Interactive Program Modularization for Software Understanding via Formal Concept Analysis

Kunihiko Saito[1], Takeshi Kurita[2], Shinichiro Yamamoto[2]

[1] Shiga University, 1-1-1 Banba, Hikone, Shiga 522-8522, Japan,
saito@biwako.shiga-u.ac.jp,
[2] Aichi Prefectural University, 1522-3 Ibaragabasama Kumabari Nagakute-cho
Aichi-gun Aichi 480-1198, Japan

Abstract. Software understanding tools involve program modularization and visualization capabilities. However provided program modules do not always represent the accurate structure. Formal Concept Analysis and concept partition are methods to identify desirable program modules. Using concept partition, we define vertical and horizontal relation among partitions and cluster different abstract concept partitions to provide optimal program modules for software understanding. The optimality and applicability of our modularization method are illustrated with experimental results using C programs.

1 Introduction

One of the important themes of software engineering is to provide software understanding techniques that support the improving process of currently developing or legacy software; refactoring, testing and maintenance. Software understanding techniques provide effective representation of software, using various kinds of views. The views are implemented with program visualization, illustrating the overall software structure. To design and to utilize the views, optimal program modularization and efficient navigation method with which the user explores program modules, are necessary [1].

As a sophisticated program partitioning technique, Formal Concept Analysis [2] and concept partition [3] are presented. Concept analysis finds a potential group of modules that share common properties. [3] present a method to identify modules of C programs. However, concept partition gives rise to many alternative partitions at a time therefore optimal partitions are to be chosen for software understanding. If the partitions are ordered based on the abstraction level or clustered into groups, a complicated concept partition can be simplified and classified.

This paper presents techniques to choose an optimal set of modules and to relate each module to the others. We define a *partition hierarchy lattice* and *path* (See Section 3.1), and present a clustering method of different abstract concept partitions (See Section 3.2). A *partition hierarchy lattice* represents vertical and horizontal relations among program modules. A *path* is used for navigation in the modules. The clustering method relates different abstract concept partitions. By combining the concept partitions, a schematic view of a C program is depicted with different abstract modules.

To automate the program modularization process, we have implemented a prototype concept partition processor (See Section 4) that partitions a C program into optimal module groups. The tool builds a formal context, concept lattice, concept partition and *partition hierarchy lattice*. The tool supports visualization of program modularization, using function call graphs.

As an example, a program modularization of gzip-1.2.4 is presented (See Section 4). Through the experimental results we discuss the optimality and efficiency of the program modularization (See Section 5). Concept partition ensures the accurate partitioning of programs and provides the navigation capability that leads the user to desirable modules using resources of Formal Concept Analysis as a guide index.

2 Related Works

Software understanding becomes the center of attention in software engineering researches, because recent software systems have been developed by reusing and improving existing software. In this paper, we concentrate on the program partitioning techniques to provide optimal modules applicable to software understanding.

Siff and Reps [3] provide module identifying techniques via Formal Concept Analysis. They present a concept partition algorithm and decompose a program into modules. These techniques are useful for automatic program modularization and our research uses an algorithm explained in the research [3]. However, the algorithm sometimes yields many partition alternatives, therefore we present techniques to select optimal partitions from them.

Approaches of [3], [4] and [5] use Formal Concept Analysis for program refactoring and [6] applies the technique to program slicing. Our approach aims at utilizing the partitions for program understanding with program visualization. For example, we provide a schematic view of a program, using modules obtained from concept partitions (See Fig. 4).

Many works on software understanding present sophisticated representation of various kinds of programs [7] [8]. Almost software understanding tools decompose a program into modules using software metrics and graph layout algorithm [9]. In some case, the modules do not represent accurate structures. Compared to the tools, we try to provide program modules representing accurate structures of C programs.

3 Basic Methods

This section describes basic program modularization techniques with Formal Concept Analysis and concept partition. Based on the concept partition, we present *partition hierarchy* (Section 3.1) and *clustering of concept partitions* (Section 3.2) to identify optimal program modules. We obtain below concept partitions with the bottom up partition algorithm explained by Siff and Reps [3].

3.1 Partition Hierarchy

Many partitions are derived from a formal context if the number of *attributes* are large, therefore the partitions are to be ordered and classified. The partitions can be hierarchically ordered with *sub-partition* relation, by which a lattice structure is constructed. The lattice represents the abstraction level of each partition.

Formal Context, Concept and Partition The basis of Formal Concept Analysis is a *context* that is a triple $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$. $\mathcal{O}$ is a finite set of *objects* and $\mathcal{A}$ is of *attributes*. $\mathcal{R}$ is a binary relation between $\mathcal{O}$ and $\mathcal{A}$.

A *concept* is a pair of a finite set of objects and of attributes $(X, Y) : X \subseteq \mathscr{O}, Y \subseteq \mathscr{A}$. X and Y are bilaterally related with mappings σ and τ; Y=$\sigma(X)$, X=$\tau(Y)$: $\sigma(\mathcal{O}) = \{a \in \mathcal{A} | \forall o \in \mathcal{O} : (o, a) \in \mathcal{R}\}$ and $\tau(\mathcal{A}) = \{o \in \mathcal{O} | \forall a \in \mathcal{A} : (o, a) \in \mathcal{R}\}$. X is called *extent* and Y is called *intent*. A concept partition p is a set of concepts whose *extent* forms a partition of $\mathscr{O}$: $p = \{c_1, c_2, \cdots, c_n\}$. An atomic partition consists of atomic concepts and a trivial partition holds a concept that contains all objects of $\mathscr{O}$.

Partition Hierarchy Lattice Partition hierarchy is represented by the *sub-partition* relation in which partitions are hierarchically ordered according to the abstraction level. Given two partitions p_A and p_B, and let $c_i \in p_A$ and c'_{i_1}, $c'_{i_2} \in p_B$ hold a relation $c_i = c'_{i_1} \sqcup c'_{i_2}$, and two partitions share other concepts ($\because$ a difference set p_A-$\{c_i\}$ equals to p_B-$\{c'_{i_1}, c'_{i_2}\}$), we define *sub-partition* relation as follows.

Definition 1 (Sub-partition Relation) *Given two concept partitions p_A and p_B in a concept partition $\mathscr{P}$, partition hierarchy relation $\prec$ is defined as*

$$p_B \prec p_A$$

where: $c_i \in p_A$, c'_{i_1}, $c'_{i_2} \in p_B$, $c_i = c'_{i_1} \sqcup c'_{i_2}$, p_A-$\{c_i\} \equiv p_B$-$\{c'_{i_1}, c'_{i_2}\}$

The relation $\prec$ is defined with $c_i = c'_{i_1} \sqcup c'_{i_2}$ (other concepts are common), therefore the relation $\prec$ forms a complete partial order. A lattice structure can be constructed from a concept partition, using an atomic partition as top and trivial partition as bottom.

Definition 2 (Partition Hierarchy Lattice) *A partition hierarchy lattice is a lattice in which each node corresponds to a partition p_i. An edge between two nodes is drawn according to the sub-partition relation $\prec$.*

Along with a partition hierarchy lattice, a path that starts from a trivial partition and ends to an atomic partition, is defined. Abstraction level of the corresponding partitions is represented by a path.

Definition 3 (Partition Path) *A partition path is a sequence of partitions p_{s_i} such that*

$$p_{s_0(=atomic)} \prec p_{s_1} \prec p_{s_2} \prec \cdots \prec p_{s_n(=trivial)}$$

Two consecutive partitions p_A and p_B in a path P_S are differed using only concepts $c_i \in p_A$ and c'_{i_1}, $c'_{i_2} \in p_B$. Each concept holds a set of *attributes* Y_{c_i}, $Y_{c'_{i_1}}$ or $Y_{c'_{i_2}}$, consequently there exists a pair of difference sets of *attributes* $\{Y_{c'_{i_1}}$ - Y_{c_i}, $Y_{c'_{i_2}}$ - $Y_{c_i}\}$ that we call *partitioning attributes pair*. The pair specifies the difference between two consecutive partitions.

Definition 4 (Partitioning Attributes Pair) *Given two consecutive partitions p_A and p_B, partitioning attributes pair is defined as*

$$\{Y_{c'_{i_1}} - Y_{c_i}, Y_{c'_{i_2}} - Y_{c_i}\}$$

where: $c_i \in p_A$, c'_{i_1}, $c'_{i_2} \in p_B$, $c_i = c'_{i_1} \sqcup c'_{i_2}$, $Y_{c_i} \subset Y_{c'_{i_1}}$, $Y_{c_i} \subset Y_{c'_{i_2}}$

3.2 Clustering of Concept Partitions

To implement a schematic view of a program with various viewpoints, we provide a method to cluster multiple concept partitions derived from a program. Different abstract concept partitions are built by re-choosing *objects* and *attributes*. For example, when an *attribute* a is replaced by a set of *attributes* $\{a'_i\}$, a concept lattice is expanded. In contrast, a complicated concept lattice can be reduced into a manageable size. We define *attribute decomposition* as follows.

Definition 5 (Attribute Decomposition) *Given two contexts* $\mathscr{C}_A = (\mathscr{O}, \mathscr{A}, \mathscr{R})$ *and* $\mathscr{C}_B = (\mathscr{O}, \mathscr{A}', \mathscr{R}')$*, and given an attribute* $a \in \mathscr{A}$ *and* $\{a'_i\} \subseteq \mathscr{A}'$*, attribute decomposition* $\rightarrow$ *is defined as*

$$a \rightarrow \{a'_i\}, \;\; where \;\; \tau(a) \equiv \bigcup \tau(2^{\{a'_i\}}) - \tau(\emptyset)$$

Similarly, *attribute composition* $\leftarrow$ is defined.

Then we define a relation between two concept partitions built by *attribute decomposition* as *re-partition relation*.

Definition 6 (Re-partition relation) *Given two contexts* $\mathscr{C}_A$ *and* $\mathscr{C}_B$*, and suppose* $\mathscr{C}_B$ *is built from* $\mathscr{C}_A$ *with attribute decomposition. Then a set of objects* X_c *of each atomic concept* $c \in \mathscr{C}_A$ *equals to a union of sets of objects* $\{X_{c'_i}\}$ *of* $\{c'_i\} \subseteq \mathscr{C}_B$*:* $\{c'_i\}$ *are also atomic concept.*

$$\forall c = (X_c, Y_c) \in \mathscr{C}_A, \exists \{c'_i = (X_{c'_i}, Y_{c'_i})\} \subseteq \mathscr{C}_B, \; s.t. \; X_c \equiv \bigcup X_{c'_i}$$

For example an it attribute ”human” can be decomposed into {man, women}. Generic-specific relation used in object oriented programming also generates *attribute (de)composition*.

4 Example of Program Modularization

This section describes experimental results of program modularization. Using gzip1.2.4, formal contexts, concept partitions (Section 4.1) and more detail information on program modularization such as *partition lattice, path* (Section 4.2) and combination of different abstract concept partitions (Section 4.3) are presented.

To create a *context*, concept lattice and partition, a prototype analyzer and concept partition processor is implemented on **Sapid** [10]. **Sapid** provides comprehensive information on a target program written in ANSI-C and Java. The analyzer creates a *context* from the source program according to a choice of *objects* and *attributes*.

4.1 Building a Formal Context and Partition

We analyze well-known open-source software, gzip-1.2.4 written in C, whose size is approximately 231.9K LOC (Line Of Code). To build a *context* representing program modularization, *function* is used as *objects* and *aggregate type* variables in C programs

Table 1. Experimental results of gzip-1.2.4

context	using attribute as	object	attribute	concept/atomic		partition	ave/max module		layer	path
$\mathscr{A}$	struct type	76	3	6/4	$\mathscr{P}_A$	5	18.8/64	$\mathscr{P}_{L_A}$	4	3
$\mathscr{B}$	struct type variables	76	11	53/11	$\mathscr{P}_B$	246	6.8/44	$\mathscr{P}_{L_B}$	11	-

as *attributes*, following the research [3]. These variables relate each *function* to the others.

We present two *attribute* choices of the aggregate type according to whether occurrences of different variables that belong to a struct type, are dealt with one *attribute* or not. Each *context* (Table 1.) are automatically built by the prototype tool. For example, *context* $\mathscr{A}$ holds 76 *objects*, 3 *attributes* and 6 concepts. Concept partition $\mathscr{P}_A$ holds 5 partitions and $\mathscr{P}_B$ holds 246 partitions.

4.2 Hierarchy of Partitions

From a concept partition $\mathscr{P}_A$ and $\mathscr{P}_B$, a partition lattice $\mathscr{P}_{L_A}$ and $\mathscr{P}_{L_B}$ are derived. One of the partition path of $\mathscr{P}_{L_B}$ = $\{p_{190}, p_{248}, p_{260}, p_{263}, \cdots, p_0\}$ is listed below.

	partition	concepts	*partitioning attribute pair*
original	p_{190}	$\{c_{53}\}$	
bi-partition	p_{248}	$\{c_{21}, c_{32}\}$	{} {d_desc, l_desc}
tri-partition	p_{260}	$\{c_{21}, c_{42}, c_{64}\}$	{istat} {dyn_dtree}
tetrameric-partition	p_{263}	$\{c_7, c_{326}, c_{42}, c_{46}\}$	{} {d_desc, l_desc}
$\vdots$	$\ddots p_0$	$\{c_0, c_1, \cdots, c_{10}\}$	

The path presents from bi-partition to eleventh-partition according to the *partitioning attributes pair* (Fig. 1-3). Bi-partition p_{248} = $\{c_{21}, c_{32}\}$ and an original graph p_{190} are differed with *partitioning attribute pair* ($\emptyset$, $\{d\_desc, l\_desc\}$). Each concept is figured with function call graph.

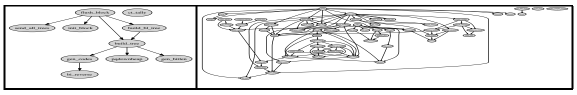

Fig. 1. Bi-partition

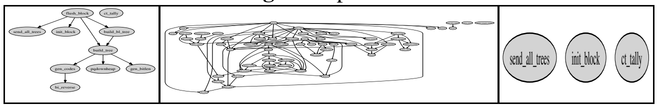

Fig. 2. Tri-partition

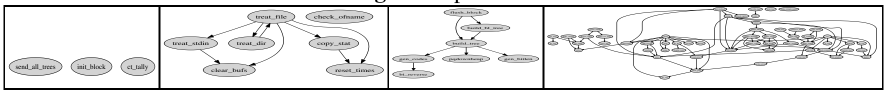

Fig. 3. Tetrameric-partition

4.3 Combination of Concept Partitions

Concept partitions $\mathscr{P}_A$ and $\mathscr{P}_B$ can be used complementary. For example, *Partition hierarchy lattice* $\mathscr{P}_{L_B}$ is too complicate to obtain a desirable path. In contrast, partition lattice $\mathscr{P}_{L_A}$ holds 4 layers and 3 paths (See Table 1.) therefore $\mathscr{P}_{L_A}$ can be utilized to depict a schematic view.

A *re-partition relation* of two partitions $\mathscr{P}_A$ and $\mathscr{P}_B$ is illustrated below. For example, an extent of C_2: $\{ct\_tally, send\_all\_trees, ct\_init, init\_block, set\_file\_type, scan\_tree, send\_tree\}$ in $\mathscr{P}_A$ is re-partitioned into $\{c_2 : (send\_all\_tree, init\_block), c_4 : (scan\_tree, send\_tree), c_6 : set\_file\_type, c_7 : ct\_tally, c_8 : ct\_init\}$ in $\mathscr{P}_B$. Other *attributes* of $\mathscr{P}_A$ are also decomposed into some *attributes* of $\mathscr{P}_B$.

concept $\mathscr{A}$	attribute	concept $\mathscr{B}$
C_2	ct_data={ d_desc, bl_tree, l_desc, dyn_dtree, dyn_ltree, bl_desc }, not-tree_desc	c_2, c_4, c_6, c_7, c_8
C_3	ct_data, tree_desc ={static_dtree, configuration_table istat, static_ltree, longopts}	c_1, c_3
$C_5 = (C_2, C_3)$	ct_data	$c_1, c_2, c_3, c_4, c_6, c_7, c_8$

Fig. 4 is a schematic view of gzip-1.2.4 with two different abstract concept partitions $\mathscr{P}_A$ and $\mathscr{P}_B$. There exist 18 segments in the nest structure and its depth is 4. Function call relations are used to connect corresponding two segments.

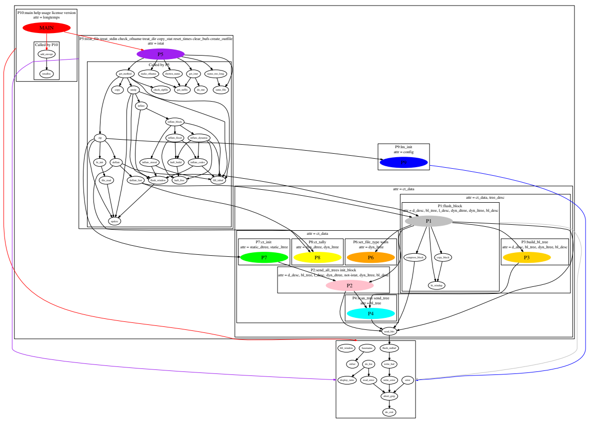

Fig. 4. A schematic view of program modularization: gzip-1.2.4

5 Discussion

This section discusses the program modularization technique presented in this paper. Section 5.1 describes the optimality and efficiency of the modularization. Section 5.2 presents other case studies applying concept partition to a number of C programs. To obtain schematic views of various programs, we use alternative choices of concept partition based on a reverse engineering tool **Sapid** [10].

5.1 Schematic View of Programs

Our research aims at providing a *user interface* for software understanding as an "Entrance" of CASE tools. To depict the *user interface*, a schematic view that links to more detail information is necessary. First we asses the optimality of the size and the number of modules, then we demonstrate that the modules are effectively related with *attributes*.

Optimal Modularization An optimal size of modules for depicting a schematic view depends on the usage of modules. It is widely known that the optimal size is less than 20 and sufficient size less than 50.

Program modules are represented with atomic concepts therefore the number of modules is counted by the number of atomic concepts. An experimental result using gzip-1.2.4, which has 231.9 KLOC and 76 user-defined functions, shows that *context* $\mathscr{A}$ holds 4 and $\mathscr{B}$ holds 11 atomic concepts. The max module size derived from a $\mathscr{A}$ is 64 and $\mathscr{B}$ is 44. Hence, the size and the number of modules of $\mathscr{B}$ are optimal.

Fig. 4 is depicted with the 11 atomic concepts of $\mathscr{B}$ using function call relation. Moreover, Fig. 4 uses a nested relation among partitions in a partition lattice $\mathscr{P}_{\mathscr{L}_{\mathscr{A}}}$ that has 4 layers.

Interactive Navigation Formal Concept Analysis ensures that each concept partition represents a specific program structure according to the choice of *objects* and *attributes*. Gzip-1.2.4 example illustrates a program structure with aggregate types of C programs, therefore each module (partition) is related based on the "accurate" program structure, and obtained schematic views are similar to UML.

The hierarchical relation among partitions in $\mathscr{P}_{\mathscr{B}}$ is presented in Fig. 1,2 and 3, where each module diagram is obtained by an interactive choice of concerning *attributes* (*partitioning attributes pair*). The prototype tool supports the interactive choice of the partitions. Using the tool, the user can find desirable modules and access detail information on programs, provided as hyper documents [11].

5.2 Other Case Studies

Table 2. presents concept partitions of various C programs, using *struct type* variables as *attribute* (See B in Table 1.). However, some programs can not be optimally partitioned as gzip-1.2.4. $Gnugo$ and $whetstone$ derive few concepts in comparison to its program size. In contrast, $bison$ and $bash$ derive many concepts and partitions therefore it is difficult to calculate concepts and partitions.

To provide a schematic view of the programs, alternative choices of *attributes* are introduced. $Gnugo$ uses *global variable*, and $bash$ and $bison$ use *struct type* (See A in Table 1.) as *attributes*. Then, we obtain optimal concept partitions given in the right part of Table 2. Alternative choices of concept partition require a reverse engineering method. Our prototype tool supports a user defined design of formal context and concept partition. The analyzer is developed on the software repository database of **Sapid** [10], which is written in XML [11] [12]. Therefore alternative concept partitioning is implemented with DOM programs.

Table 2. Experimental results of well-known C programs

	LOC	object	attribute	concept	partition	alternative attribute	object	attribute	concept	partition
whetstone	37.6	6	1	4	2					
gnugo-1.2	98.6	30	1	4	2	global variable	30	22	217	172
fileutils-ls-3.14	1870.2	115	15	52	778					
bison-1.5	986.0	162	30	-	-	struct type	162	11	168	68572
bash	9550.0	1078	82	-	-	struct type	1078	8	117	23121

6 Summary and Future Work

We have motivated to provide easily understandable program representation with modules of manageable size. This paper presented a program modularization technique with Formal Concept Analysis. We defined hierarchy of partitions with *partition hierarchy lattice*. To provide schematic views of programs, a clustering technique of modules were presented and obtained modules could be depicted in a diagram. We implemented a prototype tool for program modularization and through the experimental results, the optimality of modularization and efficiency of navigation method were demonstrated.

We plan to develop a practical software understanding tool based on our techniques. Then the prototype concept processor is to be more sophisticated. For example, the generating process of an alternative *context* and concept partition should be fully automated. The tool can be linked to hyper documents based on XML. The support for the modularization of Java programs is also the next trial.

References

1. K. Wong. Rigi User's Manual. The Rigi Group, Victoria, Canada, version 5.4.4 edition. 1998.
2. B. L. Achee and L. Carver. A greedy approach to obtain object identification in imperative code. Third workshop on Program Comprehension. 1994.
3. Michael Siff and Thomas Reps. Identifying modules via concept analysis. International Conference on Software Maintenance. October 1997.
4. Geogor Sneltug. Reengineering Class Herarchies Using Concept Analysis. ACM SIGSOFT Symposium on the Foundation of Software Engineering. 1998.
5. Gregor Snelting. Concept Lattices in Software Analysis. Formal Concept Analysis. 2005.
6. Raihan Al-Ekram and Kostas Kontogiannis. Source Code Modularization Using Lattice of Concept Slices. 8th European Conference on Software Maintenance and Reengineering. 2004.
7. Horwitz, S.and Reps. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12. January 1990.
8. K. Wong. On inserting program understanding technology into the software change process. Workshop on Program Comprehension 1996 Proceedings. 1996.
9. M.-A. D. Storey and H. A. Muller. Graph layout adjustment strategies. Graph Drawing 1995 Proceedings. 1995.
10. Yoshida Atsushi, Yamamoto Shinichirou and Agusa Kiyoshi. A Software Manipulating Language for a MetaCASE. The First International Congress on META-CASE. 1995.
11. K.Maruyama and S.Yamamoto. A CASE Tool Platform Using an XML Representation of Java Source Code. Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM04). 2004.
12. K.Maruyama and S.Yamamoto. Design and Implementation of an Extensible and Modifiable Refactoring Tool, 13th IEEE International Workshop on Program Comprehension. 2005.