A Method to *Dust* Legacy Code

G. Antoniol*, G. Casazza**, M. Di Penta *, E.Merlo***
antoniol@ieee.org {gec,dipenta}@unisannio.it ettore.merlo@polymtl.ca

(*) *University of Sannio, Faculty of Engineering - Piazza Roma, I-82100 Benevento, Italy*
(**) *University of Naples "Federico II", DIS - Via Claudio 21, I-80125 Naples, Italy*
(***) *Dep. Electrical and Computer Engineering Ecole Politechnique, C.P. 6079, Succ. Centre Ville, Montreal, Quebec, Canada*

Abstract

This paper proposes a method to restructure the architectural source code files organization. The method exploits concept analysis and is based on the extraction of compiling, linking and inclusion dependencies.

The method was applied to public domain and freely available software systems developed with C programming language.

Results presented and discussed in the paper are encouraging: the method was always able to identify, for each executable (i.e., main), all the other required modules to successfully compile a working system. Moreover, the candidate libraries retrieved by the method almost always match those created by developers. Finally, the directories organization was judged very effective even if different from the original one.

Keywords: architectural restructuring, dependency removals, program understanding

1. Introduction

A large legacy software system typically underwent and is subject to many maintenance interventions to remove defects, to cope with platform changes (hardware as well as software) and, more generally, to adapt to ever changing organizational needs.

It often happens that such interventions are undocumented, spread along several years, deteriorating the code structure and the files/directories organization, introducing unwanted/unexpected dependencies. Documentation, executables, sub-systems, modules and files/directories organization, the *as-is* architecture, may no longer correspond to the original software architecture.

In this scenario program understanding and software modification activities are extremely difficult: any modification may produce unpredictable side effects on other portions of the system. Moreover, repeated undocumented maintenance interventions on the code reduce one's understanding of the system, substituting reliable information on the software with smoky perceptions and folklore [7]. When reliable information are lost, a large system would require a very large compile time for any change, no matter how small a modification is, the entire system needs to be recompiled and extra modules linked to safely re-build executables [16]. As a side effect, executables may well be quite larger than needed, wasting valuable resources such as central memory and disk space [16].

This paper proposes a method to restructure the architectural source code files organization. The method is based on the analysis of compiling, linking and inclusion dependencies and, as widely adopted for Unix applications, a correspondence between directories and systems/sub-systems is assumed. In other words, cross-directories dependencies are considered undesired. The method identifies libraries and cross-directory/cross-sub-system dependencies, then it restructures the directories/files organization in order to highlight such dependencies. It can be decomposed into three sequential steps. The first step was inspired by the work [16], it accomplishes with different tools to the same purpose: *main* modules are identified and by performing the transitive closure of modules dependencies compilable units are recovered. The subsequent steps are semi-automatic steps: concept analysis is exploited to infer a new architectural organization clustering modules together. More precisely, the second step reorganizes files into library, while the third step defines a new directories organization.

The method was applied to public domain and freely available software systems developed with C programming language; the systems were chosen to constitute an *Oracle*: the systems and their organizations were well known to the authors.

The *legacy system* situation was mimicked by flattening

out, when present, the directory organization (all files were placed in a common directory). In other words, the worst case situation, where all information were lost, was artificially obtained. By qualitatively comparing the new libraries and directories organization with the original one, a measure of the method performance was achieved. In the experimentation, since C does not have a construct to identify modules, it was assumed a correspondence between files and modules.

Results presented and discussed in the paper are encouraging: the method was always able to identify, for each executable (i.e., main), all the other required modules to successfully compile a working system. Moreover, the candidate libraries retrieved by the method almost always match the actual *as is* libraries. Finally, the directories organization was judged very effective even if different from the original one.

The remainder of the paper is organized as follows. First, basic concept analysis notions are summarized for the sake of completeness. Then, Section 3 introduces the proposed method. Section 4 presents the case study, followed by a discussion of the obtained experimental results. Then, Section 3 describes the tool support adopted. The conclusion section summarizes lesson learned and outlines foreseeable research directions.

2. Background Notions

Concept Analysis (CA) is a mathematical tool that allows to identify groups of objects having common attributes. First CA study dates back to 1940, when G. Birkoff proved the possibility of construction of a lattice starting from binary relations between a set of "objects" and their "attributes" [3].

Since 1982 [17] there has been a flourishing of activities in several area of software engineering: for example, CA has been used for remodularization of legacy code [12], for identifying objects [8] and software configurations [6].

CA can be thought of as the process of searching "rectangles" in a boolean table representing a relation between objects and attributes. Thus, a concept is a maximal rectangle in a table for a given set of attribute, rectangle perhaps obtained by means of columns/rows permutations.

More precisely [2], CA starts with a *context*, a triple, $C = (O, A, R)$, where O is a finite set of objects, A is a finite set of attributes, and $P \subseteq O \times A$ is a relation between O and A. If the pair $(o, a) \in P$, it can be said that object o has attribute a. Given a set of objects $X \subseteq O$,

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

is the set of *common attributes* while, given $Y \subseteq A$,

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

is the set of *common objects*.

These two mappings ($\sigma : 2^O \rightarrow 2^A \; and \; \tau : 2^A \rightarrow 2^O$) form a *Galois* connection. Note that the mappings are antimonotone:

$$X_1 \subseteq X_2 \Rightarrow \sigma(X_2) \subseteq \sigma(X_1)$$

$$Y_1 \subseteq Y_2 \Rightarrow \tau(Y_2) \subseteq \tau(Y_1)$$

and extensive:

$$X \subseteq \tau(\sigma(X)) \; and \; Y \subseteq \sigma(\tau(Y)).$$

A concept is a pair of sets (X, Y) where $X \subseteq O$ is called the *extent*, $Y \subseteq A$ is called the *intent*, and $Y = \sigma(X)$, $X = \tau(Y)$. That is, a concept is a maximal collection of objects sharing common attributes. The set of all concepts is denoted by $B(O, A, R)$. Furthermore, a concept (X_1, Y_1) is a subconcept of another concept (X_2, Y_2) if $X_1 \subseteq X_2$. This imposes a partial order relation on $B(O, A, R)$, and it can be written that $(X_1, Y_1) \preceq (X_2, Y_2)$. The partial order $\preceq$, can be used to build a lattice called *concept lattice*, where each node represents a concept. The concept lattice introduces a hierarchical clustering of objects and attributes, where suprema factor out common attributes, while infima factor out common objects. More details can be found in [2, 8]

3. Method

The method presented in this paper aims to reorganize the structure of a software system highlighting the actual dependency structure. It assumes that software system's source files are contained in a flat directory while external libraries (i.e., libraries not specifically created by the system developers but linked by the system) are spread across operating system directories.

In the following the term *module* will be used to indicate a file containing source code while the term *object*, if not differently stated, identifies a file containing object code. Moreover, the term *executable* will be used to refer the result which is obtained through the compilation and the linking of a main module (i.e. the module containing the main function) with all the other required modules and libraries (i.e. all the files defining symbols used, directly or transitively, by the function main). Finally, it is assumed that a system is composed by several executables.

As shown in Fig. 1, the method is based on three sequential steps:

1. *Executables Identification and Module Dependencies Detection*: the main module of each system executable is detected; modules and external libraries required to produce each executable are identified;

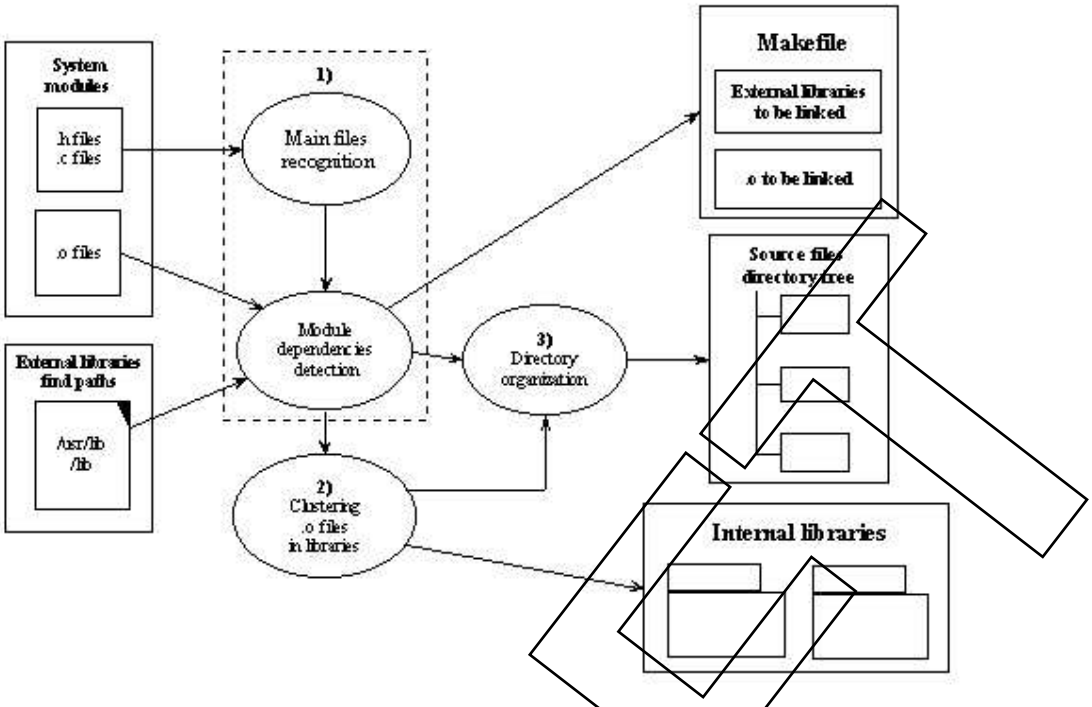

Figure 1. Main blocks of the process

2. *Clustering Objects into Libraries*: files containing object code linked to build several executables are clustered into libraries, (the libraries identified in this step will be referred in the following as internal libraries);

3. *Directories Organization*, the source code files (e.g. .c and .h, or .java) are organized into a new directory hierarchy.

The following subsections details each step using the system described by Table 1 as a running example. The example is based on an hypothetical system having four different executables: exe1, exe2, exe3 and exe4. The files required for the compilation of each executable are shown in the second column of the table. Table 1 could be thought of as a compact representation of a CA context: objects are contained in the first column (the executables), attributes are the elements of the second column (objects required).

3.1. Executables Identification and Modules Dependencies Detection

This step, given a platform configuration, produces a makefile that generates all the executables constituting the analyzed software system. In particular, it works as follows: *main* modules of the executables composing the system are detected. For languages such as C, C++, Java the *main* module is the file containing the definition of the *main* function. Then, the object file of each executable main module is analyzed, and the list of unresolved symbols is retrieved. These symbols are recursively resolved, searching the definitions in the other object files: modules resolving symbols are tagged as required to build the executable.

Finally, in order to determine which libraries must be linked to the executable, all remaining unresolved symbols are searched into external libraries (system libraries and third-part libraries).

Executable	Objects required
exe1	f1, f12, f5, fl1, f2, fB
exe2	f2, f12, f23, f6, fl1, f2, fB
exe3	f3, f23, f7, fl1
exe4	f4, f8, fl1

Table 1. Objects needed to build each executable

3.2. Clustering Objects into Libraries

By analyzing the lists of objects linked to produce executables, hints on potential libraries can be gained. The goal

of this step is to identify sets of objects linked by several executables. To cast library identification into a CA problem a *context* has to be specified. In the proposed approach, the *set of objects* corresponds to the set of executables, the *set of attributes* is obtained by collecting the set of object files. The *binary relation* is represented by the *is-required-to-link* relation.

The library identification process was based on the observation that object files listed into nodes close to the top (or listed in the top itself) are used by many (all) executables. Therefore, nodes near to the top are to be considered good candidates for the creation of a library. For example, the object *fl1* in Table 1, being used by all executable, is a candidate for a library; furthermore, *fl2* and *fl3* are used by `exe1` and `exe2`, so they also could be clustered into a different library. In some cases, the clusterization into a library of a large number of object files used by few executables may be convenient. This requires the analysis of the nodes near the bottom.

Like many other CA approaches [13, 12] to software reengineering problems, this step is semi-automatic: while the construction of concept lattice is performed automatically from results of the previous step, lattice analysis requires a manual inspection to verify the usefulness of the candidate libraries. Once candidate files to build libraries are identified, each object file is assigned to a library according to the node position in the lattice (i.e., depth first traversal). This avoids assigning a file to more than one library.

3.3. Directories Organization

This step takes as input the results produced by the two previous steps. The directories containing the system source code files (i.e. .c and .h files) are organized exploiting the makefile produced by the first step and taking into account the .o files grouped into libraries by the second step.

A top level a directory named `lib` is created: it contains a subdirectory for each identified library. Each subdirectory stores the modules required to generate the library.

Concepts		
Id	Objects	Attributes
c0	{∅}	{f1, f2, f3, f4, f5, f6, f7, f8, f12, f23}
c1	{exe1}	{f1, f5, f12}
c2	{exe2}	{f2, f6, f12, f23}
c3	{exe3}	{f3, f7, f23}
c4	{exe4}	{f4 f8}
c5	{exe1, exe2, exe3}	{∅}
c6	{exe1, exe2}	{f12}
c7	{exe2, exe3}	{f23}

Table 2. Sample Example: Concepts

The directories organization of the remaining modules (i.e. the .c and .h files not required to produce the .o files clustered into libraries) follows an executables perspective. As above, the *set of objects* corresponds to the set of executables, while the *set of attributes* is obtained by collecting the .c and .h files. The *binary relation* is represented by the *is-required-to-compile* relation (information recovered by step 1 and 2).

Table 2 reports the concepts retrieved by the concept analysis algorithm for the running example of Table 1.

The procedure used to derive the directories organization takes as input the set of concepts evaluated by the CA algorithm. The procedure firstly eliminates all the concepts having empty sets of objects and/or attributes (Table 2 concepts c0 and c5). Then, all the attributes that are shared by several concepts are identified (e.g., the attribute f12 is shared by c2 and c6). Common attributes are removed from all the concepts having a set of objects with a cardinality equal to one. Giving raise to what we called "*pruned concepts*". Table 3 reports the pruned concepts corresponding to Table 2.

Concepts		
Id	Objects	Attributes
c1	{exe1}	{f1, f5}
c2	{ exe2}	{f2, f6}
c3	{ exe3}	{ f3, f7}
c4	{exe4}	{ f4, f8}
c6	{exe1, exe2}	{ f12}
c7	{exe2, exe3}	{f23}

Table 3. Sample Example: Concepts After Pruning Common Attributes

A hierarchy with 2 levels of directories is created. A directory is created for each pruned concept: the directory content is defined by the attributes set of the pruned concept (i.e. the set of .c files grouped by the leaned concept), while the directory hierarchy level depends on the cardinality of the pruned concept objects set (i.e. the set of executables grouped by the pruned concept).

The directories belonging to the first level are obtained by the pruned concepts having a set of objects with a cardinality exactly equal to one. In fact, such concepts group for each executable the maximum number of .c file required exclusively to compile that executable. It is worth noting that the procedure always creates a number of first level directories equal to the number of executables. In fact, there will always be a pruned concept having a set of objects with cardinality equal to one for each system executable, due to the *main* module.

In the running example, four first-level directories, see

figure 2, are created starting from Table 3: `dir-exe1`, `dir-exe2`, `dir-exe3` and `dir-exe4`.

The directories belonging to the second level are created taking into account the pruned concepts having a set of objects with a cardinality greater than one. Such concepts capture an important class of information: they actually group the maximum set of executables requiring for the compilation the same set of .c files. In other words, such concepts highlight cross-directory/cross-sub-system dependencies. For example, the concept c6 highlights that the compilation of both exe1 and exe2 requires the file f12. In this case a physical directory, `dir-exe1_exe2`, is created into `dir-exe1` and it is symbolically linked to `dir-exe2`. The directory `dir-exe1_exe2` will contain the file f12 (see Figure 2).

Finally .h files are organized: in each first level directory previously created a subdirectory, named `include` is added. The `include` subdirectory will hold all the .h files directly included by the source files contained in the first level directory.

It is worth noting that whenever a .c or a .h file is required by several directories it will physically be contained in one only directory, then a symbolic link will be created to all the other directories requiring it.

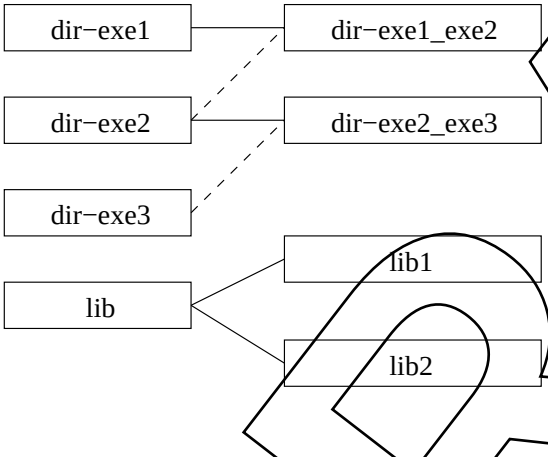

Figure 2. Directories Organization (dotted lines are intended as symbolic links)

4. Case Study

To assess the feasibility and the effectiveness of the approach, several versions of two software systems, *mSQL* and *Samba* were considered. MiniSQL (mSQL) is a free, open source relational database management system that implements a subset of SQL; it is manufactured and distributed by Hughes Technologies (http://www.hughes.com/au).

The first generation product, i.e. mSQL 1.0, was designed to provide high speed access to small data sets using very few system resources on an average UNIX workstations. mSQL 1.0 has been available in various forms since June 1994 and has undergone many enhancements to become a very popular and stable database. Mini SQL 2.0.X is the evolution of the mSQL 1.X database system; mSQL 2 goes beyond the initial design goals of mSQL 1 and provides functionality suited to larger applications. Moreover, mSQL 2.0.X is bundled with W3-mSQL 2.0.X, the second generation WWW interface package. The new W3-mSQL code provides a complete scripting language, with full access to the mSQL API, within an HTML tag.

System	KLOC	# .c	# .h	# Executables
Samba 1.9.08	30	25	11	7
Samba 2.0.5	145	211	38	12
mSQL 1.0.6	18	33	14	6
mSQL 2.0.8	54	82	42	12

Table 4. Main Features of the Analyzed Systems

Samba is a software system freely available under the GNU General Public License (http://www.samba.org); basically, it is a file server that runs on Unix and other operating systems (usually to share resources between Unix-based systems and Microsoft-based systems). The code has been written to be as portable as possible. It has been "ported" to many unixes (Linux, SunOS, Solaris, SVR4, Ultrix, etc.), porting very often requires changing only a few lines of code. Samba consists of two key programs, plus a bunch of other utilities. The two key programs are `smbd` and `nmbd`. They implement four basic services: file sharing & print services, authentication and authorization, name resolution and service announcement (browsing). Moreover, Samba comes with a variety of utilities. The most commonly used utilities are: `smbclient` a simple SMB (Server Message Block, a protocol for sharing general communications abstractions such as files, printers, etc.) client; `nmblookup` a NetBIOS name service client, and `swat` which is the Samba Web Administration Tool, it allows the configuration of Samba remotely, using a web browser.

5. Tool Support

A Perl script was developed to support the reconstruction of the the system makefile; it uses the `nm` Unix utility to identify both the defined and undefined symbols used by each .o file. The script takes as input the .o file containing the main function and it begins to extract unresolved

	mSQL 1.0.6			mSQL 2.0.8		
	Defined symbols	Objects to link	Libraries to link	Defined symbols	Objects to link	Libraries to link
mSQL	61	2 (2)	1 (1)	65	4 (4)	1 (1)
mSQLd	110	9 (9)	2 (2)	140	26 (23)	1 (1)
mSQLadmin	88	9 (4)	2 (2)	62	3 (3)	1 (1)
relshow	87	9 (3)	2 (2)	57	3 (3)	1 (1)
insert_test	45	2 (2)	1 (1)	52	3 (3)	1 (1)
select_test	45	2 (2)	1 (1)	52	3 (3)	1 (1)
mSQLdump	89	9 (4)	2 (2)	60	3 (3)	1 (1)
mSQLimport	-	-	-	62	3 (3)	1 (1)
mSQLexport	-	-	-	55	3 (3)	1 (1)
lite	-	-	-	164	41 (19)	2 (2)
w3-msql	-	-	-	147	41 (21)	2 (2)
w3-auth	-	-	-	59	5 (6)	1 (1)

Table 5. mSQL module dependencies

symbols; it continues recursively, analyzing all the other .o files in the same directory, resolving some symbols and, at the same time, finding new unresolved symbols. Finally, the script searches for libraries into directories specified in a configuration file to resolve remaining (unresolved) symbols. The output produced is represented by the list of .o files and libraries that have to be linked to the .o file containing the function main to obtain an executable application.

A CA tool was developed using the Java programming language. The tool was used to identify both candidate libraries and directories organization. The tool takes as input an objects/attributes matrix and produces a textual lattice representation. The lattice representation can be converted in `dot` format using a Perl script and then lattice can be displayed and navigated using a `dot` file viewer, like *At&T Dotty*.

6. Discussion

To validate the proposed method, a *legacy system* situation was mimicked by disregarding the directories organization (all files were placed in a common directory) of case study software systems. 15 versions of Samba (from 1.9.00 to 1.9.08 and from 2.0.0 to 2.0.5) and 11 versions of mSQL (from 1.06 to 1.08 and from 2.0.1 to 2.0.8) were analyzed. The results obtained did not significatively vary within each family (two versions belong to the same family if the first two version numbers are the same, i.e. 1.9.00 and 1.9.08 are considered to be members of the same family), for this reason a single version, extracted by each family, is discussed in this section: it can be thought as representative of the entire versions family. Table 4 reports the main features of the software systems.

Tables 5 and 7 lists, for each representative system, the results obtained by applying step 1 of the reorganization process. In particular, for each executable the following identified information are shown:

1. the number of defined symbols; and
2. the number of modules required for a successful compilation and linking, in parenthesis is reported such a number as extracted by the original system `Makefile`; and
3. the number of external libraries required for a successful compilation and linking. In parenthesis it is reported the number as extracted by the original `Makefile`.

It is worth noting that all the modules and the libraries necessary for a successful compilation and linking were always retrieved.

Tables 5 and 7 highlight that both the analyzed systems had, during the time, dramatic changes. The changes implemented on both the systems added new functionalities, improved existing features and reorganized systems architecture. However, such a reorganization for mSQL required a decreasing of the modules that each executable has to link, while for Samba the reorganization implied an increasing of the linked modules.

As can be seen in Table 7, the number of objects used to build Samba 2.0.5 applications is usually smaller than the retrieved from the `Makefile`. This happens because whenever a symbol defined in a library is needed, developers linked the entire library (composed by several object files), even if an unnecessary larger executable was obtained. On the contrary, the proposed method takes into account only objects effectively needed to resolve symbols.

The second step is related to identifying internal libraries through the clustering of object files. The application of this

step on mSQL 1.0.6 identified a candidate library: the objects `libmSQL.o` and `net.o` are used in all the executables. It is worth noting that such objects have been actually organized in a library by the original developers. Another candidate library was found within mSQL 1.0.6: in fact `mSQL_lex.o, mSQL_yacc.o, mSQL_proc.o, acl.o, regexp.o, mSQLdb.o` are used by `mSQL-dump, relshow, mSQLadmin and mSQLd`. However, developers did not build a library with that files even if it could be convenient: four over seven executables used those objects.

In mSQL 2.0.8 three candidate libraries were identified; two of these, `Lib1` and `Lib2`, are quite similar respectively to `libmsql` and `liblite` created by developers.

mSQL 1.X and 2.X developers clustered `regexp.o` and `regsub.o` into `libregexp.a`; however, this library is needed only by one executable in mSQL 1.X and by three executables in mSQL 2.X. Thus, `libregexp.a` was not highlighted as a "candidate" library by the search heuristic. A complete comparison between extracted libraries and real libraries is shown in Table 6.

RETRIEVED LIBRARIES			REAL LIBRARIES		
Lib1	**Lib2**	**Lib3**	**libmsql**	**liblite**	**libregexp**
net	parser	tmpnam	net	parser	regexp
config	symtab	regexp	config	symtab	regsub
libmsql	code-gen	sort	libmsql	code-gen	
	code-sim	regex	tmpnam	code-sim	
	extern	debug	time	extern	
	lib	strtoul		lib	
	modules	distinct		types	
	mod_w3msql	msqldb		fcrypt	
	regsub	avl_tree			
	auth	sysvar			
	mod_std	time			
	mod_msql	index			
	mod_lite	msql_yacc			
		table			
		funct			
		lexer			
		cra			
		acl			
		msql_proc			
		select			
		msql_lex			
		types			

Table 6. mSQL 2.0.8 retrieved libraries vs. real libraries (.o files)

Samba 1.9.08 developers did not create any library; however, two candidates libraries could be identified. In particular,

- `util.o` is used by all applications and it may be obviously considered a library;
- `pcap.o, params.o` and `loadparams.o` are used by `testprns` and `smbtatus`, so they could be clustered in a library.

Samba 2.0.5 was the most complex system among the analyzed ones. Figure 3 shows the concept lattice produced by applying the second step to such a software system: two candidate libraries could be identified, the first one of which is used by all executables. A comparison between retrieved libraries and libraries built by developers is shown in Table 8. It can be stated that:

- For *Lib2*, there is both a good precision and a good recall [5];
- For *Lib1*, while the precision is 100%, the recall is above 65%.

The *Lib1* lower recall value is due to the fact that only the chosen files are linked by all the executables (16 files), disregarded files are not. The manual *Lib1* analysis of dependencies revealed that almost each disregarded file was specific to a particular *main*, in other words, no discarded file was linked by more than the 30% of all executables. Files such as `getsmbpass.o, interface.o` and `netmask.o` appearing several times were moved by the approach from *Lib1* to *Lib2*.

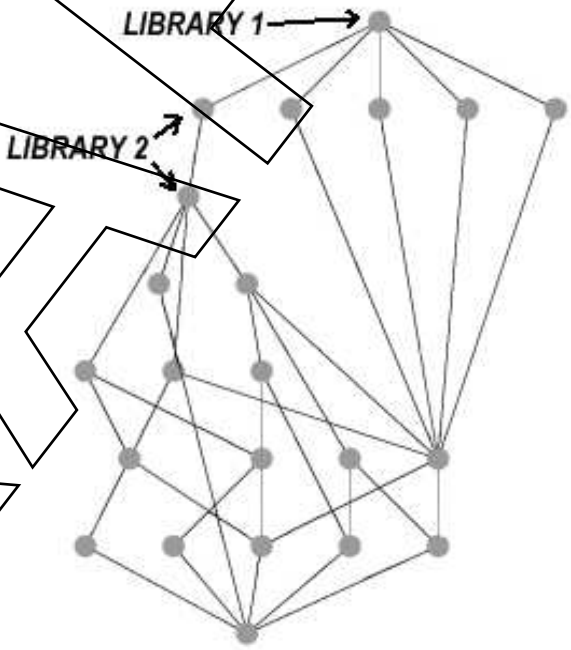

Figure 3. Samba 2.0.5 objects lattice

The original directories organization [1] implemented by both Samba and mSQL developers follows the *de-facto Linux standard directories organization*: separate directories are created for include files, applications source code,

[1] Samba 1.9.08 had a completely flat structure: all the files were contained in one only directory

	Samba 1.9.08			Samba 2.0.5		
	Symbols defi ned	Objects to link	External to link	Symbols defi ned	Objects to link	External to link
smbclient	150	1 (2)	1 (1)	264	30 (50)	1 (1)
smbd	384	17 (17)	1 (1)	287	101 (120)	2 (2)
nmbd	130	2 (3)	1 (1)	260	61 (75)	1 (1)
nmblookup	-	-	-	183	22 (49)	1 (1)
rpcclient	-	-	-	231	61 (81)	1 (1)
smbpasswd	-	-	-	198	44 (74)	1 (1)
smbstatus	116	4 (5)	1 (1)	189	21 (44)	1 (1)
smbrun	11	0 (0)	1 (1)	-	-	-
swat	-	-	-	238	50 (86)	2 (2)
testparam	119	5 (6)	1 (1)	194	19 (39)	1 (1)
testprns	103	4 (4)	1 (1)	166	20 (42)	1 (1)
make_printerdef	-	-	-	158	18 (39)	1 (1)
make_smbcodepage	-	-	-	157	18 (39)	1 (1)

Table 7. Samba module dependencies

RETRIEVED LIBRARIES		REAL LIBRARIES	
Lib1	**Lib2**	**Lib1**	**Lib2**
charcnv	namequery	charcnv	namequery
system	nmblib	system	nmblib
kanji	nterr	kanji	nterr
util_fi le	smberr	util_fi le	smberr
signal	smbencrypt	signal	smbencrypt
slprintf	smbdes	slprintf	smbdes
util_sock	clientgen	util_sock	clientgen
doscalls	pwd_cache	doscalls	pwd_cache
debug	interface	debug	credentials
util_str	netmask	util_str	passchange
username	getsmbpass	username	
util		util	
md4		md4	
time		time	
charset		charset	
genrand		genrand	
		fault	
		getsmbpass	
		interface	
		netmask	
		pidfi le	
		replace	
		ufc	
		access	
		bitmap	
		crc32	
		util_sid	
		util_unistr	
		util_sec	
		ssl	

Table 8. Samba 2.0.5 retrieved libraries vs. real libraries (.o files)

libraries and binary files. Such a organization is mainly thought for the end-user who is not in charge of the software application maintenance and evolution. On the other hand, the new directories organization aims to support maintenance and evolution activities.

The comparison between the new directories organization and the actual directories structure of the analyzed systems was encouraging. In particular, whenever the original developers grouped into distinct directories the source code files related to the system executables, the directories organization proposed matched the actual one. For example, the original Samba 2.0.5 developers placed the .c files related to 4 executables (`nmbd, swat, smbd` and `rpcclient`) into 4 distinct directories. In this case there is a perfect correspondence between the .c files contained in such actual directories and the .c files contained in the 4 directories created by the method and associated to those 4 executables. On the other hand mSQL developers, for both the analyzed system versions, concentrated almost 50% the source code related to the system executables in a single directory (`src/msql`). In particular, for mSQL 1.0.6. all the source code files related to the system executables were contained in `src/msql`. In mSQL 2.0.8 such a directory contains the source code files related to 9, on a total of 12, executables. In this case the retrieved directories organization differs from the actual one.

However, redefining the directories organization aims mainly to highlight cross-directory/cross-subsystem dependencies and, even when it differs from the original one, it is not less effective. In particular, highlighting dependencies may be helpful for software systems frequently evolved. Two principal benefits may derive from this organization:

1. when a system executable has to be modified the suggested directories organization allows the quick identification of the source code files of the executable;
2. grouping together source code files shared by different executables may help to avoid undesired side effects due to source code modifications. In other words, whenever a source code file is modified, the suggested directories organization highlights all the executables potentially impacted by such a modified file.

7. Related Works

An overview of CA applied to software reengineering problems was shown by G. Snelting in his seminal works

[11], where he used CA in several remodularization problems such as exploring configuration spaces(see also [6]), transforming class hierarchies, and remodularizing COBOL systems. In [13] Kuipers and Moonen combined CA and type inference in a semi-automatic approach to find objects in Cobol legacy code: once candidate classes are extracted from a legacy system, they can be refined by manually manipulating the concept lattice. Another semi-automatic application to find object into COBOL system was presented in [12]. CA and a vector space approach were applied to the same COBOL system and results compared.

We share with [6, 11, 13, 12] the general idea to apply CA to program understanding and reengineering. As in [6, 13, 12] we believe that with the present level of technology a programmer-centric approach is required: the user is left in charge to choose the proper remodularization based on his/her knowledge.

The need of a human intervention may be reconducted to the size of the concept lattice, the number of nodes, links (in other words the number of discovered relations). The problem of redundant information returned by a concept lattice was illustrated by Anquetil [1] where he proposed methods to prune the lattice. Another approach for extracting significant information from a concept lattice was proposed by Siff and Reps [8], where they exhibit an algorithm to recover partitions from a concept lattice. Our paper applies CA to identify libraries and reorganize the directory structure of software systems developed with C programming language. We experienced the same information *overflow*; our strategy is based on an heuristic suggesting to navigate the lattice starting from the top node: libraries are likely to be group of files *close* to the top concept.

Commonalities can be found with [9] and [4]. In [9] a methods for decomposing complex software systems into independent subsystems was proposed by Anquetil and Lethbridge. Source files were clustered based on file names and their decomposition. Merlo et al., [4] exploited comments, as well as variable and function names to clustered files. We share with the authors of [4, 9] the final goal: we believe that a system remodularization can help during program understanding activities or when performing maintenance interventions.

Recently CA was applied to remodularize C system based on memory usage [14, 15] and to identify design pattern on C++ systems [10]. Commonalities are limited to the programming language features in that C and C++ allows to use pointers and function pointers. Our case studies take into consideration system developed with C, however, while in [14, 10, 15] points to relations need to be handled our approach does not require points to relation resolution as preliminary step.

8. Conclusions

A method to reorganize source files into libraries and directories has been presented. The method recovers an architectural source code files organization analyzing compiling, linking and inclusion dependencies and using a cluster technique such as CA. The method has been applied to software system written with the C programming language. The results obtained are encouraging, it was always possible to reconstruct the `Makefile` to generate the executables of the analyzed systems, moreover the candidate libraries identified matches the actual system library; finally, the new directory organization highlights dependencies, thus it may help to maintain and evolve systems frequently modified.

Further work will be devoted to investigate the real usefulness, if any, of the proposed organization in everyday maintenance activities.

References

[1] N. Anquetil. A comparison of graphs of concept for reverse engineering. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pages 231–240, June 2000.

[2] R. V. B. Ganter. *Formal Concept Analysis*. Mathematical Foundations, Springer Verlag, 1999.

[3] G. Birkoff. *Lattice Theory*. American Mathematical Society, Providence, R.I., 1940.

[4] R. D. M. Ettore Merlo, Ian McAdam. Source code informal information analysis using connectionist model. In *IJCAI,*, pages 1339–44, Load Altos, Calif, 1993.

[5] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, PHADD,, 1992.

[6] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proc. of the 16th International Conference on Software Engineering*, pages 49–57, Sorrento, Italy, May 1994.

[7] M. M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

[8] T. R. Michael Siff. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25:749–768, Nov-Dec 1999.

[9] T. L. Nicolas Anquetil. Extracting concepts from file names; a new file clustering criterion. In *Proceedings of the International Conference on Software Engineering*, pages 84–93, April 1998.

[10] G. A. Paolo Tonella. Object-oriented design pattern inference. In *ICSM,*, pages 230–238, Sep 1999.

[11] G. Snelting. Software reengineering based on concept lattices. In *ICSM,*, pages 3–10, March 2000.

[12] A. v. D. Tobias Kuipers. Identifying objects using cluster and concept analysis. In *ICSE*, pages 246–255, June 1999.

[13] L. M. Tobias Kuipers. Types and concept analysis for legacy systems. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pages 221–230, June 2000.

[14] P. Tonella. Using the o-a diagram to encapsulate dynamic memory access. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 326–335, Bethesda, Maryland, November 1998.

[15] P. Tonella. Concept analysis for module restructuring. *IEEE Transactions on Software Engineering*, (To appear).

[16] P. Tonella, G. Antoniol, R. Fiutem, and F. Calzolari. Reenginering 4.7 million lines of code. *SPE*, 30(2):129–150, January 2000.

[17] R. Ville. Restructuring lattice theory: an approach based on hierarchies of concepts. *Ordered Sets, Reidel*, pages 445–470, 1982.