Applying Concept Formation Methods to Object Identification In Procedural Code

Houari A. Sahraoui
CRIM
1801, av McGill College, #800 Montreal (QC), Canada H3A 2N4
hsahraou@crim.ca

Walcélio Melo
Oracle do Brasil and Universidade Catolica de Brasilia, Dept. de Ciencia da Computacao
wmelo@br.oracle.com

Hakim Lounis
CRIM
1801, av McGill College, #800 Montreal (QC), Canada H3A 2N4
hlounis@crim.ca

François Dumont
CRIM
1801, av McGill College, #800 Montreal (QC), Canada H3A 2N4
fdumont@crim.ca

Abstract

Legacy software systems present a high level of entropy combined with imprecise documentation. This makes their maintenance more difficult, more time consuming, and costlier. In order to address these issues, many organizations have been migrating their legacy systems to new technologies. In this paper, we describe a computer-supported approach aimed at supporting the migration of procedural software systems to the object-oriented (OO) technology, which supposedly, fosters reusability, expandability, flexibility, encapsulation, information hiding, modularity, and maintainability. Our approach relies heavily on the automatic formation of concepts based on information extracted directly from code to identify objects. The approach tends, thus, to minimize the need for domain application experts. We also propose rules for the identification of OO methods from routines. A well-known and self-contained example is used to illustrate the approach. We have applied the approach on medium/large procedural software systems, and the results show that the approach is able to find objects and to identify their methods from procedures and functions.

1. Introduction

Many sources agree that programmers' efforts are mostly devoted to maintaining systems [4, 24]. Pressman estimates that typical software development organizations spend anywhere from 40 to 70 percent of all dollars performing maintenance [21]. This is not surprising when one considers the quantity of code to maintain. For instance, the average Fortune-100 company maintains 35 millions lines of code and adds an additional 10 percent each year, just in enhancements, updates and normal maintenance. As a result of maintenance alone, software inventories will double in size every seven years.

This problem stems in part from the fact that most of the software maintenance effort is spent changing legacy software which suffers from a lack of up to date and reliable documentation. In order to adequately maintain such systems, software engineers need understandable, consistent, and complete documentation about such systems (e.g., requirements specification, design documents, change requests, bugs reports, etc.). However, most of the documentation that software engineers have is the source code of the system they are supposed to maintain. After such code has been put through a number of changes over the years, it can present a high level of entropy; that is, the source code may become ill-structured, highly redundant, poorly self-documented, and weakly modular. Documents describing the architecture and design of such systems may present an inaccurate representation of “what is” actually implemented. Higher level of entropy combined with imprecise documentation about the design and architecture of legacy software systems make their maintenance more difficult, time consuming, and costly.

In order to address these issues, many organizations have been migrating their legacy systems to emerging technologies, e.g., object-oriented technology. Lehman and Belady present this migration as an economical choice through their three laws on the evolution of large systems [13].

The object oriented paradigm is the target architecture of choice for the reorganization of systems, since object-oriented (OO) representations are supposed to be much easier to understand than their classical “structured” counterparts. Further, encapsulation limits the complexity of maintenance. Presumably modifications in the implementation of an object (class) does not affect other objects since only the object's interface is visible.

OO approaches and languages have become quite popular, partially because of their potential benefits in terms of maintenance (reusability, separation of concerns and information hiding). However, the vast majority of the software available today is not OO. The effort necessary to simply rewrite them from scratch using an OO approach would be prohibitive, and significant expertise recorded in the procedural software would be lost. The cost of manual conversion would also be prohibitive. A tool (or a tool set) that would support the conversion of procedural code to OO, even in a semi-automatic fashion, would ease the introduction of OO technology in many organizations. This kind of reengineering tool could be especially helpful to integrate existing systems with new ones developed with OO approaches.

Several tools have been built in the last ten years that support the migration of legacy software systems to OO technology. The main difference between these tools is the level of involvement of domain experts in the migration process. Some tools are called *domain dependent* in the sense that, in addition to the source code, they need domain knowledge as input (see for example [6]).

The other category of tools is called *domain independent*. The only input required for them is the source code, although, they need some domain knowledge to make some decisions (see for example [3]). Domain dependent approaches need domain expertise that is not always available for the legacy systems, and even when it is, its cost may be very high. But because such tools are guided by domain models, the results are more reliable. Domain independent tools do not need domain expertise; they use heuristics to make the necessary decisions when identifying objects, and the results are not always reliable.

In this paper, we describe a computer-based approach aimed at supporting the migration of procedural software systems to the object-oriented technology. The approach relies heavily on the automatic formation of concepts [9]. To do so, the approach uses exclusively information extracted directly from code. The approach tends, thus, to minimize the need for application domain experts. However, when available, application domain experts can make decisions about the objects discovered by our approach. The approach consists of identifying objects in procedural code, a first step towards an object-oriented design. We also propose rules for the identification of OO methods from procedures/functions. In order to present our approach, we use a self-contained example already used to illustrate other approaches.

Furthermore, we have applied our approach on a sample of large-scale commercial software systems written in C. The results show that our approach can automatically find potential objects in non-OO code.

Section 2 presents some existing works on graph-based object identification. Section 3 describes our approach trough an example. Section 4 discusses the limitations of the approach and lessons learned. Section 5 concludes and enumerates future work.

2. Related work

Procedural code does not contain an explicit representation of objects. It contains only global variables, data structure (records) and routines (functions and procedures). However, often the designer isolates the access and modification of a data structure to a limited number of routines in order to foster design modularity. The identification of such a grouping of routines and records is the intuition behind many of the object identification techniques in the literature [1]. The other type of grouping involves routines and global variables.

Liu & Wilde [16] have proposed to group data structures with routines that use them as parameters or return value. Later some heuristics were proposed to enhance Liu & Wilde’s work [20, 15, 11]. In [25], Yeh & al. combine data structures with global variables in order to form groups of routines, data structures and global variables. Each group would consist of an object where the routines will be methods and data structures and global variables signatures of such methods. In [7], Gall & Klösch present another approach which focuses on the file access to identify the data structures that should provide candidate objects.

Other algorithms use reference graphs as introduced in [5]. In [3], Canfora & al. propose an algorithm that decomposes a reference graph into a set of strongly connected sub-graphs. Each sub-graph represents an object. This decomposition is based on the notion of connectivity.

Finally, concept formation methods have been applied in software engineering for remodularization (see [23 and 14]). In these two projects, Galois (concept) lattices are used to identify modules in legacy code.

3. Concept formation based approach

3.1 principle of Galois lattice

Our approach relies heavily on the automatic concept formation [9]. It is based exclusively on information extracted directly from code. To do so, we have used Galois (or concept) lattices. In this section we present the basic definitions for Galois lattices, proposed by Godin in [9]. Algorithms based on this method are described in [10].

Let us take two finite sets E and E' and a binary relationship R between the two sets. The Galois lattice (see example of fig. 1) is the set of elements *(X, X')*, where $X \in P(E)$ and $X' \in P(E')$. *P(S)* is the powerset of *S*. Each element *(X, X')* must be complete.

A couple *(X, X')* from $P(E) \times P(E')$ is complete if it satisfies the two properties :

1. $X' = f(X)$ where $f(X) = \{x' \in E' \mid \forall\, x \in X, xRx'\}$
2. $X = f'(X')$ where $f'(X') = \{x \in E \mid \forall\, x' \in X', xRx'\}$

					E'					
	R	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
	1	*1*	*0*	*1*	*0*	*0*	*1*	*0*	*1*	*0*
E	*2*	*1*	*0*	*1*	*0*	*0*	*0*	*1*	*0*	*1*
	3	*1*	*0*	*0*	*1*	*0*	*0*	*1*	*0*	*1*
	4	*0*	*1*	*1*	*0*	*0*	*1*	*0*	*1*	*0*
	5	*0*	*1*	*0*	*0*	*1*	*0*	*1*	*0*	*0*

Figure 1.a. Representation of binary relation R

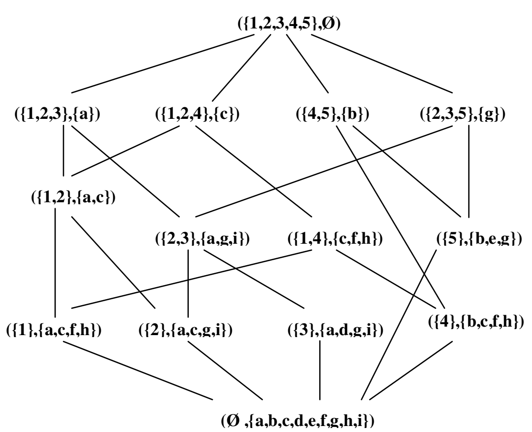

Figure 1.b. Galois lattice for relation R

Given two elements $N_1 = (X_1, X'_1)$ and $N_2 = (X_2, X'_2)$ of a Galois lattice *G*, $N_1 < N_2$ implies that $X_2 \subset X_1$ and $X'_1 \subset X'_2$.

This property defines a partial order between elements of *G*. A graph is constructed using this partial order (see figure 1.b). There is an edge between N_1 and N_2 if

1. $N_1 < N_2$
2. there does not exist $N_3 \mid N_1 < N_3 < N_2$

N_1 is said more general than N_2. Edges are directed from up to down.

3.2 Applicability to object identification

In an OO design, an application is modeled by a set of objects where objects are composed of a set of data and a set operations that manipulate this data. Most of graph based approaches to object identification group data with the routines that use them

Using this grouping approach, Galois lattices can provide all significant groups. Let *E* (c.f. 3.1) be the set of global variables, and *E'* the set of routines, and let *R* be the relation defined as $\forall\, v \in E$ *and* $\forall\, f \in E'$, *vRf* means that the function *f* uses (refers to) the variable *v*, then the resulting Galois lattice has the following properties :

1. Each node *(X, X')* denotes a group of data (*X*) relatively to a set of functions (*X'*) which can be taken as a candidate object (the criteria are defined in 3.4).
2. There does not exist $(Y, Y') \neq (X, X') \mid Y \subseteq X$ and $Y' \subseteq X'$. Only significant groups are in the lattice.
3. An edge between two nodes $N_1 = (X_1, X'_1)$ and $N_2 = (X_2, X'_2)$ can be interpreted either as
   - a generalization/specialization link. From a behavioral point of view, the set of functions in N_1 is a subset of the set of functions in N_2 ($X'_1 \subset X'_2$).
   - or an aggregation link. From a data point of view, the set of data in N_2 is a subset of the set of data in N_1 ($X_2 \subset X_1$).

3.3 An example

To illustrate our approach, let us take the well-known example introduced in [3] (call it *collections*). This example presents a part of a C program (see the following code). The program manipulates a stack, a queue and a list. For each function, the body is replaced by a comment that indicates the list of data used by the function. This example has the advantage of being self-contained, well-known in the literature, small, and yet relatively complex. Later, we will provide an actual example which shows that our approach is able to deal with large-scale software systems.

```
#define MAXDIM 99
typedef int ELEM_T;
typedef int BOOL;
ELEM_T stack_struct[MAXDIM];
int stack_point;
ELEM_T queue_struct[MAXDIM];
```

```
int queue_head, queue_tail, queue_num_elem;
struct list_struct
   {  ELEM_T node_content;
      struct list_struct * next_node; } list;
main()
{/* this program exploits a stack, a queue, and
a list of items of type */}
/* list of fuctions with as comment the list of
global variables referenced */
void stack_push(el)  {/* stack_point and
stack_struct */}
ELEM_T stack_pop()   {/* stack_point and
stack_struct */}
ELEM_T stack_top()   {/* stack_point and
stack_struct */}
BOOL stack_Empty()   {/* stack_point */}
BOOL stack_full()    {/* stack_point */}
void queue_insert(el)   {/* queue_struct,
queue_head and queue_num_elem */}
ELEM_T queue_extract()  {/* queue_struct,
queue_tail and queue_num_elem */}
BOOL queue_Empty()   {/* queue_num_elem */}
BOOL queue_full()    {/* queue_num_elem */}
void list_add(el)    {/* list */}
void list_elim(el)   {/* list */}
BOOL list_is_in()    {/* list */}
BOOL list_empty()    {/* list */}
void global_init()   {/* stack_point, list,
queue_head, queue_tail and queue_num_elem */}
void stack_to_list() {/* stack_point,
stack_struct and list */}
void stack_to_queue()   {/* stack_point,
stack_struct, queue_struct, queue_head
and queue_num_elem */}
void queue_to_stack()   {/* queue_struct,
queue_tail, queue_num_elem, stack_point
and stack_struct */}
void queue_to_list() {/* queue_struct,
queue_tail, queue_num_elem, and list */}
void list_to_stack() {/* list, stack_point and
stack_struct */}
void list_to_queue() {/* list, queue_struct,
queue_head and queue_num_elem */}
```

From this program, a reference graph is extracted (figure 2) [3]. Functions are represented by ellipses and global variables by rectangles. Edges are always directed from functions to global variables.

3.4 The approach

The object identification approach we propose in this paper consists of four steps. First, we extract the reference graph from the source code (we propose an improved version of this graph). Then, we identify candidate objects from the corresponding Galois lattice. In the third step, we identify objects. Finally, we identify the methods of these objects.

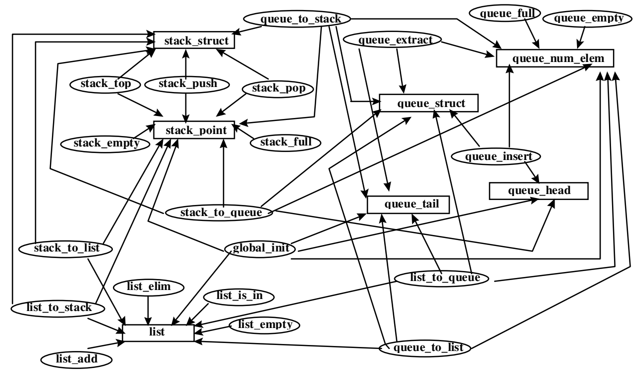

Figure 2. Reference graph for *collections* program

3.4.1 Graph extraction

In [3], the relationship between a function and a global variable simply indicates that the function uses the variable. In our case, the way in which the function uses the variable is important. We define three modes : modification or write mode (m) when the function modifies the value of the variable, access or read mode (a) when it access its value to compute something else, and predicate mode (p) when the variable is used to control the execution of the function (in a predicate). This classification is based on some work on module coupling [19, 17]. This improvement can help us for two reasons :

1. A global variable in the reference graph that has no link in m mode can be considered as a constant, and removed from the graph (such decisions are not easy to make when pointer arithmetic is used).
2. When we identify methods, the mode can be considered in conflict situations (see 3.4.4).

The extraction process is performed in two steps. First, an abstract syntax tree (AST) is built from the program. Then, this AST is used by a pattern recognition and transformation program to extract the necessary information (in our case the reference graph). The result of this process (a file) is a set of facts *refers_to(f, v, t, m)* where *f* is a function, *v* is a global variable, *t* is the type of *v*, and *m* is the usage mode. For example, in *collections*, *refers_to ("stack_push", "stack_point", "int", "m")* means that the function *stack_push* uses the variable *stack_point* which is an integer in modification mode.

R'	b. stack_struct	c. stack_point	d. list	e. queue_tail	f. queue_head	g. queue_struct	h. queue_num_elem
2. stack_push	1	1					
3. stack_top	1	1					
4. stack_pop	1	1					
5. stack_empty		1					
6. stack_full		1					
7. stack_to_queue	1	1			1	1	1
8. global_init		1	1	1	1		1
9. list_is_in			1				
10. list_empty			1				
11. stack_to_list	1	1	1				
12. list_to_stack	1	1	1				
13. list_add			1				
14. list_elim			1				
15. queue_to_stack	1	1		1		1	1
16. queue_extract				1		1	1
17. queue_full							1
18. queue_empty							1
19. queue_insert					1	1	1
20. list_to_queue			1		1	1	1
21. queue_to_list			1	1		1	1

Figure 3. Matrix representation of reference graph for *collections* program

3.4.2 Candidate object identification

As presented above, E is the set of global variables, E' the set of functions and R the relation which indicates that $v \in E$ is used by $f \in E'$. Figure 3 shows the matrix representation of R' instead of R (for the *collections* program) for readability reasons. For the same reasons, names of functions and global variables are replaced by codes (number for a function and letter for variables) when building the Galois lattice. The Galois lattice constructed from R presents all the significant groups of data (see figure 4 for the *collections* program). The goal of this step is to identify candidate objects. To this end, we define some criteria to select a subset of groups.

In order to identify candidate objects from the Galois lattice, we first define the set NS that contains the not-yet-selected variables. In the initial state $NS = E$. The identification process stops when $NS = \varnothing$. In the identification process, groups are checked starting from the bottom up. This order is motivated by the fact that the deeper a group is in the lattice the higher is the cardinality of its function set (X'). In other words, our hypothesis is that a group of variables can be considered as a candidate object if these variables are simultaneously accessed by a large number of functions. In case of a tie (same cardinality of functions sets), groups are ordered by the cardinality of their variables sets (X) in a descendant mode. This is done to avoid large objects. These two criteria define a static order. If two groups have the same rank in this order, a priority is given to the one that has the higher cardinality of the set $ns = X \cap NS$. This defines a dynamic order. Each time a group is selected, the variables it contains are removed from NS. A group with ns = $\varnothing$ is ignored. The last criterion for selection is if a group has only one variable, the type of this variable must be non basic type (e.g. int, char, etc).

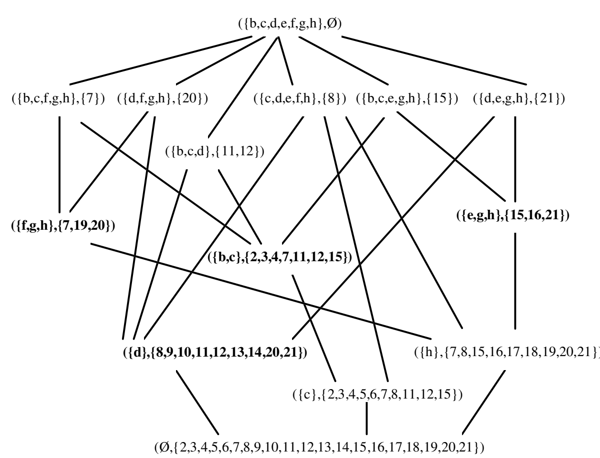

Figure 4. Galois lattice for reference relation (*collections* program)

The application of these criteria to the example of figure 4 gives the following four candidate objects :

co1 = {b,c} = {stack_struct, stack_point}

co2 = {d} = {list}

co3 = {f,g,h} = {queue_head, queue_struct, queue_num_elem}

co4 = {e,g,h} = {queue_tail, queue_struct, queue_num_elem}

3.4.3 Object identification

If we consider candidate objects co3 and co4, we notice that they share two variables out of three. Such situations motivate the introduction of a new step that automatically merges these two objects. To detect these situations, we apply the same technique (Galois lattice) with a new relation. In this step, E is the set of candidate objects found in step 2. E' is the set of global variables. We define the relation R as

$\forall g \in E$ *and* $\forall v \in E'$, gRv means that g contains v.

Figure 5 shows that co3 and co4 can be grouped in the same object. This decision is made relative to the cardinality of the set of variables in ({co3, co4},{g, h}) which is fixed to 2 by default in our prototype. However, in our prototype an expert can be involved to make decisions based on her knowledge about the application domain, like merging candidate objects, or breaking a candidate object in two or more objects.

In the *collections* program example, we obtain the following objects:

o1 = co1 = {b,c} = {stack_struct, stack_point}

o2 = co2 = {d} = {list}

o3 = co3 ∪ co4 = {e,f,g,h} = {queue_tail, queue_head, queue_struct, queue_num_elem}

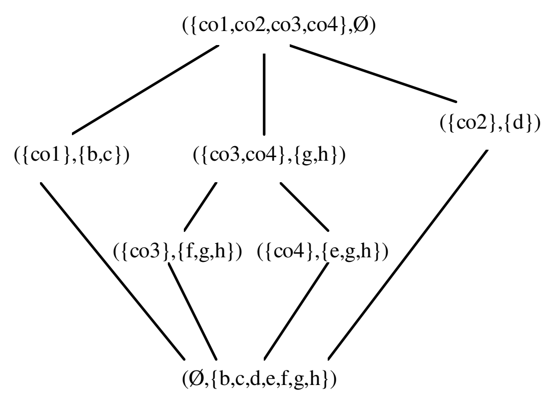

Figure 5. Galois lattice for grouping relation (*collections* program)

3.4.4 Method identification

So far, we have identified the structure of the objects (variables). To be complete, an object must have a behavior (i.e. methods). In our approach, we identify methods from functions. In the remainder of this section, we present an overview of the rules we use to form methods from procedures/functions. A detailed description of method identification process is beyond the scope of this paper. Some ideas we exploit can be found in [18].

Let O be the set of identified objects, F the set of functions in the legacy code, and V the set of global variables. For each function f, we define two sets *ref(f)* and *modif(f)* as follows:

$\forall f \in F$,

$ref(f) = \{o_i \in O \mid \exists v_j \in V \text{ and } v_j \text{ in } o_i \text{ and } v_i R f\}$ where R denotes the relation *is used by*.

$modif(f) = \{o_i \in O \mid \exists v_j \in V \text{ and } v_j \text{ in } o_i \text{ and } v_i M f\}$ where M denotes the relation *is modified by*.

The relation M is derived from R with the condition that the mode of usage is m (see 3.4.1).

There are three possible cases :

1. cardinality of ref(f) = 1
2. cardinality of ref(f) > 1 and cardinality of modif(f) = 1
3. cardinality of modif(f) > 1

For each case we define a rule.

Rule 1*: For a function f, if cardinality of ref(f) = 1, then f becomes a method of the unique object in ref(f).*

The first case is trivial. For example in *collections*, $ref(stack\_full) = \{o_1\}$. *stack_full* becomes a method of o_1.

Rule 2*: For a function f, if cardinality of ref(f) > 1 and cardinality of modif(f) = 1, then f becomes a method of the unique object in modif(f).*

This rule is motivated by the fact that conceptually we consider a function as a behavior of an object if it modifies its state. For example, ref(stack_to_list) = {o1, o2} and modif(stack_to_list) = {o2}, stack_to_list becomes a method of o2. stack_to_list is a conversion function. In object oriented programming, there are two possibilities to convert an object o1 into another object o2 : (1) ask o1 to become o2 (e.g. in smalltalk, method asPolyline in Circle class which convert a circle into a polyline), and (2) create o2 from o1 (e.g. in smalltalk, method fromDays: in Date class which create a date from an integer). With our approach the second solution is automatically taken. When available, an expert can make such a decision.

Rule 3*: For a function f, if cardinality of ref(f) > 1 and cardinality of modif(f) > 1, then f must be sliced when possible to create a method for each object in modif(f).*

For example, *ref(global_init)* = $\{o_1, o_2, o_3\}$ and *modif(global_init)* = $\{o_1, o_2, o_3\}$. *global_init* can be sliced to create three methods *init_stack, init_list, init_queue*. Actually, it is not always possible to break a function into cohesive methods. Other solutions can be used depending on the target OO language. In C++ for example, it is possible to define a function independently from any class. In other languages, a method can be associated to more than one class. Finally, it is possible to define a new object that aggregates the objects involved in *modif(f)*, and put *f* as a method in that object.

3.5 Complexity

Let n be the cardinality of the set E (c.f. 3.1), and assume that there is a finite upper bound K on the number of relations for an element of E, formally $\mathbf{K = Max(\{Cardinality(f(x))} \mid \mathbf{x} \in E\})$, Godin & al. show that in this case, the worst case complexity of the Galois lattice (number of nodes *nl*) is linearly with respect to n : $nl \leq 2^K n$ [9].

In the same time, it is proved that the relations R and R' gives the same lattice. We can then replace n by n' (cardinality of the set E') and K by K' (the upper bound of the number of relations for an element of E').

In our case n' indicates the number of routines in the reference graph, and K' the maximum number of variables that can be referenced by a routine. The increase of the size of a program can increase the number of routines n', but the maximum number K' of variables referenced by a routine is in general stable. K' depends much more on other factors (programming style for example) than the size of the program.

4. Discussion and lessons learned

We developed a prototype (named COBOI) to implement our approach. This prototype was developed using a graphical-description based application generator (MÉTAGEN [22]) and a pattern recognition extractor generator. Figure 6 shows a graphical editor of our prototype which allows to display and manipulate Galois lattices.

Using this prototype, we have applied our approach to different C applications (sizes between 3,000 and 47,000 lines of code). To illustrate the advantages and limitations of our approach, we present two examples of the applications we have studied. The first (and the larger) one is a system for education record processing (Proverbe). We applied the approach without any expert intervention. The resulting Galois lattice was very large. Identified candidate objects were numerous and generally too large to be thought of as objects. Due to a lack of space and confidentiality considerations, figure 6 is not complete. It just gives an idea about the size of the Galois lattice. It also shows the capability of our prototype to be applied to large software applications.

During validation with the expert, we noticed that a number of global variables are related to the windows interface library (e.g. DLL handles), and identified object are composed of both domain data and library data.

Actually, there are two problems with this case : (1) the code of the library is not available, and (2) in object oriented programming, the user interface of an application is generally dissociated from the model. We decided to remove from the reference graph all the variables related to the graphical library, since the goal is not to migrate the library, but the application which uses the library. The size of the resulting Galois lattice was reduced considerably. According to the expert evaluation, the identified objects were more meaningful (see figure 7).

The main lesson we learned from this case is that we need human intervention to decide which data are domain related. Our tool is not able to know automatically which components under analysis belong to either a domain independent library or to the application domain. We consider, however, that this kind of information can be easily obtained from the maintainers. Once we know which routines should not be analyzed (since they belong to the library), our prototype is able to work properly without further help from the maintainer. Of course, we should tell COBOI which routines should not be considered when building the Galois lattice.

The second case is a library that allows to recognize geometric objects. It is medium size (12,000 lines). Since the first step, we noticed that there were very few global variables. This limits the applicability of our approach. We decided to use the same approach with a different graph as an input, *data visibility graph*. Like reference graph, this one has two types of nodes (data and functions) and a single type of edge (function refers to data). The difference is that the data consist of both global variables and the local variables that are transmitted as parameter to other functions. A variable v is visible to the function f in which it is declared and all the functions that receive it as parameter, either directly from f, or via other functions recursively.

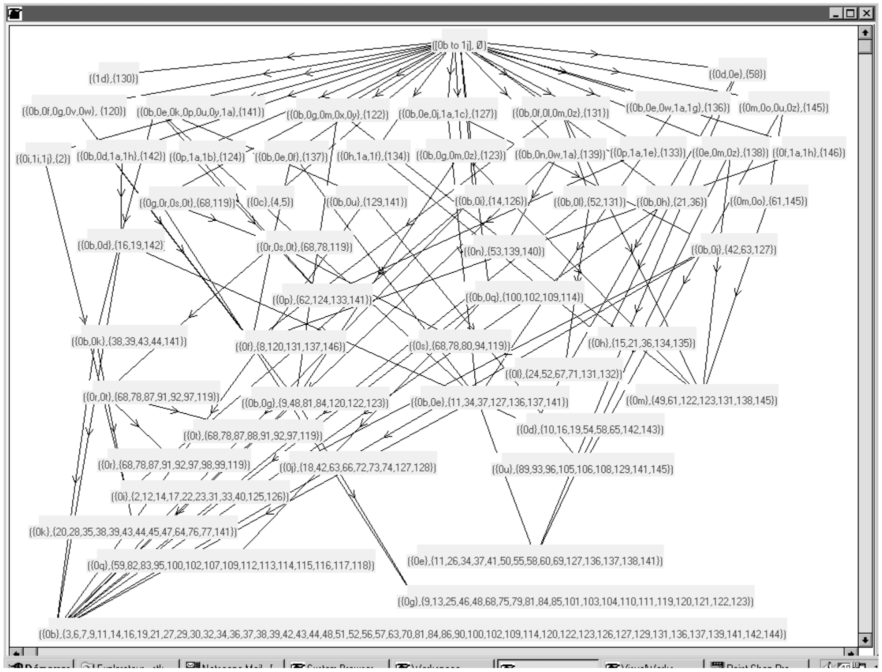

Figure 6. An overview of Galois lattice for a large example

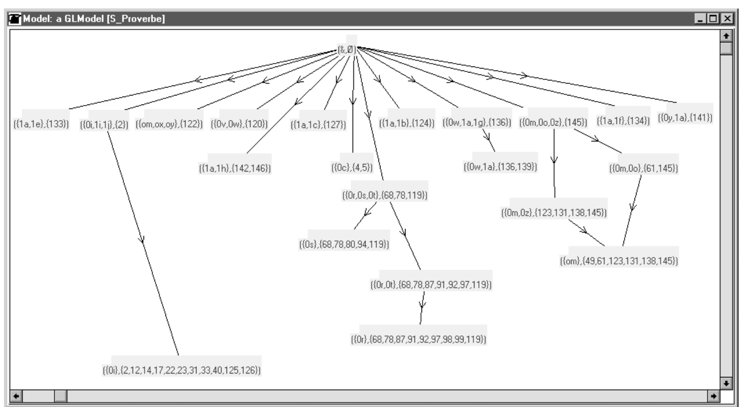

Figure 7. An overview of the same Galois lattice as figure 6 without library related variables

The main lesson learned from this case is that applications are different depending on the domain and programming style. Any approach of finding objects must be flexible. This leads us to introduce a preliminary step which defines the profile of the application based on metrics (e.g. average number of functions per global variable). Depending on this profile, different graphs can be used as an input (reference, data visibility or type visibility graph which includes types rather than data [25]).

5. Conclusion and future work

In this paper, we propose a technique for identifying objects in procedural code. It differs from other work by the fact that it borrows part of its inspiration from the artificial intelligence sub-field of concept formation. The main difficulty in graph based approaches is the identification of the sub-graphs that can be interpreted as groups of data and related functions. In our case, the identification of significant groups is automatic using Galois lattices. The part that can be improved is the set of criteria that determine witch groups can be selected as candidate objects. The prototype we built can work in an automatic fashion. It is also open to human intervention when an expert is available. The approach can take different types of bipartite graphs depending on the profile of the application at hand.

The cases we have studied show that there is room for improvement. In the near future, we will develop an incremental version of our approach to help the expert validate the results by reducing the complexity of the resulting models. To do that, we will use an incremental algorithm for building Galois lattices (see [10]). Another project we have started is the effective migration of code. We are currently implementing slicing algorithms (see [26, 8, 12, 2]) which allows us to generate two or more methods from a function according to the results of method identification step.

Acknowledgment

The authors would like to thanks professors R. Godin, H. Mili, and C. Hamzaoui for their comments on this work.

References

[1] R. S. Arnold, *Software Reengineering*, IEEE Computer Society Press, 1994.

[2] G. Canfora, A. Cimitile, A. De Lucia, & A. Di Lucca, Software Salvaging Based on Conditions, In *Proc. of ICSM'94*, IEEE Computer Society Press, pp. 424-433, September 1994.

[3] G. Canfora, A.Cimitile, and M.Munro, An Improved Algorithm for Identifying Objects in Code, *Software Practice and Experience*, 26(1):25-48, January 1996.

[4] T.A. Corbi. Program understanding: Challenge for the 1990s, *IBM System Journal*, 28(2):294–306, 1989.

[5] M. F. Dunn and J. C. Knight, Automating the Detection of Reusable Parts in Existing, *In Proc. of International Conference on Software Engineering*, pp 381-390, Baltimore, Maryland, 1993, IEEE Computer Society Press.

[6] H. C. Gall, R. R. Klösch and R. T. Mittermeir, Architectural Transformation of Legacy Systems, Workshop on Program Transformation for Software Evolution, ICSE, 1995.

[7] H. C. Gall and R. R. Klösch, Finding objects in procedural programs, In L.Wills, P.Newcomb, and E.Chikofsky, editors, *Second Working Conference on Reverse Engineering*, pp. 208–217, Los Alamitos, California, July 1995. IEEE Computer Society Press.

[8] K.B. Gallagher & J.R. Lyle, Using Program Slicing in Software Maintenance, *IEEE Transactions on Software Engineering*, 17(8): 751-761, August 1991.

[9] R. Godin, G. Mineau, R. Missaoui, M. St-Germain and N. Faraj, Applying Concept Formation Methods to Software Reuse, *International Journal of Knowledge Engineering and Software Engineering*, 5(1): 119-142, 1995.

[10] R. Godin, R. Missaoui and H. Alaoui, Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices, *Computational Intelligence*, 11(2): 246-267, 1995.

[11] D. Harris, H.Reubenstein, and A.S. Yeh. Recognizers for extracting architectural features from source code. In L.Wills, P.Newcomb, and E.Chikofsky, editors, *Second Working Conference on Reverse Engineering,* pp. 252–261, Los Alamitos, California, July 1995. IEEE Computer Society Press.

[12] F. Lanubile and G. Visaggio, Function Recovery Based on Program Slicing, In Proc. of ICSM'93, IEEE Computer Society Press, pp. 396-404, Montreal, September 1993.

[13] M. M. Lehman and L. A. Belady, *Program evolution*, Academic Press, New York, 1985.

[14] C. Lindig and G. Snelting, Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis, *In Proc. of International Conference on Software Engineering*, ACM Press, Boston, 1997, pp 349-359.

[15] P.E. Livadas and P.K. Roy, Program dependence analysis, *In Conference on Software Maintenance 1992*, pp 356–365, 1992.

[16] S.S. Liu and N.Wilde. Identifying objects in a conventional procedural language: An example of data design recovery. *In Conference in Software Maintenance*, pp. 266–71. IEEE Computer Society Press, November 1990.

[17] H. Lounis, W. Melo, Identifying and Measuring Coupling in Modular Systems, *8th International Conference on Software Technology ICST'97*, Curitiba, Brazil, June 1997. To appear

[18] H. Mili, On Behavioral Description in Object-Oriented Modeling, *The Journal of Systems and Software*, 34(2):105-121, August 1996.

[19] J. Offutt, M. J. Harrold and P. Kolte, A Software Metric System for Module Coupling, *The Journal of Systems and Software*, 20(3):295-308, March 1993.

[20] R.M. Ogando, S.S. Yau, and N.Wilde. An object finder for program structure understanding, *In Journal of Software Maintenance*, 6(5):261–83, September-October 1994.

[21] R.Pressman, *Software Engineering: a Practioner's approach*, McGraw-Hill, second edition, 1987.

[22] N. Revault, H.A. Sahraoui, G. Blain and J.F. Perrot, A Metamodeling technique: The METAGEN system, *Proceedings of TOOLS 16*, pp. 127-139, Versailles, march 1995.

[23] M. Siff, T. Reps, Identifying Modules via Concept Analysis, In *Proc. of ICSM'97,*

[24] I.Sommerville, *Software Engineering*, Addison Wesley, fourth edition edition, 1992.

[25], A. S. Yeh, D. R. Harris, and H. B. Reubenstein, Recovering Abstract Data Types and Object Instances from a Conventional Procedural language, In L.Wills, P.Newcomb, and E.Chikofsky, editors, Second *Working Conference on Reverse Engineering*, pp. 252–261, Los Alamitos, California, July 1995. IEEE Computer Society Press.

[26] M. Weiser, Program Slicing, *IEEE Transactions on Software Engineering*, 10(4): 352-357, July 1984.