# Identifying Modules Via Concept Analysis*

Michael Siff and Thomas Reps
University of Wisconsin-Madison
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.*

## 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 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.)
- 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 1a. 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 1b.

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 supplied to a suitable transformation tool that maps C code to C++ code, as in the aforementioned example. Although other modularization algorithms are

*To appear in ICSM '97: IEEE Int. Conf. on Softw. Maint. (Bari, Italy, Sept. 29 – Oct. 3, 1997).

able to identify the same decomposition [3, 21], 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. 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* [20].

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.

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 $\sigma(X) = \{a \in \mathcal{A} | \forall o \in X : (o, a) \in \mathcal{R}\}$ (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, $\sigma(\{\text{cats}, \text{chimpanzees}\}) = \{\text{hair-covered}\}$ 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 2.

The fundamental theorem for concept lattices [20] 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 as follows:

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

This computation corresponds to the fact that $c_1 \sqcup c_2 = c_5$ in the lattice shown in Figure 2.

There are several algorithms for computing the concept lattice for a given context [6, 18]. 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 (which is often the empty set, as in our example).

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. The atomic concepts correspond to those elements in the concept lattice reachable from the bottom element in one step. Computation of one of the atomic concepts for the mammal example is shown below:

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

```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 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); }
```

(a)

```cpp
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); }
};
```

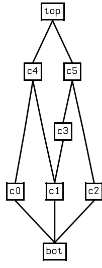
(b)

Figure 1: Code to implement a queue using two stacks in: (a) C and (b) C++

| | | attributes | | | | |
|---|---|---|---|---|---|---|
| | | four-legged | hair-covered | intelligent | marine | thumbed |
| objects | cats | √ | √ | | | |
| | chimpanzees | | √ | √ | | √ |
| | dogs | √ | √ | | | |
| | dolphins | | | √ | √ | |
| | humans | | | √ | | √ |
| | whales | | | √ | √ | |

Table 1: A crude characterization of mammals.

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,



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

Figure 2: The concept lattice (and accompanying key) for the mammal example.

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.

## 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 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 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 1. (page 3).

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 $\theta_i$'s and $\alpha_i$'s correspond to functions and properties of functions as indicated by the tables below:

| | |
|---|---|
| $\theta_0$ | `initStack` |
| $\theta_1$ | `initQ` |
| $\theta_2$ | `isEmptyStack` |
| $\theta_3$ | `isEmptyQ` |
| $\theta_4$ | `push` |
| $\theta_5$ | `enq` |
| $\theta_6$ | `pop` |
| $\theta_7$ | `deq` |

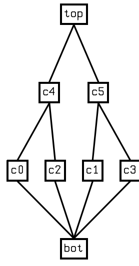
| | |
|---|---|
| $\alpha_0$ | return type is `struct stack *` |
| $\alpha_1$ | return type is `struct queue *` |
| $\alpha_2$ | has argument of type `struct stack *` |
| $\alpha_3$ | has argument of type `struct queue *` |
| $\alpha_4$ | uses fields of `struct stack` |
| $\alpha_5$ | uses fields of `struct queue` |

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

| | $\alpha_0$ | $\alpha_1$ | $\alpha_2$ | $\alpha_3$ | $\alpha_4$ | $\alpha_5$ |
|---|---|---|---|---|---|---|
| $\theta_0$ | $\surd$ | | | | $\surd$ | |
| $\theta_1$ | | $\surd$ | | | | $\surd$ |
| $\theta_2$ | | | $\surd$ | | $\surd$ | |
| $\theta_3$ | | | | $\surd$ | | $\surd$ |
| $\theta_4$ | | | $\surd$ | | $\surd$ | |
| $\theta_5$ | | | | $\surd$ | | $\surd$ |
| $\theta_6$ | | | $\surd$ | | $\surd$ | |
| $\theta_7$ | | | | $\surd$ | | $\surd$ |

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 in Figure 3.

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 (con-




| 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 |

Figure 3: The concept lattice (and corresponding key) for the stack and queue example.

cepts $c_2$ 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 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 1a into two modules, and techniques such as those described in [3, 21] 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 , rather than calling **isEmptyStack** and **push**:

```
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++; }
```

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 $\alpha_4$ ("uses fields of **struct stack**") now applies to **isEmptyQ** and **enq**.

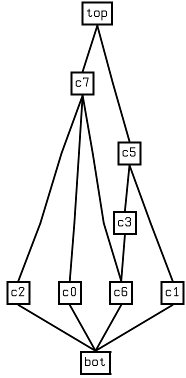
The context relation for the tangled stack and queue code with the original sets of objects and attributes is as follows:

| | $\alpha_0$ | $\alpha_1$ | $\alpha_2$ | $\alpha_3$ | $\alpha_4$ | $\alpha_5$ |
|---|---|---|---|---|---|---|
| $\theta_0$ | $\surd$ | | | | $\surd$ | |
| $\theta_1$ | | $\surd$ | | | | $\surd$ |
| $\theta_2$ | | | $\surd$ | | $\surd$ | |
| $\theta_3$ | | | | $\surd$ | $\surd$ | $\surd$ |
| $\theta_4$ | | | $\surd$ | | $\surd$ | |
| $\theta_5$ | | | | $\surd$ | $\surd$ | $\surd$ |
| $\theta_6$ | | | $\surd$ | | $\surd$ | |
| $\theta_7$ | | | | $\surd$ | | $\surd$ |

The resulting concept lattice is shown in Figure 4. 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 {**initStack**} and the extent of $c_2$ is {**isEmptyStack**, **push**, **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.

The problem is that the attributes listed in Section 3.1. 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 let $\alpha_6$ be the complement of "uses fields of **struct queue**" (i.e., "does not use fields of **struct queue**"). The corresponding context is now:

| | $\alpha_0$ | $\alpha_1$ | $\alpha_2$ | $\alpha_3$ | $\alpha_4$ | $\alpha_5$ | $\alpha_6$ |
|---|---|---|---|---|---|---|---|
| $\theta_0$ | $\surd$ | | | | $\surd$ | | $\surd$ |
| $\theta_1$ | | $\surd$ | | | | $\surd$ | |
| $\theta_2$ | | | $\surd$ | | $\surd$ | | $\surd$ |
| $\theta_3$ | | | | $\surd$ | $\surd$ | $\surd$ | |
| $\theta_4$ | | | $\surd$ | | $\surd$ | | $\surd$ |
| $\theta_5$ | | | | $\surd$ | $\surd$ | $\surd$ | |
| $\theta_6$ | | | $\surd$ | | $\surd$ | | $\surd$ |
| $\theta_7$ | | | | $\surd$ | | $\surd$ | |




| 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 4: The concept lattice (and corresponding key) for the "tangled" stack and queue example using the attributes listed in Section 3.1.

The resulting concept lattice (and corresponding key) is shown in Figure 5. This concept lattice contains all of the concepts in the concept lattice from Figure 4, 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 below:

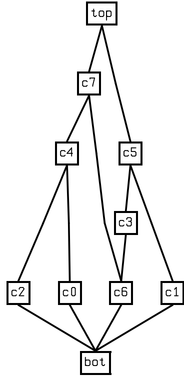
```
class queue;

class stack {
  friend class queue;
private:  // ...
public:  // ...
};

class queue {
private:
  stack *front, *back;
public:  // ...
  int isEmptyQ() {
    return (front->sp == front->base
      && back->sp == back->base);
  }
  void enq(int i) {
    *(front->sp) = i;
    front->sp++;
  }
  // ...
};
```

### 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 functions.[1] 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**" [10, 15].
- *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 [16, 14]. For example, if $f$ is a function declared to be of type `int` $\times$ `int` $\rightarrow$ `bool`, type inference might discover that $f$'s most general type is of the form $\alpha \times \beta \rightarrow$ `bool`. This reveals that the type of $f$'s first argument is distinct from the type of

[1] 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 [19, 7] rather than functions.




| 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 |

Figure 5: The concept lattice (and corresponding key) for the "untangled" stack and queue example.

its second argument (even though they had the same declared type). Attributes might then be of the form "has argument of type $\alpha$" 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 "$\pi_1$ or $\pi_2$". 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.

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 5. 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 in Section 2.) A concept lattice need not have an atomic partition. For example, the lattice in Figure 2 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). Details of how this can be done, along with an algorithm to find all the partitions of a concept lattice, can be found in [17].

## 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 109.27) in conjunction with the SmlTK interface to Tcl/Tk. It runs on a Sun Sparc under Solaris 2.5.1.

The prototype takes a C program as input and builds a context. This is fed into a concept analyzer, which builds the concepts bottom up as described in Section 2. The system's front end builds an abstract syntax tree that is annotated with type information. A context is constructed by routines that walk this structure. Default context-construction routines are provided that build contexts in which the object set is the set of all functions defined in the input program and the 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 examples in this paper were analyzed by the implementation. We have begun to investigate larger examples. 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 partitioner computed the 153 possible partitions of the concept lattice in roughly two seconds.

The atomic partition groups the functions into the eight concepts listed in Table 2. 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. 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.

## 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., [11, 5, 13, 12, 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, 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.

| 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 2: The atomic partition of the concept lattice derived for `chull.c`.

Liu and Wilde [11] 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, attributes $\alpha 0, \ldots, \alpha_3$ in the attribute table in Section 3.1). 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 [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 the present 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 [19, 7]. 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 [11, 5, 12, 21, 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.

There has been a certain amount of work involving the use of cluster analysis to identify potential modules (e.g., [8, 1, 9, 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.

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 [18].

Contemporaneously with our own work, Lindig and Snelting [10] and Sahraoui et al. [15] 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.

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

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[8] 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.

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

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] 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.

[17] Michael Siff and Thomas Reps. Identifying modules via concept analysis. Technical Report CS-TR-97-1337, University of Wisconsin-Madison, January 1997.

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

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

[20] 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.

[21] 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.