Source Code Modularization Using Lattice of Concept Slices

Raihan Al-Ekram and Kostas Kontogiannis
Dept. of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
Email: {rekram | kostas}@swen.uwaterloo.ca

Abstract

Most legacy systems have been altered due to prolonged maintenance to the point that they deviate significantly from their original and intended design and consequently, they lack modularity. Static source code analysis techniques like concept assignment, formal concept analysis and program slicing, have been successfully used by researchers for program understanding and for restoring system design properties. In our approach we combine these three techniques, aiming to gain on their individual strengths and overcoming their weaknesses. In this paper we present a program representation formalism that we call the Lattice of Concept Slices and a program modularization technique that aims to separate statements in a code fragment according to the concept they implement or they may belong to. The lattice shows the relationship between the statements of a program and the domain concepts that might be implemented by the statements. Using the lattice as a primary data structure we present two algorithms for decomposing the program into appropriate modules. The goal is to achieve a modularization such that the modules are self-contained, side effect free and the code duplication among nodes is minimal. The modularization process is illustrated with an example C program.

1. Introduction

Most legacy software systems have been altered due to prolonged maintenance activities so that they deviate from their original design and consequently they lack among other qualities modularity. Such systems may have been degenerated to a point that they consist of monolithic low cohesive subroutines each of which implementing numerous distinct domain concepts[1]. This makes these systems difficult to understand and maintenance tasks hard to perform. Automatic or semi-automatic decomposition of such systems into a more modular structure with each module preferably implementing a single domain concept facilitates better understanding and maintenance of the application, program parallelization, object-oriented migration or other re-engineering activities.

Static source code analysis techniques like concept assignment, formal concept analysis, and program slicing, have long been used by researchers for program modularization. Concept assignment aims to associate specific meaning to specific parts of a program. This technique can be used to extract program fragments that may be associated with a particular domain concept in a program. But the problem of concept assignment is that the code fragment that has been identified as a domain concept is not self-contained and not executable independently as a separate module. Program slicing is another useful technique that extracts an executable subset of the original program that preserves the behavior of the program with respect to a variable at a program point. The problem of slicing is that the decomposition is done based on very fine-grained program variables instead of domain concepts. Moreover different decompositions overlap with each other and may have a significant amount of duplicated code among the extracted modules. Even though each of the decompositions is executable by itself, the code duplication may cause side effects when implemented as separate modules. Formal concept analysis is used from a different perspective in modularization. Instead of decomposition, it has been used to identify grouping of subroutines and global data structures into modules.

[1] A domain concept is an idea or a task in the problem domain that is being implemented in the program, e.g. calculate interest, book ticket etc. It is totally different from the concepts of formal concept analysis.

Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR'04)
1534-5351/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

In this paper we introduce a program representation formalism that we call the Lattice of Concept Slices and propose a modularization technique based on it. This approach combines concept assignment, slicing, and concept analysis aiming to capitalize on the strengths of these three analysis techniques while overcoming their shortcomings. The goal is to achieve a modularization such that each module implements preferably a single domain concept, each module is self-contained, there is minimal duplication in code and there are limited side effects among the modules. Firstly, we use concept assignment to identify domain concepts in the program and extract code fragments associated with them. Secondly, we define a new type of slicing criteria based on the domain concepts and compute concept slices. Thirdly, we build a lattice from the concept slices using formal concept analysis, which we call the Lattice of Concept Slices. We use this lattice to identify code duplication and side effects among the concept slices. Finally, we perform modularization by clustering and restructuring the lattice.

The paper is organized as follows. Section 2 provides a brief description of the existing modularization techniques and summarizes their strengths and weaknesses. Section 3 introduces the Lattice of Concept Slices representation formalism and describes the mechanism for building it. Section 4 presents the algorithms for modularization based on the lattice. The complete process is illustrated with an example C program. Section 5 describes some related work in the area of combining source code analysis techniques for program comprehension and re-engineering. Finally Section 6 concludes with paper with directions for future work.

2. Background

Researchers have used static source code analysis techniques like concept assignment, slicing and formal concept analysis for program comprehension and modularization. This section provides a brief description of these techniques.

2.1 Concept Assignment

As defined by Biggerstaff [1] the Concept Assignment problem is the identification of human oriented domain concepts and assigning them to implementation oriented source code within a program.

There are two approaches taken for identifying domain concepts – the structural analysis and the plausible reasoning. Structural analysis is based on parsing technology. A domain concept is defined as a structural pattern, e.g. uses of a variable, call to a method, regular expression matching on variable naming etc. The source code is then parsed to match the signature of the pattern. The matching lines of source are considered to be part of the domain concept. The atomic concepts are recognized first and based on them larger grained composite domain concepts are identified.

Plausible reasoning systems are based on heuristics, rules of thumb, informal information, weight of accumulated evidence and so forth. Some example plausible reasoning systems are hypothesis-based concept assignment (HB-CA) [2], domain model – the adaptive observer (DM-TAO) [1]. HB-CA is a three-stage process. The process uses a knowledge base that contains a list of domain concepts implemented in the program and their indicators. The indicators can be identifiers, keywords, comments, regular expressions etc. In the hypothesis generation stage the source code is taken as input and scanned through to generate hypotheses of domain concepts and based on the knowledge base. The hypotheses are then sorted by the indicator position in the source code. In the segmentation stage the sorted hypotheses are analyzed to group them into segments using an unsupervised competitive learning neural network. The output of the stage is a collection of segments each containing a number of hypotheses. In the concept binding stage the segments hypotheses are analyzed to identify the most evident concept. The segments are then labeled with their corresponding domain concepts.

Code segments corresponding to the domain concepts are candidates for modules. The advantage of this technique is that the fragmentation is done at right level of granularity, the domain concepts. But the drawback is that the code segments corresponding the domain concepts are not self-contained and not executable independently as separate modules.

2.2 Program Slicing

Slicing as originally described by Weiser [3] is an abstraction of a program based on a particular behavior. A slice is defined to be an executable subset of the original program that preserves the original behavior of the program with respect to a slicing criteria <*P*, *V*>, which is a given variable *V* at a given program point *P*. The slice will consist of all the statements of the program that may affect the value of *V* at point *P*.

The original slicing algorithm was based on statement deletion using data flow analysis. More

IEEE COMPUTER SOCIETY

widely used algorithms [4], [12] work on the Dependence Graph of the program. First, a program dependence graph (PDG) [10], [11] is created for the program at hand. Some additional nodes are inserted at the start of the PDG to correspond to the initial definitions of all variables used in the program without first being defined and at the end to correspond to the final uses of all the variables. The algorithm starts by traversing the PDG from the node corresponding the program point *P* and then traces back to all the nodes that has a direct or indirect control or data flow dependency on this node. All the visited nodes are marked. All the unmarked nodes are deleted. The program corresponding the resulting PDG is the computed slice. This type of slicing is known as static intra-procedural slicing. [15], [16] gives a comprehensive list of all the slicing variations and techniques.

Slicing has the advantage that the slices are self-contained and executable by themselves. But the problem of slicing is that the decomposition is done based on very fine-grained program variables instead of domain concepts. Modularization based on slicing may result into modules that contain a dignificant amount of duplicaed code because of overlapping control flows. Moreover, even though each of the decompositions is self-contained, if the duplicated code modifies global program ressorces it may cause significant and undesirable side effects when deployed in separate modules.

2.3 Formal Concept Analysis

Formal Concept Analysis is a mathematical technique used for identifying groupings of objects that have common attributes and representing them in a lattice structure to show the generalization-specialization relationship among the groups.

Concept analysis starts with a context (O, A, R), a binary relation R between a set of objects O and their attributes A. A concept C (E, I) is a maximal collection of objects E (the extent) sharing common attributes I (the intent). A concept C_1 (E_1, I_1) is a sub-concept of another concept C_2 (E_2, I_2) if $E_1 \subseteq E_2$ or equivalently $I_2 \subseteq I_1$. The sub-concept relation is a partial order relationship that forms a lattice over the set of the concepts, each of the nodes of the lattice being a concept. For the infimum of the lattice the intent is empty and the extent contains all the objects, whereas for the supremum the intent contains all the attributes and the extent is empty. Concepts in the lattice are then grouped together depending on the relationships among them.

Concept analysis has been used as a data analysis method in other disciplines for a while. In software engineering its applications include program understanding, automatic modularization of legacy [5] [6], detection of configuration interference, class hierarchy transformation [13] and, source code restructuring [14].

In modularization, instead of decomposition concept analysis is used to identify grouping of program elements into modules. For example it is used to group together subroutines and global data structures into ADTs for object-oriented migration. As a result this technique is not directly applicable to the type modularization we are interested in.

3. Lattice of Concept Slices

We now propose a program representation formalism that we call the Lattice of Concept Slices. Based on this representation we are going to propose modularization techniques in the next section. The goal is to achieve a modularization such that each module implements preferably a single domain concept, each module is self-contained, there is minimal duplication in code and there is no side effect among modules

The formation of the lattice is a three-stage process – domain concept identification, computation of concept slices and finally, building and analyzing the lattice.

3.1 Identification of Domain Concepts

The first step is the identification of domain concepts in the program. This can be done using exhaustive concept assignment techniques like HB-CA or DM-TAO. Instead we take the simpler approach of structural and informal analysis of the source code.

In our approach the software engineer provides a list of domain concepts that are taken from the functional specifications of the system and are implemented in the given program. Furthermore, she associates such domain concepts with one or more program elements such as variables and structural idioms in the source code. The associations may be based on the *use* or *def* of a particular data type or variable, call to a procedure or method, a particular variable passed as parameter in a call, expressions matching on identifier naming or comments etc. These associations cannot be by no means exhaustive and only serve as a starting point of the analysis. In addition, the software engineer identifies some statements as key statements [8] that are believed to contribute the most in the computation of that domain concept.

IEEE COMPUTER SOCIETY

```
1: #include <stdio.h>
2: #define YES 1
3: #define NO  2
4: void main()
5: {
6:   int nl = 0;
7:   int nw = 0;
8:   int nc = 0;
9:   int inword = NO;
10:  int c = getchar();
11:  while (c!=EOF)
12:  {
13:    char ch = (char) c;
14:    nc = nc + 1;
15:    if (ch=='\n')
16:       nl = nl + 1;
17:    if (ch==' ' || ch=='\n' || ch=='\t')
18:      inword = NO;
19:    else if (inword == NO)
20:     {
21:      inword = YES;
22:      nw = nw + 1;
      }
23:    c = getchar();
    }
24:  printf("%d \n", nl);
25:  printf("%d \n", nw);
26:  printf("%d \n", nc);
  }
```

Figure 1: The line count program

Some domain concepts can be identified automatically based on a set of general criteria. The rationale behind the criteria is that any information being sent outside from the program or any change in the internal state that is externally visible are information that will be used by other parts of the program and hence are candidates for being part of a domain concept. Such criteria can be the identifiers such as – return parameters of a function or method, modified formal parameters that have been called by reference, variables in output/print statements, global variables or class attributes been modified. These identifiers are candidates for domain concepts and the statements that modify these identifiers will be considered as part of the corresponding domain concept. The software engineer may accept or reject the suggestions made automatically.

The outcome of this step is a set of domain concepts and associated with each of them is a set of program statements that implement the concept, where some of the statements are marked as key statements. Each of the concepts is a candidate to form a possible module and the associated statements will comprise the statements for the module.

As an illustration of the technique consider Figure 1 that illustrates a simple line count program taken from [7] that counts the number of lines, words and characters in a text file. We are attempting to modularize *main* function. The function outputs the calculation results of three variables – *nl*, *nw* and *nc* statement 24, 25 and 26 respectively. Hence the automatic identification technique suggests the possible presence of three domain concepts corresponding to these three variables. The software engineer confirms the suggestion and names the domain concepts as *Lines*, *Words* and *Chars* respectively. The *nl* variable is being computed (def-ed) in statement number 6 and 16. Statement 6 is the declaration and initialization and does not directly contribute to the computation of *nl,* whereas Statement 16 is the place where the main computation is being done. In this respect the *Lines* domain concept consists of statement 6, 16 and 24, where statement 16 is the key statement. Table 1 shows the list of domain concepts identified in this step and the statements associated with each of them.

In addition to the domain knowledge used by the software engineer to collect the significant variable that are believed to be associated with a specific domain concept, other semi-automated techniques can be also used. These include data mining, cohesion metrics, and data usage analysis [17].

Table 1: The Domain Concepts

Domain Concepts	Statements
Lines	6, **16**, 24
Words	7, **22**, 25
Chars	8, **14**, 26

3.2 Computation of Concept Slices

In this step we apply concept slicing as introduced in [8]. A concept slice is a slice of the program with respect to the domain concept. It is computed by taking slices of the program with respect to each of the statements belonging to a domain concept and then taking union of the slices.

The outcome of this step adds more statements to the domain concepts and makes the domain concept executable. The concept slices corresponding to the domain concepts are candidates for possible modules.

```
4: void main()
5: {
6:   int nl = 0;
10:  int c = getchar();
11:  while (c!=EOF)
12:  {
13:    char ch = (char) c;
15:    if (ch=='\n')
16:       nl = nl + 1;
23:    c = getchar();
    }
24:  printf("%d \n", nl);
  }
```

Figure 2: Slice on the Lines concept

1534-5351/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

```
4: void main()
5: {
7:   int nw = 0;
9:   int inword = NO;
10:  int c = getchar();
11:  while (c!=EOF)
12:  {
13:    char ch = (char) c;
17:    if (ch==' ' || ch=='\n' || ch=='\t')
18:       inword = NO;
19:    else if (inword == NO)
20:     {
21:       inword = YES;
22:       nw = nw + 1;
      }
23:    c = getchar();
     }
25:  printf("%d \n", nw);
  }
```

Figure 3: Slice on the Words concept

```
4: void main()
5: {
8:   int nc = 0;
10:  int c = getchar();
11:  while (c!=EOF)
12:  {
14:    nc = nc + 1;
23:    c = getchar();
     }
26:  printf("%d \n", nc);
   }
```

Figure 4: Slice on the Chars concept

Figures 2, 3 and 4 illustrate the concept slices corresponding to the domain concepts *Lines*, *Words* and *Chars*. Table 2 shows the refined statement list associated with the domain concepts after performing this step.

Table 2: The Concept Slices

Domain Concepts	Statements
Lines	4, 5, 6, 10, 11, 12, 13, 15, **16**, 23, 24
Words	4, 5, 7, 9, 10, 11, 12, 13, 17, 18, 19, 20, 21, **22**, 23, 25
Chars	4, 5, 8, 10, 11, 12, **14**, 23, 26

3.3 Building the Lattice

The third step is building the lattice of concept slices. This is done by performing a formal concept analysis on the concept slices. The context is formed by the relationship between the domain concepts and the program statements. The domain concepts make up the objects (extents) and statements make up the attributes (intents) of the context. A domain concept is considered to have a relationship with a statement if the statement is member of the concept slice for that domain concept. A concept lattice is formed by performing a formal concept analysis on the context.

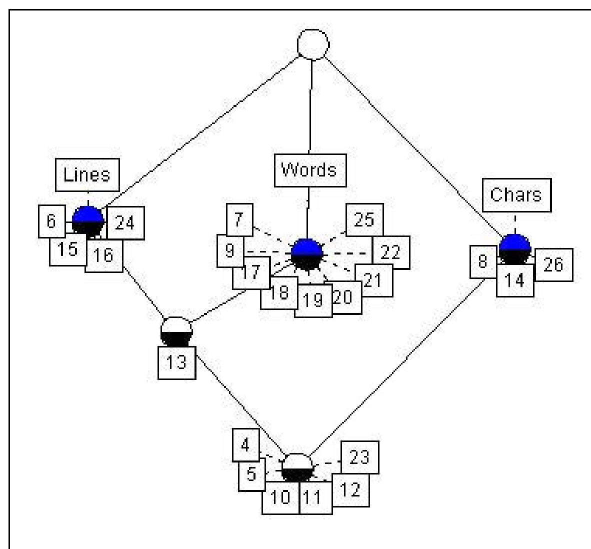

Figure 5: The Lattice of Concept Slices

The nodes of the lattice are called concepts[2], some of which correspond to the domain concepts. In the lattice each domain concept is related to all the statements in its node and all the nodes below it, whereas each statement is related to all the domain concepts in its node and all the nodes above it.

After applying a formal concept analysis on the elements illustrated in Table 2, we obtain the lattice given in Figure 5. The lattice consists of 6 nodes, 3 of which are the domain concepts. The domain concept *Lines* consists of statements 6, 15, 16, 24, 13, 4, 5, 10, 11, 12 and 23. The statement 13 belongs to both *Lines* and *Words* domain concepts.

An observation from the lattice is that the flow in the lattice structure does not correspond to the control flow or data flow in the program. Rather the statements are shuffled up and down among the nodes. This lattice serves as an input to the modularization algorithms that is discussed in more detail in the following section.

4. Modularization Algorithms

Once the concept lattice has been built, the next step of the process is to perform modularization based on it. We present two different algorithms to do so. The first one is a lattice clustering algorithm that aims at keeping the modules as separate from each other as possible and only merge them together if there is any side effects identified. The second one targets the elimination of code duplication and organizes the program in a module/sub-module hierarchy.

---

[2] A concept is a node of the concept lattice. It is totally different from domain concept.

IEEE COMPUTER SOCIETY

Lattice Clustering Algorithm
Input: Lattice of concepts
Output: A collection of modules

Step 1. In the bottom-up traversal of the lattice

- **1.1**. If this node is a domain concept create a new cluster containing this domain concept
- **1.2**. If this node is a critical node create a new cluster containing all the domain concepts in the sub-lattice starting from and above this node.

Step 2. Merge clusters with non-empty intersection

Step 3. Create a separate module for each of the clusters consisting of the union of the statements corresponding to the concept slices of the domain concepts inside each cluster.

Figure 6: Clustering algorithm

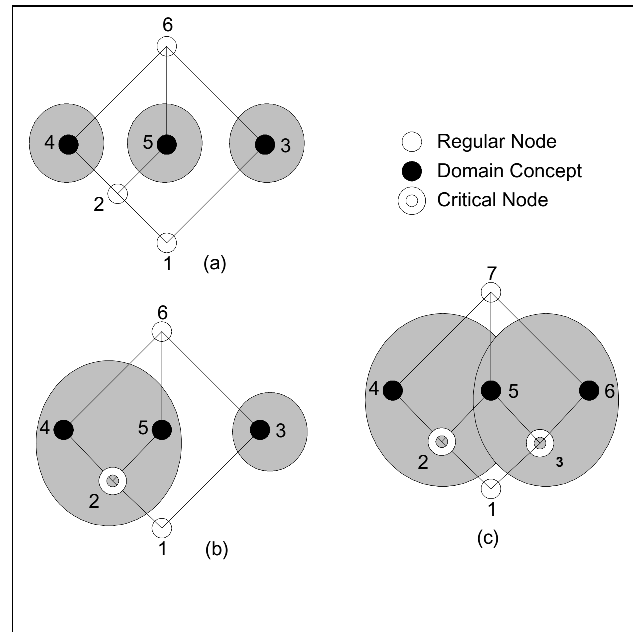

Figure 7: Lattice clustering

4.1 Lattice Clustering Algorithm

A node of the lattice is called a *critical node* if any statement in the node contains a Pricipal Variable[3] [8]. The existence of a critical node makes all other nodes above this node in the lattice interfering with each other. Furthermore, if the concept slices corresponding to the domain concepts above a critical node are decomposed as sperate modules then they might have side-effect with each other since one module will change the internal state of the program and it might cause the other module to work possibly with incorrect data.

In this step we run a clustering algorithm on the lattice to group domain concepts together to form a non-interfering set of modules. Figure 6 presents the algorithm.

For a lattice in Figure 7(a) there will be three clusters consisting of nodes 3, 4, and 5 respectively. For the lattice in Figure 7(b) due to the presence of the critical node 2, the node 4 and 5 will be clustered together and node 3 will be a in a separate cluster. In Figure 7(c) node 4 and 5 will be in one cluster due to the presence of critical node 2 and node 5 and 6 will be in another cluster sue to the critical node 3. But since the clusters overlap for node 5, they will be joined together forming a single cluster for the lattice.

[3] A Variable *V* in a set of statements *S* is a Principal Variable iff *V* is global or call-by-reference and is assigned in *S*.

In our line count example there is no critical node present in the lattice structure. Hence we have three clusters corresponding to three domain concepts. Resulting three modules, one for each *Lines*, *Words* and *Chars,* consisting of the statements in their corresponding concept slice given in Figure 2, 3 and 4.

4.2 Lattice Restructuring Algorithm

The modularization we have achieved in the previous section contains some duplication of code among the modules. This is mostly because of the control flow they share while computing the domain concepts. Also if the computation of one domain concept depends on the result of another domain concept, all the code from the second domain concept will be duplicated in the module for the first one.

Since the statements of a node of the lattice belongs to all the domain concepts above it, the shared control flow and the shared computations among the modules will appear at the lower part of the lattice. Domain concept specific computations will reside in their corresponding nodes. But since statements in the nodes are not necessarily consecutive, rather shuffled up and down among the nodes of the lattice, the statements in a node may not be self-contained in a control flow.

We attempt to eliminate the code duplication by restructuring the lattice is a way such that the statements in the nodes will have a complete control flow and the lattice structure will correspond to a module sub-module hierarchy.

IEEE COMPUTER SOCIETY

Lattice Restructuring Algorithm
Input: Lattice of concepts
Output: A collection of modules

Step 1. In the top-down traversal of the lattice

- **1.1.** If there is a sub-lattice starting and below this node, push all the statements down from this node to the bottom node of the sub-lattice
- **1.2.** Identify groups of statements in this node such that the statements of the group and the statements of the nodes above this node have a self-contained and complete control flow
- **1.3.** If this node is a domain concept, and consecutive statements are not found, identify each key statement as a group
- **1.4.** Push down the other statements in this node to the nodes right below it
- **1.5.** If there are multiple groups in this node, split them into separate nodes.
- **1.6.** Create a module from each statement group in this node. Declare all the variables used and defined in a module as parameters with variables being defined as call-by-reference and being used as call-by-value

Step 2. In the bottom-up traversal of the lattice

- **2.1**. From the module of this node call modules above this node from appropriate location.

Figure 8: Lattice Restructuring algorithm

The restructuring is done by pushing down the statements from the nodes that disrupt the control flow. Since only the domain concepts of the nodes above the current node depend on the statements of the current node, pushing a statement down to the node below the current node still keeps it as part of those domain concepts. The pushed down statement now becomes part of other domain concepts above the new location of the statement. This does not affect the computation of these other domain concepts since the other domain concepts were never dependent on the pushed down statement. It is to be noted that in worst case all the statements from all the nodes will be pushed down to the bottom node of the lattice and there will be no modularization at all. Figure 8 shows the restructuring algorithm.

In our example Statement 22 is a key statement of Words, which is part of the consecutive statement group 17, 18, 19, 20, 21 and 22. Also the control flow of this group is self-contained, hence it will form the module for Words and the statements 7, 9 and 25 will be pushed down to the node right below it. Since the statements being pushed down from Words does not affect the computations for Lines or Chars, putting the statements on the path of Lines or Chars will not hamper the computation of them. Figure 9 shows the final structure of the lattice after applying the algorithm on the lattice in Figure 5.

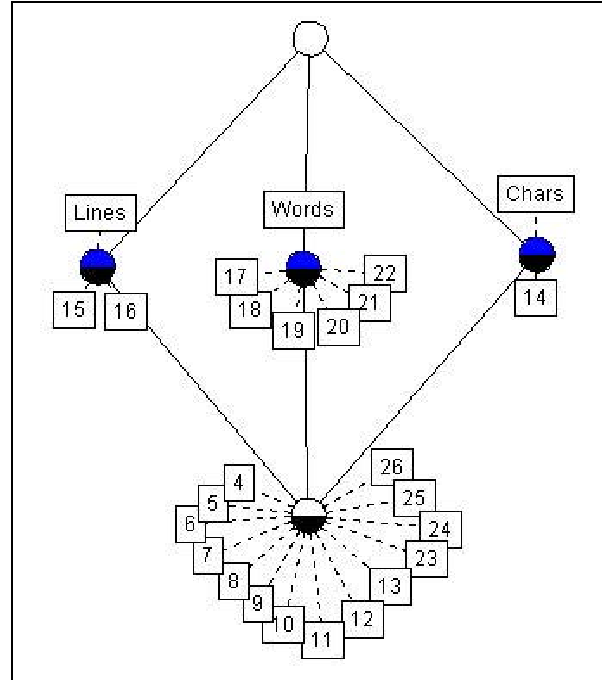

Figure 9: Restructured lattice

Statements in the node at the bottom of the lattice will form the main controller module, which will call the other modules that actually computes the domain concepts. Figure 10 shows the line count program after the modularization is performed.

5. Related Work

Source code analysis techniques like concept assignment, program slicing and formal concept analysis has been used in different combinations in the area of program comprehension and re-engineering.

Harman et al. [8] described a framework for unifying concept assignment with slicing. The paper presents three algorithms for combining the two analysis techniques and demonstrates their application in reuse and reverse engineering. The algorithms are for computing an executable concept slice, key statement analysis and concept dependence analysis.

IEEE COMPUTER SOCIETY

```
#include <stdio.h>
#define YES 1
#define NO  2

void chars(int &nc)
{
  nc = nc +1;
}
void lines(int &nl, char ch)
{
  if (ch=='\n')
    nl = nl + 1;
}
void words(int &nw, char ch, int &inword)
{
  if (ch==' ' || ch=='\n' || ch=='\t')
    inword = NO;
  else if (inword == NO)
  {
    inword = YES;
    nw = nw + 1;
  }
}
void main()
{
  int nl = 0;
  int nw = 0;
  int nc = 0;
  int inword = NO;
  int c = getchar();
  while (c != EOF)
  {
    char ch = (char) c;
    chars(nc);
    lines(nl, ch);
    words(nw, ch, inword);
    c = getchar();
  }
  printf("%d \n", nl);
  printf("%d \n", nw);
  printf("%d \n", nc);
}
```

Figure 10: The modularized line count program

Gallagher et al. [7] introduced the idea of a lattice of a slice to analyze the relationship among the decomposition slices of a program. The lattice is basically a relationship graph of the decomposition slices of a program. The lattice is used it to identify the ripple affects of changes made to a program during the maintenance of the program. The paper also gives a list of modifications that can be done in the program without any affect on the rest of the program.

Tonella [9] extended the work of [7] by performing formal concept analysis to build the concept lattice of decomposition slices as an extension of decomposition slice graphs. The paper demonstrates the use of the data structure in change impact analysis.

6. Conclusion and Future Work

This paper presents a modularization algorithm that aims to identify stand-alone code fragments that can be implemented as modules that deliver cohesive domain functionality. The algorithms are applied on a lattice based program representation formalism that is built by combining concept assignment, concept analysis, and slicing. The lattice representation models the contributions of the individual statements of a program towards the computation of different domain concepts implemented by it. The lattice structure serves as a primary data structure for program modularization. The algorithms to discover modules and sub-modules are applied on the concept lattice by clustering and restructuring it so that nodes in the lattice contain consecutive statements that form complete control flows.

Currently, we apply this modularization technique in migration of monolithically developed Servlet-based web applications into Model-View-Controller (MVC) pattern J2EE architecture. The work is motivated by early web-based multi-tier applications that were developed using Java Servlet technology in a monolithic way by embedding all the business logic and presentation logic inside one Java program. The business logic in such programs computes the domain concepts implemented in the Servlet. Similarly, the presentation logic relates also special type of domain concepts that deal with the user interface. Decomposed modules from the domain concepts that correspond to the business logic can be migrated as JavaBeans/EJBs while modules that correspond to the presentation logic can be migrated into JavaServer Pages.

In this paper, the proposed modularization process has been illustrated with an example C program. However, additional work is to be done in order to evaluate and apply the modularization process on larger programs with inter-procedural dependencies. One future direction is to use Inter-Procedural Dependency graphs, and incremental analysis so that larger program segments and groups of functions that relate to specific concepts can be individually analyzed.

This work is being conducted in collaboration with IBM Toronto Laboratory, Center for Advanced Studies, and co-sponsored by the Consortium for Software Engineering Research.

References

[1] Ted J. Biggerstaff, Bharat G. Mitbander and Dallas Webstar. The Concept Assignment Problem in Program Understanding. Proceedings of the 15th International Conference on Software Engineering. May 1993.

[2] Nicolas Gold and Keith Bennett. Hypothesis-based Concept Assignment in Software Maintenance. IEE Proceedings on Software. August 2002.

[3] Mark Weiser. Program Slicing. IEEE Transactions on Software Engineering. July 1984.

[4] Susan Horwitz, Thomas Reps and David Binkley. Interprocedural Slicing using Program Dependence Graphs. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 12 Issue 1. January 1990.

[5] C. Lindig, and G. Snelting. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. Proceedings of the 19th International Conference on Software Engineering. May 1997.

[6] M. Siff and T. Reps. Identifying Modules via Concept Analysis. IEEE Transactions on Software Engineering, Volume 25 Issue 6. November 1999.

[7] Keith B. Gallagher and James R. Lyle. Using Program Slicing in Software Maintenance. IEEE Transactions on Software Engineering, Volume 17 Issue 8. August 1991.

[8] Mark Harman, Nicolas Gold, Rob Hierons and Dave Binkley. Code Extraction Algorithms which Unify Slicing and Concept Assignment. Proceedings of Ninth Working Conference on Reverse Engineering. October 2002.

[9] Paolo Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. IEEE Transactions on Software Engineering, Volume 29 Issue 6. June 2003

[10] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. ACM Transactions on Programming Languages and Systems. July 1987.

[11] Susan Horwitz and Thomas Reps. The Use of Program Dependence Graphs in Software Engineering. Proceedings of the 14th International Conference on Software Engineering. May 1992.

[12] K.J Ottenstein and L.M. Ottenstein. The Program Dependence Graph in a Software Development Environment. Proceedings of the ACM Software Engineering Symposium on Practical Software Development Environments. April, 1984.

[13] G. Snelting. Software Reengineering based on Concept Lattices. Proceedings of the Fourth European Conference on Software Maintenance and Reengineering, March 2000.

[14] G. Antoniol, G. Casazza, M. di Penta and E. Merlo. A method to re-organize legacy systems via concept analysis. Proceedings. 9th International Workshop on Program Comprehension. May 2001.

[15] Frank Tip. A Survey on Program Slicing Techniques. Journal of programming languages. 1995.

[16] Andera De Lucia. Program Slicing: Methods and Application. Proceedings of First IEEE International Workshop on Source Code Analysis and Manipulation. November 2001.

[17] K. Sartipi, K. Kontogiannis. A User-assisted Approach to Component Clustering. In Journal of Software Maintenance: Research and Practice (to appear 2004).

1534-5351/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY