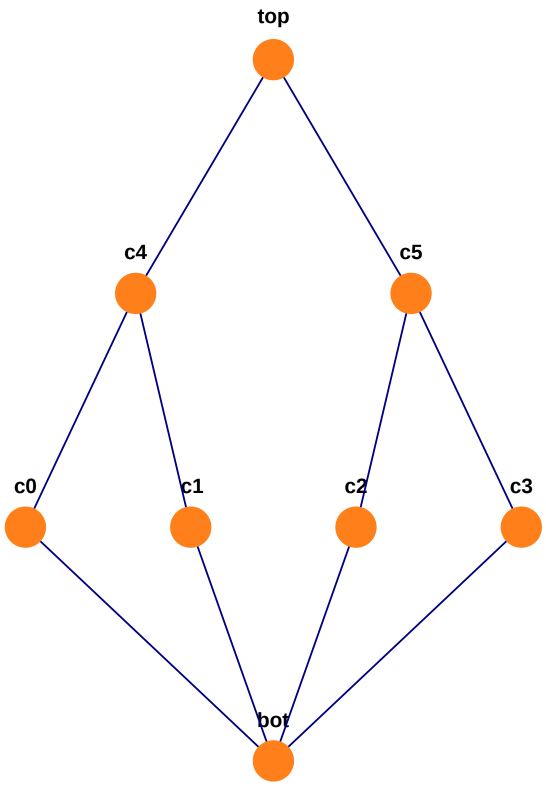

Figure 7: The concept lattice for the stack and queue example.

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

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

Figure 8: "Tangled" `isEmptyQ` and `enq` functions.

that the stack concept consists of a constructor concept and a member-function concept.

3.2 Adding complementary attributes

The stack and queue example, as considered thus far, has not demonstrated the full power that concept analysis brings to the modularization problem. It is relatively straightforward to separate the code shown in Figure 1 into two modules, and techniques such as those described in [3, 24] will also create the same grouping. We now show that concept analysis offers the possibility to go beyond previously defined methods: It offers the ability to tease apart code that is, in some sense, more "tangled".

To illustrate what we mean by more tangled code, consider a slightly modified stack and queue example. Suppose the functions `isEmptyQ` and `enq` have been written so that they modify the stack fields directly, as in Figure 8, rather than calling `isEmptyS` and `push`. While this may be more efficient, it makes the code more difficult to maintain—simple changes in the stack implementation may require changes in the queue code. Furthermore, it complicates the process of identifying separate modules. If we apply concept analysis using the same set of attributes as we did above, attribute "uses `stack` fields" now applies to `isEmptyQ` and `enq`. Table 5 shows the context relation for the tangled stack and queue code with the original sets of objects and attributes. The resulting concept lattice is shown in Figure 9. (The extent and intent of the concepts corresponding to the nodes in the lattice are shown in Table 6.) Observe that concept c_5 can still be identified with a queue module, but none of the concepts coincide with a stack module. In particular, even though the extent of c_0 is $\{\texttt{initStack}\}$ and the extent of c_2 is $\{\texttt{isEmptyS}, \texttt{push}, \texttt{pop}\}$, the concept $c_0 \sqcup c_2 = c_7$ is not the stack concept:

$$\tau(\sigma(\{\texttt{initStack}, \texttt{isEmptyS}, \texttt{push}, \texttt{pop}\})) =$$

$$\{\texttt{initStack}, \texttt{isEmptyS}, \texttt{isEmptyQ}, \texttt{push}, \texttt{enq}, \texttt{pop}\}$$

In other words, the extent of c_7 mixes the stack operations with some, but not all, of the queue operations.

	returns `stack`	*returns* `queue`	*has* `stack` *arg.*	*has* `queue` *arg.*	*uses* `stack` *fields*	*uses* `queue` *fields*
`initStack`	$\surd$				$\surd$	
`initQ`		$\surd$				$\surd$
`isEmptyS`			$\surd$		$\surd$	
`isEmptyQ`				$\surd$	$\surd$	$\surd$
`push`			$\surd$		$\surd$	
`enq`				$\surd$	$\surd$	$\surd$
`pop`			$\surd$		$\surd$	
`deq`				$\surd$		$\surd$

Table 5: The context for the "tangled" stack and queue example.

top	(*all objects*, $\emptyset$)
c_7	({`initStack`, `isEmptyS`, `isEmptyQ`, `push`, `enq`, `pop`}, {uses `stack` fields})
c_5	({`initQ`, `isEmptyQ`, `enq`, `deq`}, {uses `queue` fields})
c_3	({`isEmptyQ`, `enq`, `deq`}, {has `queue` argument, uses `queue` fields})
c_6	({`isEmptyQ`, `enq`}, {has `queue` argument, uses `queue` fields, uses `stack` fields})
c_2	({`isEmptyS`, `push`, `pop`}, {has `stack` argument, uses `stack fields`})
c_1	({`initQ`}, {returns `queue`})
c_0	({`initStack`}, {returns `stack`})
bot	($\emptyset$, *all attributes*)

Table 6: The extent and intent of the concepts for the tangled stack/queue example.

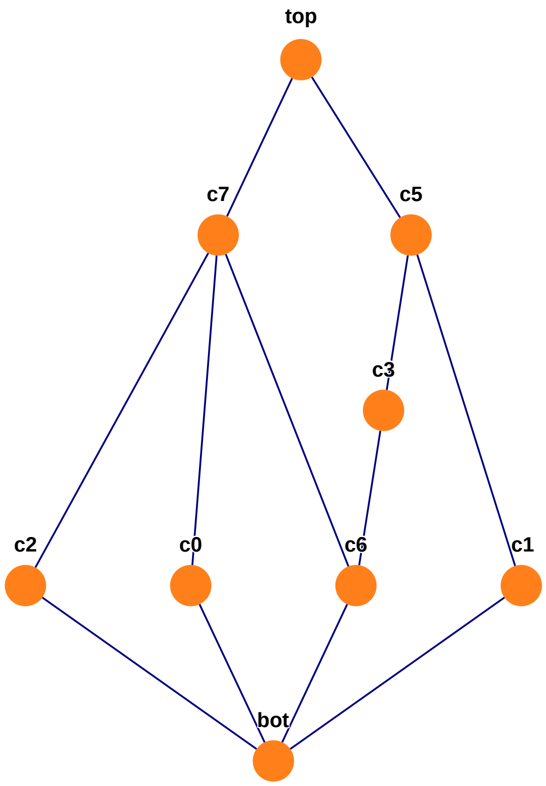

Figure 9: The concept lattice for the tangled stack and queue example.

	returns `stack`	*returns* `queue`	*has* `stack` *arg.*	*has* `queue` *arg.*	*uses* `stack` *fields*	*uses* `queue` *fields*	*not* `queue` *fields*
`initStack`	√				√		√
`initQ`		√				√	
`isEmptyS`			√		√		√
`isEmptyQ`				√	√	√	
`push`			√		√		√
`enq`				√	√	√	
`pop`			√		√		√
`deq`				√		√	

Table 7: The context for the "untangled" stack and queue example.

The problem is that the attributes in this context reflect only "positive" information. A distinguishing characteristic of the stack operations is that they depend on the fields of `struct stack` but *not* on the fields of `struct queue`. To "untangle" these components, we need to augment the set of attributes with "negative" information—in this case, we add a new attribute—the complement of "uses `queue` fields" (i.e., "does *not* use `queue` fields"). The corresponding context is shown in Table 7. The resulting concept lattice is shown in Figure 10. (The extent and intent of the concepts corresponding to the nodes in the lattice are shown in Table 8.) This concept lattice contains all of the concepts in the concept lattice from Figure 9, as well as an additional concept, c_4, which corresponds to a stack module. This modularization identifies `isEmptyQ` and `enq` as being part of a queue module that is separate from a stack module, even though these two operations make direct use of stack fields.

This example raises some issues for the subsequent C-to-C++ code-transformation phase. Although one might be able to devise transformations to remove these dependences of queue operations on the private members of the stack class (e.g., by introducing appropriate calls on member functions of the stack class), a more straightforward C-to-C++ transformation would simply use the C++ friend mechanism, as shown in Figure 11.

3.3 Other choices for attributes

A concept is a maximal collection of objects having common properties. A cohesive module is a collection of functions (perhaps along with a data structure) having common properties. Therefore, when employing concept analysis to the modularization problem, it is reasonable to have objects correspond to func-

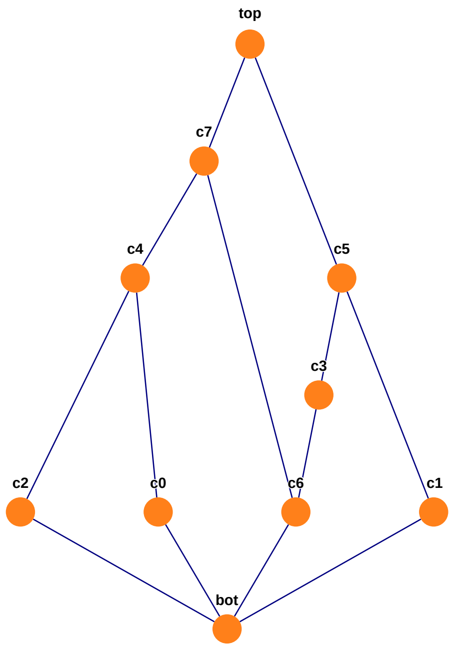

Figure 10: The concept lattice for the untangled stack and queue example.

top	(*all objects*, $\emptyset$)
c_7	({`initStack`, `isEmptyS`, `isEmptyQ`, `push`, `enq`, `pop`}, {uses `stack` fields})
c_5	({`initQ`, `isEmptyQ`, `enq`, `deq`}, {uses `queue` fields})
c_4	({`initStack`, `isEmptyS`, `push`, `pop`}, {uses `stack` fields, *not* `queue` fields})
c_3	({`isEmptyQ`, `enq`, `deq`}, {has `queue` argument, uses `queue` fields})
c_6	({`isEmptyQ`, `enq`}, {has `queue` arg., uses `queue` fields, uses `stack` fields})
c_2	({`isEmptyS`, `push`, `pop`}, {has `stack` arg., uses `stack` fields, *not* `queue` fields})
c_1	({`initQ`}, {returns `queue`})
c_0	({`initStack`}, {returns `stack`, does *not* use `queue` fields})
bot	($\emptyset$, *all attributes*)

Table 8: The extent and intent of the concepts for the untangled stack/queue example.

```
const int QUEUE_SIZE = 10

class queue;

class stack {
  friend class queue;
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++; } // no overflow check
};

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

Figure 11: Queue and stack classes in C++ with friends.

tions.[2] However, we have more flexibility when it comes to attributes. There are a wide variety of attributes we might choose in an effort to identify concepts (modules) in a program. Our examples have used attributes that reflect the way `struct` data types are used. But in some instances, it may be useful to use attributes that capture other properties. Other possibilities for attributes include the following:

- *Variable-usage information*: Related functions can sometimes be identified by their use of common global variables. An attribute capturing this information might be of the form "uses global variable `x`" [13, 18].
- *Dataflow and slicing information* can be useful in identifying modules. Attributes capturing this information might be of the form "may use a value that flows from statement s" or "is part of the slice with respect to statement s".
- *Information obtained from type inferencing*: Type inference can be used to uncover distinctions between seemingly identical types (see [19, 17]). For example, if f is a function declared to be of type $\texttt{int} \times \texttt{int} \rightarrow \texttt{bool}$, type inference might discover that f's most general type is of the form $\alpha \times \beta \rightarrow \texttt{bool}$. This reveals that the type of f's first argument is distinct from the type of its second argument (even though they had the same declared type). Attributes might then be of the form "has argument of type α" rather than simply "has argument of type `int`". This would prevent functions from being grouped together merely because of superficial similarities in the declared types of their arguments.
- *Disjunctions of attributes*: The user may be aware of certain properties of the input program, perhaps the similarity of two data structures. Disjunctive attributes allow the user to specify properties of the form "π_1 or π_2". For example, "uses fields of stack or uses fields of queue".

Any or all of these attributes could be used together in one context. This highlights one of the advantages of the concept-analysis approach to modularization: It represents not just a single *algorithm* for modularization; rather, it provides a *framework* for obtaining a collection of different modularization algorithms.

4 Concept Partitions

Thus far, we have discussed how a concept lattice can be built from a program in such a way that concepts represent potential modules. However, because of overlaps between concepts, not every group of concepts represents a potential modularization.

For the above examples involving stacks and queues, it is straightforward to look at the small collection of concepts and find the desired modules by hand.

[2] Some legacy code is monolithic—multiple tasks are contained within one function. In such cases, it may be preferable to have objects correspond to slices [22, 10] rather than functions.

In practice, programs to be modularized will be much larger. Modularization is unlikely to ever be a fully automated process, but it would be helpful to have a tool that suggests a palatable number of potential modularizations from which a software engineer might reasonably choose. Not all combinations of potential modules work together. Some concepts *overlap*. That is, a function may appear in multiple concepts, but it should only reside in one module.

Feasible modularizations are partitions: collections of modules that are disjoint, but include all the functions in the input code. To limit the number of choices that a software engineer would be presented with, it is helpful to identify such partitions.

Below, the notion of a *concept partition* is made more formal and an algorithm to identify such partitions from a concept lattice is presented.

4.1 Concept Partitions

Given a context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$, a *concept partition* is a set of concepts whose extents form a partition of $\mathcal{O}$. That is, $P = \{(X_0, Y_0), \ldots, (X_{k-1}, Y_{k-1})\}$ is a concept partition iff the extents of the concepts *cover* the object set (i.e. $\bigcup X_i = \mathcal{O}$) and are pairwise disjoint ($X_i \cap X_j = \emptyset$ for $i \neq j$ and $X_i, X_j \in P$). In terms of modularizing a program, a concept partition corresponds to a collection of modules such that every function in the program is associated with exactly one module.

As a simple example, consider the concept lattice shown in Figure 10 on page 17. The concept partitions for that context are listed below:

P_1	$\{c_0, c_1, c_2, c_3\}$
P_2	$\{c_0, c_2, c_5\}$
P_3	$\{c_1, c_3, c_4\}$
P_4	$\{c_4, c_5\}$
P_5	$\{\text{top}\}$

P_1 is the *atomic partition*. P_2 and P_3 are combinations of atomic concepts and larger concepts. P_4 consists of one stack module and one queue module. P_5 is the trivial partition: All functions are placed in one module.

By looking at concept partitions, the software engineer can eliminate nonsensical possibilities. In the preceding example, c_7 does not appear in any partition—if it did, then to what module (i.e., nonoverlapping concept) would `deq` belong?

An *atomic partition* of a concept lattice is a concept partition consisting of exactly the atomic concepts. (Recall that the atomic concepts are the concepts with smallest extent containing each of the objects treated as a singleton set. For instance, see the atomic concepts in the mammal example on page 8 in Section 2.) A concept lattice need not have an atomic partition. For example, the lattice in Figure 3 on page 7 does not have an atomic partition: The atomic concepts are c_0, c_1, c_2, and c_3; however, c_1 and c_3 overlap—the object "gibbons" is in the extent of both concepts.

When it exists, the atomic partition of a concept lattice is often a good starting point for choosing a modularization of a program. In order to develop tools to work with concept partitions, it is useful to be able to guarantee the existence of atomic partitions. Contexts that result in atomic concepts that, in turn, form a concept partition can be characterized precisely by the following definition: a context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ is *well-formed* if and only if, for every pair of elements $x, y \in \mathcal{O}$, $\sigma(\{x\}) \subseteq \sigma(\{y\})$ implies $\sigma(\{x\}) = \sigma(\{y\})$.

While not every context results in a concept lattice that has an atomic partition, we can *extend* any context—by adding additional attributes—to make it well-formed. Informally, a context extension is another context over the same set of objects (but a possibly augmented set of attributes) whose concept lattice offers as least as many ways of grouping the objects as did the lattice derivable from the original context. More formally, a context $(\mathcal{O}, \mathcal{A}', \mathcal{R}')$ is an *extension* of context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ if and only if $\mathcal{A} \subseteq \mathcal{A}'$ and $\mathcal{R} \subseteq \mathcal{R}'$.

There are several ways in which a non-well-formed context can be extended into a well-formed context. The important step in any such process is to identify the 'offending' pairs of objects x and y for which $\sigma(\{x\}) \subsetneq \sigma(\{y\})$. This inequity may be counterbalanced by the addition of an attribute such that, in the resulting context, $\sigma(\{x\}) \not\subseteq \sigma(\{y\})$. Two such ways to extend a context to well-formed context are described below:

1. A context can be extended via the addition of *unique identification attributes*: for each pair of objects, x, y such that $\sigma(\{x\}) \subsetneq \sigma(\{y\})$, a new attribute a_x that uniquely identifies x is added to the extended attribute set. x becomes the *only* object that has attribute a_x in the extended context relation (i.e., $\tau(\{a_x\}) = \{x\}$).

   As an example, consider the mammal context shown in Table 1 on page 3. The context is not well-formed because the attributes of "human" are a proper subset of the attributes of "gibbon". To make a *uniquely attributed* extension, we augment the attribute set to include the attribute a_{human}. The resulting context is shown in Table 9. The resulting concept lattice shown in Figure 12. Table 10 shows the extent and intent corresponding to the nodes in the lattice. Table 11 shows the partitions of the lattice. Partition P_1 is the atomic partition.

2. A context can be extended to a well-formed context by augmenting a context with negative information (similar to what is done in Section 3.2).

   Given a context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$, a *complement* of an attribute $a \in \mathcal{A}$ is an attribute $\overline{a}$ such that $\tau(\{\overline{a}\}) = \{x \in \mathcal{O} | (x, a) \notin \mathcal{R}\}$. That is, $\overline{a}$ is an attribute of exactly the objects that do not have property a. For example, the attribute "does *not* use `queue` fields" from the tangled stack and queue example (page 16) is the complement of the attribute "uses `queue` fields".

   The *complemented extension* of a context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ is a new context $(\mathcal{O}, \mathcal{A}', \mathcal{R}')$ formed by the algorithm in Figure 14.

   For example, we can form a different well-formed extension of the mammal context shown in Table 1 (page 3) by creating the complemented extension.

	four-legged	hair-covered	intelligent	marine	thumbed	a_{human}
cats	$\surd$	$\surd$				
dogs	$\surd$	$\surd$				
dolphins			$\surd$	$\surd$		
gibbons		$\surd$	$\surd$		$\surd$	
humans			$\surd$		$\surd$	$\surd$
whales			$\surd$	$\surd$		

Table 9: The uniquely-attributed extension of the mammal context shown in Table 1 on page 3.

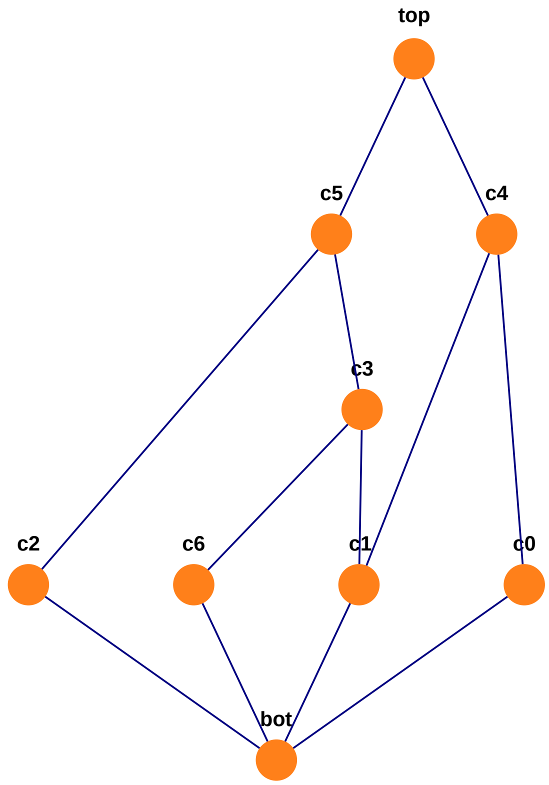

Figure 12: The concept lattice for the uniquely-attributed mammal example.

To make the complemented extension, we augment the attribute set to include the complementary attribute "not hair-covered". The resulting context is shown in Table 12. The resulting concept lattice shown in Figure 13. Table 13 shows the extent and intent corresponding to the nodes in the lattice. Table 14 shows the partitions of the lattice. Partition P_1 is the atomic partition.

It should be clear that both forms of extension result in well-formed contexts.

Both uniquely-attributed extensions and complemented extensions result in a concept lattice with at least as many (and frequently many more) nodes than the lattice derived from the original context. We say that a concept lattice $\mathcal{L}'$ derived from a $\mathcal{C}' = (\mathcal{O}, \mathcal{A}', \mathcal{R}')$ is an *extension* of a concept lattice $\mathcal{L}$ derived from a $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$ if $\mathcal{A} \subseteq \mathcal{A}'$, and for every concept c in $\mathcal{L}$, there is a concept c' in $\mathcal{L}'$ with the same extent. More formally, if $X \subseteq \mathcal{O}$ such that $\tau(\sigma(X)) = X$, then $\tau'(\sigma'(X)) = X$ where τ' and σ' are the common-object and common-attribute relations, respectively, for context $\mathcal{C}'$.

Given a context $\mathcal{C}$, both uniquely-attributed extensions and complemented extensions of $\mathcal{C}$ result in concept lattices that are extensions of the lattice derived from $\mathcal{C}$. In both cases, attributes are added to the extended context to make in easier to distinguish objects. For example, if $\sigma\{x\} \subsetneq \sigma\{y\}$ then there is at least one attribute which is a property of y that is not a property of x. Whether adding unique attributes or complementary attributes, negative information is represented in a positive form in the extended context to help distinguish such x and y.

4.2 Finding Partitions from A Concept Lattice

We say that concept lattice derived from a well-formed context is a well-formed concept lattice. Given a well-formed concept lattice, we define the following relations on its elements:

A concept d *covers* concept c if $c < d$ and there is no concept e such that $c < e < d$. If d covers c, we say "c is covered by d". The set of covers of concept c, denoted by $\mathrm{covs}(c)$, is the set of lattice elements d such that d covers c. The set of elements *subordinate* to d, denoted by $\mathrm{subs}(c)$, is the set of lattice elements c such that $c < d$.

The algorithm builds up a collection of all the partitions of a concept lattice. Let P be the collection of partitions that we are forming. Let W be a worklist of partitions. We begin with the atomic partition, which is the covers of the bottom element of the concept lattice. P and W are both initialized to the singleton set containing the atomic partition.

The algorithm works by considering partitions from worklist W until W is empty. For each partition removed from W, new partitions are formed (when possible) by selecting a concept of the partition, choosing a cover of that concept, adding it to the partition, and removing overlapping concepts. The algorithm is given in Figure 15.

top	$(\{\text{cats}, \text{gibbons}, \text{dogs}, \text{dolphins}, \text{humans}, \text{whales}\}, \emptyset)$
c_5	$(\{\text{gibbons}, \text{dolphins}, \text{humans}, \text{whales}\}, \{\text{intelligent}\})$
c_4	$(\{\text{cats}, \text{dogs}, \text{gibbons}\}, \{\text{hair-covered}\})$
c_3	$(\{\text{gibbons, humans}\}, \{\text{intelligent}, \text{thumbed}\})$
c_2	$(\{\text{dolphins, whales}\}, \{\text{intelligent}, \text{marine}\})$
c_6	$(\{\text{humans}\}, \{\text{intelligent}, \text{thumbed}, a_{\text{human}}\})$
c_1	$(\{\text{gibbons}\}, \{\text{hair-covered}, \text{intelligent}, \text{thumbed}\})$
c_0	$(\{\text{cats}, \text{dogs}\}, \{\text{hair-covered}, \text{four-legged}\})$
bot	$(\emptyset, \{\text{four-legged}, \text{hair-covered}, \text{intelligent}, \text{marine}, \text{thumbed}, a_{\text{human}}\})$

Table 10: The extent and intent of the concepts for the uniquely-attributed mammal example.

P_1	$\{c_0, c_1, c_2, c_6\}$
P_2	$\{c_2, c_6, c_4\}$
P_3	$\{c_0, c_2, c_3\}$
P_4	$\{c_0, c_5\}$
P_5	$\{\text{top}\}$

Table 11: Concept partitions corresponding to the concept lattice in Figure 12. P_1 is the atomic partition.

	four-legged	hair-covered	intelligent	marine	thumbed	not hair-covered
cats	$\surd$	$\surd$				
dogs	$\surd$	$\surd$				
dolphins			$\surd$	$\surd$		$\surd$
gibbons		$\surd$	$\surd$		$\surd$	
humans			$\surd$		$\surd$	$\surd$
whales			$\surd$	$\surd$		$\surd$

Table 12: The complemented extension of the mammal context shown in Table 1 on page 3.

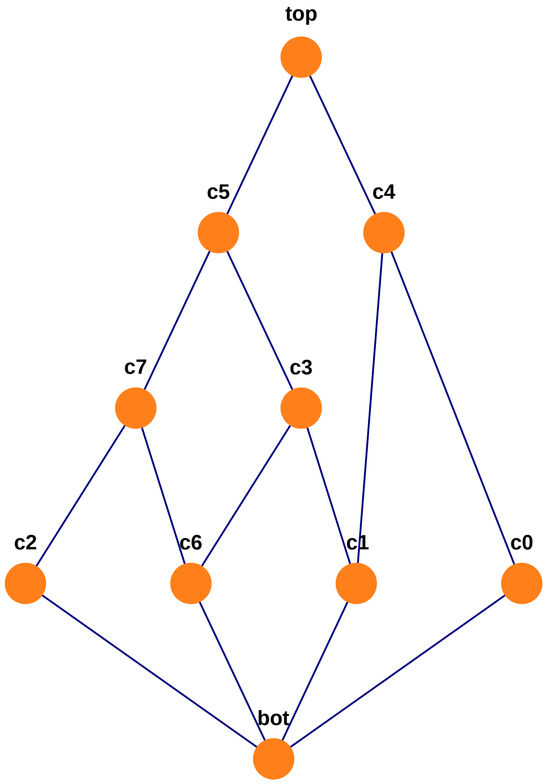

Figure 13: The concept lattice for the complemented mammal example.

top	$(\{\text{cats}, \text{gibbons}, \text{dogs}, \text{dolphins}, \text{humans}, \text{whales}\}, \emptyset)$
c_5	$(\{\text{gibbons}, \text{dolphins}, \text{humans}, \text{whales}\}, \{\text{intelligent}\})$
c_4	$(\{\text{cats}, \text{dogs}, \text{gibbons}\}, \{\text{hair-covered}\})$
c_7	$(\{\text{dolphins, humans, whales}\}, \{\text{not hair-covered}\})$
c_3	$(\{\text{gibbons, humans}, \{\text{intelligent}, \text{thumbed}\})$
c_2	$(\{\text{dolphins, whales}\}, \{\text{intelligent}, \text{marine}, \text{not hair-covered}\})$
c_6	$(\{\text{humans}\}, \{\text{intelligent}, \text{thumbed}, \text{not hair-covered}\})$
c_1	$(\{\text{gibbons}\}, \{\text{hair-covered}, \text{intelligent}, \text{thumbed}\})$
c_0	$(\{\text{cats}, \text{dogs}\}, \{\text{hair-covered}, \text{four-legged}\})$
bot	$(\emptyset, \{\text{four-legged}, \text{hair-covered}, \text{intelligent}, \text{marine}, \text{thumbed}, \text{not hair-covered}\})$

Table 13: The extent and intent of the concepts for the complemented mammal example.

P_1	$\{c_0, c_1, c_2, c_6\}$
P_2	$\{c_2, c_6, c_4\}$
P_3	$\{c_0, c_1, c_7\}$
P_4	$\{c_0, c_2, c_3\}$
P_5	$\{c_7, c_4\}$
P_6	$\{\text{top}\}$

Table 14: Concept partitions corresponding to the concept lattice in Figure 13. P_1 is the atomic partition.

$\mathcal{A}' \leftarrow \mathcal{A}$
$\mathcal{R}' \leftarrow \mathcal{R}$
`while` $(\mathcal{O}, \mathcal{A}', \mathcal{R}')$ `is not well formed do`
  `let` $x, y \in \mathcal{O}$ `be such that` $\sigma(\{x\}) \subsetneq \sigma(\{y\})$
  `let` $a \in \mathcal{A}'$ `be such that` $a \notin \sigma(\{x\})$`,` $a \in \sigma(\{y\})$
  $\mathcal{A}' \leftarrow \mathcal{A}' \cup \{\overline{a}\}$`, where` $\overline{a}$ `is a new attribute`
  $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{(x, \overline{a}) | (x, a) \notin \mathcal{R}'\}$
`endwhile`

Figure 14: An algorithm to compute the complemented extension of a context.

As an example, consider the atomic partition of the concept lattice derived from the uniquely-attributed mammal context (see Figure 12). The algorithm begins with the atomic partition (consisting of concepts, c_0, c_1, c_2, and c_6) as the sole member of the worklist. The algorithm removes the atomic partition from the worklist, as p in line [5] of Figure 15. Suppose that in the first iteration of the for loop in line [6], c refers to c_0. The covering set of c_0 is the singleton set consisting of c_4, so c' is assigned c_4 in line [7]. In line [8], p' is assigned the value of p minus the subordinate concepts of c_4 (i.e., c_1, c_0, and bottom), so p' is $\{c_2, c_6\}$. In line [9], the union of the extents of c_2 and c_6 is disjoint with the extent of c_4; thus, in line [10], the partition $p'' = \{c_2, c_6\} \cup \{c_4\}$ is formed. p'' is added to the set of partitions and to the worklist in line [12] and line [13].

In the worst case, the number of partitions can be exponential in the number of concepts. Furthermore, the techniques for making contexts well-formed, discussed in Section 4.1, only exacerbate the problem: More precise means of distinguishing sets of objects translates to more concepts. This is turns leads to more possibilities for partitions.

If the number of concepts in a concept lattice is large, it may be impractical to consider every possible partition of the concepts. In such a case, it is possible to adapt the algorithm to work interactively, with guidance from the user. Before attempting to find a new partition, the algorithm would pause for the user to specify *seed sets* of concepts, which would be used to force the algorithm to find only coarser partitions than the seed sets (i.e., partitions that do not

```
A ← covs(⊥)                          // the atomic partition
P ← {A}
W ← {A}
while W ≠ ∅ do
  remove some p from W
  for each c ∈ p
    for each c′ ∈ covs(c)
      p′ ← p − subs(c′)
      if (⋃p′) ∩ c′ = ∅              // if p′ and c′ are disjoint
        p″ ← p′ ∪ {c′}
        if p″ ∉ P
          P ← P ∪ {p″}
          W ← W ∪ {p″}
        endif
      endif
    endfor
  endfor
endwhile
```

Figure 15: An algorithm to find the partitions of a well-formed concept lattice.

subdivide the seed sets).

A proof of the correctness of the algorithm can be found in Appendix A.

5 Implementation and Results

We have implemented a prototype tool that employs concept analysis to discover potential modularizations of C programs. Our implementation is actually a collection of tools, working together as depicted in Figure 16. As the diagram indicates, the tool consists of five components:

1. The AST builder ("FrontEndC") takes a compilable, preprocessed C program and generates an abstract syntax tree that is annotated with type information.

2. The context builder takes as input a typed abstract syntax tree and generates formal contexts by traversing the tree. This component consists of a collection of functions allowing for contexts to be constructed based on a wide assortment of objects and attributes, including, but not limited to, the following criteria:

   - Objects are functions, attributes are types. (f, t) is in the context relation if and only if an expression of type t occurs in function f.

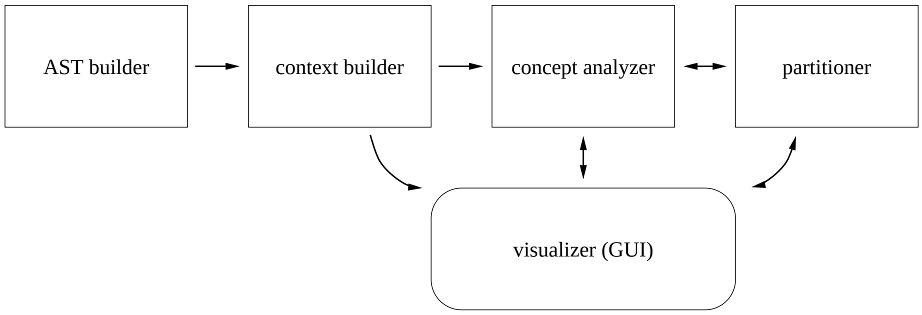

Figure 16: A C concept-analysis tool: the different components of our implementation.

- Objects are functions, attributes are aggregate types (i.e., `struct` or `union` types). (f, t) is in the context relation if and only if a field selector (i.e., `.` or `->`) is applied to an expression of type t in f. (This is the kind of context used to produce the data in Table 15 on page 30.)
- Objects are functions, attributes are aggregate types. (f, t) is in the context relation if and only if f has a parameter type or return type that uses t. (This is the kind of context used to produce the data in Table 16 on page 30.)
- Objects are functions, attributes are global variables. (f, v) is in the context relation if and only if f uses v.
- Objects are labels of case statements. The attributes are functions. (c, f) is in the context relation if and only if a statement of the form "`case` c" occurs in function f. Labels are only counted as objects when they are textual labels (i.e. enumerated values)—integer and character constants are ignored, as is the `default` label.

The context builder also includes the ability to merge a collection of contexts into one context. This feature is quite useful for generating one context from many input C files. The context merger assumes that the attributes and objects across files are represented in a consistent fashion. For example, if an attribute represented by the string `"Stack"` is present in more than one context, it is assumed to refer to the same property (whether it means "uses field of type `Stack`", "returns type `Stack`, etc.) in each context in which it appears.

3. The concept analyzer takes as input a formal context and builds the concept lattice bottom up, as described in Section 2. The transitively reduced graph representation of the lattice is then computed. The concept analyzer also provides functions to compute common-attribute and common-object sets.

4. The partitioner takes as input a concept lattice and generates the set of all partitions of that lattice, as described in Section 4. In order to construct partitions, the concept lattice must be derived from a well-formed context. The partitioner offers two ways to transform an arbitrary context into a well-formed context: by adding unique attributes or by computing the complemented extension. Since the number of partitions can be very large, the partitioner also provides the ability to compute partitions incrementally.

5. The visualizer has two main functions: to display context and concept lattice information in an interconnected fashion and to provide a graphical user interface through which the user can manipulate the other components of the C concept-analysis system.

The entire system is written in Standard ML except for the visualization component, which is written in Standard ML in conjunction with the SmlTK interface to Tcl/Tk.

The simple examples mentioned thus far in this paper have been analyzed using our implementation. We have also applied the prototype tool on the integer portion of the SPEC 95 benchmark suite, as well as several other programs of comparable size.

As an example, consider the SPEC 95 benchmark *go* ("The Many Faces of Go"). The program consists of roughly 28,000 lines of C code, 372 functions, and 8 user-defined aggregate types. The concept lattice for the fully complemented context (i.e., the context including all the complements of the original attributes) associated with these functions and data types consists of thirty-four concepts and was constructed in less than thirty seconds of user time (using Standard ML of New Jersey version 110 on a SPARCstation 20 with 256MB of RAM running Solaris 2.5.1). The partitioner identified 63 possible partitions of the lattice in roughly the same amount of time.

Table 15 summarizes results from contexts where objects are functions, attributes are aggregate types (`struct` or `union`), and (f, t) is in the context relation if and only if a field selector (i.e., `.` or `->`) is applied to an expression of type t in f. The programs listed are a variety of programs that make significant use of `struct` types, but do not necessarily take the `struct` types as arguments to the functions. (For example, in *go*, the variables of `struct` types are all global.) The programs include *go*, *chull* (computes the convex hull of a set of points), *bdd* (manipulates and manages binary decision diagrams) and some switching-configuration control code from Lucent Technologies.

Table 16 summarizes results from contexts where objects are functions, attributes are aggregate types, and (f, t) is in the context relation if and only if f has a parameter type or return type that uses t. In this case, a type t' 'uses' type t, if one of the following cases holds:

- $t' = t$
- t' is equivalent to t via a sequence of `typedef` associations

program	KLOC	objects	attributes	concepts
chull	1.0	26	3	9
bdd	2.5	51	13	20
go	28.0	372	16	34
Lucent code	160.0	45	44	95

Table 15: Results from applying concept analysis to various C programs. Objects are functions and attributes are of the form "uses the fields of `struct t`".

program	KLOC	objects	attributes	concepts
li	7.6	299	3	7
m88ksim	19.9	24	14	14
perl	26.9	189	17	42
ijpeg	31.2	407	29	48
vortex	67.2	654	48	93
bash	73.6	266	32	50
gcc	205.1	1,358	38	61

Table 16: Results from applying concept analysis to C programs from the SPEC benchmark (and *bash*). Objects are functions and attributes are of the form "uses `struct t` in parameter list or return type".

- t' is a pointer to a type that 'uses' t.

The programs listed are those from the integer portion of the SPEC 95 benchmark that make moderate to heavy use of aggregate types by passing them into and returning them from functions, as well as *bash*, a Unix shell, which also makes heavy use of `struct` types. The data indicates that from a *quantitative* point of view, concept analysis is practical. In the worst case, the number of concepts can be exponential in the number of objects; however, in the cases we have studied, there are actually fewer concepts than objects.

The data shown in Table 15 and Table 16 is generated from contexts that are not well-formed, so the partition algorithm cannot be applied to the concept lattices. Instead, we use the two techniques for extending contexts to well-formed contexts described in Section 4. Table 17 summarizes the results of applying concept analysis to the well-formed extensions of the contexts used in Table 16, where the extensions are produced by adding unique identification attributes. Table 18 summarizes the results of applying concept analysis to the complemented extensions of the contexts used in Table 16.

The data from the well-formed extensions indicate that it is more tractable to form concept lattices (and to generate partitions from the lattices) from well-formed contexts made by adding unique identification attributes than it is from complemented extensions. The reason for this is that while the number of

program	KLOC	objects	attributes	concepts	partitions
li	7.6	299	299	303	6
m88ksim	19.9	24	26	26	9
perl	26.9	189	172	197	5,760
ijpeg	31.2	407	303	322	5,713
vortex	67.2	654	575	620	?
bash	73.6	266	211	229	231
gcc	205.1	1,358	1,288	1,311	?

Table 17: Results from applying concept analysis to C programs from the SPEC benchmark (and *bash*). Objects are functions and attributes are of the form "uses `struct t` in parameter list or return type" plus unique identification attributes. "?" indicates that the partitioning algorithm never terminated due to exponential behavior.

program	KLOC	objects	attributes	concepts	partitions
li	7.6	299	6	15	18
m88ksim	19.9	24	17	24	16
perl	26.9	189	32	13,826	?
ijpeg	31.2	407	50	?	?
vortex	67.2	654	91	?	?
bash	73.6	266	44	1,942	?
gcc	205.1	1,358	65	?	?

Table 18: Results from applying concept analysis to C programs from the SPEC benchmark (and *bash*). Objects are functions and attributes are of the form "uses `struct t` in parameter list or return type" plus complemented extensions. "?" indicates that the concept-lattice generator or the partitioning algorithm never terminated due to exponential behavior.

attributes rises sharply by adding unique identification attributes, the density of the context does not really change. On the other hand, complemented extensions have a much smaller increase in the number of attributes, but a much sharper rise in the density of the context. This is reflected in the increased number of concepts in the generated lattices. In some cases (e.g. *gcc*, *vortex*, *ijpeg*), the concept-lattice generation component did not complete even after running for several days.

In the worst case, the number of partitions of a given concept lattice can be exponential in the number of nodes (i.e., concepts) in the lattice. As the tables indicate, in several instances exponential behavior is apparently exhibited.

We now describe a case study.

Case study: *chull*

Chull is a program taken from a computational-geometry library that computes the convex hull of a set of vertices in the plane. The program consists of roughly one thousand lines of C code. It has twenty-six functions and three user-defined `struct` data types: `tVertex`, `tEdge`, and `tFace`, representing vertices, edges, and faces, respectively. Thus, one might hope that three modules—one for each `struct` type—would fall out naturally.

Initially, we formed the context consisting of the twenty-six functions as the object set and three attributes ("uses fields of `tVertex`", "uses fields of `tEdge`", and "uses fields of `tFace`"). The context is not well formed: For example, the attribute set of the `MakeEdge` function is the singleton set consisting of "uses fields of `struct tEdge`", which is a proper subset of the attributes of the `CleanEdges` function ("uses fields of `struct tEdge`" and "uses fields of `struct tFace`"). To extend the context to a well-formed context, we added unique attributes, such as $a_{\texttt{MakeEdge}}$. The resulting context consisted of the twenty-six functions as objects and a total of twenty-four attributes (the original three attributes plus twenty-one unique identification attributes). The resulting concept lattice had thirty concepts and is shown in Figure 17. The partitioning algorithm discovered eight partitions, of which one seemed particularly interesting: It consisted of eleven concepts, ten of which had singleton extents (corresponding to individual functions) and one concept that consisted of the sixteen functions that use the fields of `struct tVertex`. A possible interpretation of this partition as a modularization would be one vertex class (with sixteen member functions), and the remaining functions standing on their own, some of them being friend functions.

We also extended the original *chull* context to a well-formed context by adding complementary attributes. The resulting context consisted of the twenty-six functions as objects and a total of six attributes (the original three attributes plus their complements: "does not use fields of `struct tEdge`", etc.) The resulting concept lattice had twenty-eight nodes and is shown in Figure 18. The partitioning algorithm discovered 154 possible partitions of the concept lattice.

The atomic partition groups the functions into the eight concepts listed in Table 19. Thus, this partition does not break the code directly into three

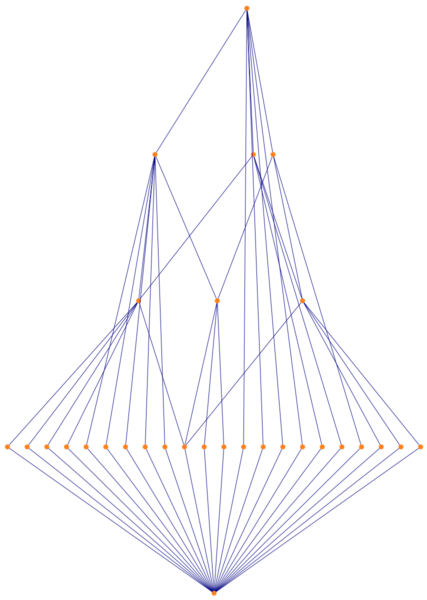

Figure 17: A concept lattice for *chull* after unique identification attributes were added.

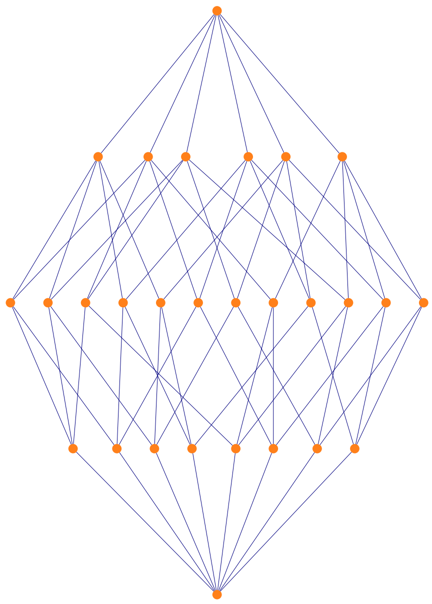

Figure 18: A concept lattice for *chull* after three complementary attributes were added.

concept number	user-defined `struct` types	functions
1	none	`main`, `CleanUp`, `CheckEuler`, `PrintOut`
2	`tVertex`	`MakeVertex`, `ReadVertices`, `Collinear`, `ConstructHull`, `PrintVertices`
3	`tEdge`	`MakeEdge`
4	`tFace`	`CleanFaces`, `MakeFace`
5	`tVertex`, `tEdge`	`CleanVertices`, `PrintEdges`
6	`tVertex`, `tFace`	`Volume6`, `Volumed`, `Convexity`, `PrintFaces`
7	`tEdge`, `tFace`	`MakeCcw`, `CleanEdges`, `Consistency`
8	`tVertex`, `tEdge`, `tFace`	`Print`, `Tetrahedron`, `AddOne`, `MakeStructs`, `Checks`

Table 19: The atomic partition of the concept lattice for *chull* shown in Figure 18.

modules (e.g., one for each `struct` type). However, assuming that the goal is to transform *chull* into an equivalent C++ program, the eight concepts do suggest two possible modularizations: Concepts 2, 3, and 4 would correspond to three classes, for vertex, edge, and face, respectively; concept 1 would correspond to a "driver" module; and the functions in concepts 5 through 8 would form four "friend" modules, where each of the functions would be declared to be a `friend` of the appropriate classes. Alternatively, one could group concepts 2–8 into a polyhedron class with nested vertex, edge, and face classes. Concept 1 would still represent a "driver" module. This possibility corresponds to one of the non-atomic partitions.

Another interesting partition is summarized in Table 20. It consists of four concepts. The first concept identifies the four functions not associated with any of the `struct` types. The second, third, and fourth concepts identify possible face, edge, and vertex classes, respectively. This partition comes close to our expectation (i.e., one module for each `struct` type); however, several functions that, from their names, we might have expected to end up in the edge and face modules ended up in the vertex module (e.g., `Tetrahedron`, `Volume6`, `Volumed`, `PrintEdges`, and `PrintFaces`). These member functions would need to be declared as `friend`s of the other classes.

6 Related Work

Because modularization reflects a design decision that is inherently subjective, it is unlikely that the modularization process can ever be fully automated. Given that some user interaction will be required, the concept-analysis approach offers certain advantages over other previously proposed techniques (e.g., [14, 5, 16, 15, 2]), namely, the ability to "stay within the system" (as opposed to applying ad hoc methods) when the user judges that the modularization that the system suggests is unsatisfactory. If the proposed modularization is on too fine a scale,

concept number	user-defined struct types	functions
1	none	main, CleanUp, CheckEuler, PrintOut
4	tFace	CleanFaces, MakeFace
10	tEdge	MakeCcw, MakeEdge, CleanEdges, Consistency
26	tVertex	MakeVertex, ReadVertices, Print, Tetrahedron, ConstructHull, AddOne, Volume6, Volumed, MakeStructs, CleanVertices, Collinear, Convexity, Checks, PrintVertices, PrintEdges, PrintFaces

Table 20: A four-concept partition of the concept lattice for *chull* shown in Figure 18.

the user can "move up" the partition lattice. (See Section 4.) If the proposed modularization is too coarse, the user can add additional attributes to generate more concepts. (See Section 3.) Furthermore, concept analysis really provides a family of modularization algorithms: Rather than offering one fixed technique, different attributes can be chosen for different situations.

The reader is referred to [2, pp. 27–32] for an extensive discussion of the literature on the modularization problem. In the remainder of this section, we discuss only the work that is most relevant to the approach we have taken.

Liu and Wilde [14] make use of a table that is very much like the object-attribute relation of a context. However, whereas our work uses concept analysis to analyze such tables, Liu and Wilde propose a less powerful analysis. They also propose that the user intervene with ad hoc adjustments if the results of modularization are unsatisfactory. As explained above, the concept-analysis approach can naturally generate a variety of possible decompositions (i.e., different collections of concepts that partition the set of objects).

The concept-analysis approach is more general than that of Canfora et al. [3], which identifies abstract data types by analyzing a graph that links functions to their argument types and return types. The same information can be captured using a context, where the objects are the functions, and the attributes are the possible argument and return types (for example, the first four attributes in the context shown in Table 3 on page 11). By adding attributes that indicate whether fields of compound data types are used in a function, as is done in the example used in Section 3, concept-analysis becomes a more powerful tool for identifying potential modules than the technique described in [3].

The work described in [4] and [5] expands on the abstract-data-type identification technique described in [3]: Call and dominance information is used to introduce a hierarchical nesting structure to modules. It may be possible to combine the techniques from [4] and [5] with the concept-analysis approach of this paper.

Canfora et al. discuss two types of links that cause undesirable clustering of functions [2]. The first type, "coincidental links", caused by routines that implement more than one function, can be overcome by program slicing [22, 10]. The second type, "spurious links", is caused by functions that access supporting data structures of more than one object type. In most of the approaches mentioned above, spurious links arise from a function that accesses several global variables of different types. The work described in [14, 5, 15, 24, 2] will all stumble on examples that exhibit spurious links. In our approach, an analogous kind of spurious link arises due to functions that access internal fields of more than one `struct`. An example is found in the tangled-code example discussed in Section 3.2, where the `enq` function uses the fields of both `struct stack` and `struct queue`. The additional discriminatory power of the concept-analysis approach is due to the fact that it is able to exploit both positive and negative information.

In contrast with the approach to identifying *objects* described in [1], our technique is aimed at analyzing relationships among functions and types to identify *classes*. In [1], the aim is to identify objects that link functions to specific variables. A similar effect can be achieved via concept analysis by introducing one attribute for each actual parameter.

There has been a certain amount of work involving the use of cluster analysis to identify potential modules (e.g., [11, 1, 12, 2]). This work (implicitly or explicitly) involves the identification of potential modules by determining a similarity measure among pairs of functions. We are currently investigating the link between concept analysis and cluster analysis.

[6] offers background on lattice theory and an introduction to concept analysis. [23] formalizes the notions of concept analysis and provides a proof of the fundamental theorem.

Concept analysis has been applied to many kinds of problems. Concept analysis was first applied to software engineering in the NORA/RECS tool, where it was used to identify conflicts in software-configuration information [21].

Contemporaneously with our own work, Lindig and Snelting [13] and Sahraoui et al. [18] independently explored the idea of applying concept analysis to the modularization problem. In both of these studies, the context relations used for concept analysis relate each function of the program to the global variables accessed by the function.

The results reported by Lindig and Snelting on two case studies of small to medium-sized Fortran and Cobol programs are not encouraging. In both cases, the concept lattice that resulted did not identify any useful ways to decompose the program into modules. However, we believe that the results achieved by our approach to using concept analysis are more promising than those of Lindig and Snelting and Sahraoui et al. This is due to several factors:

- The languages on which the techniques were applied—i.e., Fortran and Cobol (in the case of Lindig and Snelting) versus C. The C-to-C++ conversion problem is a variant of the modularization problem that has more structure than Fortran-to-X and Cobol-to-X conversion/modularization problems. In particular, the C program's `struct` types serve as a natural starting point for the C++ program's classes.

- Lindig and Snelting and Sahraoui et al. use context relations that relate each function of a program to the global variables accessed by the function. In our work, context relations relate each function of a program to (i) the fields of user-defined `struct` types that the function accesses, (ii) the types of sub-expressions that occur within the function, and (iii) the complements of (i) and (ii).

- In our work, we employ negative information (e.g., "attributes of the form `f` does not use fields of `struct t`"). This allows the concepts identified to be based not only on the similarities between functions, but also on their differences.

References

[1] B. L. Achee and Doris L. Carver. A greedy approach to object identification in imperative code. In *Third Workshop on Program Comprehension*, pages 4–11, 1994.

[2] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Software — Practice and Experience*, 26(1):25–48, January 1996.

[3] G. Canfora, A. Cimitile, M. Tortorella, and M. Munro. Experiments in identifying reusable abstract data types in program code. In *Second Workshop on Program Comprehension*, pages 36–45, 1993.

[4] G. Canfora, A. De Lucia, G. A. Di Lucca, and A. R. Fasolino. Recovering the architectural design for software comprehension. In *Third Workshop on Program Comprehension*, pages 30–38, 1994.

[5] A. Cimitile, M. Tortorella, and M. Munro. Program comprehension through the identification of abstract data types. In *Third Workshop on Program Comprehension*, pages 12–19, 1994.

[6] B.A. Davey and H.A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 1990.

[7] Jean-François Girard. Personal communiation., July 1998.

[8] Jean-François Girard and Rainer Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *International Conference on Software Maintenance*, pages 58–65, Bari, Italy, October 1997.

[9] R. Godin and R. Missaoui H. Alaoui. Incremental concept formation algorithms based on Galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.

[10] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[11] David H. Hutchens and Victor R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, SE-11(8):749–757, August 1985.

[12] Thomas Kunz. Evaluating process clusters to support automatic program understanding. In *Fourth Workshop on Program Comprehension*, pages 198–207, 1996.

[13] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359, 1997.

[14] Sying-Syang Liu and Norman Wilde. Identifying objects in a conventional procedural language: An example of data design recovery. In *Conference on Software Maintenance*, pages 266–271. IEEE Computer Society Press, November 1990.

[15] Panos E. Livadas and Theodore Johnson. A new approach to finding objects in programs. *Software Maintenance: Research and Practice*, 6:249–260, 1994.

[16] Philip Newcomb. Reengineering procedural into object-oriented systems. In *Second Working Conference on Reverse Engineering*, pages 237–249, July 1995.

[17] Robert O'Callahan and Daniel Jackson. Practical program understanding with type inference. Technical Report CMU-CS-96-130, Carnegie Mellon University, May 1996.

[18] Houari A. Sahraoui, Walcélio Melo, Hakim Lounis, and François Dumont. Applying concept formation methods to object identification in procedural code. Technical Report CRIM-97/05-77, CRIM, 1997.

[19] Michael Siff and Thomas Reps. Program generalization for software reuse: From C to C++. In *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 135–146, San Francisco, October 1996.

[20] Michael Siff and Thomas Reps. Identifying modules via concept analysis. In *International Conference on Software Maintenance*, pages 170–179, Bari, Italy, October 1997.

[21] Gregor Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.

[22] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[23] Rudolf 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.

[24] Alexander Yeh, David R. Harris, and Howard B. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *Second Working Conference on Reverse Engineering*, pages 227–236, 1995.

A Correctness of the Concept-Partition Algorithm

We assume familiarity with the basic definitions for contexts, concepts, and concept lattices given in Section 2, and concept partitions given in Section 4. Recall that a *concept* is a pair of sets—a set of objects (the *extent*) and a set of attributes (the *intent*) (X, Y)—such that $Y = \sigma(X)$ and $X = \tau(Y)$. A concept is uniquely identified by its extent. In the following, we abuse the language slightly by referring to a concept as a set when really we mean to treat the extent of the concept as a set, which it is. For example, "the union of two concepts" refers to the concept identified by taking the smallest concept which has an extent containing the union of the extents of the two concepts.

Recall the following definitions:

- *Well-formed context*: A context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ is *well-formed* if and only if, for every pair of elements $x, y \in \mathcal{O}$, $\sigma(\{x\}) \subseteq \sigma(\{y\})$ implies $\sigma(\{x\}) = \sigma(\{y\})$. (See page 21.)
- *Atomic partition*: An *atomic partition* of a concept lattice is a concept partition consisting of exactly the concepts with smallest extent containing each of the objects treated as a singleton set (the atomic concepts). (See page 20.)

We say that a set of concepts q *blankets* a concept c if the extent of c is a subset of the union of the extents of q.

Lemma 1 *The concept lattice derived from a well-formed context has an atomic partition.*

***Proof*:**

For any context, the collection of atomic concepts is $\{(\tau(\sigma(\{x\})), \sigma(\{x\})) | x \in \mathcal{O}\}$

By extensivity, $x \in \tau(\sigma(\{x\}))$ and thus the union of the extents of atomic concepts is equal to the set of all objects.

If $\sigma(\{x\}) = \sigma(\{y\})$ then x, y are in the extent of the same atomic concept.

Suppose $\sigma(\{x\}) \neq \sigma(\{y\})$.

For any $z \in \tau(\sigma(\{x\}))$, $\sigma(\{x\}) \subseteq \sigma(\{z\})$.

By well-formedness, $\sigma(\{x\}) = \sigma(\{z\})$.

So, $z \notin \tau(\sigma(\{y\}))$.

Likewise, if $z \in \tau(\sigma(\{y\}))$ then $z \notin \tau(\sigma(\{x\}))$

Thus, either $\tau(\sigma(\{x\})) = \tau(\sigma(\{y\}))$ or $\tau(\sigma(\{x\})) \cap \tau(\sigma(\{y\})) = \emptyset$

$\therefore$ The concept lattice derived from a well-formed context has an atomic partition.

$\square$

Lemma 2 *If c is a non-atomic concept in a concept lattice derived from a well-formed context, then the set of atomic concepts that are subordinate to c in the lattice form a partition of c.*

***Proof*:**

Let q be the set of atomic concepts that are subordinate to c. By the definition of an atomic concept, the concepts in q are pairwise disjoint.

Consider any x in the extent of c. The atomic concept containing x (i.e., $\tau(\sigma\{x\})$ must be a member of q, so x is in the union of the extents of q. So q blankets c.

$\therefore$ Concept set q is a partition of c.

$\square$

Lemma 3 *If c is a non-atomic concept in a concept lattice derived from a well-formed context, then there exists a $q \subseteq subs(c)$ and a $c' \in q$ that is covered by c such that the set of extents of the elements of q partition the extent of c.*

***Proof*:**

Let c' be any concept that is covered by c (i.e., $c \in \text{covs}(c')$). Let q' be the set of atomic concepts that are subordinate to c'. By Lemma 2, q' partitions c'.

Let q be the set consisting of c' and the atomic concepts that are subordinate to c except for those atomic concepts in q'. By definition, q is disjoint.

Consider any x in the extent of c. The atomic concept containing x (i.e., $\tau(\sigma\{x\})$ must either be in q or its extent must be a subset of c'. Either way, x is a member of the union of the extents of q. So q blankets c.

□

Definition 1 (Level order for a concept lattice) *Given a concept lattice, a* ***level order*** *for the lattice is a linear order induced by a breadth-first navigation through the lattice starting at the bottom element, not including the top element. The top element is defined to be greater than all the other elements of the lattice.*

Definition 2 (Partition order) *Given any two partitions p, p' of a concept lattice and a level order for the lattice $<_c$, suppose the concepts in p are $c_{\alpha_0}, \ldots, c_{\alpha_{j-1}}$ such that $c_{\alpha_{j-1}} <_c \ldots <_c c_{\alpha_1} <_c c_{\alpha_0}$ and the concepts of p' are $c_{\beta_0}, \ldots, c_{\beta_{k-1}}$ such that $c_{\beta_{k-1}} <_c \ldots <_c c_{\beta_1} <_c c_{\beta_0}$. We then can think of p as the "word" $c_{\alpha_0} c_{\alpha_1} \cdots c_{\alpha_{j-1}}$ and the word p' as the "word" $c_{\beta_0} c_{\beta_1} \cdots c_{\beta_{k-1}}$. We define the* ***partition order*** *on the set of partitions of the concept lattice to be the lexicographic ordering of these "words."*

Lemma 4 *In the partitioning algorithm (Figure 15, page 27), every $p \in P$ is a partition.*

***Proof*:** By induction on the number of iterations in the algorithm.

Base case: $p = \text{covs}(\perp)$, so p is the atomic partition (i.e. the partition consisting of all the atomic concepts). The partition algorithm assumes the context to be well-formed and hence p is a partition by Lemma 1.

Induction hypothesis: For iterations less than i, all p inserted into P at line [12] are partitions.

Induction step: At iteration i, if p'' is inserted into P at line [12], then $p'' = p - \text{subs}(c') \cup \{c'\}$. The union of the extents of p'' is equal to $\mathcal{O}$ because the union of the extents of p is equal to $\mathcal{O}$ and the union of the extents of the concepts subordinate to c' is a subset of the extent of c'. Line [9] ensures that the objects in the extent of c' are disjoint from $p - \text{subs}(c')$. Thus p'' is also a partition.

□

Lemma 5 *If p is any partition of a concept lattice derived from a well-formed context, then the partitioning algorithm (Figure 15, page 27) finds p.*

***Proof*:** By Lemma 1, there is at least one partition, namely the atomic partition. Suppose the claim is false. Fix a partition order for the partitions of the lattice. Suppose p_* is the least such partition (according to the partition order) that is not found by the algorithm. We now show that this assumption leads to a contradiction.

The atomic partition is discovered by the algorithm, so p_* must contain a non-atomic concept; call it c_*. By Lemma 3, there exists a $q \subseteq \text{subs}(c_*)$ and a $c_0 \in q$ that is covered by c_* (i.e., $c_* \in \text{covs}(c_0)$) in the concept lattice such that the set of extents of the elements of q partitions the extent of c_*. Let $p_0 = p_* - \{c_*\} \cup q$. p_0 is a partition because $c_* = \bigcup q$ and the elements of q are pairwise disjoint. $p_0 <_p p_*$ because for all $c \in q$, $c <_c c_*$. By hypothesis, p_0 is discovered by the algorithm. Because $c_* \in \text{covs}(c_0)$, at some iteration of the while loop beginning at line[4] the p in line [5] is p_0; at some iteration of the for loop beginning at line [6], c is c_0; for some iteration of the for loop beginning at line [7], c' is c_*; and finally, at line [12], p'' is p_*. Therefore, partition p_* is discovered by the algorithm, which contradicts our assumption. That is, there is no such least partition, and hence all partitions are discovered by the partitioning algorithm.

$\square$

Theorem 6 *Given a concept lattice derived from a well-formed context, the partition algorithm (Figure 15, page 27) finds exactly all partitions of the lattice.*

***Proof*:** Immediate from Lemma 4 and Lemma 5.

$\square$