Identifying Modules Via Concept Analysis

Michael Siff and Thomas Reps
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706
phone: (608) 262-1204
fax: (608) 262-9777
{siff, reps}@cs.wisc.edu

January 31, 1997

Abstract

We describe a general technique for identifying modules in programs that do not designate them explicitly. 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 describe an algorithm that, given a concept lattice, identifies possible ways of partitioning the program into modules. We discuss a prototype implementation and some results on small and medium-sized programs.

Keywords: Concept analysis, modularization, software migration, software restructuring, reverse engineering, design recovery.

1 Introduction

Many existing software systems were developed using programming languages and paradigms that do not incorporate object-oriented features and design principles. These systems often have a monolithic style that makes 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.

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

- 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 have implemented a prototype tool that uses concept analysis to propose modularizations of C programs. The implementation has been tested on several small and medium-sized examples. The largest example consists of about 28,000 lines of source code. (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 code that does not delineate them explicitly. The resulting information can then be 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 [2, 14], they are unable to handle a variant of this example in which stack and queue 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.

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 and results. Section 6 concerns related work.

```
#define QUEUE_SIZE 10

struct stack { int* base; int* sp; int size; };

struct queue { struct stack* front; struct stack* 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++; } /* no overflow check */

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: C code to implement a queue using two stacks.

```
#define 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++; } // no overflow check
};

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++.

2 A Concept Analysis Primer

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

To illustrate concept analysis, we consider the example of a crude classification of a group of mammals: cats, chimpanzees, 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.

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

Table 1: A crude characterization of mammals.

In order to understand the basics of concept analysis, a few definitions are required. We follow the presentation in [11]. 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 $\sigma(X) = \{a \in \mathcal{A} | \forall o \in X : (o, a) \in \mathcal{R}\}$ (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 satisfy:

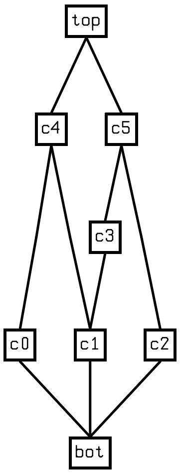
$$X_1 \subseteq X_2 \Rightarrow \sigma(X_2) \subseteq \sigma(X_1) \quad \text{and} \quad Y_1 \subseteq Y_2 \Rightarrow \tau(Y_2) \subseteq \tau(Y_1)$$

and

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

(The mappings are *antimonotone* and *extensive*.) In the mammal example, $\sigma(\{\text{cats}, \text{chimpanzees}\}) = \{\text{hair-covered}\}$ 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, chimpanzees}, {hair-covered}) is not a concept. A concept (X_0, Y_0) is a *subconcept* of concept (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 ({chimpanzees, 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.



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

Figure 3: The concept lattice (and accompanying key) for the mammal example.

The fundamental theorem for concept lattices [12] relates subconcepts and superconcepts as follows:

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

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 taking the union of the extents and then finding the common objects of the set of common attributes of that resulting union. An example of the application of the fundamental theorem is as follows:

$$\begin{aligned}
&(\{\text{chimpanzees}\}, \{\text{hair-covered}, \text{intelligent}, \text{thumbed}\}) \sqcup (\{\text{dolphins}, \text{whales}\}, \{\text{intelligent}, \text{marine}\}) \\
&\quad = (\tau(\sigma(\{\text{chimpanzees}, \text{dolphins}, \text{whales}\})), \{\text{intelligent}\}) \\
&\quad = (\tau(\{\text{intelligent}\}), \{\text{intelligent}\}) \\
&\quad = (\{\text{chimpanzees}, \text{humans}, \text{dolphins}, \text{whales}\}, \{\text{intelligent}\})
\end{aligned}$$

This computation corresponds to the fact that $c_1 \sqcup c_2 = c_5$ in the lattice shown in Figure 3.

There are several algorithms for computing a concept lattice from a given context [11]. 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 those objects that have all the attributes (often the empty set, as in our example).

The initial step of the algorithm is to compute the bottom 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. The atomic concepts correspond to those nodes in the concept lattice reachable from the bottom node in one step. Computation of some of the atomic concepts for the mammal example is shown below:

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

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 $c'' = c_0 \sqcup c_1$ using the fundamental theorem of concept analysis. 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 below:

$$\begin{aligned}
Worklist &= [(c_0, c_1), (c_0, c_2), (c_0, c_3), (c_1, c_2), (c_2, c_3)] \\
c_4 &= c_0 \sqcup c_1 = (\{\text{cats}, \text{chimpanzees}, \text{dogs}\}, \{\text{hair-covered}\}) \\
Worklist &= [(c_0, c_2), (c_0, c_3), (c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_0 \sqcup c_2 &= \top = (\{\text{cats}, \text{chimpanzees}, \text{dogs}, \text{dolphins}, \text{humans}, \text{whales}\}, \emptyset) \\
Worklist &= [(c_0, c_3), (c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_0 \sqcup c_3 &= \top \\
Worklist &= [(c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_5 &= c_1 \sqcup c_2 = (\{\text{chimpanzees}, \text{dolphins}, \text{humans}, \text{whales}\}, \{\text{intelligent}\}) \\
Worklist &= [(c_2, c_3), (c_2, c_4), (c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_2 \sqcup c_3 &= c_5 \\
Worklist &= [(c_2, c_4), (c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_2 \sqcup c_4 &= \top \\
Worklist &= [(c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_3 \sqcup c_4 &= \top \\
Worklist &= [(c_0, c_5), (c_4, c_5)] \\
c_0 \sqcup c_5 &= \top \\
Worklist &= [(c_4, c_5)] \\
c_4 \sqcup c_5 &= \top \\
Worklist &= \emptyset
\end{aligned}$$

3 Using Concept Analysis to Identify Potential Modules

The main idea of this paper is to apply concept analysis to the problem of identifying potential modules within monolithic 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 Applying concept analysis to 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 (page 4).

First, we define a context. Let the objects be $\theta_0, \theta_1, \ldots, \theta_7$, and the attributes be $\alpha_0, \alpha_1, \ldots, \alpha_5$, where the θ_i's and α_i's correspond to functions and properties of functions as indicated by the tables below:

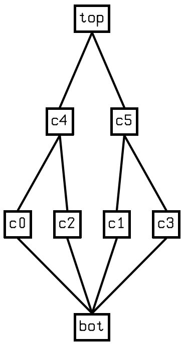
θ_0	`initStack`
θ_1	`initQ`
θ_2	`isEmptyStack`
θ_3	`isEmptyQ`
θ_4	`push`
θ_5	`enq`
θ_6	`pop`
θ_7	`deq`

α_0	return type is `struct stack *`
α_1	return type is `struct initQ *`
α_2	has argument of type `struct stack *`
α_3	has argument of type `struct queue *`
α_4	uses fields of `struct stack`
α_5	uses fields of `struct queue`

The context relation for the stack and queue example is then:

	α_0	α_1	α_2	α_3	α_4	α_5
θ_0	$\surd$				$\surd$	
θ_1		$\surd$				$\surd$
θ_2			$\surd$		$\surd$	
θ_3				$\surd$		$\surd$
θ_4			$\surd$		$\surd$	
θ_5				$\surd$		$\surd$
θ_6			$\surd$		$\surd$	
θ_7				$\surd$		$\surd$

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, together with a key, identifying lattice-node labels with corresponding concepts, is shown below:



top	$(\{\theta_0, \theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7\}, \emptyset)$	universal concept
c_5	$(\{\theta_1, \theta_3, \theta_5, \theta_7\}, \{\alpha_5\})$	queue concept
c_4	$(\{\theta_0, \theta_2, \theta_4, \theta_6\}, \{\alpha_4\})$	stack concept
c_3	$(\{\theta_3, \theta_5, \theta_7\}, \{\alpha_3, \alpha_5\})$	`isEmptyQ`, `enq`, `deq`
c_2	$(\{\theta_2, \theta_4, \theta_6\}, \{\alpha_2, \alpha_4\})$	`isEmptyStack`, `push`, `pop`
c_1	$(\{\theta_1\}, \{\alpha_1, \alpha_5\})$	`initQ`
c_0	$(\{\theta_0\}, \{\alpha_0, \alpha_4\})$	`initStack`
bot	$(\emptyset, \{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\})$	empty concept

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_2) are distinct concepts from other functions (concepts c_1 and c_3). The former two concepts correspond to constructors and the latter two to 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 that the stack concept consists of a constructor concept and a member-function concept.

3.2 Adding complementary attributes to "untangle" code

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 [2, 14] will also create the same grouping. In essence, the concept analysis described above emulates these techniques. This shows that concept analysis encompasses previously defined methods for modularization. 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 (see Figure 4), rather than calling `isEmptyStack` 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 α_4 ("uses fields of `struct stack`") now applies to `isEmptyQ` and `enq`. Table 2 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 5. 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 $\{\texttt{isEmptyStack}, \texttt{push}, \texttt{pop}\}$, the concept $c_0 \sqcup c_2 = c_7$ is not the stack concept: c_7 consists of `initStack`, `isEmptyStack`, `isEmptyQ`, `push`, `enq`, and `pop`, which mixes the stack operations with some, but not all, of the queue operations.

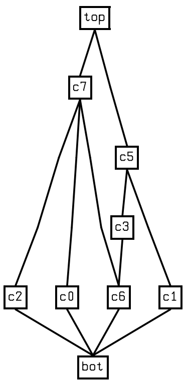
```
int isEmptyQ(struct queue* q) {
  return (q->front->sp == q->front->base && q->back->sp == q->back->base);
}

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

Figure 4: The queue and stack example revisited: “Tangled” C code.

	α_0	α_1	α_2	α_3	α_4	α_5
θ_0	$\surd$				$\surd$	
θ_1		$\surd$				$\surd$
θ_2			$\surd$		$\surd$	
θ_3				$\surd$	$\surd$	$\surd$
θ_4			$\surd$		$\surd$	
θ_5				$\surd$	$\surd$	$\surd$
θ_6			$\surd$		$\surd$	
θ_7				$\surd$		$\surd$

Table 2: The context relation for the “tangled” stack and queue example.



top	$(\{\theta_0, \theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7\}, \emptyset)$	universal concept
c_7	$(\{\theta_0, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}, \{\alpha_4\})$	???
c_5	$(\{\theta_1, \theta_3, \theta_5, \theta_7\}, \{\alpha_5\})$	queue concept
c_3	$(\{\theta_3, \theta_5, \theta_7\}, \{\alpha_3, \alpha_5\})$	`isEmptyQ`, `enq`, `deq`
c_6	$(\{\theta_3, \theta_5\}, \{\alpha_3, \alpha_4, \alpha_5\})$	`isEmptyQ`, `enq`
c_2	$(\{\theta_2, \theta_4, \theta_6\}, \{\alpha_2, \alpha_4\})$	`isEmptyStack`, `push`, `pop`
c_1	$(\{\theta_1\}, \{\alpha_1, \alpha_5\})$	`initQ`
c_0	$(\{\theta_0\}, \{\alpha_0, \alpha_4\})$	`initStack`
bot	$(\emptyset, \{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\})$	empty concept

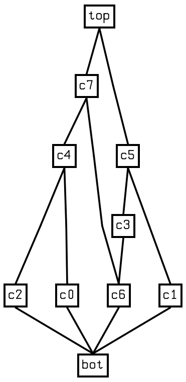
Figure 5: The concept lattice (and corresponding key) for the “tangled” stack and queue example using the attributes listed on page 8.

The problem is that the attributes listed on page 8 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 the complement of "uses fields of `struct queue`" (i.e., "does not use fields of `struct queue`"). The revised set of attributes and the corresponding context relation are shown below:

α_0	return type is `struct stack *`
α_1	return type is `struct initQ *`
α_2	has argument of type `struct stack *`
α_3	has argument of type `struct queue *`
α_4	uses fields of `struct stack`
α_5	uses fields of `struct queue`
α_6	does not use fields of `struct queue`

	α_0	α_1	α_2	α_3	α_4	α_5	α_6
θ_0	$\surd$				$\surd$		$\surd$
θ_1		$\surd$				$\surd$	
θ_2			$\surd$		$\surd$		$\surd$
θ_3				$\surd$	$\surd$	$\surd$	
θ_4			$\surd$		$\surd$		$\surd$
θ_5				$\surd$	$\surd$	$\surd$	
θ_6			$\surd$		$\surd$		$\surd$
θ_7				$\surd$		$\surd$	

The resulting concept lattice (and corresponding key) is now:



top	$(\{\theta_0, \theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7\}, \emptyset)$	universal concept
c_7	$(\{\theta_0, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}, \{\alpha_4\})$	
c_5	$(\{\theta_1, \theta_3, \theta_5, \theta_7\}, \{\alpha_5\})$	queue concept
c_4	$(\{\theta_0, \theta_2, \theta_4, \theta_6\}, \{\alpha_4, \alpha_6\})$	stack concept
c_3	$(\{\theta_3, \theta_5, \theta_7\}, \{\alpha_3, \alpha_5\})$	`isEmptyQ`, `enq`, `deq`
c_6	$(\{\theta_3, \theta_5\}, \{\alpha_3, \alpha_4, \alpha_5\})$	`isEmptyQ`, `enq`
c_2	$(\{\theta_2, \theta_4, \theta_6\}, \{\alpha_2, \alpha_4, \alpha_6\})$	`isEmptyStack`, `push`, `pop`
c_1	$(\{\theta_1\}, \{\alpha_1, \alpha_5\})$	`initQ`
c_0	$(\{\theta_0\}, \{\alpha_0, \alpha_4, \alpha_6\})$	`initStack`
bot	$(\emptyset, \{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6\})$	empty concept

This concept lattice contains all of the concepts in the concept lattice from Figure 5, 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 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 6.

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

```
#define 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 6: Queue and stack classes in C++ with friends.

correspond to functions. 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`".
- *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 [10, 9]. 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 and Module 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. 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.

We now formalize the notion of a *concept partition* and present an algorithm to identify such partitions from a concept lattice.

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 if and only if 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 on page 11. 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 7.) A concept lattice need not have an atomic partition. For example, the lattice in Figure 3 (page 6) 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 "chimpanzees" is in the extent of both concepts.

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. This can be achieved by augmenting a context with negative information (similar to what we did 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, in the attribute table on page 11, α_5 and α_6 are complements.

Given a context $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$, the *complemented extension* of $\mathcal{C}$ is the the context $\mathcal{C}' = (\mathcal{O}, \mathcal{A}', \mathcal{R}')$ such that $\mathcal{A}' = \mathcal{A} \cup \{\overline{a} | a \in \mathcal{A}\}$ and $\mathcal{R}' = \mathcal{R} \cup \{(x, \overline{a}) | x \in \mathcal{O}, (x, a) \notin \mathcal{R}\}$. A complemented extension of a context is the original context with the attribute set augmented by the addition of a complement for every original attribute.

It is straightforward to see that every context has a complemented extension, and that the concept lattice of a complemented extension has an atomic partition. Using this fact, we can now present an algorithm to find the all the partitions of a concept lattice.

4.2 Finding partitions from a concept lattice

Given a concept lattice, we define the following relations on its elements: The set of immediate suprema of concept x, denoted by $\text{sups}(x)$, is the set of lattice elements y such that $x \leq y$ and there are no elements z for which $x \leq z \leq y$. The set of ancestors of x, denoted by $\text{ancs}(x)$, is the set of lattice elements y such that $y \leq x$ and $y \neq x$.

```
A ← sups(⊥)                          // the atomic partition
P ← {A}
W ← {A}
while W ≠ ∅ do
  remove some p from W
  for each c ∈ p
    for each c′ ∈ sups(c)
      p′ ← p − ancs(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 7: An algorithm to find the partitions of a concept lattice.

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 set of immediate suprema 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 supremum of that concept, adding it to the partition, and removing overlapping concepts. The algorithm is given in Figure 7.

In the worst case, the number of partitions can be exponential in the number of concepts. In such a case (or any case where the number of partitions is large), it is possible to adapt the algorithm to work interactively. After each new partition is discovered, the algorithm would pause for the user to consider that partition. If it is on too fine a scale, the user would allow the algorithm to iterate further to find coarser-grained partitions.

5 Implementation and Results

We have implemented a prototype tool that employs concept analysis to propose modularizations of C programs. It is written in Standard ML of New Jersey (version 109.24) and runs on a Sun Sparc under Solaris 2.5.1.

The prototype takes a C program as input. The default object set is the set of all functions defined in the input program. The default attribute set consists of one attribute of the form "uses the fields of `struct t`" for each user-defined `struct` type (or equivalent `typedef`) in the input program. The user has the option to include attributes of the form "has a parameter or return type of type `t`." The context can be formed as is, or can be formed in *fully complemented form*, where for each user-defined `struct` type, the attributes of the form"does not use fields of `struct t`" (or "does not have a parameter or return type of type `t`") are included in the attribute set of the context.

The context is then fed into a concept analyzer, which builds the concepts bottom up as described in Section 2. The user can then view the concept lattice or feed the lattice into the partitioner, which computes (depending on the user's choice) all possible partitions or one partition at a time.

The examples in this paper were analyzed by the implementation. Preliminary results on larger examples appear promising. In particular, we have used the prototype tool on 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 data types. The concept lattice for the fully complemented context associated with these functions and data types consists of thirty-four concepts and was constructed in 30 seconds of user time (on a SPARCstation 10 with 64MB of RAM). The partitioner identified 63 possible partitions of the lattice in roughly the same amount of time.

5.1 Case study: `chull.c`

`chull.c` 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. The context fed into the concept analyzer consisted of the twenty-six functions as the object set, six attributes ("uses fields of `tVertex`", "does not use fields of `tVertex`", etc.), and the binary relation indicating whether or not function `f` uses fields of one of the `struct` types. The concept analyzer built twenty-eight concepts and the corresponding lattice in roughly one second of user time. The lattice appears in Figure 8. The partitioner computed the 153 possible partitions of the concept lattice in roughly two seconds.

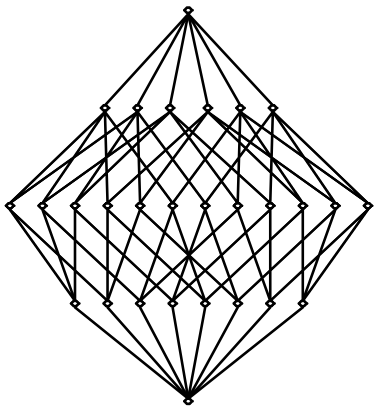


Figure 8: The concept lattice derived from `chull.c`.

The atomic partition groups the functions into the eight concepts listed in Table 3. This partition indicates that the code does not cleanly break into three modules (e.g., one for each `struct` type). However, assuming that the goal is to transform `chull.c` into an equivalent C++ program, the eight concepts do suggest a possible modularization based on the three types: 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.

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 3: The atomic partition of the concept lattice derived for `chull.c`.

6 Related Work

Although there is a growing body of literature concerning module and abstract-data-type recovery from non-modular code (e.g., [13, 6]), we are unaware of previous work on the problem involving the use of concept analysis. 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, 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, 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 conditions.

The work most closely related to ours is that of Liu and Wilde [8], which makes 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. [2], 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, attributes $\alpha 0, \ldots, \alpha_3$ in the attribute table on page 8). By adding attributes that indicate whether fields of compound data types are used in a function, as is done in the example used in this paper, concept-analysis becomes a more powerful tool for identifying potential modules than the technique described in [2].

The work described in [3] and [4] expands on the abstract-data-type identification technique described in [2]: Call and dominance information is used to introduce a hierarchical nesting structure to modules. It may be possible to combine the techniques from [3] and [4] with the concept-analysis approach of the present paper.

The concept-analysis approach is also more general than technique used in the OBAD tool [14], which is designed to identify abstract data types in C programs. OBAD analyzes a graph that consists of nodes representing functions and `struct` types, and edges representing the use of internal fields of a `struct` type by a function. This recovers similar information to a concept analysis in which the attributes are exactly those indicating the use of fields of `struct` types (for example, α_4 and α_5 in Table 3.1 on page 8). However, OBAD will stumble on tangled code like that in the example discussed in Section 3.2. 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., [5, 1, 7]). 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.

Concept analysis has previously been applied in a software-engineering tool, albeit for a problem much different from modularization: the NORA/RECS tool uses concept analysis to identify conflicts in software-configuration information [11].

Acknowledgements

This work was supported in part by the National Science Foundation under grant CCR-9625667 and 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).

The comments of Manuvir Das on the work reported in the paper are greatly appreciated.

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, 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.

[3] 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.

[4] 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.

[5] 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.

[6] IEEE. *IEEE Third Workshop on Program Comprehension*. IEEE Computer Science Press, November 1994.

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

[8] 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.

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

[10] 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.

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

[12] 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.

[13] Lida Wills, Phillip Newcomb, and Elliot Chikofsky, editors. *Second Working Conference on Reverse Engineering*. IEEE Computer Science Press, July 1995.

[14] 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.