AN APPROACH TO INDENTIFYING COMPONENTS FOR SOFTWARE REENGINEERING

THESIS

Presented to the Graduate Council of

Southwest Texas State University

in Partial Fulfillment of

the Requirements

For the Degree

Master of Science

By

Sidharth J. kodikal

San Marcos, Texas

December, 2000

Dedication

This work is dedicated to my wonderful family.

Acknowledgements

I am thankful to Dr. Hall for the patient guidance and unlimited support that he gave me. Thanks to Mr. Davis and Dr. Hazlewood for being on the committee and for supporting this work.

CONTENTS

Table of Figures

Chapter 1

INTRODUCTION

Software maintenance involves making evolutionary updates to an existing program. Increasing expectations for functionality, flexibility, and ease of use have driven software projects to become larger. Moreover, modern business demands more information. Increased project size, coupled with this need for more information, can make maintenance difficult. Larger, more complex software systems, geographically distributed computing, and a consistently growing need for additional functionality have created a complex computing world. Miller observed that "maintenance is the logical consequence of establishing this critical mass of technology. When a critical mass of technology is reached, it is logical to build on that base. Throwing out technology and starting over again without a clear financial return does not make good business sense" [pg. vii, Miller 98]. An ever-increasing proportion of software development labor is expended on maintaining previously developed software. Maintenance and reengineering tasks cannot be performed without understanding the code that is being modified. Studies show that programmers spend as much as 80% of their time in understanding the program they are maintaining [Brooks 95], [Sommerville '96]. Considering that as much as 75% of the software lifecycle is devoted to maintenance [Fairly '85], this suggests that a tremendous amount of software engineering resource are spent on code understanding. If automated tools could reduce this percentage, the economic impact could be staggering [Woods, Quilici, Yang 98].

It is the goal of this research to present techniques for automatic program comprehension and software restructuring. This research has also led to implementation of a prototype for such an automatic program-restructuring tool based on the techniques to be proposed in this thesis.

This research aims particularly at

- Reverse engineering a legacy system with structured architecture into a conceptual model that contains the system information required for effectively reengineering it.
- Restructuring the conceptual model into an equivalent system with object-oriented architecture.

While some aspects of this research apply specifically to the problem of restructuring legacy C programs, the general techniques described in this thesis focus on modularization and subclassing. Modularization is the identification and extraction of modules with high cohesion and low coupling. We define a module as a unit that is directly usable, discrete, and ideally separable from its original context and usable in other contexts. As such, a module should combine data representation and methods for manipulating that data as a neat encapsulated package. Subclassing is the identification of hierarchies amongst modules with the intent of promoting code reuse by way of inheritance.

The motivation for focusing on the structured to object-oriented architecture reengineering problem may be summarized as practical needs, encapsulation/abstraction, and reuse.

Practical needs: The systems built by object-oriented programming are more flexible than traditional systems [Fayad, Laitinen 98]. Practitioners find that adoption of object-oriented programming is beneficial. With companies recognizing the superiority of object-oriented programming languages for large system implementation and maintenance activities, object-oriented development is becoming standard [Brooks 95]. Maintenance tasks are often carried out by new software engineers who are trained primarily in object-oriented languages [Siff 98]. There is therefore a practical need to transform traditional systems to object-oriented systems.

Encapsulation/Abstraction: Object-oriented programming aids encapsulation. Encapsulation involves enclosing related data, routines and definitions in a capsule. Implementation hiding is a concept that is possible because of encapsulation. In a class, both data and routines are classified according to their role as interface or implementation. Encapsulation enhances localization by providing the means to separate the abstract interface of a class from its implementation: the interface being the visible surface of the capsule with the implementation hidden in the capsule [Joyner 96]. Encapsulation effectively reduces coupling by reducing implementation interdependence between modules and reduces the impact of correction. By making modules maximally cohesive and minimally coupled, encapsulation makes the system less brittle to change. Structured programs on the other hand keep data separate from the operations on those data. The structured programming paradigm also suggests that isolation of variables and procedures is good; too much flow of information across boundaries (interfaces) is bad; the flow of values from one procedure to another should be minimized; the clustering of functionality within a procedure or module should be maximized [Lewis 95]. However the procedure-oriented paradigm offers no support for data hiding, as does the object-oriented paradigm. McConnell states that "with older languages, any information hiding counts on voluntary compliance" [pg. 150, McConnell 99]. This often leads to the formation of modules that display high coupling and low cohesion.

> "Abstraction affords ignoring irrelevant details and concentrating on relevant characteristics. Object-oriented design is especially good at abstraction because it uses bigger intellectual building blocks than do functional approaches such as structured design. In structured design, the unit of abstraction is a function. In object-oriented design, the unit of abstraction is an object. Since an object includes both functions and the data that's affected by them, it's possible to work with chunks of a problem that are bigger than functions. Such ability increases the level of abstraction at which one can think about a problem" [pg. 150, McConnell 99].

Reuse: Software development is expensive. Software developers strive to get results in less time and for lower costs. The ultimate dream, however, has been to avoid programming altogether. A very good way to not program is to reuse existing code. A key part of the attraction of object-oriented technology is that a class once created and tested can be reused. Reuse also allows substantial reduction in mistakes.

Construction of an application from tested, reused parts lessens the possibility of errors in the parts and relegates testing to the design and assembly of the correct objects [Miller 98]. Reuse of code is a desirable feature that object-oriented programming languages and tools have taken to new levels.
Inheritance is an object-oriented feature whereby a programmer can model relationships between objects. Inheritance promotes code reuse: the inheriting objects reuse the code of the parent object and may also add new features to it. Miller states that "inheritance is a benefit for programmers because it reduces the amount of code that goes toward duplicating what went before. This reduces errors and may result in faster operations" [pg. 175, Miller 98]. The need to define classes and subclasses of objects, the objects themselves and the attributes, methods, messages and interrelationships of objects, forces a better model of the system to be developed. The ability to model real world objects and their inheritance at multiple levels can be said to be the basis for object-oriented technology's real value [Fayad, Laitinen 98].

Systems built on object-oriented techniques lend themselves to distribution, integration, and faster, error-free application development. They allow systems to evolve and to be reengineered, as business, human and technological factors change. Existing software development practice has produced monolithic applications that are hard to evolve and harder yet to evaluate in terms of cost and value. Functionality of such systems is not easy to extract, making it unavailable for next generation products. With object-oriented technology, functionality can be easily found, extracted and reused. Object-oriented technology places greater emphasis on the analysis and design of the system rather than on the system integration and integration testing. This shifted emphasis gives both the customer and the development team a better understanding of the product being built. Using functional decomposition, only the high level analysis is accessible to customers and management. They must take on faith the connection between the analysis (in terms of application objectives) and the more detailed analysis and design (in terms of programming constructs). Object-oriented analysis and design allows a deeper view: objects model known entities [Fayad, Laitinen 98].

Software is inherently complex. Brooks observers that "software artifacts are more complex for their size than perhaps any other human construct, and the complexity of software is an essential property, not an accidental one" [Brooks 95]. Understanding software is consequently difficult, not only because the software itself is complex, but also because the process of understanding requires knowledge from a variety of sources. Studies of expert programmers for conventional languages have shown that knowledge is not organized simply around syntax, but in larger conceptual structures such as algorithms, data structures and idioms and also in plans that indicate steps necessary to fulfill a particular goal. Reengineering of software is not normally effective unless some automated tool support can be deployed to support the process [Sommerville '96]. It is the purpose of this research to propose and implement techniques that can be used for automating the program comprehension and reengineering process. Although it is unlikely that such architectural restructuring activity can ever be completely automated, it is the goal of this research to aid assisted program reengineering by designing a tool that can suggest the potential modules and the hierarchical arrangement between them for the restructurer to verify. Although the techniques described herein may be used generally for the stated purpose with a variety of source and target languages[1], we will particularly aim at the problem of transforming the structured architecture of legacy C code into object-oriented architecture because there is a plethora of large-scale legacy systems written in C that are actively being used [Siff 98].

Organization of the Thesis

The remaining chapters of this thesis are organized as follows:

In chapter 2, we present a primer for mathematical concept analysis by discussing the notions of concept and concept lattice. We introduce a bottom-up algorithm for constructing a concept lattice, and demonstrate how a concept lattice can be used to visualize a system.

In chapter 3, we discuss in detail the application of concept analysis to the structured to object-oriented restructuring problem. We demonstrate how a concept lattice can be used to visualize a software system, and we also present an algorithm to partition such a concept lattice into modules (or classes).

Chapter 4 discusses methods to identify class hierarchies in a structured software system. We introduce some programming idioms commonly used to emulate object-oriented features such as inheritance and dynamic method lookup. We also present inference rules used in capturing these idioms.

[1] Source language that supports structured programming and target language that supports object-oriented programming

Chapter 5 presents a survey of some of the related work done in the field of software reengineering. Here we compare our work with the others, and identify our contribution to automatic program understandıng and software reengineering.

Chapter 6 dıscusses the future directions that this research may take.

Chapter 7 summarizes this thesis and presents some conclusions.

Chapter 2

MATHEMATICAL CONCEPT ANALYSIS PRIMER

Introduction

Garrett Birkhoff [Birkhoff 73], who proved that for every binary relation between certain objects and attributes a lattice could be constructed, laid the mathematical foundation for concept lattices. Based on his work, Rudolph Wille and Bernhard Ganter presented techniques for data analysis [Ganter, Wille 96], [Snelting 96]. In this chapter, we will show that this theory of concept lattices offers promising approaches, not only to the visualization of old software, but also to its automatic restructuring. Using concept analysis, data can be visualized as a conceptual model that offers meaningful and comprehensible interpretation of the system. This conceptual model can be reengineered into units that represent formal abstractions of the concepts of human thought. The mathematics required for the concept analysis has been taken directly from lattice theory [Ganter, Wille 96], [Szasz 64]. Our goal is to apply this theory to the modeling and reengineering of software. This chapter will introduce the notions of concept analysis and lattice theory to better understand its application to the structured to object-oriented reengineering problem.

Basic Notions of Concept Analysis

This section lays a primer for concept analysis using the example classification shown in figure 2.11. The table in figure 2.11 relates some vehicles of interest with some features of interest. For instance, from the first entry in the table, we can see that Camry is a sedan, is 2WD, and Japanese.

	Sedan	**Sports**	**SUV**	**2WD**	**4WD**	**Luxury**	**Japanese**
Camry	X			X			X
740iL	X			X		X	
Passat	X			X		X	
X5			X		X	X	
A6	X				X	X	
CCar1		X			X	X	
CCcar2		X			X	X	
ML320			X		X	X	
Maxima	X			X			X

Figure 2.11: Classification of few vehicle models

In other words, this context defines a relationship between the set of vehicles (objects)

O=`{Camry,740iL,Passat,X5,A6,CCar1,CCar2,ML320,Maxima}`,

and the set of features (attributes)

A=`{Sedan,Sports,SUV,2WD,4WD,Luxury,Japanese}`.

The corresponding visualization of the classification is shown in figure 2.12.

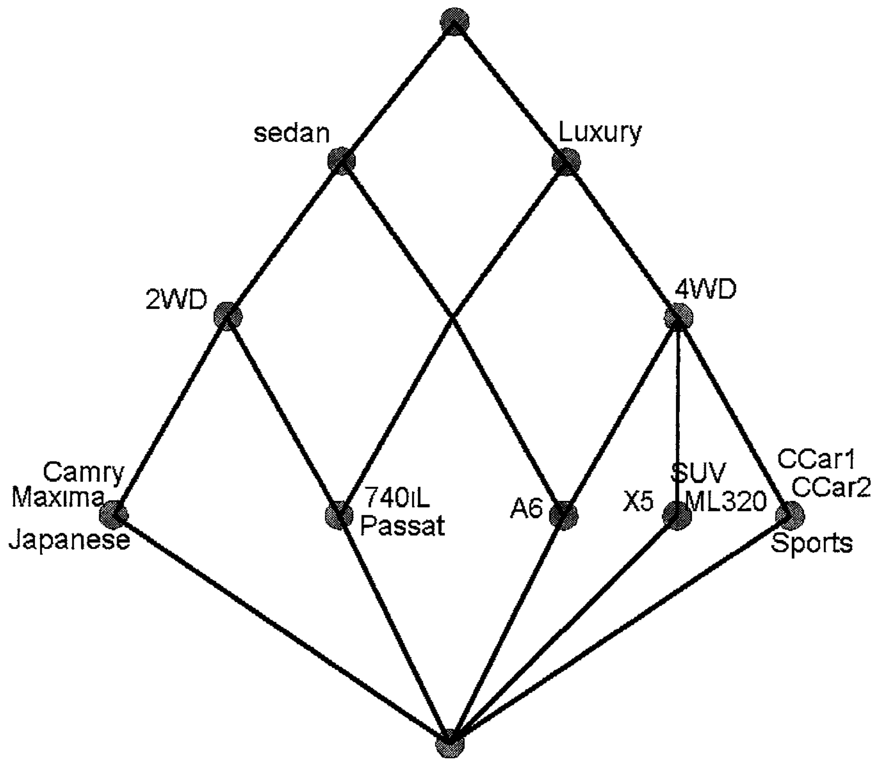

Figure 2.12: Abstraction of the classification in figure 2.11

As we will see later, the lattice offers insight that is not obvious from the table like:

1. A6 and ML320, are both 4WD
2. Any vehicle that is 4WD is also a luxury vehicle.
3. Any vehicle that is a 2WD is a sedan.

To understand the basics of concept analysis a few definitions are in order. A triple (O, A, R) is called a context if O and A are finite sets of objects and attributes respectively, and, $R \subseteq O \times A$ is a binary relation between O and A.

For a set of objects $X \subseteq O$, the set of common attributes is denoted by

$\sigma(X) = \{a \in A \mid \forall o \in X : (o, a) \in R\}$

As an instance from the classification in figure 2.11, for a set of objects

X={Camry, 740iL, Passat, Maxima}, the set of common attributes is

$\sigma(X)$ = {Sedan, 2WD}.

Similarly, for a set of attributes $Y \subseteq A$, the set of common objects is denoted by

$\tau(Y) = \{o \in O \mid \forall a \in A : (o,a) \in R\}$

For example, for the set of attributes Y={4WD, Luxury}, the set of common objects

is $\tau(Y)$ = {X5, A6, CCar1, CCar2}.

These operators, σ and τ, satisfy the following rules:

For all $A, A1, A2 \subseteq O$, and all $B, B1, B2 \subseteq A$

1 $A1 \subseteq A2 \Rightarrow \sigma(A2) \subseteq \sigma(A1)$ and

$B1 \subseteq B2 \Rightarrow \tau(B2) \subseteq \tau(B1)$

For example, for the following two object sets from the vehicle classification example:

$A1$={Camry}, and $A2$={Camry, 740iL, Passat}, with

$A1 \subseteq A2$,

$\sigma(A2) \subseteq \sigma(A1)$, as $\sigma(A2)$={Sedan, 2WD}, and

$\sigma(A1)$ = {Sedan, 2WD, Japanese}

Conversely, for the following two attribute sets

$B1$={Sedan, 2WD}, and $B2$={Sedan, 2WD, Japanese} with

$B1 \subseteq B2$, $\tau(B2) \subseteq \tau(B1)$, as $\tau(B2)$={Camry, Maxima}, and

$\tau(B1)$ = {Camry, 740iL, Passat, Maxima}

2. $A \subseteq \tau(\sigma(A))$ and $\sigma(A) = \sigma(\tau(\sigma(A)))$, and

$B \subseteq \sigma(\tau(B))$ and $\tau(B) = \tau(\sigma(\tau(B)))$

For example, for A={Camry},

$\sigma(A)=$ {Sedan,2WD,Japanese}, and

$\tau(\sigma(A))=$ {Camry,Maxima}

And, $A \subseteq \tau(\sigma(A))$ as required.

And since, $\sigma(\tau(\sigma(A)))=$ {Sedan,2WD,Japanese}, $\sigma(A)=\sigma(\tau(\sigma(A)))$

With similar examples, we can illustrate the converse: $B \subseteq \sigma(\tau(B))$ and

$\tau(B)=\tau(\sigma(\tau(B)))$

3. $A \subseteq \tau(B) \Leftrightarrow B \subseteq \sigma(A)$

For A={740iL,Passat}, B={Luxury}, since

$\tau(B)$ = {740iL,Passat,X5,A6,CCar1,CCar2}, we have, $A \subseteq \tau(B)$.

And also, with $\sigma(A)$={Sedan,2WD,Luxury}, we have, $B \subseteq \sigma(A)$.

Galois Connection

For two sets *S* and *T*, if there exists a single valued mapping σ of *S* into T, and a single valued mapping τ of *T* into *S*, it is said that the pair of mappings σ and τ establish a Galois connection between the sets *S* and *T*.

The mappings

$$\sigma : 2^{O} \rightarrow 2^{A} \text{ and } \tau : 2^{A} \rightarrow 2^{O}$$

form a galois connection and can be characterized by the condition:

for all $A, A1, A2 \subseteq O$, and all $B, B1, B2 \subseteq A$

$$A1 \subseteq A2 \Rightarrow \sigma(A2) \subseteq \sigma(A1)$$

and

$$B1 \subseteq B2 \Rightarrow \tau(B2) \subseteq \tau(B1)$$

Concept

A pair *(X, Y)* is a concept of the context *(O, A, R)*, if and only if

$$X \subseteq O,\ Y \subseteq A,\ X = \tau(Y) \text{ and } Y = \sigma(X)$$

For instance, the pair

X= {`Camry, 740iL, Passat, Maxima`} and

Y= {`Sedan, 2WD`} form a concept.

For such a context *(X, Y), X* is called the extent of the concept and *Y* is called its intent. In other words, the extent covers all the objects (or entities) for the concept, and the intent covers all the attributes (or properties) of the objects under consideration. For any given context *c,* we define the operators $ext(c)$ which gives the extent of the concept *c,* and $\operatorname{int}(c)$, which gives the intent of the concept *c.*

The concepts of a given context are naturally ordered by the subconcept-superconcept relation defined by

$$(A1, B1) \le (A2, B2) \Leftrightarrow A1 \subseteq A2,$$

or equivalently,

$$(A1, B1) \le (A2, B2) \Leftrightarrow B2 \subseteq B1$$

With these notions defined, we can proceed to define the concept lattice as an ordered set of all formal concepts of (O, A, R). Other notions of interest are infimum and supremum, given formally by:

$$\bigwedge_{i\in I}(Xi, Yi) = \left(\bigcap_{i\in I} Xi, \sigma\left(\tau\left(\bigcup_{i\in I} Yi\right)\right)\right) \quad \text{and}$$

$$\bigvee_{i\in I}(Xi, Yi) = \left(\tau\left(\sigma\left(\bigcup_{i\in I} Xi\right)\right), \bigcap_{i\in I} Yi,\right)$$

Informally, the infinum of two concepts is the greatest common subconcept of those concepts, and is found by intersecting their extents and joining the intents of the objects of the intersection. Consider the lattice in figure 2.12. Those nodes that factor out the common objects are the infima. Thus, the fact that A6 is both a sedan and 4WD, is denoted by the node labeled A6.

Analogously, the supremum of two concepts is the smallest common superconcept of those concepts, and is found by intersecting their intents and joining the objects that have the attributes found in the intersection [Snelting 96]. In the lattice in figure 2.12, the nodes that factor out the common attributes are the suprema. For example, both A6 and ML320 are 4WDs.

Constructing the concept lattice

For constructing the lattice, a standard lattice construction algorithm can be used [Godin, Alaoui 95], [Snelting 96], [Siff 98]. The algorithm may employ a bottom up or a top down approach. The algorithm we employ uses a bottom-up approach, in that the atoms (or the smallest concepts of the lattice) are computed first, and then the others are computed as the suprema of the existing ones, this being an iterative process. The concepts represent the nodes in the lattice.

Remember that for a concept *(X, Y),* $Y = \sigma(X)$ and $X = \tau(Y)$.

Hence, the algorithm can iterate over *O*, and for each $o \in O$, it can compute an atomic concept *c* such that *int*(*c*) is σ(*o*), and *ext*(*c*) is then τ(*int*(*c*)) or $\tau(\sigma(o))$. Figure 2.13 shows the computation of all the atomic concepts in the vehicle classification. Figure 2.14 lists the atomic concepts computed.

$\tau(\sigma(\{\texttt{Camry}\})) = \tau(\{\texttt{Sedan, 2WD,Japanese}\}) = (\{\texttt{Camry, Maxima}\})$

$\tau(\sigma(\{\texttt{740iL}\})) = \tau(\{\texttt{Sedan, 2WD,Luxury}\}) = (\{\texttt{740iL,Passat}\})$

$\tau(\sigma(\{\texttt{Passat}\})) = \tau(\{\texttt{Sedan, 2WD,Luxury}\}) = (\{\texttt{740iL,Passat}\})$

$\tau(\sigma(\{\texttt{X5}\})) = \tau(\{\texttt{SUV, 4WD,Luxury}\}) = (\{\texttt{X5,ML320}\})$

$\tau(\sigma(\{\texttt{ML320}\})) = \tau(\{\texttt{SUV, 4WD,Luxury}\}) = (\{\texttt{X5,ML320}\})$

$\tau(\sigma(\{\texttt{CCar1}\})) = \tau(\{\texttt{Sports, 4WD,Luxury}\}) = (\{\texttt{CCar1,CCar2}\})$

$\tau(\sigma(\{\texttt{CCar2}\})) = \tau(\{\texttt{Sports, 4WD,Luxury}\}) = (\{\texttt{CCar1,CCar2}\})$

$\tau(\sigma(\{\texttt{A6}\})) = \tau(\{\texttt{Sedan, 4WD,Luxury}\}) = (\{\texttt{A6}\})$

Figure 2.13: Computing the atomic concepts in the vehicle classification

Concept C0 $= (\{\texttt{Camry,Maxima}\}, \{\texttt{Sedan, 2WD,Japanese}\})$

Concept C1 $= (\{\texttt{740iL, Passat}\}), (\{\texttt{2WD,Luxury,Sedan}\})$

Concept C2 $= (\{\texttt{ML320,X5}\}), (\{\texttt{SUV, 4WD, Luxury}\})$

Concept C3 $= (\{\texttt{CCar1,CCar2}\}), (\{\texttt{Sports, 4WD,Luxury}\})$

Concept C4 $= (\{\texttt{A6}\}), (\{\texttt{Sedan, 4WD, Luxury}\})$

Figure 2.14: Atomic concepts in the vehicle classification

For a concept c, operation $\mathrm{int}(c)$ requires O($|A|$) time. We can therefore compute the intents of all the atoms in O($|O| \times |A|$) time. The next step is to compute the extents for these concepts. Computing the extent involves applying the τ operator. The time complexity for that is O ($|O| \times |A|$). For a context of size $n \times n$, the worst-case number of elements in the lattice will be polynomial in n: i.e. O(n^3). Typically, however, lattices with contexts of size $n \times n$ *have* O(n^2) or even O(n) elements. Hence, the typical run-time for the construction of the entire lattice is O(n^3) [Snelting 96].

A queue may be used as a worklist for building the lattice upon these atoms. Initially, the queue is populated with pairs of atomic concepts, such that, in no pair of concepts (C1, C2), is C1 ≤ C2 or C2 ≥ C1; i.e. no concept in a pair is a subconcept of the other. For the vehicle classification, the initial queue will be:

```
Queue = {(C0,C1),(C0,C2),(C0,C3),(C0,C4),(C1,C2),
           (C1,C3),(C1,C4),(C2,C3),(C2,C4),(C3,C4)
        }
```

By dequeuing a pair in the queue at a time, the supremum of the two concepts is created. For example, the pair (C1, C2) yields the supremum computed as:

$$\begin{aligned} \mathtt{C0} \vee \mathtt{C1} &= (\tau(\mathrm{int}(\mathtt{C0}) \cap \mathrm{int}(\mathtt{C1})),\ \mathrm{int}(\mathtt{C0}) \cap \mathrm{int}(\mathtt{C1})) \\ &= (\tau\,(\mathtt{2WD,Sedan}), \{\mathtt{2WD,Sedan}\}) \\ &= (\{\mathtt{740iL,Camry,Maxima,Passat}\}, \{\mathtt{2WD,Sedan}\}) \end{aligned}$$

If this new concept does not already exist, new pairs are generated by pairing the existing concepts with this newly generated supremum, again seeing that there is no subconcept-superconcept relationship between the two concepts in a pair. This iterative process is continued until the queue is empty. Figure 2.15 shows the computation of the concept lattice for the vehicles classification.

```
    C0 = ({Camry,Maxima},{2WD,Sedan,Japanese})
    C1 = ({740iL,Passat},{2WD,Luxury,Sedan})
    C2 = ({ML320,X5},{SUV,4WD,Luxury})
    C3 = ({CCar1,CCar2},{Sports,4WD,Luxury})
    C4 = ({A6},{Sedan,4WD,Luxury})
 Queue = {(C0,C1),(C0,C2),(C0,C3),(C0,C4),(C1,C2), (C1,C3), (C1,C4),
          (C2,C3),(C2,C4), (C3, C4)
         }
```

We dequeue the first pair from the queue and compute a supremum for the concepts in that pair

$C0 \vee C1 = C5 = (\{740iL, Camry, Maxima, Passat\}, \{2WD, Sedan\})$

This new concept C5 is now paired with those others with which C5 does not form a subconcept or superconcept relation. All such pairs are added to the queue.

```
Queue = {(C0,C2),(C0,C3),(C0,C4),(C1,C2),(C1,C3),(C1,C4),(C2,C3),
          (C2,C4),(C3,C4), (C2,C5),(C3,C5),(C4,C5)
         }
```

$C0 \vee C2 = (\{all\ objects\}, \phi\})$

This pair yields a concept with an object set *X=O*, and an empty attribute set. This corresponds to the highest concept in the lattice. This concept can not have any superconcept, and hence cannot yield a supremum when paired with any other concept. Therefore, no new pairs are formed.

```
 Queue = {(C0,C3),(C0,C4),(C1,C2), (C1,C3), (C1,C4), (C2,C3),
          (C2,C4),(C3,C4), (C2,C5),(C3,C5),(C4,C5)
         }
```

$C0 \vee C3 = (\{all\ objects\}, \phi\})$

The pair `(C0,C3)` also yields the highest node as their supremum. Note that this is the unique least common upper bound for the concepts in the pair.

```
 Queue = {(C0,C4),(C1,C2), (C1,C3), (C1,C4), (C2,C3), (C2,C4),
          (C3,C4), (C2,C5),(C3,C5),(C4,C5)
         }
```

```
C0 ∨ C4 = C6 = ({Camry,Maxima,740iL,Passat,A6},{Sedan})
```

C6 represents the unique least upper bound for the concepts C0 and C4. C6 is now paired with those that are not ordered with respect to C6, and all such pairs are added to the queue. This iterative process is continued until the queue is empty.

```
 Queue = {(C1,C2), (C1,C3), (C1,C4), (C2,C3), (C2,C4),
          (C3,C4), (C2,C5),(C3,C5),(C4,C5),(C2,C6),(C3,C6)
         }
C1 ∨ C2 = C7 = ({740iL,A6,CCar1,CCar2,ML320,Passat,X5},{Luxury})
 Queue = {(C1,C3), (C1,C4), (C2,C3), (C2,C4), (C3,C4), (C2,C5),
          (C3,C5),(C4,C5),(C2,C6),(C3,C6),(C0,C7),(C5,C7),(C6,C7)
         }
C1 ∨ C3 = C7
 Queue = {(C1,C4), (C2,C3), (C2,C4), (C3,C4), (C2,C5),
          (C3,C5),(C4,C5),(C2,C6),(C3,C6),(C0,C7),(C5,C7),(C6,C7)
         }
C1 ∨ C4 = C7
 Queue = {(C2,C3), (C2,C4), (C3,C4), (C2,C5),
          (C3,C5),(C4,C5),(C2,C6),(C3,C6),(C0,C7),(C5,C7),(C6,C7)
         }
C2 ∨ C3 = C8 = ({A6,CCar1,CCar2,ML320,X5},{4WD,Luxury})
 Queue = {(C2,C4), (C3,C4), (C2,C5),(C3,C5), (C4,C5),
          (C2,C6),(C3,C6),(C0,C7),(C5,C7),(C6,C7),(C0,C8),
          (C1,C8),(C5,C8), (C6,C8),(C7,C8)
         }
C2 ∨ C4 = C8
 Queue = {(C3,C4), (C2,C5),(C3,C5), (C4,C5), (C2,C6), (C3,C6),
          (C0,C7),(C5,C7),(C6,C7),(C0,C8),(C1,C8),(C5,C8),
          (C6,C8),(C7,C8)
         }
C3 ∨ C4 = C8
 Queue = {(C2,C5),(C3,C5), (C4,C5), (C2,C6), (C3,C6),
          (C0,C7),(C5,C7),(C6,C7),(C0,C8),(C1,C8),
          (C5,C8),(C6,C8),(C7,C8)
         }
C2 ∨ C5 = ({all objects}, ϕ}
 Queue = {(C3,C5), (C4,C5), (C2,C6), (C3,C6),(C0,C7),(C5,C7),
          (C6,C7) (C0,C8),(C1,C8),(C5,C8),(C6,C8),(C7,C8)
         }
C3 ∨ C5 = ({all objects}, ϕ}
 Queue = {(C4,C5), (C2,C6), (C3,C6),(C0,C7),(C5,C7),(C6,C7)
          (C0,C8),(C1,C8),(C5,C8),(C6,C8),(C7,C8)}
C4 ∨ C5 = C6
```

```
 Queue = {(C2,C6), (C3,C6),(C0,C7),(C5,C7),(C6,C7)
          (C0,C8),(C1,C8),(C5,C8),(C6,C8),(C7,C8)}
C2∨C6 = ({all objects}, ϕ}
 Queue = {(C3,C6),(C0,C7),(C5,C7),(C6,C7)
          (C0,C8),(C1,C8),(C5,C8),(C6,C8),(C7,C8)}
C3∨C6 = ({all objects}, ϕ}
 Queue = {(C0,C7),(C5,C7),(C6,C7)
          (C0,C8),(C1,C8),(C5,C8),(C6,C8),(C7,C8)}
C0∨C7 = ({all objects}, ϕ}
 Queue = {(C5,C7),(C6,C7)
          (C0,C8),(C1,C8),(C5,C8),(C6,C8),(C7,C8)}
C5∨C7 = ({all objects}, ϕ}
 Queue = {(C6,C7)
          (C0,C8),(C1,C8),(C5,C8),(C6,C8),(C7,C8)}
C6∨C7 = ({all objects}, ϕ}
 Queue = {(C0,C8),(C1,C8),(C5,C8),(C6,C8),(C7,C8)}
C0∨C8 = ({all objects}, ϕ}
 Queue = {(C1,C8),(C5,C8),(C6,C8),(C7,C8)}
C1∨C8 = C7
 Queue = {(C5,C8),(C6,C8),(C7,C8)}
C5∨C8 = ({all objects}, ϕ}
 Queue = {(C5,C8),(C6,C8),(C7,C8)}
C6∨C8 = ({all objects}, ϕ}
 Queue = {(C7,C8)}
C7∨C8 = C7
 Queue = {ϕ}
```

Concept lattices can be represented as usual lattice diagrams, but for large lattices it would be messy to label each concept by its extent and intent. Hence, the idea of reduced labeling is used [Ganter, Wille 96]. With reduced labeling, each attribute and each object is entered exactly once in the lattice diagram. A concept is labeled with an object if it is the smallest concept subsuming that object, and analogously, a concept is labeled with an attribute if it is the largest concept subsuming that attribute. Consequently, for a concept in a lattice, the set of attributes is the union of all the attributes occurring in the labels for its superconcepts and similarly, the objects set is the union of all the objects occurring in the labels for its subconcepts. For instance, in the lattice for the vehicle classification, the node labeled `2WD`, is the concept:

`({Camry,740iL,Passat,Maxima}, {2WD,Sedan})`

`Sedan`, being upwards from `2WD` and `{Camry,740iL,Passat,Maxima}`, being downwards from `2WD`.

Properties of a Concept Lattice

1. The lattice is the smallest lattice compatible with the context table and can thus be order embedded into any other lattice compatible with the table. A concept lattice is minimal.
2. Attributes for a given concept occur upwards (in the superconcepts) in the lattice. Common attributes are thus maximally factored out. Similarly, objects occur downwards (in the subconcepts) in the lattice and are thus maximally factored out. Hence a concept lattice can be said to be maximally factorized [Wille, Ganter 96].

Summary

In this chapter, we have introduced lattice theory and concept analysis. We have discussed how a concept lattice can be used to model a context with a binary relation between a set of objects and a set of attributes. In the following chapter, we will discuss the application of concept analysis in modeling a software system.

We will,

1. Make a conceptual model of a structured program into a concept lattice with reduced labeling,
2. Decompose that lattice into potential modules based on the data extracted from the source program.

Chapter 3

MODULARIZATION OF SOFTWARE USING CONCEPT ANALYSIS

Introduction

In the previous chapter, we showed how a concept lattice could be used to model a system that exhibits a binary relationship between a set of objects and a set of attributes. In this chapter, we demonstrate how concept analysis can be used to visualize a software system as a concept lattice. A concept lattice provides a conceptual model that offers a meaningful and comprehensible interpretation of the system. We present techniques for the analyses and automatic modularization of software based on this model and some semantic information extracted from the system.

Objects and Attributes

We define a module as a unit that is directly usable, discrete, and ideally separable from its original context and usable in other contexts. A component may be viewed as a "static abstraction with plugs" [Nierstrasz 95]. By static we mean that a software module is independent of applications. By abstraction we mean that a module encapsulates software and only provides "plugs" as well-defined ways of interaction. We should therefore think about the data not only in terms of how to represent it, but also in terms of how it is to be used. As such, a module should combine data representation and methods for manipulating that data as a neat package. The methods should provide the "plugs," the data being hidden otherwise.

A program *P* in the C language (and in most languages) can be divided into the following domains: Global-variable, User-defined-data-type, Macro, Function, and file domain [Chen, Nishimoto, Ramamoorthy 90]. Of these, global variables represent data (local variables are not of importance to us); functions and macros represent the operations on the data. Functions and the macros may collectively represent objects, *O*, and data may represent attributes, *A*. The relation *R* is as expected: a procedure $o \in O$, is related with a data element $a \in A$, if and only if it uses that element. By building a concept lattice, we can thus create a conceptual model for a given software system in terms of the procedures as objects and data elements as attributes, and relate an object to a data element if and only if it uses that data element. We will demonstrate that this conceptual model can be effectively used to identify interference caused due to coupling, eliminate such interference, and decompose the lattice into meaningful modules.

User defined compound data types provide a good starting point in the detection of modules. In the C programming language, a structure provides a convenient means of keeping related information together. A structure is thus a "module" of data. Functions accessing the fields of such aggregate types are thus potential members of these structures. We therefore consider a relationship of the form "`uses_struct_x_fields`" for a function `f` accessing the fields of `struct x`.

Global variables can be used by any piece of code and, as such, are a source of coupling amongst functions. With our definition of a module, it is reasonable to have objects correspond to procedures and attributes correspond to the usage of global variables and user defined data-type fields by these procedures. A relationship capturing this information might therefore be of the form "`uses_global_variable_a`" or "`uses_struct_x_fields`". Many sets of functions being related to a set of attributes represents coupling between the sets of functions and is displayed as such in the concept lattice. Analogously, one function set being related to different attribute sets is a source of interference and is also visible in a concept lattice. The identification and elimination of such sources of coupling and interference is an important step in the formation of modules. As will be demonstrated later, the information required to resolve such sources can be extracted to a large extent from the program source. If needed, more information may be obtained with user assistance. The choice of objects and attributes in visualizing a system as a concept lattice may vary. Some legacy systems are monolithic in nature: they lack structure, all tasks being contained within one function. In such a case objects may correspond to program slices rather than procedures [Siff 98].

A generic hash table package

Appendix A3.21 is the C code for a generic hash table package, the user of which is expected to type define `Record` and to provide a function `Equal()` to check equality of two `Records`. It has twelve functions and four user defined compound data types (structs). The functions form the object set O, and the struct field usage information forms the attribute set A. Therefore,

O={`equal,print_data,initnode,initlist,list_head_insert,`
`find_element,size,print,inithashtable,find_record,`
`insert_record`}

A ={`uses_record_fields,uses_node_fields,uses_list_fields,`
`uses_hashtable`}

The relation R is as expected: a function $o \in O$, is related with attribute $a \in A$, if and only if it uses the fields of that structure. Figure 3.21 is the corresponding context table and figure 3.22 is the resultant concept lattice.

	Uses_node_fields	**Uses_list_fields**	**Uses_table_fields**
InitNode	X		
InitList		X	
List_head_insert	X	X	
Find_element	X	X	
Delete_element	X	X	
Size		X	
Print	X	X	
Destroy_list	X	X	
InitHashtable			X
Lookup			X
Insert			X
Destroy_table			X

Figure 3.21: Context table for the Hash table package

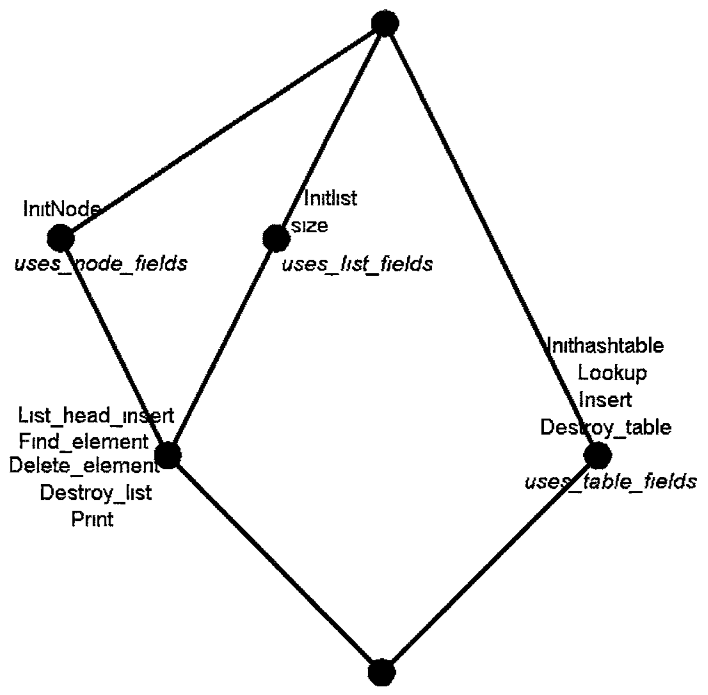

Figure 3.22: Concept lattice for the hash table package

The atomic concepts are computed as follows:

$\tau(\sigma(\{\text{InitNode}\})) = \tau(\{\text{uses\_node\_fields}\})$
$= \{\text{Initnode, List\_head\_insert, Find\_element, Delete\_element, Print, Destroy\_list}\}$

$\tau(\sigma(\{\text{InitList}\})) = \tau(\{\text{uses\_list\_fields}\})$
$= \{\text{InitList, List\_head\_insert, Find\_element, Delete\_element, Size, Print, Destroy\_list}\}$

$\tau(\sigma(\{\text{List\_head\_insert}\})) = \tau(\{\text{uses\_node\_fields, uses\_list\_fields}\})$
$= \{\text{List\_head\_insert, Find\_element, Delete\_element, Print, Destroy\_list}\}$

$\tau(\sigma(\text{Find\_element})) = \tau(\{\text{uses\_node\_fields, uses\_list\_fields}\})$
$= \{\text{List\_head\_insert, Find\_element, Delete\_element, Print, Destroy\_list}\}$

$\tau(\sigma(\{\text{Delete\_element}\})) = \tau(\{\text{uses\_node\_fields, uses\_list\_fields}\})$
$= \{\text{List\_head\_insert, Find\_element, Delete\_element, Print, Destroy\_list}\}$

$\tau(\sigma(\{\text{Size}\})) = \tau(\{\text{uses\_node\_fields}\})$
$= \{\text{Initnode, List\_head\_insert, Find\_element, Delete\_element, Print, Destroy\_list}\}$

$\tau(\sigma(\text{Print})) = \tau(\{\text{uses\_node\_fields, uses\_list\_fields}\})$
$= \{\text{List\_head\_insert, Find\_element, Delete\_element, Print, Destroy\_list}\}$

$\tau(\sigma(\{\text{Destroy\_list}\})) = \tau(\{\text{uses\_node\_fields, uses\_list\_fields}\})$
$= \{\text{List\_head\_insert, Find\_element, Delete\_element, Print, Destroy\_list}\}$

$\tau(\sigma(\{\text{InitHashtable}\})) = \tau(\{\text{uses\_table\_fields}\})$
$= \{\text{InitHashtable, Lookup, Insert, Destroy\_table}\}$

$\tau(\sigma(\{\text{Lookup}\})) = \tau(\{\text{uses\_table\_fields}\})$
$= \{\text{InitHashtable, Lookup, Insert, Destroy\_table}\}$

$\tau(\sigma(\{\text{Insert}\})) = \tau(\{\text{uses\_table\_fields}\})$
$= \{\text{InitHashtable, Lookup, Insert, Destroy\_table}\}$

$\tau(\sigma(\{\text{Destroy\_table}\})) = \tau(\{\text{uses\_table\_fields}\})$
$= \{\text{InitHashtable, Lookup, Insert, Destroy\_table}\}$

We have thus identified the following atomic concepts:

```
Concept 0 = ({InitNode, List_head_insert, Find_element,Delete_element,
             Print, Destroy_list}, {uses_node_fields})

Concept 1 = ({Initlist, List_head_insert, Find_element, Delete_element,
             Destroy_list, Print, Size},{uses_list_fields})

Concept 2 = ({List_head_insert, Find_element, Delete_element, Print,
             Destroy_list}, {uses_node_fields, uses_list_fields})

Concept 3=({InitHashtable, Lookup, Insert, Destroy_table},
            {uses_table_fields})
```

Note that C2 ≤ C0 (C2 is a subconcept of C0) and C2 ≤ C1 (C2 is a subconcept of C1).

Hence our queue will look like:

{ (C0, C1), (C0,C3), (C1,C3), (C2,C3) }

None of these pairs yields a new concept. Hence the resultant lattice in figure 3.2 contains only the four atomic concepts.

Patterns in a concept lattice

Before we devise an algorithm to identify sources of coupling and interference and to decompose a concept lattice into modules, the patterns in a lattice that are of interest to us need to be introduced. These patterns are illustrated in figures 3.31(a-c).

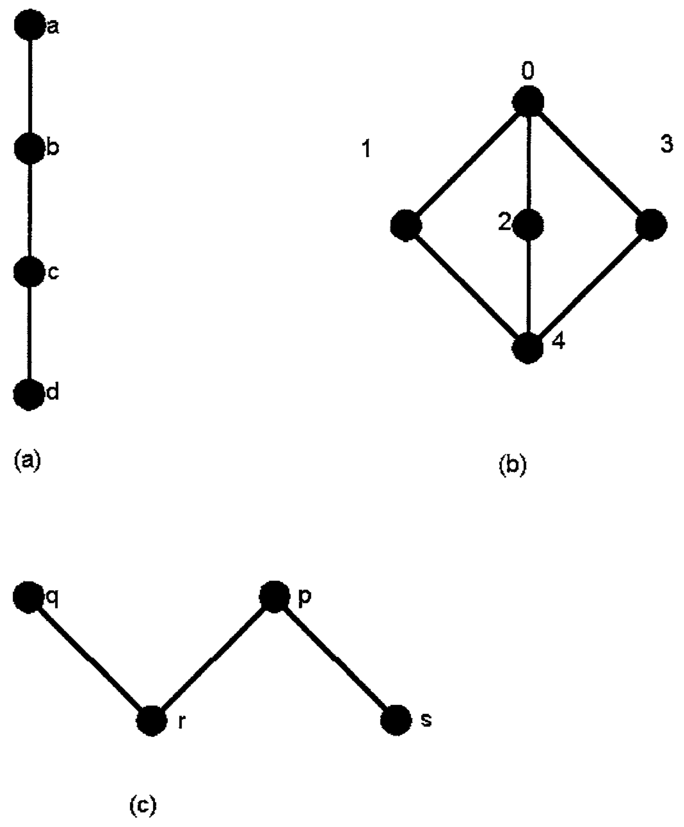

Figure 3.31: Patterns of interest in a lattice
a) Chain (b) Antichain (c) Infima, Suprema

Chain: a sequence of concepts in which a concept is a subconcept of the next one in the sequence. Such concepts run as a chain joining the bottom of a lattice to its top. Figure 3.31a shows such a chain of concepts

Anti-chain: a set of distinct chains. Figure 3.31b shows an anti-chain.

The patterns of interest in addition to these are suprema and infima, which we have already seen. Figure 3.31c shows p as the supremum of q and r, and r as the infimum of q and s.
We will also need to introduce some terminology from graph theory [Temperley 81] [Cormen, Leiserson, Rivest 90], [Tarjan 83], [Even 79]. An articulation point is a point in a graph, the removal of which, with all its incident lines, causes a graph to fall into two or more disconnected pieces. Infima in a lattice are normally these articulation points that need to be identified and resolved in order to have a horizontally decomposable lattice. The operation of contraction of a line involves not only removing it from the graph, but also replacing its two ends by a single one, all other lines incident on the two ends being retained.

From the lattice for the hashtable package, we observe that concept 2 is the infinum of concept 0 and concept 1. Again, an infimum of two concepts is the greatest common subconcept of those concepts. As such, it represents the data being shared by functions of two concepts yielding that infimum (concept 0 and concept 1 in this case). Hence an infimum represents interference in the form of coupling. If this infimum is suppressed, we will have a perfect anti-chain of three concepts that cleanly represent the three modules that the concept lattice may be decomposed into, viz. node, list and hashtable, encapsulating all the related data and functions neatly. In other words, the suppression of the interference renders the lattice horizontally decomposable into three partitions. Several algorithms have been established for decomposition of lattices [Wille,Ganter 94] , [Funk, Lewein, Snelting]. We present a horizontal decomposition algorithm that uses semantic information extracted from the source program.

Algorithm for the horizontal decomposition of a concept lattice

Horizontally Decomposable Lattice

Let *L1, L2...Ln* be lattices. The horizontal sum of these lattices is

$\sum_{i=1}^{n} Li = \{T, B\} \cup \bigcup_{i=1}^{n} Li \setminus \{Ti, Bi\}$, where, $T \geq x, B \leq x$ for all $x \in \sum_{i=1}^{n} Li$ [Funk, Lewein, Snelting]. For instance, the lattice in figure 3.31b is horizontally decomposable. The *Li* are the disjoint summands. Conversely, a lattice *L* is horizontally decomposable, if it is a horizontal sum. But in practice, *L* might not readily have a horizontal sum. The coupling and interference in a system may produce suprema and infima that will need to be eliminated before the lattice can be decomposed into modules. Therefore, in case a lattice is not horizontally decomposable, it can be made so after the interferences have been removed. Although the top element is a supremum, it is not considered as a source of coupling unless it has a non-empty intent. Similarly, although the bottom element of the lattice represents an infimum, it is not considered as a source of interference unless it has a non-empty intent. To demonstrate the efficacy of our modularization algorithm, we treat a "tangled" version of the hash table package described in the previous section. We call this a tangled version because `Find_record` and `Insert_record` have been rewritten to directly access the fields of other structures. `Record` has been type defined to be of the type `struct record`, and the function `Equal()` has also been provided. This version is listed in Appendix A3.41. Figure 3.42 shows the context table for this version of the hash table package.

	Uses_Record_fields	**Uses_Node_fields**	**Uses_list_fields**	**Uses_hashtable**
Equal	X			
Print_data	X			
Initnode		X		
Initlist			X	
List_head_ins		X	X	
Find_element		X	X	
Size			X	
Print		X	X	
Inithashtable				X
Find_record	X	X	X	X
Insert_record	X			X

Figure 3.42: Context table for the tangled hash table

The corresponding concept lattice shown in figure 3.43a is based on the following atomic concepts:

Concept C0 = `({equal, print_data, find_rec, insert_rec},`
`{uses_record_fields})`

Concept C1 = `({init_node, list_head_insert, find_element,`
`delete_element, print},{uses_node_fields})`

Concept C2 = `({init_list,list_head_insert,find_element,`
`delete_element, size, print },`
`{uses_list_fields})`

Concept C3 = `({list_head_insert, find_element,`

```
            delete_element,print, find_record },
            {uses_node_fields, uses_list_fields})
```

Concept C4 = `({Init_hashtable, Find_record,Insert_record}, {Uses_hashtable})`

Concept C5 = `({Find_record}, {uses_record_fields, uses_node_fields, uses_list_fields, uses_hashtable})`

Concept C6 = `{(Insert_record}, (Uses_record_fields, uses_hashtable})`

It may be observed that the resultant lattice ın figure 3.43a is not horizontally decomposable. The lattice shows the three infima represented by the concepts C6, C3 and C5 (in a breadth-first order).

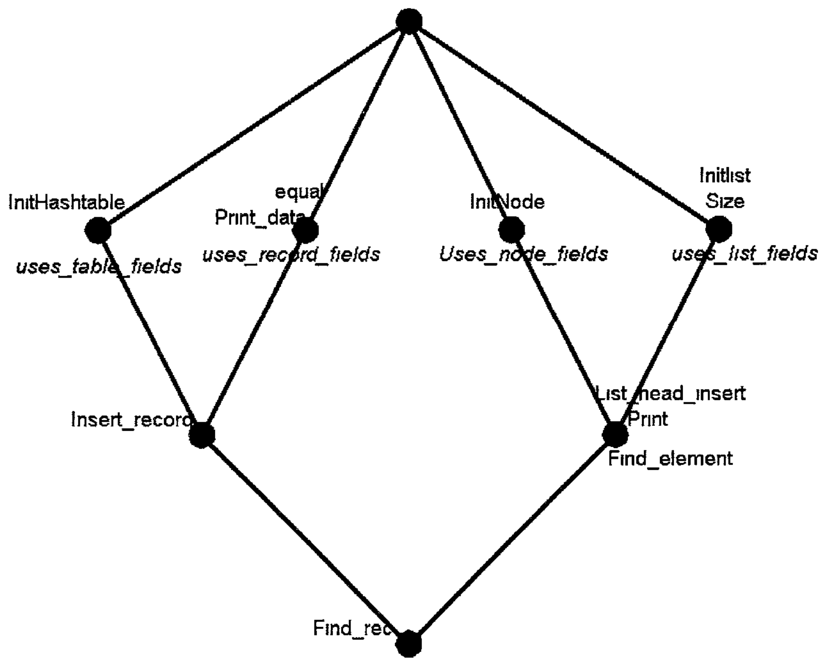

Figure 3.43a: Concept lattice for the “tangled” hash table

Infima that display interference are the $\wedge -$ reducible concepts of the lattice. This suggests that by shifting the objects of this concept upward in the lattice, the interference may be eliminated. Our next step is therefore to suppress these interferences. To compute the suitable superconcept to which these interfering objects may be relegated, we employ semantic information that can be obtained from the program. In addition, information may also be obtained with user assistance.

A breadth-first search of a concept lattice, starting at the top node, can be used to identify the infima and the suprema causing interference. The first supremum to be identified is the top node. Since it has an empty intent, it does not represent coupling. The next node of interest to be discovered in the breadth-first order is the node representing concept C6. Consider the one function `Insert_record()` in the extent of this concept. This function is causing the interference because it accesses the fields of the two structures `record`, and `hashtable`. However, from the source code, it is clear that `hashtable` struct displays a "has-a" relationship with the `list` struct; `list` struct shows a "has-a" relationship with `node` struct and `node` struct shows a "has-a" relationship with `record` struct. In other words, a `hashtable` instance contains a `list` instance that contains a `node` instance that contains a `record` instance. An object (`Insert_record`) that uses the fields of both, the contained structure (`record`) and the containing structure (`hashtable`) is more likely to belong to the module representing the containing structure. Therefore, the object `Insert_record`, in this case, is added to the extent of C4. We can now eliminate C6 and connect C0 and C4 directly to the subconcept of C6 i.e. C5. The resultant lattice is shown in figure 3.44b. If a vertex needs to be suppressed, all the edges from its direct parents are then connected to its children.

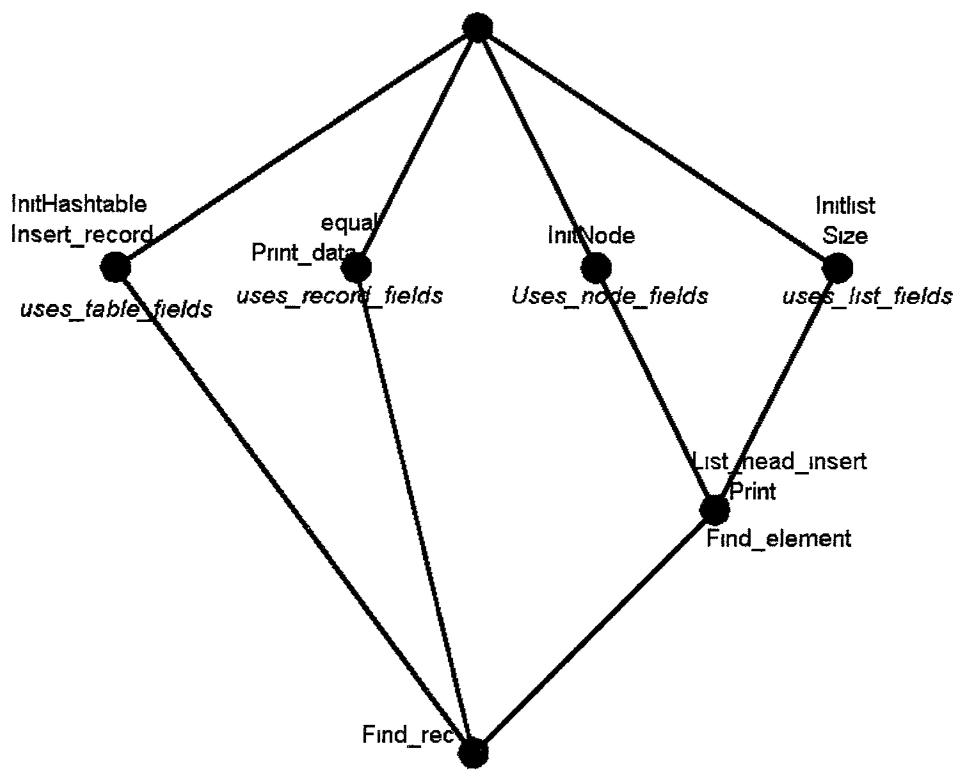

Figure 3.44b. Lattice after the suppression of the node representing concept C6

Therefore, the suppression of a vertex *v* leads to the creation of infima at most equal to the number of children of *v*. We process each of these infima until we reach the bottom node in one breadth-first traversal of the lattice. If an infimum *v* to be processed is also a supremum, we delegate all objects in its extent to the respective parents and connect *v* directly to top. All the children of *v* now become direct children of the parents of *v*. This process continues until the lattice is horizontally decomposable. The infimum identified next is the node representing concept C3. This concept produces interference because the functions in its extent ({`list_head_insert`,`find_element`,`delete_element`,`print`, `find_record`}) access the fields of both the `node` structure and the `list` structure. Since `node` is the contained structure and `list` is the containing structure, we delegate the functions in the extent of C3 to that of concept C2. After eliminating the node representing concept C3, we get the lattice shown in figure 3.43c.

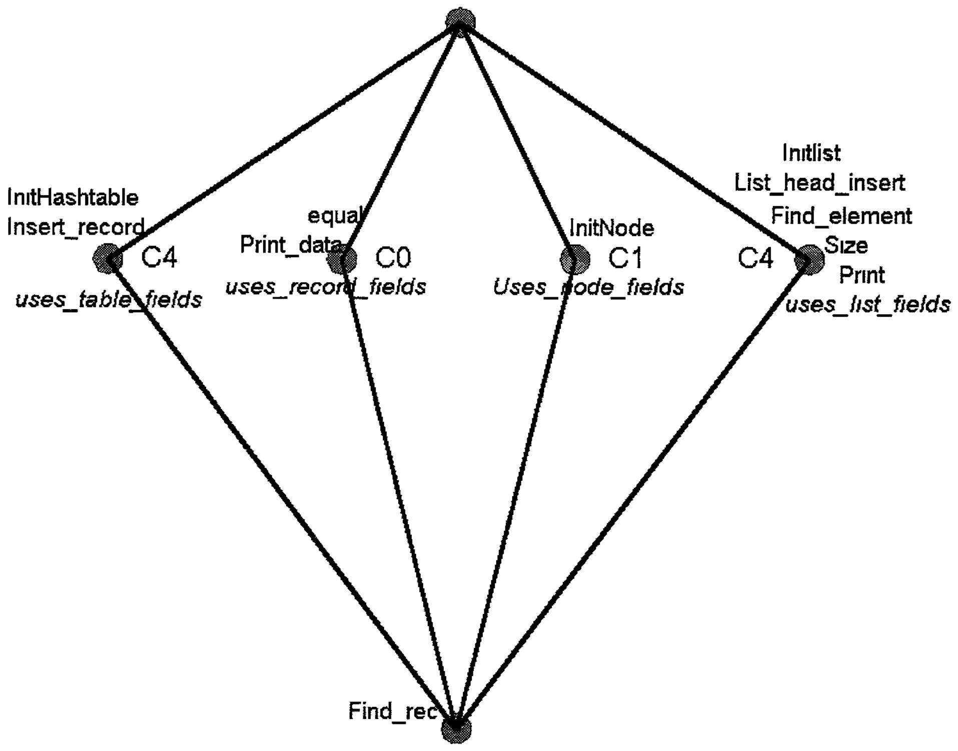

Figure 3.43c: The concept lattice after the elimination of concept C3

The next node of interest is bottom. This concept exhibits interference since the function in its extent (`find_rec`) accesses the members of all the structures. Again, with our knowledge of the relationship between these two structures, we can resolve that this sole object is more likely to belong to the node representing the hashtable i.e. C4. We therefore move `find_rec` to the extent of C4. Since this is the bottom node of the concept lattice, we do not suppress it. The resultant lattice in figure 3.43d is a horizontal sum of four antichains that represent the four meaningful modules identified in the system viz. `record`, `node`, `list` and `hashtable`.

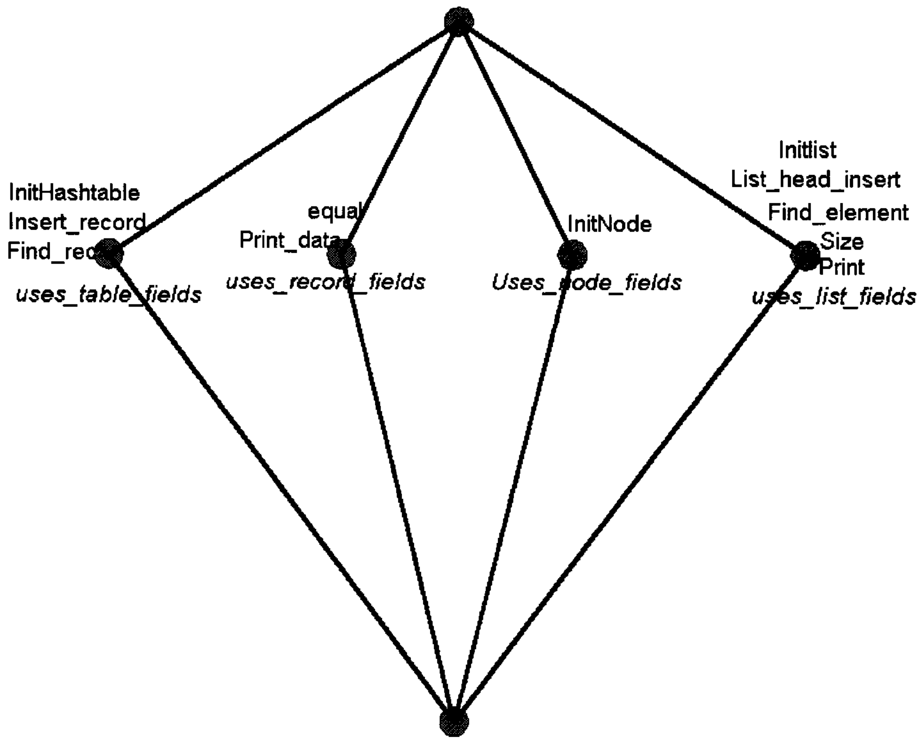

Figure 3.43d: The final horizontally decomposable lattice

The identified modules are tabulated in the table in figure 3.44.

Module	**Data members**	**Operations**

Record	char name[40]
	char SSN[9]
		Equal()
		Print_data()

Node	Record data Node* next	InitNode()
List	Node* head int length	InitList() List_head_insert() Find_element Delete_element Print

Hashtable	int tablesize
	List* Hashtable
		InitHashtable()
		Insert_record()
		Find_record()

Figure 3.44: Modules identified in the "tangled" hash table package

It is possible that there may not be enough system information available to resolve coupling and interference. Moreover, the system may have data in the form of global variables alone; such base modules as offered by user defined compound data types may not be available. Even in such cases, the algorithm will group functions and the data used by them into modules. Elimination of coupling and interference in this case may demand more user assistance. The following chapter presents a case study where our modularization algorithm is applied to a small-scale legacy system written in the C language. In the presence of both user defined compound datatypes and global variables, we show how restructuring can be carried out effectively by treating data from these two domains separately, as opposed to making a model with all the information to start with. We modularize the system based on a conceptual model formed with the relationship between functions and user defined compound datatypes and their global instances. It is then much easier to delegate the other global variables to these potential modules. The results obtained have been encouraging.

Because modularization is inherently subjective, it is not likely that this process can ever be made completely automatic. However, for existing systems with high entropy, there is an acute need for a tool that aids, as much as possible, in the understanding and restructuring process. Sommerville states that "reengineering software is not normally effective unless some automated tool support can be deployed to support the process" [Sommerville 96].

We observe that concept analysis, by providing a good method for analyzing software not only aids program restructuring, but also gives a detailed account of all dependencies due to coupling and may be used to limit the increase of entropy as software evolves.

Summary

In this chapter, we discussed the application of mathematical concept analysis to reverse engineering of software. Based on the visualization of software as a concept lattice, we described our algorithms for identifying meaningful modules from the concept lattice.

The next chapter presents a case study where our modularization technique has been applied to a computational-geometry library called *Voronoi*. The results obtained are encouraging.

Chapter 4

IMPLEMENTATION AND RESULTS

Introduction

Based on the work presented in this chapter we have implemented a prototype tool that

- Creates an abstraction of a given software system as a concept lattice, where the object set *O* is constituted by functions and macros, the attribute set *A* is constituted by global variables and user defined compound data types, and the relation R relates an object with an attribute if it uses that attribute. For an attribute that is a user defined compound data type, an object is related to it if and only if the object accesses its members.
- Identifies sources of coupling and interference in the system by analyzing the concept lattice and
- partitions the system into maximally cohesive and minimally coupled modules that encapsulate functions and data.

Figure 3.41 shows the general design of the tool. It contains the following parts:

- Program Information Extractor
- Concept Lattice Builder
- Modularizer
- GUI front end
- Lattice Drawing Tool

The Program Information Extractor: used to obtain information for generating the context table. We have not implemented the Program Information extractor, and rely on such cross referencing tools as *cxref* and *Understand for C*, for extracting

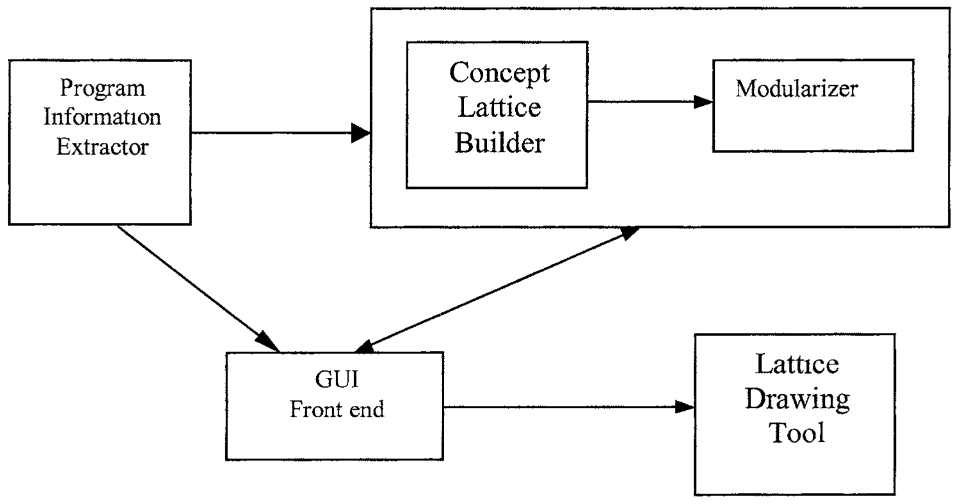

Figure 3.41: The outline design of our modularization tool

system information from C programs. Understand for C is an interactive development environment (IDE) designed for code navigation using a detailed cross referencing engine.

It also offers a C API to:

1. Report functions with their parameters and types
2. Report global objects
3. Report all structs and their member types
4. Find all references to and from an entity

The Concept Lattice Builder: takes input from the Program Information Extractor and creates the abstraction of the system in the form of a concept lattice. It deals with the construction of the context table, the identification of atomic concepts, and the building of the entire concept lattice.

The Modularizer: Based on the concept lattice generated by the Concept Lattice Builder and system information given by the Program Information Extractor, the Modularizer identifies and eliminates the sources of coupling and interference in the system and decomposes the lattice into meaningful modules that display high cohesion and low coupling.

The GUI Front end: provides a graphical user interface to our modularization tool. It generates HTML reports that offer a lot of architectural information generated by the tool in addition to suggesting restructuring actions.

The Lattice Drawing Tool: We employ a 2D graph drawing tool for drawing a concept lattice. DOT, a successor of the UNIX tool DAG, is such an open source 2 dimensional graph drawing tool presented by AT&T Research. DOTTY is the visualizer written on top of DOT.

The Concept Lattice Builder and the Modularizer have been written in ANSI standard C++. The GUI front end is written in Tcl/Tk. The program has been compiled using Microsoft Visual C++ 6.0 and runs on Microsoft Windows 95/98/NT/2000. For all the examples illustrated in this thesis the output generated by our tool has been included in the appendices. The source code for the Concept Lattice builder and the Modularizer are listed in appendix B1 and that for the GUI Front end, in appendix B2.

Case Study: *voronoi*

A Voronoi diagram is a geometric structure that represents proximity information about a set of points. This program has been taken from a computational-geometry library. The program contains roughly one thousand lines of C code, with eight source files, forty-four functions, thirty-six global variables and six structures.

Initially we create a visualization of the system with a context that relates the functions to the structures and their global instances alone. For the context table shown in figure 4.21, our tool generates twenty concepts. Figure 4.22 shows the resultant concept lattice. The nodes in color represent infima that need to be resolved. Our tool identifies eight such infima. The system source provides us with all the information necessary for suppressing these interfering concepts. These are eliminated to give us the horizontally decomposable lattice shown in figure 4.23.

	Edge	Freelist	Freenode	Halfedge	Point	Site	ELhash	PQhash	Sites
bisect	X					X			
clip_line	X					X			
deref						X			
dist						X			
ELdelete				X					
ELgethash				X			X		
ELinitialize							X		
ELinsert				X					
ELleft				X	X		X		
ELleftbnd									
ELright				X					
Endpoint	X								
Freeinit		X							
Getfree		X							
HEcreate				X					
intersect	X								
leftreg						X			
makefree				X					
nextone									X
out_bisector	X								
out_ep	X								
out_site						X			
out_triple						X			
out_vertex						X			
PQmin								X	
PQbucket				X					
PQdelete				X				X	
PQempty				X					
PQextractmin				X				X	
PQinitialize								X	
PQinsert				X		X		X	
readone						X			
readsites									X
ref						X			
right_of	X			X	X				
rightreg				X					
scomp					X				
voronoi				X	X	X			

Figure 4.21: The context table relating functions with the structures and their global instances

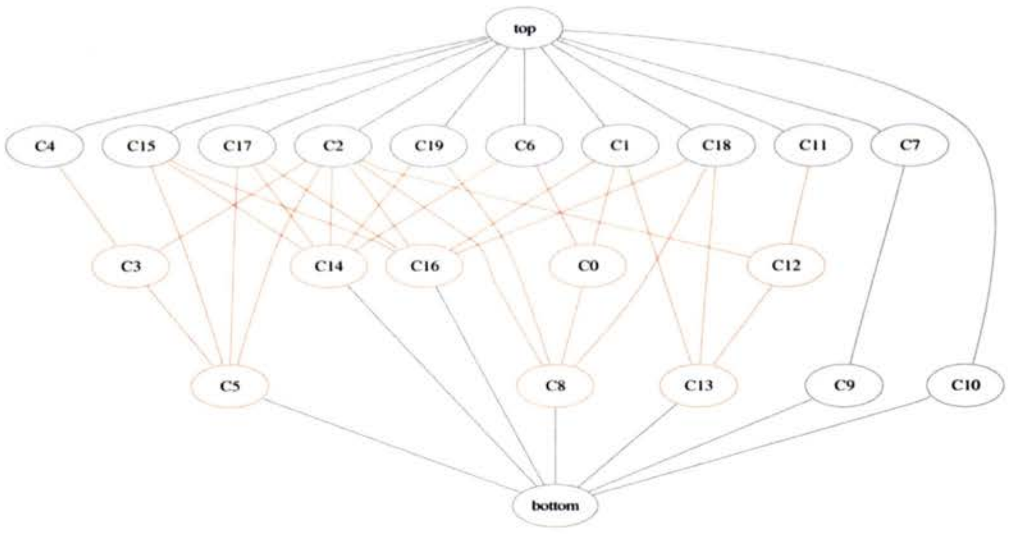

Figure 4.23: The first concept lattice for *voronoi*

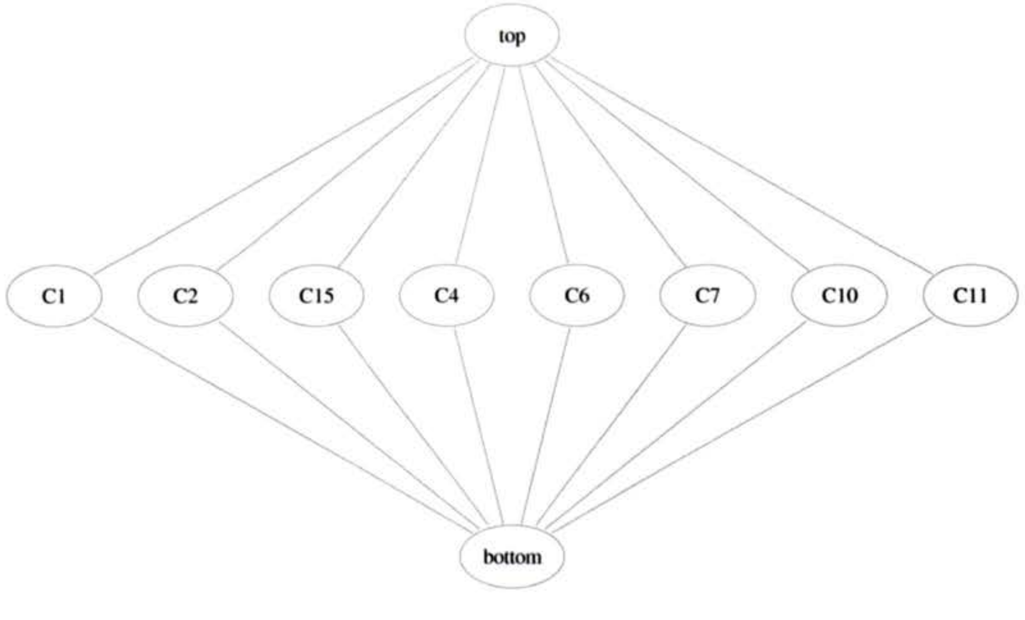

Figure 4.24: horizontally decomposable lattice suggesting eight modules

This lattice is a horizontal sum of eight antichains that represent potential modules. These eight modules are listed below

1. ({deref,dist,out_site,out_triple,out_vertex, readone,ref } {Site })
2. ({ELdelete,ELinsert,ELleft,ELright,HEcreate, PQbucket,intersect,leftreg,right_of,rightreg, voronoi} {Halfedge })
3. ({scomp} {Point})
4. ({ELgethash,ELinitialize,ELleftbnd} {ELhash})
5. ({bisect,clip_line,endpoint,out_bisector,out_ep} {Edge})
6. ({freeinit,getfree,makefree} {Freelist})
7. ({nextone,readsites} {sites})
8. ({PQ_min,PQdelete,PQextractmin,PQinitialize,PQinsert} {PQhash})

In the next step, we build a concept lattice with the identified modules as objects and the global variables as attributes. The context, shown in figure 4.25 relates a module to a global variable if any function in that module uses that global variable. By making this a two-pass process, we have effectively simplified the modularization process. A complete concept lattice built with the entire attribute set would “clutter up” the lattice making modularization difficult. With our method, we have generated potential modules by encapsulating functions and user defined compound datatype members together in the first pass. The next pass accounts for the global variable usage by the functions in these modules. The new context contains the modules identified in the first pass in its object set, *O*, and the unaccounted global variables in the attribute set *A*. Note that the objects and the attributes in the table shown in figure 4.25 have been transposed due to aesthetic reasons.

	C1	C2	C4	C6	C7	C10	C11
bottomside		X					
cradius	X						
debug	X			X			
deltax			X				
deltay		X					
efl				X			
ELhashsize			X				
ELleftend			X				
ELrightend			X				
Freeinit					X		
hfl		X	X				
leftreg		X					
nedges				X			
nsites						X	
ntry			X				
plot	X			X			
PQcount							X
PQhashsize		X					X
PQmin		X					X
pxmax				X			
pxmin				X			
pymax				X			
pymin				X			
sfl	X	X					
siteidx	X					X	
Sqrt_nsites			X		X		X
total_search			X				
triangulate	X			X			
xmax						X	
xmin			X			X	
ymax		X				X	
ymin		X				X	

Figure 4.25: Context table considering global variables

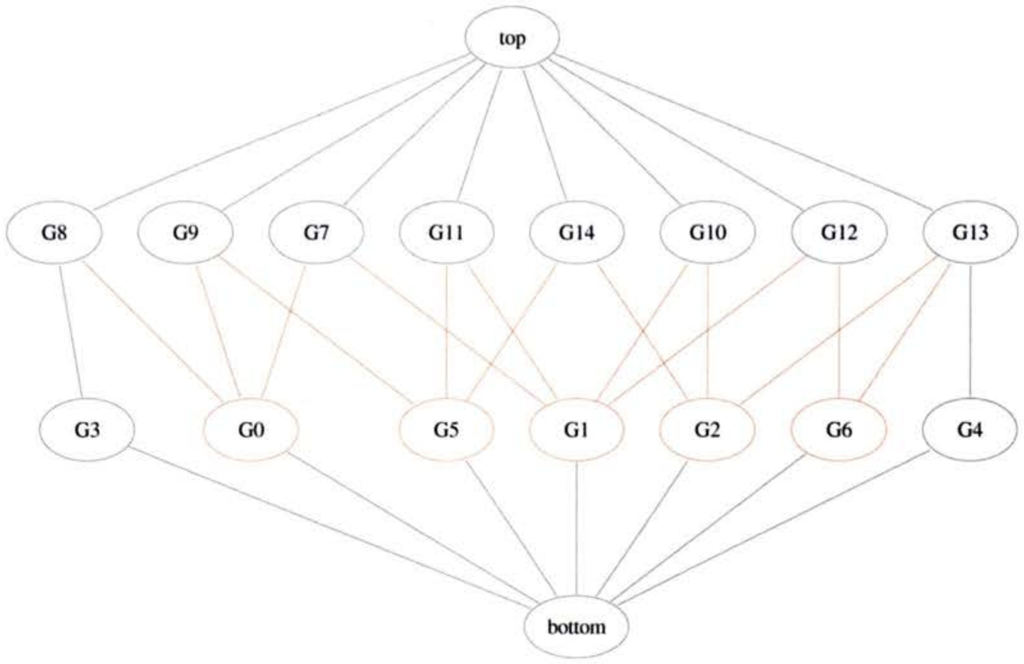

Figure 4.26: The second concept lattice for *voronoi*

The resulting concept lattice is shown in figure 4.26. The following are the reduced labels corresponding to each node

```
G0  = ({C1} {cradius})
G1  = ({C2} {deltay})
G2  = ({C4} {ELhashsize,ELleftend,ELrightend,deltax,ntry,
            totalsearch})
G3  = ({C6} {efl,nedges,pxmax,pxmin,pymax,pymin})
G4  = ({C7} {})
G5  = ({C10} {})
G6  = ({C11} {PQcount})
G7  = ({} {sfl})
G8  = ({} {debug, plot, triangulate})
```

G9 = ({} {siteidx})

G10 = ({} {hfl})

G11 = ({} {ymax, ymin})

G12 = ({} {PQhashsize,PQmin})

G13 = ({} {sqrt_nsites})

G14 = ({} {xmin})

Groups G0 through to G6 represent the potential modules. The global variables in the reduced labels corresponding to these are added to their corresponding intents. The global variables that are shared between these modules label the suprema of modules (G7 through G14), and, as such, represent a source of coupling. Whether these variables should be encapsulated into the modules or not can be decided with user assistance. The table in figure 4.28 shows all the identified modules of *voronoi*.

Module	Data members	Operations
Site	cradius Point coord int sitenbr int refcnt	deref, dist, out_site, out_triple, out_vertex, readone, out_vertex, ref

Halfedge	Deltay
	Halfedge *ELleft, *ELright;
	struct Edge *ELedge;
	int ELrefcnt;
	char ELpm;
	Site *vertex;
	double ystar;
	struct Halfedge *PQnext;
		ELdelete, ELinsert,
		ELleft, ELright,
		HEcreate, PQbucket,
		intersect, leftreg,
		right_of, rightreg,
		voronoi

Point	float x, y	scomp
Elhash	ELhashsize, ELleftend, ELrightend, deltax, ntry, totalsearch	ELgethash, ELinitialize, ELleftnd
Edge	efl, nedges, pxmax, pxmin, pymax, pymin	bisect, clip_line, end_point, out_bisector, out_ep

Freelist	Freenode *head;
	int nodesize;
		freeinit, getfree,
		makefree

Sites		nextone, readsites
Pqhash	Pqcount	PQ_min, PQdelete, PQextractmin, PQinitialize, PQinsert

Figure 4.28: The modules identified in *voronoi*

The rest of the functions that are not relegated to any modules may be considered to constitute a potential driver module and the unaccounted global variables, if any, may be added to it. From the encouraging results of this case study, we can conclude that concept analysis offers a useful method to automate program comprehension and restructuring.

Chapter 5

CONCLUSION

Based on the information extracted from the given system, we presented a technique to identify closely related data, routines and their definitions, and encapsulate them in meaningful modules that are maximally cohesive and minimally coupled. We presented the application of mathematical concept analysis to software reengineering, and demonstrated how a concept lattice provides a concise, yet vivid account of all the relationships amongst concepts. We used concept analysis to identify inherent structures in software that cannot be fully captured in quantitative analyses. With the encouraging results obtained from case study of the computational-geometric library voronoi, we established that concept analysis offers a useful method to automate program comprehension and restructuring. The following section discusses related work in the field of program comprehension and software reengineering. By comparing our results with those of these other efforts, we identify our contribution to the field of automatic program comprehension and software reengineering.

Related Work

There have been numerous prior research efforts in conceptual program understanding [Chin, Quilici 96], [Quilici 94], [Kozaczynksi, Ning 94], [Wills 92], [Wills 90], [Hartman 91], [Johnson 86]. They perform program understanding by plan recognition: the identification of a set of items in the code that meet certain constraints with heuristics used to reduce the combinatorics involved in plan recognition. None of these efforts in establishing a plan recognition algorithm has scaled to the size of real-world software systems. These efforts assume a library of program plans, and provide an engine for recognizing instances of these plans in the system. As illustrated in [Woods, Quilici, Yang 98] a typical plan recognition algorithm parses the imperative code (shown in figure 5.31), makes an internal representation for it, and with the help of a plan recognition engine, identifies a suitable existing plan from the plan library.

```
const char *s = "Hello world";

for (i = 0; s[i] = '\0'; i++)
      putchar(s[i])
. . .
putchar('\n');
. . .
for (i = 0; s[i] != '\0'; i++)
{
      if (s[i] == m)
      {
            pos = i;
            break;
      }
}
. . .
```

Figure 5.31: Example C code that manipulates strings

A typically recognized plan would be as illustrated in [Woods, Quilici, Yang 98]: *"The code declares a character string, initializes it with a constant, prints that string followed by a newline character, and searches it for a particular value."*

The corresponding "objectified" version of figure 5.31 is listed in figure 5.32.

```
const string s("Hello world");
...
cout << s << endl;
...
pos = s.findfirst(m);
...
```

Figure 5.32: The translation to object-oriented C++ using plan recognition

The success of this approach at objectification apparently depends highly on the knowledge in the plan library, which should map pieces of existing imperative code to suitable class operations. It is unlikely to have a complete code pattern library, however this approach claims success on the basis that even if the understanding of the code is partial, it will always be correct because the system will not recognize design patterns that are not present in the code. The understanding may be partial because there are design concepts that the process may not be able to identify. Our efforts differ from plan recognition in that our approach aims at comprehending and reengineering the underlying architecture of the software system.

Chen *et al.* have presented a C Information Abstraction System (CIA) that records information about the various objects in a C program in a program database. This information may then be processed by relational database tools to locate declarations and analyze their relationships. Based on the system information provided by CIA, various tools to provide graphical views of program structure have been developed. These tools visualize the system as directed graphs to model function call graphs and file include hierarchies. In addition to automating software restructuring tasks, CIA also aims to eliminate dead code from C systems. Based on this work, experiments are being conducted on software metrics and program structure comparison.

Concept analysis has been applied to a host of different problem domains. Snelting has presented work on the application of mathematical concept analysis for varied problems in the analysis and reengineering of software [Snelting 96]. One application of mathematical concept analysis was to the problem of analyzing configurations in UNIX source files. This work led to the development of the NORA/RECS tool used to identify conflicts in software-configuration management information. Snelting and Tip used a method well founded in mathematical concept analysis to reengineer class hierarchies [Snelting, Tip 98]. A concept lattice was used to visualize the access and subtype relationships between variables, objects and class members. Lindig and Snelting studied the application of mathematical concept analysis to the identification of modular structure of legacy code [Lindig, Snelting 97]. The results reported by them on the two case studies are not encouraging. In both the cases, the lattice could not be decomposed satisfactorily due to the presence of a large number of interferences. A context was built by relating the functions in the program to the global variables being accessed by them. Our work has demonstrated how the use user defined compound data types offers a good start to modularization.

Contemporaneously with their work, Siff applied concept analysis to the C to C++ conversion problem with fairly encouraging results [Siff 98]. The case studies presented have not taken global variables into consideration. Although structures present a good starting point, modularization based on structures alone does not address the problem of coupling and interference due to global objects. To resolve interference, Siff uses negative information (e.g. “function f does not use the fields of struct t”). It is our view that such added attributes complicate the concept lattice, and that better results may be achieved by using information that can be extracted from the program source. By also considering the global instances of user defined compound datatypes, we have also demonstrated how the identified modules need not necessarily correspond to the user defined types alone. By treating user defined compound types and their global instances separately from global variables, we have simplified the modularization process by avoiding "clutter" in the concept lattice that would be difficult to deal with in one pass.

To summarize, we have presented techniques

1. to reverse engineer software into a conceptual model in the form of a concept lattice, and
2. to reengineer the concept lattice into the modules

We have seen that a concept lattice provides a concise, yet vivid account of the inter-relationships between the inherent structures in software. The encouraging results of our case study have established that this technique can be effectively used to automate program comprehension and software reengineering.

Chapter 6

DIRECTIONS FOR FUTURE RESEARCH

Identification of class hierarchies

The method described in the previous chapter aids program understanding by automatically extracting and analyzing the program structure. We extracted relationships between functions and variables and presented this information visually to the restructurer in the form of a concept lattice. Structural information alone may not be enough for completely understanding a system [Sommerville '96]. Maintenance tasks typically require semantic or conceptual understanding. System structure does not address the underlying purpose of a particular function or variable, a type of information that is almost always essential for program understanding. Identification of idiosyncratic implementations of standard design concepts can be used to increase the quality of modularization. Considering the specific problem of reengineering legacy C code, there are many idioms that are commonly used by C programmers to simulate such object-oriented features as inheritance, virtual functions, and genericity. All of these involve type casting of pointers, with which a programmer can interpret any memory region to be of any desired type. Not only do type casts make programs vulnerable to type errors, but they also hinder program comprehension and maintenance by creating latent dependencies between seemingly independent pieces of code. According to McConnell, "a program that requires type casts probably has some architectural gaps that need to be revisited. Redesign if that's possible" [McConnell 93]. By identifying the usage of the common programming idioms, we may identify the intent of the programmer to emulate class-hierarchies ("is-a", "has-a" relations) and virtual functions.

Consider the example listing in Appendix 6.11 [Stroustrup 94]. Figure 6.12 shows the context table for the listing.

	Shape	**Circle**	**Square**	**Rectangle**
Initcircle		X		
Initsquare			X	
Initrectangle				X
Draw	X			

Figure 6.12: Context table for Shapes

The corresponding concept lattice in figure 6.12 can be seen to be easily horizontally decomposable; `Initcircle` being a member of the `Circle` class, `Initrectangle,` that of the `Rectangle` class, `Initsquare`, that of the `Square` class, and `Draw()`, that of `Shape` class. However, an examination of the underlying architecture shows that the function `Draw()` depends completely on the physical layout of the structure instances in memory. Figure 6.12 shows the corresponding concept lattice.

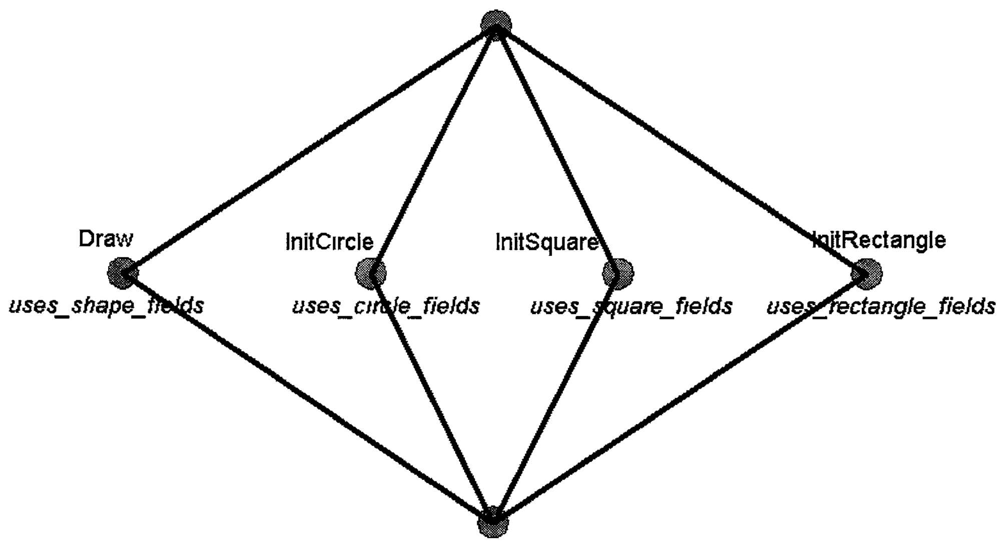

Figure 6.13: Concept lattice for “shapes”

In a programming language like C, the ability to cast variables and pointers allows a program to access a given object as if it had a type different from its declared type. The function `draw()` has been used to "draw" `Circles`, `Squares` and `Rectangles`, while treating them as `Shapes` by depending on their structural layout in memory. Physical layouts of structure and class instances are compiler and system dependent, and software architectures depending on these are, at best, non-portable. Further, any new structure added to the design listed in figure 6.11, that will need to be "drawn" will warrant changes in the `draw()` function. This makes the system less changeable and less maintainable.

This warrants a need for type checking while building the context table. Type-checking can be based on points-to analysis [Emami, Ghiya, Hendren 98], [Ghiya, Hendren 98], [Omega 85], [Steensgaard 96], [Yong, Horwitz, Reps 99]. In checking for type conversions, we also have to take into account the fact that most C compilers do not require explicit casts from one pointer type to another. Implicit type conversions at most invite a warning from a C compiler. The context table in figure 6.2 captures the correct relationship between the functions and the attributes.

	Shape	Circle	Square	Rectangle
Initcircle		X		
Initsquare			X	
Initrectangle				X
Draw		X	X	X

Figure 6.2: New context table for "shapes"

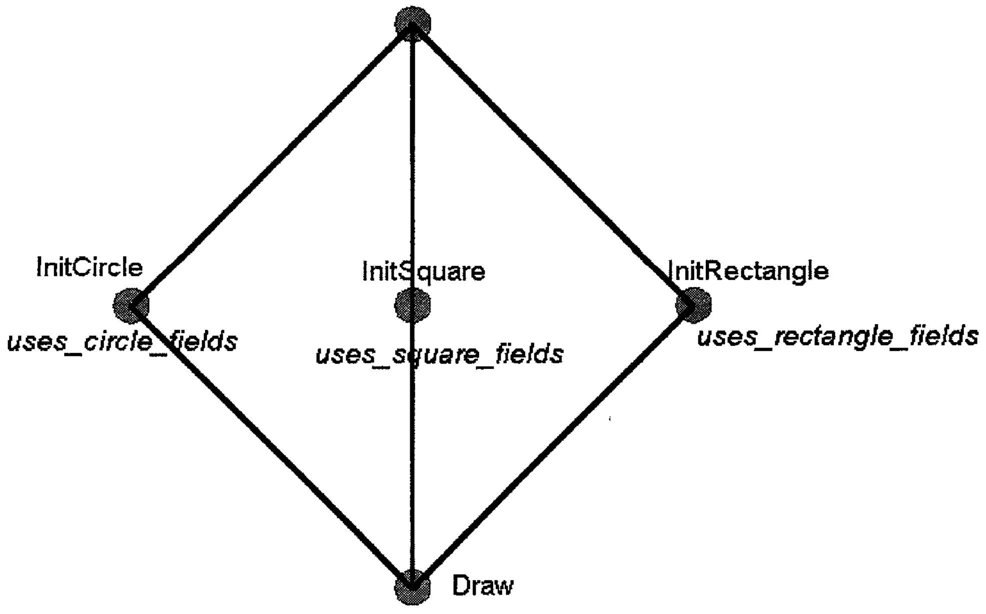

Figure 6.22: The new concept lattice for "shapes"

In resolving the interference presented by the bottom node, we observe that the structures `Circle`, `Square`, and `Rectangle` are structural subtypes of the structure `Shape`. In other words, they contain all the information in the structure `Shape` and more. The data in `Shape` is duplicated in these structures as a contiguous block, starting as the first member of the structure. This programming idiom is commonly known as the redundant declaration idiom. With this information we can identify an "is-a" relationship between each of these structures and the structure `Shape`.

Listing 6.3 in the appendix lists the equivalent source code in C++, establishing the inheritance identified in the system.

Data Organization Metric

It is possible that a system being restructured does not have enough structure or that enough information is not available to resolve sources of coupling and interference. Modularization achievable may be limited. In such cases, the concept lattice could be used to generate a metric for the data organization in the system. The concept lattice can also be used to quantify such attributes as coupling, cohesion and interference.

BIBLIOGRAPHY

[Abadi, Cardelli 96]
Martın Abadı, Luca Cardelli. *A theory of objects* Springer, 1996

[Aho, Hopcroft, Ullman 87]
Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *Data Structures and Algorithms* Addison Wesley Publication Company, 1987

[ANSI/ISO 90]
The Annotated ANSI C Standard. American National Standard for Programming Languages – C. Annotated by Herbert Schildt, ANSI/ISO 9899-1990

[Birkhoff 73]
Garrett Birkhoff, *Lattice Theory*, third edition. Amer. Math. Soc. Coll. Publ. 25, Providence, R.I., 1973.

[Brooks 95]
Brooks Frederick P. *The mythical man-month: Essays on Software Engineering.* Addison-Wesley Publication Company, Anniversary Edition.

[Chandra, Reps 99]
Satish Chandra and Thomas Reps *Physical Type Checking for C.* Technical Report BL0113590-990302-04, Lucent Technologies, Bell Labs, March 1999. http://www.bell-labs.com/~schandra/pubs/checking-tr.ps.

[Chen, Nishimoto, Ramamoorthy 90]
Yih-Farn Chen, Michael Y. Nishimoto, C. V. Ramamoorthy. *The C Information Abstraction System.* IEEE Transactions on software engineering (TSE) 16(3): 325-334 (1990)

[Chin, Quilici 96]

Quiang Yang, Alexander Quilici, *DECODE: A cooperative program understanding environment.* Journal of Software Maintenance: Research and Practice, 8(1):3-34.

[Cormen, Leiserson, Rıvest 90]
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts 1990

[Emami, Ghiya, Hendren 98]
Rakesh Ghiya and Laurie J. Hendren. Putting Pointer Analysis to work
ACM Principles of Programming Languages pg. 242-254. January '98.
[Even 79]
Shimon Even. *Graph Algorithms.* Computer Scıence Press, 1979

[Funk, Leweın, Sneltıng]
P. Funk, A. Lewein, G. Snelting
Algorithms for Concept Lattice Decomposition and their Application

[Ganter, Wille 96]
Bernhard Ganter and Rudolf Wille. *Applied Lattice theory: Formal Concept Analysis*

[Ghiya, Hendren 98]
Rakesh Ghiya and Laurie J. Hendren
Putting Pointer Analysis to Work. Proceedings of the Twenty Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California. 1998

[Grosch, Snelting 93]
F.-J. Grosch, G. Snelting: *Polymorphic Components for Monomorphic Languages.* Second International Workshop on Software Reusability, March 1993, IEEE Comp. Soc. Press, pp. 47-55.

[Godin, Alaoui 95]
R. Godin and R. Missaoui H. Alaoui. *Incremental concept formation based on Galois (concept) Lattices.* Computational Intelligence, 11(2): 246-267, 1995

[Hartman 91]
Hartman J., *Pragmatic empirical program understanding.* In workshop notes, AAAI Workshop on AI and Automated Program Understanding, San Jose, CA

[Horwitz, Reps, Binkley 90]

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

[Johnson 86]
Johnson W.L., *Intension Based Diagnostics of Novice Programming Errors*, Morgan Kaufman, Los Altos, CA

[Kozaczynksi, Ning 94]
Kozaczynksi, V., Ning, J. Q., *Automated program understanding by concept recognition*, Automated Software Engineering, 1:61-78

[Lewis 95]
Lewis and friends. *Object Oriented Application Frameworks.*
Manning Publication Co. 1995

[Lindig, Snelting 97]
C. Lindig, G. Snelting: *Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis.* Proc. International Conference on Software Engineering (ICSE 97), Boston, USA, May 1997, pp. 349-359.

[McConnell 83]
Steve McConnell. *Code Complete.* Microsoft Press, 1993

[Neirstrasz]
D.T.Oscar Neirstrasz. *Object Oriented Software Composition.* Prentice Hall International (UK) Ltd., Hempstead, UK, 1995

[Omega 85]
Omega. A data flow analysis tool for the C programming language.
IEEE Transactions on Software Engineering SE-11. No.9 September, 1985

[Prata 98]
Stephen Prata. *C++ Primer Plus*, 3rd ed. SAMS Publications

[Quilici 94]
Alexander Quilici, *A memory based approach to recognizing programming plans.* Communications of the ACM, 37(5):84-93

[Siff 98]
Michael Benjamin Siff. *Techniques for Software Renovation*
PhD Dissertation, University of Wisconsin - Madison. 1998

[Snelting 96]
Gregor Snelting. *Reengineering of configurations based on mathematical concept analysis.* ACM Transactions on Software Engineering and Methodology. S(2): 146-189. April 1996.

[Snelting, Tip 98]
G. Snelting, F. Tip: *Reengineering Class Hierarchies Using Concept Analysis.* Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering, November 1998, pp. 99-110.

[Sommerville 96]
Sommerville, I. *Software Engineering.* Addision-Wesley, Reading, MA, 5th edition

[Steensgaard 96]
Bjarne Steensgaard. Point-to analysis in almost linear time.
In *Proceedings of the Twenty-Third ACM Syposium on Principles of Programming Languages,* Pg. 32-41, St. Petersburg, FL, January 1996

[Stroustrup 94]
Bjarne Stroustrup. *The Design and Evolution of C++.* Addison Wesley 1994

[Stroustrup]
Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley. 3rd ed.

[Szasz 63]
Gabor Szasz *Introduction to Lattice theory* Academic Press, 1963

[Temperley 81]
H. N. V Temperley. *Graph Theory and Applications.* Ellis Horwood Limited. 1981

[Tarjan 83]
Robert E. Tarjan. *Data Structures and Network Algorithms.*
Society for Industrial and Applied Mathematics, 1983

[Wills 90]
Wills, L. M., *Automated program recognition by graph parsing.* Ph.D. thsis, MIT, Department of Computer Science

[Wills 92]
Wills, L. M., *Automated program recognition: A feasibility demonstration.* Artificial Intelligence, 45(2):113-172

[Yong, Horwitz, Reps 99]
Suan YI Yong, Susan Horwitz, Thomas Reps
Pointer Analysis for programs with Structures and Casting
Proceedings of the SIGPLAN '99 Conference on
Programming Language Design and Implementation (PLDI) 1999 ACM

APPENDICES

Appendix A2.1

```
<<list.h>>

#define true 1
#define false 0


struct Node{
      struct Node *next;
      Record data;
};


typedef struct Node Node;


typedef struct {
      Node *head;
      int length;
}List;



void initNode (Node* n, Record* d) {
      n->data = *d;
      n->next = 0;
}



void initList (List * l) {
            l->head = 0;
            l->length = 0;
}


void list_head_insert (List * l, Record* d) {
      Node *n = (Node *) malloc (sizeof(Node));
      if (!n) exit(1);
      n->data = *d;
      n->next = l->head;
      l->head = n;
      l->length++;
}


int find_element (List* l, Record* d, Node* prev) {
            Node* p = l->head;
            while (p) {
                  if(p->data == d) return true;
                  prev = p;   p = p->next;
            }
            return false;
}



void print (List * l) {
      struct Node *p = l->head;
      while(p) {
            print_data(&(p->data));
            p=p->next;
            printf("\n");
      }
}
```

```
void delete_element (List* l, Record* d) {
      Node* p = 0;
      if(find_element(l,d,p)) {
            Node* temp = p;
                  if(p==0) { temp=l->head; l->head = l->head->next;
                             free(temp); }
            else {temp = temp->next; p->next = temp->next; if(temp)
                  free(temp); }
            l->length--;
      }
      return;
}


int size (List * l) {
      return l->length;
}


void destroy_list(List* l) {
      Node *p = l->head;
      while (p) {
            l->head = l->head->next;
            free(p);
            p = l->head;
      }
}



<<hash.h>>


#include "list.h"


typedef struct  {
      int tablesize;
      List* Hashtable;
}Table;


void InitHashtable(Table* T, int size) {
      int i;
      T->tablesize = size;
      T->Hashtable = (List*) malloc (T->tablesize * sizeof(List));
      for (i=0; i<T->tablesize; i++)
                  initList(&(T->Hashtable[i]));
}


void Insert(Table* T, Key k, Record* r) {
      if(!Lookup(T, k, r))
       list_head_insert(&(T->Hashtable[HashFn(k)%T->tablesize]), r);
}


int Lookup (Table* T, Key k, Record* r) {
      List* l = &(T->Hashtable[HashFn(k)%T->tablesize]);
      if(l->head) {
            struct Node* p = l->head;
            while(p) {
                  if (Match(k, &(p->data))==0) {
                        *r = p->data;
                        return true;
                  }
                        p = p->next;
            }
      }
      return false;
}
```

```
void destroy_table (Table* T) {
      int i;
      for(i=0; i<T->tablesize; i++)
            destroy_list(&(T->Hashtable[i]));
      free(T->Hashtable);
}
```

Listing 2.11: A generic hash table package

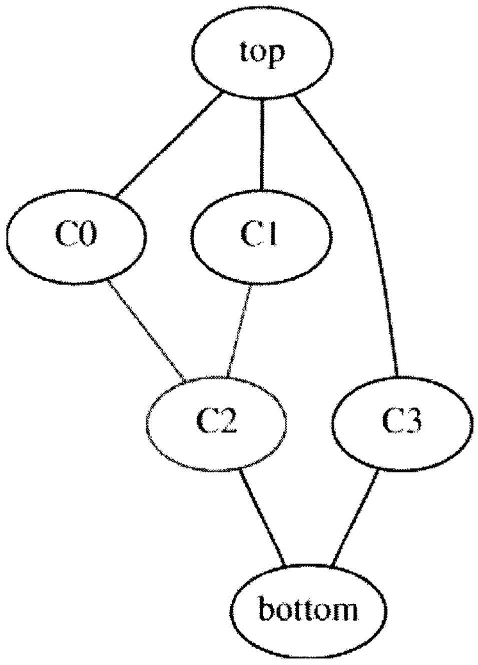

Figure 2.12: Concept Lattice for the hash table package

```
1. ({InitNode}, {struct node })

2. ({Initlist, List_head_insert, Find_element, Delete_element,
          Destroy_list, Print, Size},{struct list })

3. ({Inithashtable, Lookup, Insert, Destroy_table}, {struct
hashtable})
```

Figure 1.13: The identified modules

Appendix A3.41

```
<<variables.h>>

struct record {
      char name[40];
      char SSN [9];
};


typedef struct record Datatype;


bool equal (Datatype r1, Datatype r2) {
      return strcmp(r1.name,r2.name)==0 && strcmp(r1.SSN,r2.SSN)==0);
}


void print_data (Datatype r1) {
      printf ("Name : %s\nSSN: %s \n", r1.name, r1.SSN);
}


<<hash.h>>

#define TABLESIZE 117
struct List* Hashtable[TABLESIZE];

void initHashtable() {
      for (int i=0; i<TABLESIZE; i++) {
            Hashtable[i]  =  (struct  List  *)malloc  (sizeof(struct
List));
            initList(Hashtable[i]);
      }
}


bool find_record (const char* k, struct record* r) {
      struct List* l = Hashtable[atoi(k)%TABLESIZE];
      if(l->head) {
            struct Node* p = l->head;
            while(p) {
                  if (strcmp(p->data.SSN,k)==0) {
                        strcpy(r->name,p->data.name);
                        strcpy(r->SSN,k);
                        return true;
                  }
                        p = p->next;
            }
      }
      return false;
}


void insert_record (struct record* r) {
      if(!find_record(r->SSN, r))
            list_head_insert(Hashtable[atoi(r->SSN)%TABLESIZE], *r);
}
```

Listing 3.41: Tangled hash table package

	Uses_Record_f	Uses_Node_fielld	Uses_list_fields	Uses_hashtable
Equal	X			
Print_data	X			
Initnode		X		
Initlist			X	
List_head_inst		X	X	
Find_element		X	X	
Size			X	
Print		X	X	
Inithashtable				X
Find_record	X	X	X	X
Insert_record	X			X

Figure 3.42: Context table for the tangled hash table

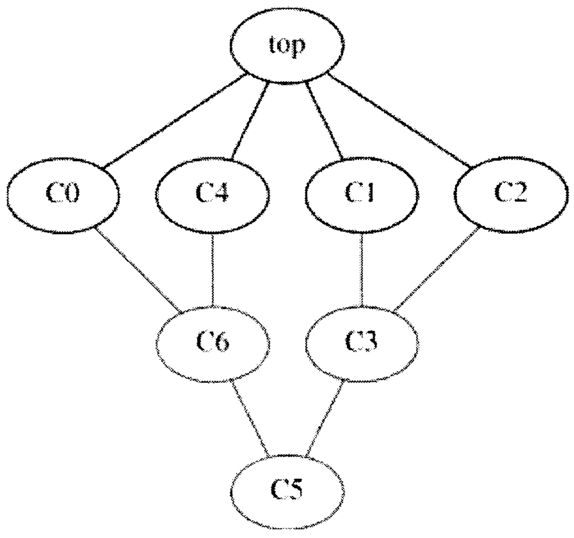

Figure 2.43: Concept lattice for the tangled hash table

Appendix A6.1

```
typedef enum {CIRCLE, SQUARE, RECTANGLE, ELLIPSE} type;


typedef struct {
      type k;
      int x;
      int y;
} Shape;


typedef struct {
      type k;
      int x;
      int y;
      int radius;
} Circle;


int initcircle (Circle* c, int a, int b, int r, enum type t) {
      c->x = a;
      c->y = b;
      c->radius = r;
      if (t != CIRCLE) return 0;
      c->k=t;
      return 1;
}


typedef struct {
      type k;
      int x;
      int y;
      int side;
} Square;


int initsquare (Square* s, int a, int b, int si, enum type t) {
      s->x = a;
      s->y = b;
      s->side = si;
      if (t != SQUARE) return 0;
      s->k = t;
      return 1;
}


typedef struct {
      type k;
      int x;
      int y;
      int length;
      int breadth;
} Rectangle;
```

```
int initrectangle (Rectangle* r, int a, int b, int ln, int br, enum
type t) {
      r->x = a;
      r->y = b;
      r->length = ln;
      r->breadth = br;
      if (t != RECTANGLE) return 0;
      r->k=t;
      return 1;
}


void draw (Shape* s) {
      switch (s->k) {
      case CIRCLE:
            printf ("I am a circle\n");
            printf ("Center: %d, %d, radius: %d\n", s->x, s->y,
                              *(&(s->y)+1));
            break;
      case RECTANGLE:
            printf ("I am a rectangle\n");
            printf ("Center: %d, %d, length: %d, breadth: %d\n",
                        s->x, s->y, *(&(s->y)+1), *((&(s->y)+1)+1));

            break;
      case SQUARE:
            printf ("I am a square\n");
            printf ("Center: %d, %d, length: %d\n", s->x, s->y,
                        *(&(s->y)+1));
            break;
      }
}


int main() {

      int i;
      Circle c;
      Square s;
      Rectangle r;
      Shape* shapes[3];

      initcircle(&c,10,20,30,CIRCLE);
      initsquare(&s,15,25,20,SQUARE);
      initrectangle(&r,20,30,15,20,RECTANGLE);

      shapes[2] = &c;
      shapes[1] = &r;
      shapes[0] = &s;

      for(i=0; i<3; i++)
            draw(shapes[i]);

      return 0;
}
```

Listing 6.1: "Shapes"

Appendix A6.4

```
class Shape {
protected:
      int x; int y;
public:
      virtual void Draw () = 0;
};


class Circle : public Shape {
protected:
      int radius;
public:
      Circle (int a,int b,int r);
      void Draw ();
};


Circle::Circle (int a,int b,int r) {. . . }


void Circle::Draw () {
// Draw only Circle
}


class Square : public Shape {
protected:
      int side;
public:
      Square (int a,int b,int si);
      void Draw ();
};


Square::Square (int a,int b,int si) { . . . }

void Square::Draw () {
//Draw only Square
}



class Rectangle : public Shape {
protected:
      int length;
      int breadth;
public:
      Rectangle (int a,int b,int ln,int br);
      void Draw ();
};


Rectangle::Rectangle (int a,int b,int ln,int br) {. . .}


void Rectangle::Draw () {
//Draw only rectangle
}
```

Listing 6.4: Restructured Shapes with virtual `draw()`

Appendix B1

```
#ifndef WORD_ITERATOR
#define WORD_ITERATOR

#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

class word_iterator {

protected:

      std::istream *in;       // pointer to the istream to iterate
through
      std::string value;            // current value of the iterator
      char delimiter;               // delimiter to decide "next"

      bool at_end;                  // flag to indicate position of
iterator

      void read() {
            if (*in) {
                  *in >> value;
            }
            at_end = (*in) ? true : false;
      }


public:

      typedef std::input_iterator_tag iterator_category;
      typedef std::string value_type;
      typedef ptrdiff_t difference_type;
      typedef const std::string* pointer;
      typedef const std::string& reference;


      word_iterator() : in(&std::cin), at_end(false), delimiter('\t')
{}


      word_iterator(std::istream& s, const char delim = '\t') :
in(&s), delimiter(delim)
      { read(); }


      reference operator * () const { return value; }


      pointer operator -> () const { return &value; }


      word_iterator operator ++ () {
            read();
            return *this;
      }


      word_iterator operator ++ (int) {
            word_iterator temp = *this;
            read();
            return temp;
      }
```

```
        bool operator == (const word_iterator& i) const {
              return (in == i.in && at_end == i.at_end) ||
                          (at_end == false && i.at_end==false);
        }

        bool operator == (bool flag) const {
              return at_end==flag;
        }


        bool operator != (const word_iterator& i) const {
              return !(*this == i);
        }


        bool operator != (bool flag) const {
              return at_end!=flag;
        }

};


#endif




<<line_iterator.h>>

#ifndef WORD_ITERATOR
#define WORD_ITERATOR

#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

class word_iterator {

protected:

      std::istream *in;       // pointer to the istream to iterate
through
      std::string value;            // current value of the iterator
      char delimiter;               // delimiter to decide "next"

      bool at_end;                  // flag to indicate position of
iterator

      void read() {
            if (*in) {
                  *in >> value;
            }
            at_end = (*in) ? true : false;
      }


public:

      typedef std::input_iterator_tag iterator_category;
      typedef std::string value_type;
      typedef ptrdiff_t difference_type;
      typedef const std::string* pointer;
      typedef const std::string& reference;
      word_iterator() : in(&std::cin), at_end(false), delimiter('\t')
{}

      word_iterator(std::istream& s, const char delim = '\t') :
in(&s), delimiter(delim)
      { read(); }

      reference operator * () const { return value; }

      pointer operator -> () const { return &value; }

      word_iterator operator ++ () {
            read();
            return *this;
      }

      word_iterator operator ++ (int) {
            word_iterator temp = *this;
            read();
            return temp;
      }

      bool operator == (const word_iterator& i) const {
            return (in == i.in && at_end == i.at_end) ||
                        (at_end == false && i.at_end==false);
      }

      bool operator == (bool flag) const {
            return at_end==flag;
      }

      bool operator != (const word_iterator& i) const {
            return !(*this == i);
      }

      bool operator != (bool flag) const {
            return at_end!=flag;
      }

};

#endif

/*
Sidharth Kodikal

Class concept_base implements a context table entry.
It being similar to a concept structurally has been factored out as
the base class.

File:         concept_base.h
Last revised: 11/08/00

*/

#ifndef CONCEPT_BASE
```

```
#define CONCEPT_BASE

#include <iostream>
#include <set>
#include <string>
#include <algorithm>

extern class lattice;
extern class context_table;


class concept_base
{

protected:

      std::set<std::string> objects;
      std::set<std::string> attributes;


public:

      friend class lattice;
      friend class context_table;

      friend bool operator < (const concept_base& c1, const
concept_base& c2);

      friend bool operator == (const concept_base& c1, const
concept_base& c2);

      typedef std::set<std::string>::iterator SETITER;

      void add_object(std::string s)
      {
            // add a function to O
            objects.insert(s);
      }


      void add_attribute(std::string s)
      {
            // add an attribute to A
            attributes.insert(s);
      }


      virtual bool is_akin (const concept_base& other)
      {
            return std::includes(other.attributes.begin(),
other.attributes.end(), attributes.begin(), attributes.end());
      }


      virtual std::ostream& output (std::ostream& out) const
      {
            out << "( {";
            std::copy(objects.begin(), objects.end(),
std::ostream_iterator<std::string>(out,", "));
            out << "} ";
            out << " {";
            std::copy(attributes.begin(), attributes.end(),
std::ostream_iterator<std::string>(out,", "));
            out << "} )";
            return out;
      }


};
```

```
bool operator < (const concept_base& c1, const concept_base& c2)
{
      // some order
      return (c1.objects.size() < c2.objects.size());
}



bool operator == (const concept_base& c1, const concept_base& c2)
{
      return (c1.objects == c2.objects);
}


std::ostream& operator << (std::ostream& out, const concept_base& c)
{
      c.output(out);
      return out;
}


std::set<std::string> operator + (const std::set<std::string>& s1,
const std::set<std::string>& s2)
{
      std::set<std::string> temp;
      std::set_union(s1.begin(), s1.end(), s2.begin(), s2.end(),
std::inserter(temp, temp.begin()));
      return temp;
}


std::set<std::string> operator - (const std::set<std::string>& s1,
const std::set<std::string>& s2)
{
      std::set<std::string> temp;
      std::set_difference(s1.begin(), s1.end(), s2.begin(), s2.end(),
std::inserter(temp, temp.begin()));
      return temp;
}

#endif

#ifndef CONCEPT_H_
#define CONCEPT_H_

#include "concept_base.h"
#include <list>



class concept : public concept_base
{

      int number;

      std::set<int> superconcept_list; // for o/p compatible with
Ralph Freese's Lattice drawing tool
      std::set<int> subconcept_list;       // for o/p compatible to
ATT Dot

public:

      friend class lattice;

      friend bool operator < (const concept& c1, const concept& c2);
      friend bool operator == (const concept& c1, const concept& c2);


      concept() : concept_base() { }

      void set_name (int x) { number = x; }

      int get_name () const { return number; }

      bool is_superconcept (const concept& other)
      {
            // returns true if
            // a. this includes the objects of other AND
            // b. other includes the attributes of this
            if ( *this == other ) return false;
            return ( std::includes (objects.begin(), objects.end(),
other.objects.begin(), other.objects.end()) ||  std::includes
(other.attributes.begin(), other.attributes.end(),
attributes.begin(), attributes.end()));
      }


      bool is_subconcept (const concept& other)
      {
            // returns true if
            // a. other includes the objects of this AND
            // b. this includes the attributes of other
            if ( *this == other ) return false;
            return ( std::includes (other.objects.begin(),
other.objects.end(), objects.begin(), objects.end()) ||
std::includes (attributes.begin(), attributes.end(),
other.attributes.begin(), other.attributes.end()));
      }

      void add_superconcept (int num)
      {
            superconcept_list.insert(num);
      }

      void add_subconcept (int num)
      {
            subconcept_list.insert(num);
      }

      std::ostream& output (std::ostream& out) const
      {
            out << "C" << number << " = ";
            concept_base::output(out);
            return out;
      }

};


bool operator < (const concept& c1, const concept& c2)
{
      // for sorting using sort. sort to be able to eliminate
duplicates using unique
      return (c1.objects.size() > c2.objects.size()) &&
(c1.attributes.size() < c2.attributes.size());
            // ordered according to galois connectivity
}


bool operator == (const concept& c1, const concept& c2)
{
      return (c1.objects == c2.objects) && (c1.attributes ==
c2.attributes);
}
```

```
std::ostream& operator << (std::ostream& out, const concept& c)
{
      c.output(out);
      return out;
}


// predicate for find_if function. finding by key (number) instead of
==
struct predicate
{
      int number;
      predicate (int num) : number(num) { }
      bool operator () (const concept& c)
      {
            return (number == c.get_name());
      }
};

// strict weak ordering for std::sort
struct ltstr
{
      bool operator () (const concept& c1, const concept& c2)
      {
            return c1 < c2;
      }
};


#endif



#ifndef CONTEXT_TABLE
#define CONTEXT_TABLE

#include <list>
#include "concept_base.h"
#include "line_iterator.h"
#include "word_iterator.h"


extern class lattice;

class context_table {

      std::list<std::string> objects;
      std::list<std::string> attributes;

      std::list<concept_base> table;

public:

      friend class lattice;

      typedef concept_base* pointer;
      typedef const concept_base* const_pointer;
      typedef concept_base& reference;
      typedef const concept_base& const_reference;

      typedef std::list<concept_base>::iterator ctable_iterator;
      typedef std::list<concept_base>::const_iterator
const_ctable_iterator;
      typedef std::list<concept_base>::reverse_iterator
reverse_ctable_iterator;
      typedef std::list<concept_base>::const_reverse_iterator
const__reverse_ctable_iterator;
```

```
      ctable_iterator begin () { return table.begin(); }
      ctable_iterator end () { return table.end(); }


      reverse_ctable_iterator rbegin () { return table.end(); }
      reverse_ctable_iterator rend () { return table.begin(); }

      std::istream& input(std::istream& in)
      {
            line_iterator file_begin(in, '*');
            line_iterator file_end;

            word_iterator row_start;
            word_iterator row_end;

            // read O ( objects )
            row_start = file_begin.begin();
            std::copy(row_start, row_end, std::inserter(objects,
objects.begin()));

            // read A ( attributes )
            ++file_begin;
            row_start = file_begin.begin();

            std::copy(row_start, row_end, std::inserter(attributes,
attributes.begin()));

            for ( ++file_begin; file_begin != file_end; ++file_begin)
            {
                  concept_base cb;
                  row_start = file_begin.begin();

                  cb.add_object(*row_start++);

                  for ( ; row_start != row_end; row_start++)
                        cb.add_attribute(*row_start);
                  table.push_back(cb);
            }

            return in;
      }



      std::ostream& output(std::ostream& out) const
      {
            out << "Objects: ";
            std::copy(objects.begin(), objects.end(),
std::ostream_iterator<std::string>(out, "\t"));
            out << std::endl;
            out << "Attributes: ";
            std::copy(attributes.begin(), attributes.end(),
std::ostream_iterator<std::string>(out, "\t"));
            out << std::endl;
            std::copy(table.begin(), table.end(),
std::ostream_iterator<concept_base>(out, "\n"));
            return out;
      }


      std::ostream& html_output(std::ostream& out) const
      {

            // start table
            out << "<html>\n <!-- Generated by Sidharth's Reverse
Engineering Tool>\n";
            out << "<head><title> Context table </title> </head>
<body> &nbsp;\n";
```

```
            out << "<table BORDER=1>\n" ;
            out << "<tr>\n";

            // print objects: heading row
            out << "<td></td>\n<td>";
            std::copy(attributes.begin(), attributes.end(),
std::ostream_iterator<std::string>(out, "</td>\n<td>"));
            out.seekp (-4, std::ios::cur);
            out << "</tr>\n";


            // print rows
            const_ctable_iterator titer = table.begin();
            std::list<std::string>::const_iterator obiter =
objects.begin();
            std::list<std::string>::const_iterator atiter;

            for( ; titer != table.end(), obiter != objects.end();
++titer, ++obiter)
            {
                  out << "<tr><td>" << *obiter << "</td>\n";

                  for( atiter = attributes.begin(); atiter !=
attributes.end(); ++atiter)
                  {
                        out << "<td>";
                        if( (*titer).attributes.find(*atiter) !=
(*titer).attributes.end())
                        {
                              out << "X";
                        }
                        out << "</td>";
                  }

                  out << "</tr>\n";
            }

            // end table
            out << "</table>\n</body>\n</html>";
            return out;
      }



};




std::istream& operator >> (std::istream& in, context_table& c)
{
      c.input(in);
      return in;
}



std::ostream& operator << (std::ostream& out, const context_table& c)
{
      return c.output(out);

}



#endif
```

```
#ifndef LATTICE_H_
#define LATTICE_H_

#include "context_table.h"
#include "concept.h"
#include <set>
#include <algorithm>


struct concept_pair
{
      concept x;
      concept y;

      concept_pair() { }
      concept_pair(const concept& a, const concept& b) : x(a), y(b) {
}

};


std::ostream& operator << (std::ostream& out, const concept_pair& cp)
{
      out << "\n\n ----------------------------------------- \n\n";
      out << cp.x << std::endl << cp.y;
      return out;
}


/*
  a. generate the atomic concepts from the context table, name them.
     compute 'top', and 'bottom'
  b. create a worklist containing distinct pairs of these atomic
concepts
  c. if any concept is a subconcept of another, add the superconcept
to the
       superconcept list  of the subconcept
  d. starting with the first pair in the worklist, compute suprema.
if there is one, name it,
     add it to the superconcept lists of each of the subconcepts.
  e. create new pairs of this new suprema with the concepts in the
atom_list
       bearing that two concepts in a pair cannot have a relation
  f. add this new concept to the atom_list
  g. voila! bottom up approach.
*/


class lattice
{

      context_table context;
      std::list<concept> atoms_list;
      std::list<concept_pair> work_list;
      std::list<concept> partition_list;
      int concept_counter;
      concept top;
      concept bottom;


      bool supremum (const concept& c1, const concept& c2, concept&
supremum)
      {
            // returns true if this and other yield a supremum, false
otherwise;
            // puts the supremum concept in the supremum parameter

            bool flag = false;
            if (c1 == c2) return flag;

            std::set_intersection(c1.attributes.begin(),
c1.attributes.end(), c2.attributes.begin(), c2.attributes.end(),
std::inserter(supremum.attributes, supremum.attributes.begin()));

            context_table::ctable_iterator citer;

            if ( supremum.attributes.size() )
            {
                  for(citer = context.begin(); citer !=
context.end(); ++citer)
                  {
                        if(std::includes((*citer).attributes.begin(),
(*citer).attributes.end(), supremum.attributes.begin(),
supremum.attributes.end()))
                        {
                              flag = true;
                              std::set<std::string>::iterator siter;

                              for(siter = (*citer).objects.begin();
siter != (*citer).objects.end(); ++siter)
                              {
                                    supremum.add_object(*siter);
                              }
                        }
                  }

                  if(flag)
                        return true;
            }

            return false;
      }



      bool infimum (const concept& c1, const concept& c2, concept&
infimum)
      {
            // returns true if this and other yield an infimum, false
otherwise;
            // puts the infimum concept in the infimum parameter
      }


      std::string resolve (const concept& c)
      {
            std::string ht = "uses_hashtable", lst =
"uses_list_fields", nd = "uses_node_fields", rec =
"uses_record_fields";
            if ( std::find(c.attributes.begin(), c.attributes.end(),
ht) != c.attributes.end() )
                  return ht;
            if ( std::find(c.attributes.begin(), c.attributes.end(),
lst) != c.attributes.end() )
                  return lst;
            if ( std::find(c.attributes.begin(), c.attributes.end(),
nd) != c.attributes.end() )
                  return nd;
            if ( std::find(c.attributes.begin(), c.attributes.end(),
rec) != c.attributes.end() )
                  return rec;
      }


      void clean_principle_ideal ( std::list<concept>::iterator i,
const std::set<std::string>& minus)
      {
            (*i).objects = (*i).objects - minus;
            std::set<int>::iterator iter =
(*i).superconcept_list.begin();
            for ( ; iter != (*i).superconcept_list.end(); ++iter)
            {
                  std::list<concept>::iterator temp_iter;
                  temp_iter = std::find_if(partition_list.begin(),
partition_list.end(), predicate(*iter));
                  if ( temp_iter != partition_list.end() )
                  {
                        clean_principle_ideal(temp_iter, minus);
                  }
            }
      }



public:

      lattice() : concept_counter(0) { bottom.set_name(-1);
top.set_name(-2); }

      void build_atoms ()
      {
            context_table::ctable_iterator citer1, citer2;

            // compute atomic concepts in O(N) time

            for(citer1 = context.begin(); citer1 != context.end();
++citer1)
            {
                  concept ctemp;
                  std::set<std::string>::iterator siter;


                  // have to do this because set::insert(iterator1,
iterator2) is not supported
                  // by most compilers. some work with
set::const_iterators...not M$VC
                  // TRY std::set_union() instead
                  for(siter = (*citer1).objects.begin(); siter !=
(*citer1).objects.end(); ++siter)
                  {
                        ctemp.add_object(*siter);
                  }


                  for(siter = (*citer1).attributes.begin(); siter !=
(*citer1).attributes.end(); ++siter)
                  {
                        ctemp.add_attribute(*siter);
                  }


                  for(citer2 = context.begin(); citer2 !=
context.end(); ++citer2)
                  {
                        if (citer1 == citer2)
                              continue;

                        if((*citer1).is_akin(*citer2))
                        {
                              for(siter = (*citer2).objects.begin();
siter != (*citer2).objects.end(); ++siter)
                              {
                                    ctemp.add_object(*siter);
                              }
                        }
```

```
                    }

                    // no duplicates in atom_list: O(N)
                    if( (std::find(atoms_list.begin(),
atoms_list.end(), ctemp)) == atoms_list.end() )
                    {
                          ctemp.set_name(concept_counter++);
                          atoms_list.push_back(ctemp);
                    }
              }


              //compute the direct superconcepts of "bottom"

              std::copy(context.attributes.begin(),
context.attributes.end(), std::inserter(bottom.attributes,
bottom.attributes.begin()));
              bottom.set_name(-1);

              std::list<concept> dummy = atoms_list;

              std::list<concept>::iterator aiter = atoms_list.begin(),
del_iter = atoms_list.end();
              for ( ; aiter != atoms_list.end(); ++aiter)
              {
                    if(bottom.attributes == (*aiter).attributes)
                    {
                          std::copy((*aiter).objects.begin(),
(*aiter).objects.end(), std::inserter(bottom.objects,
bottom.objects.begin()));
                          bottom.set_name((*aiter).get_name());
                          dummy.remove(*aiter);
                    }
              }

              atoms_list = dummy;

              for ( aiter = atoms_list.begin(); aiter !=
atoms_list.end(); ++aiter)

        bottom.superconcept_list.insert((*aiter).get_name());


              // creating concept "top"

              std::copy(context.objects.begin(), context.objects.end(),
std::inserter(top.objects, top.objects.begin()));

              // compute "top" and its subconcepts in build_lattice

        }


        void print_atoms (std::ostream& out) const
        {
              std::copy(atoms_list.begin(), atoms_list.end(),
std::ostream_iterator<concept_base>(out, "\n\n"));
        }



        void initialize_worklist ()
        {
              std::list<concept>::iterator aiter1, aiter2;

              // initialize the work list as pairs of distinct concepts
              // in O(N*N) time
```

```
            for(aiter1 = atoms_list.begin(); aiter1 !=
atoms_list.end(); ++aiter1)
            {
                  for(aiter2 = aiter1,++aiter2; aiter2 !=
atoms_list.end(); ++aiter2)
                  {
                        if( (*aiter1).is_subconcept(*aiter2) )
                        {

      bottom.superconcept_list.erase((*aiter2).get_name());
                              (*aiter1).add_superconcept(
(*aiter2).get_name() );
                              continue;
                        }
                        if( (*aiter1).is_superconcept(*aiter2) )
                        {

      bottom.superconcept_list.erase((*aiter1).get_name());
                              (*aiter2).add_superconcept(
(*aiter1).get_name() );
                              continue;
                        }
                        concept_pair cptemp(*aiter1, *aiter2);
                        work_list.push_back(cptemp);
                  }
            }
      }


/*
  d. starting with the first pair in the worklist, compute suprema.
if there is one, name it,
     add it to the superconcept lists of each of the subconcepts.
  e. if the new concept does not already exist in the atom_list,
       add it there and proceed to step f.
  f. create new pairs of this new suprema with the concepts in the
atom_list
       bearing that two concepts in a pair cannot have a relation
  g. voila! bottom-up approach.

*/

      void build_lattice()
      {
            build_atoms();
            initialize_worklist();

            std::list<concept_pair>::iterator iter;
            std::list<concept>::iterator aiter;
            for(iter = work_list.begin(); iter != work_list.end();
++iter)
            {
                  concept stemp;
                  if(supremum((*iter).x, (*iter).y, stemp))
                  {
                        if( (std::find(atoms_list.begin(),
atoms_list.end(), stemp)) == atoms_list.end() && !(stemp == bottom) )
                        {
                              stemp.set_name (concept_counter++);
                              atoms_list.push_back(stemp);

                              for(aiter = atoms_list.begin(); aiter
!= atoms_list.end(); ++aiter)
                              {

                                    if( (*aiter).is_subconcept(stemp)
)
                                    {
```

```
                                                (*aiter).add_superconcept(
(stemp).get_name() );
                                                continue;
                                          }

                                          if(
(*aiter).is_superconcept(stemp) )
                                          {
                                                (stemp).add_superconcept(
(*aiter).get_name() );
                                                continue;
                                          }

                                          concept_pair cptemp(*aiter,
stemp);
                                          work_list.push_back(cptemp);
                                    }
                              }

                        }

                  }

                  // compute "top"

                  std::list<concept> dummy = atoms_list;

                  for( aiter = atoms_list.begin(); aiter !=
atoms_list.end(); ++aiter)
                  {
                        if((*aiter).objects == top.objects)
                        {
                              std::copy((*aiter).attributes.begin(),
(*aiter).attributes.end(), std::inserter(top.attributes,
top.attributes.begin()));
                              top.set_name((*aiter).get_name());
                              dummy.remove(*aiter);
                        }
                        else
                              top.set_name(-2);
                  }

                  atoms_list = dummy;

                  for( aiter = atoms_list.begin(); aiter !=
atoms_list.end(); ++aiter)
                  {
                        if((*aiter).superconcept_list.size() == 0)
                              (*aiter).add_superconcept(top.get_name());
                  }

            }


            void partition_lattice( )
            {
                  /*
                  a.for each concept, create a subconcept list: visit
concept, find_if each of it's superconcepts
                        and add concept to their subconcept list. write a
Pred for find_if
                  b.sort in ascending order of number of attributes in
concepts
                  c.iterate thru this list; find infima, resolve, find_if
the appropriate superconcept,
                    the objects to it's objects list, add its subconcept to
all the superconcepts' subconcept lists
                    remove this concept from the subconcepts list of all
the superconcepts
```

```
            */

            typedef std::list<concept>::iterator list_iterator;

            partition_list = atoms_list;
            partition_list.push_front(bottom);
            partition_list.push_back(top);

      //    a. computing the subconcept lists of each concept

            list_iterator piter = partition_list.begin(), temp_iter =
partition_list.end();
            std::set<int>::iterator supiter;

            for ( ; piter != partition_list.end(); ++piter)
            {
                  supiter = (*piter).superconcept_list.begin();
                  for ( ; supiter !=
(*piter).superconcept_list.end(); ++supiter)
                  {
                        temp_iter =
std::find_if(partition_list.begin(), partition_list.end(),
predicate(*supiter));
                        if ( temp_iter != partition_list.end() )
                        {

      (*temp_iter).add_subconcept((*piter).get_name());
                        }
                  }
            }

            // b.sort in ascending order of number of attributes in
concepts
            partition_list.sort();

            std::cout << "after sorting....before messin\n";
            std::copy(partition_list.begin(), partition_list.end(),
std::ostream_iterator<concept>(std::cout, "\n"));
            /*
            c.iterate thru this list; find infima, resolve, find_if
the appropriate superconcept,
              the objects to it's objects list, add its subconcept to
all the superconcepts' subconcept lists
              remove this concept from the subconcepts list of all
the superconcepts
            */

            std::set<int>::iterator setiter;

            for ( piter = partition_list.begin(); piter !=
partition_list.end(); ++piter )
            {
                  if ( (*piter).superconcept_list.size() > 1 )    //
infimum
                  {
                        std::string prior = resolve( *piter );
                        std::cout << "resolving " << *piter << prior
<< "------------\n";

                        for ( supiter =
(*piter).superconcept_list.begin(); supiter !=
(*piter).superconcept_list.end(); ++supiter )
                        {
                              temp_iter =
std::find_if(partition_list.begin(), partition_list.end(),
predicate(*supiter));
```

```
                                for ( setiter =
(*piter).subconcept_list.begin(); setiter !=
(*piter).subconcept_list.end(); ++setiter )
                                {

      (*temp_iter).add_subconcept(*setiter);
                                }

                                // compute node to which objects of
*piter to be moved
                                if (
(*temp_iter).attributes.find(prior) != (*temp_iter).attributes.end()
)
                                {
                                      (*temp_iter).objects =
(*temp_iter).objects + (*piter).objects;
                                      std::cout << "Moving to ..." <<
(*temp_iter) << "++++\n";
                                }
                                else
                                {
                                      //(*temp_iter).objects =
(*temp_iter).objects - (*piter).objects;
                                      clean_principle_ideal(temp_iter,
(*piter).objects);

                                      std::cout << "Taking from ..." <<
(*temp_iter) << "++++\n";
                                }

                          }
                    }
            }

            // suppress infima

            std::list<concept> dummy_list = partition_list;

            for ( piter = partition_list.begin(); piter !=
partition_list.end(); ++piter )
            {
                  if( ((*piter).superconcept_list.size() > 1) ||
(*piter).attributes.size() == 0 || (*piter).objects.size() == 0)
                  {
                        dummy_list.remove(*piter);
                  }
            }

            partition_list = dummy_list;

            std::copy(partition_list.begin(), partition_list.end(),
std::ostream_iterator<concept>(std::cout, "\n"));

      }


      void print_partitions (std::ostream& out) const
      {
            std::copy ( partition_list.begin(), partition_list.end(),
std::ostream_iterator<concept>(out , "\n\n"));
      }




      void print_worklist(std::ostream& out)
      {
            std::copy(work_list.begin(), work_list.end(),
std::ostream_iterator<concept_pair>(out, "\n"));
```

```
	}


	// html output compatible with ralph freese's lattice drawing
tool
	void print_lattice_bottomup(std::ostream& out) const
	{
		std::list<concept>::const_iterator aiter;

		out << "<html> \n <!-- Textual bottomup lattice generated
by Sidharth's reengineering tool> \n"
			<< "<head> <title> Lattice - Text form </title> \n
</head> \n <body>";
		out << "( <br>\n";
		out << "(";
		if (bottom.get_name() == -1)
			out << "bottom (";
		else
			out << bottom.get_name() << " (";

		std::copy((bottom).superconcept_list.begin(),
(bottom).superconcept_list.end(), std::ostream_iterator<int>(out, "
"));
		out << " ) )<br>\n";

		std::set<int>::const_iterator sliter;
		for( aiter = atoms_list.begin(); aiter !=
atoms_list.end(); ++aiter)
		{
			out << "(" << (*aiter).get_name() << " (";
			for( sliter = (*aiter).superconcept_list.begin();
sliter != (*aiter).superconcept_list.end(); ++sliter)
			{
				if(*sliter == -2)
					out << "top ";
				else
					out << *sliter << " ";
			}

			out << " ) )<br>\n";
		}

		if (top.get_name() == -2)
			out << "top () )<br>\n";
		else
			out << top.get_name() << "() )<br>\n";
		out << ")<br>\n";
		out << "</body> \n </html>\n";
	}


	// plain text output compatible with ATT dot
	void print_lattice_topdown(std::ostream& out) const
	{
		std::set<int>::const_iterator sliter;

		out << "digraph G {\n";

		for( std::list<concept>::const_iterator aiter =
atoms_list.begin(); aiter != atoms_list.end(); ++aiter)
		{
			// check if this is an infimum
			if((*aiter).superconcept_list.size() > 1)
			{
				/*
				"5" [label = "\N"
					color = "black"
					width = "0.750000"
```

```
                                 shape = "ellipse"
                                ]
                         */
                         out << "C" << (*aiter).get_name() <<
"[color=red]\;\n";
                         /*
                         "6" -> "1" [color = "black"
                                     arrowheadhead = "none"]
                         */

                         for( sliter =
(*aiter).superconcept_list.begin(); sliter !=
(*aiter).superconcept_list.end(); ++sliter)
                         {
                               if( (*sliter) == -2 )
                               {
                                     out << "top->C" <<
(*aiter).get_name()  << "[arrowhead=none]\;\n";
                               }
                               else
                                     out << "C" << (*sliter) << "->C"
<< (*aiter).get_name() << "[color=red\narrowhead=none]\;\n\n";

                         }
                   }

                   else
                   {
                         for( sliter =
(*aiter).superconcept_list.begin(); sliter !=
(*aiter).superconcept_list.end(); ++sliter)
                         {
                               if( (*sliter) == -2 )
                               {
                                     out << "top->C" <<
(*aiter).get_name()  << "[arrowhead=none]\;\n";
                               }
                               else
                                     out << "C" << *sliter << "->C" <<
(*aiter).get_name() << "[arrowhead=none]\;\n";

                         }

                   }
             }


             if(bottom.get_name() != -1)
             {
                   out << "C" << bottom.get_name() <<
"[color=red]\;\n";

                   for( sliter = bottom.superconcept_list.begin();
sliter != bottom.superconcept_list.end(); ++sliter)
                   {
                         out << "C" << *sliter << "->" << "C" <<
bottom.get_name() << "[color=red\narrowhead=none]\;\n";
                   }
             }
             else
             {
                   for( sliter = bottom.superconcept_list.begin();
sliter != bottom.superconcept_list.end(); ++sliter)
                   {
                         out << "C" << *sliter << "->bottom" /*<<
bottom.get_name()*/ << "[arrowhead=none]\;\n";
                   }
             }
             out << "}";
```

```
	}


	std::istream& input (std::istream& in)
	{
		in >> context;
		return in;
	}


	std::ostream& output (std::ostream& out) const
	{
		out << context;
		return out;
	}

	std::ostream& print_contexttable (std::ostream& out) const
	{
		context.html_output(out);
		return out;
	}

};


std::istream& operator >> (std::istream& in, lattice& l)
{
	return l.input(in);
}



std::ostream& operator << (std::ostream& out, const lattice& l)
{
	return l.output(out);
}



#endif
```

```
#pragma warning(disable: 4786)
#include <iostream>
#include <string>
#include <string.h>
#include "lattice.h"


using namespace std;

char* parsefilename (char *str)
{
	char *token, *temp;
	char seps[] = "\\";
	token = strtok(str, seps);
	while( token != NULL )
	{
		temp = token;
		token = strtok(NULL, seps);
	}
	return strtok(temp, ".");
}
```

```
string directorypath (const char *filename, char *path)
{
      int limit = strlen(path) - strlen(filename) - 4;
      char* di = new char[limit+1];
      strncpy(di, path, limit);
      di[limit] = '\0';
      return string(di);


}
```

```
string dir;
string filename;


void generate_html_report ( const lattice& l, std::ostream& out )
{
      out << "<html> \n <!-- Textual bottomup lattice generated by
Sidharth's reengineering tool> \n"
            << "<head> <title> Reegneering Report </title> \n </head>
\n <body>";

      out << "<h1> Report </h1>\n";

      out << "<a href=\"#contab\"> <h3> View context table </h3> </a>
<br> <br> <br>\n";

      out << "<a href=\"#concepts\"> <h3> View all concepts </h3>
</a> <br> <br> <br>\n";

      out << "<a href=\"#textlat\"> <h3> Textual representation of
the lattice generated </h3></a> <br> <br> <br>\n";

      out << "<a href=\"#lat\"> <h3> View the lattice generated </h3>
</a> <br> <br> <br>\n";

      out << "<a href=\"#suggest\"> <h3> View suggested restructuring
actions </h3> </a> <br> <br> <br>\n";

      out << "<a href=\"#lattop\"> <h3> Top view of the lattice </h3>
</a> <br> <br> <br> <br>\n";

      out << "<a name=\"contab\"> <h2> Context table </h2> </a>\n";
      l.print_contexttable (out);
      out << "<br><br><br>\n";

      out << "<a name=\"concepts\"> <h2>List of the Concepts
generated </h2></a>\n";
      out << "<pre>\n";
      l.print_atoms(out);
      out << "</pre>\n";
      out << "<br><br><br>\n";

      out << "<a name=\"textlat\"> <h2> Textual Representation of the
lattice </h2> </a>\n";
      l.print_lattice_bottomup (out);
      out << "<br><br><br>\n";

      out << "<a name=\"lat\"> <h2> Behold....the Lattice </h2> </a>
<br> <br> <br>\n";
```

```
	//out << "<img src=" << filename << ".jpg height=600 width=600
BORDER=\"0\">\n";
	out << "<img src=contab1.jpg height=900 width=900
BORDER=\"0\">\n";

	out << "<a name=\"suggest\"><h2> Suggested restructuring
actions </h2></a>\n";
	out << "<h3> Suggested Partitions: </h3>\n";
	out << "<pre>\n";
	l.print_partitions (out);
	out << "</pre>\n";

	out << "<a name=\"lattop\"> <h2> Lattice topview </h2> </a>
<br> <br/> <br>\n";
	//out << "<img src=" << filename << "_top.jpg  height=700
width=900 BORDER=\"0\">\n";
	out << "<img src=contab1_top.jpg height=900 width=900
BORDER=\"0\">\n";

	out << "</body></html>\n";

}



int main(int argc, char* argv[])
{
	if(argc < 2)
	{
		cout << "Usage: lattice <context_file>\n";
		exit(1);
	}

	string path = argv[1];

	filename = parsefilename(argv[1]);

	ifstream fin;
	fin.open(path.c_str());

	dir = directorypath(filename.c_str(), (char *)path.c_str());

	string dotfile = dir + filename + ".dot";
	string reesefile = dir + filename + "_reese.html";
	string htmlopfile = dir + filename + ".html";
	string conceptsfile = dir + filename + "_concepts.dat";
	string partitionsfile = dir + filename + "_partitions.dat";

	ofstream fout1;
	fout1.open(conceptsfile.c_str());
	if(fout1.fail())
	{
		cout << "Cannot open op file\n";
		exit(1);
	}

	ofstream fout2;
	fout2.open(dotfile.c_str());
	if(fout2.fail())
	{
		cout << "Cannot open op file\n";
		exit(1);
	}

	ofstream fout3;
	fout3.open(htmlopfile.c_str());
	if(fout3.fail())
	{
		cout << "Cannot open op file\n";
```

```
            exit(1);
      }

      ofstream fout4;
      fout4.open(reesefile.c_str());
      if(fout4.fail())
      {
            cout << "Cannot open op file\n";
            exit(1);
      }


      lattice t;
      fin >> t;

      cout << t;

      t.build_lattice();
      t.print_atoms(fout1);
      t.print_lattice_topdown(fout2);
      t.print_lattice_bottomup(fout4);
      t.partition_lattice();

      generate_html_report(t, fout3);

      fin.close();
      fout1.close();
      fout2.close();
      fout3.close();

      return 0;

}
```

Appendix B2

```
<<gui.tcl>>


#usr/bin/wish -f

#window
wm withdraw .
set w [toplevel .t]
wm title .t {Reegineering Tool}

# menu

set m [menu $w.menubar -tearoff 0]
$m add cascade -label File -menu [menu $m.file]
$m.file add command -label "Load Context" -command "load_context
$w.f2"
$m.file add command -label "Save Partitions" -command {puts
Load_context"}
$m.file add command -label Exit -command exit

$m add cascade -label Action -menu [menu $m.action]
$m.action add command -label Partition -command {puts Partition}
$m.action add command -label Suggest -command {puts Suggest}

$m add cascade -label View -menu [menu $m.view]
$m.view add command -label "Report" -command viewreport

$m.view add command -label "Lattice Diagram" \
           -command {exec c:/progra~1/graphviz/bin/dotty $dotfile &}

$m.view add command -label "Lattice Topview" \
           -command {exec c:/progra~1/graphviz/bin/lneato $dotfile &}

$m.view add command -label "Lattice Representation" -command {exec
c:/progra~1/netscape/communicator/program/netscape -k $reesefile &}

$m.view add command -label Partitions -command {puts "show
partitions"}

$m add cascade -label Help -menu [menu $m.help]
$m.help add command -label About -command "show_message Sidharth"
$m.help add command -label Bugs -command {puts "Known bugs"}

$w configure -menu $m


# frame1: buttons

set f [frame $w.f1 -relief groove -bd 3]
button $f.b1 -text "Load Context" -default active -command
"load_context $w.f2"

button $f.b3 -text "Report" -default normal \
          -command viewreport

button $f.b6 -text "Draw Lattice" -default normal \
           -command {exec c:/progra~1/graphviz/bin/dotty $dotfile &}

button $f.b4 -text "Show Topview" -default normal \
           -command {exec c:/progra~1/graphviz/bin/lneato $dotfile &}

button $f.b5 -text "View labels" -default normal -command {exec
notepad $rlabel &}
```

```
button $f.b8 -text "View Partitions" -default normal -command {exec
notepad $partitionsfile &}

button $f.b7 -text "Text Lattice" -default normal -command {exec
c:/progra~1/netscape/communicator/program/netscape -k $reesefile &}

button $f.b9 -text "Restructured Lattice" -default normal \
           -command {exec c:/progra~1/graphviz/bin/dotty $dotfile2 &}

button $f.b10 -text "Restructured topview" -default normal \
           -command {exec c:/progra~1/graphviz/bin/lneato $dotfile2
&}

pack $f.b1 $f.b6 $f.b4 $f.b3 $f.b5 $f.b8 $f.b7 $f.b9 $f.b10 -padx 1 -
side left
pack $f -pady 2

#frame2: text for displaying all latice concepts
set f [frame $w.f2]
text $f.t -wrap word  -width 90 -spacing1 1m -spacing2 0.5m -spacing3
1m \
        -height 15 -yscrollcommand ".y set"
scrollbar .y -command "$f.t yview"
#$f.t insert end {All the concepts will be displayed}
pack $f.t -fill x -expand true -side left
pack $f -pady 2


proc load_context { f1 } {
      global filename path dotfile htmlopfile conceptsfile
partitionsfile reesefile rlabel dotfile2
      set filename [import_file]
      exec c:/users/sidharth/ms_thesis/tool_01/debug/lattice
$filename

      set path [file root $filename]
      set dotfile ${path}.dot
      set dotfile2 ${path}_2.dot
      set reesefile ${path}_reese.html
      set htmlopfile ${path}.html
      set conceptsfile ${path}_concepts.dat
      set partitionsfile ${path}_partitions.dat
      set rlabel ${path}_rlabels.dat

      load_concepts $f1.t
}

proc load_concepts {t1 } {
      global conceptsfile dotfile path
      if [catch {open $conceptsfile} in] {
            puts stderr "Cannot open file"
      }

      while {[gets $in line] >= 0} {
            $t1 insert end $line\n
      }
}

proc import_file {  } {
      global filename
      set file_types {
        { "Context Files" { .con .dat .txt }}
        { "All Files" * }}
      set filename [tk_getOpenFile -initialdir [pwd] -filetypes
$file_types \
                    -title "Load Context"]
      set initialdir [file dirname $filename]
      return $filename
}
```

```
proc show_message {msg} {
#     message about -text "Sidharth Kodikal"
}


proc viewreport { } {
      global dotfile path htmlopfile

      set img ${path}.ps

      exec c:/progra~1/graphviz/bin/dot -Tps $dotfile -o $img
      exec c:/aladdin/gs6.01/bin/gswin32 -sDEVICE=jpeg -
sOutputFile=${path}.jpg \
            -sNOPAUSE -q $img

      set img ${path}_top.ps

      exec c:/progra~1/graphviz/bin/neato -Tps $dotfile -o $img
      exec c:/aladdin/gs6.01/bin/gswin32 -sDEVICE=jpeg -
sOutputFile=${path}_top.jpg -sNOPAUSE -q $img

      exec c:/progra~1/netscape/communicator/program/netscape
$htmlopfile &


}
```