A Concept Formation Based Approach to Object Identification in Procedural Code*

Houari A. Sahraoui[1], Hakim Lounis[1], Walcélio Melo[2] and Hafedh Mili[3]

[1]*Centre de Recherche Informatique de Montréal, {hsahraou, hlounis}@crim.ca*

[2]*Oracle do Brasil and Universidade Catolica de Brasilia, Dept. de Ciencia da Computacao, wmelo@br.oracle.com*

[3]*Université du Quebec À Montréal, Dept. Informatique, hafedh.mili@uqam.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 emerging 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. Our approach is based on the automatic formation of concepts, and uses information extracted directly from code to identify objects. The approach tends, thus, to minimize the need for domain application experts.

1. Introduction

Many sources agree that programmers' efforts are mostly devoted to maintaining systems [Corbi, 89 and Sommerville, 92]. Pressman estimates that typical software development organizations spend from 40 to 70 percent of their budget to maintenance [Pressman, 87]. This is not surprising when one considers the quantity of code to maintain. This problem stems in part from the fact that most of the software maintenance effort is spent modifying legacy software that suffers from a lack of up to date and reliable documentation. In order to adequately maintain such systems, software engineers need to have understandable, consistent, and complete documentation about such systems (e.g., requirements specification, design documents, change requests, bug 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

* An early and shorter version of this paper was published in [Sahraoui et al., 97].

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 [Lehman & Belady, 85].

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) do not affect other objects since only the objects 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 [Gall et al., 95]).

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 [Canfora et al., 96]). 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.

This paper presents some approaches and algorithms for identifying objects proposed in the literature. It identifies some their limitations and propose a five step approach (domain independent) which take into account the profile of the application to migrate. The object identification step is presented in details through one of the two algorithm we use. Finally, a section is devoted to the presentation of some results.

2. Related work

As stated by [Jacobson & Lindstrom, 91], the process of re-engineering can be defined by the simple formula:

$$Re\text{-}engineering = Reverse\ engineering + Changes + Forward\ engineering$$

where "Reverse engineering" is the activity of defining an abstract representation of the system, "Changes" the activity of changing the implementation technique and eventually the functionality, and where "Forward engineering" is the activity of creating a representation that is executable. In the particular case of program migration to the object paradigm, the formula above can be written as follows:

$$Re\text{-}engineering = Program\ abstraction + Object\ identification + Code\ generation$$

In the remainder of this section, we describe some existing work related to the first two steps, program abstraction and object identification ; the third step is not specific to this problem, and will not be discussed.

2.1 Program abstraction

The source code contains part of the knowledge about the application. To identify object-like features in it, we have to decide which information must be used. Different techniques use different program abstractions. In this paper we limit ourselves to three examples : (1) routine interdependence graphs, (2) reference graphs, and (3) type visibility graphs. The first two allow us to identify objects, the last one allows us to identify classes.

Proposed by Liu and Wilde, routine interdependence graphs show the dependence between routines consequent to their common coupling to the same global data [Liu and Wilde, 90]. A node $P(x)$ in the graph denotes the set of routines that reference a global variable x. An edge between $P(x_1)$ and $P(x_2)$ means that the two sets are not disjoint ($P(x_1) \cap P(x_2) \neq \emptyset$). Figure 1.a shows the reference relation between the routines f_is and the global data d_is of a program. The t_is represent the global data types. Figure 1.b gives the corresponding routine interdependence graph. Such a kind of graphs is used to identify objects. Each isolated sub-graph is considered to embody an object.

In reference graphs, nodes are either routines or global variables, and an edge between a routine and a variable means that the routine uses the variable [Dunn & Knight, 93]. Figure 2 shows the reference graph of the relation of figure 1.a. Like the routine interdependence graph, this kind of graph is used to identify objects through isolated sub-graphs.

	d1 (t1)	***d2 (t1)***	***d3 (t2)***	***d4 (t3)***	***d5 (t3)***	***d6 (t4)***
r1			1			
r2	1		1			
r3		1				
r4		1				
r5	1					1
r6					1	
r7	1					1
r8		1		1		
r9				1		
r10					1	
r11	1		1			

Figure 1.a Reference relation between routines and global data

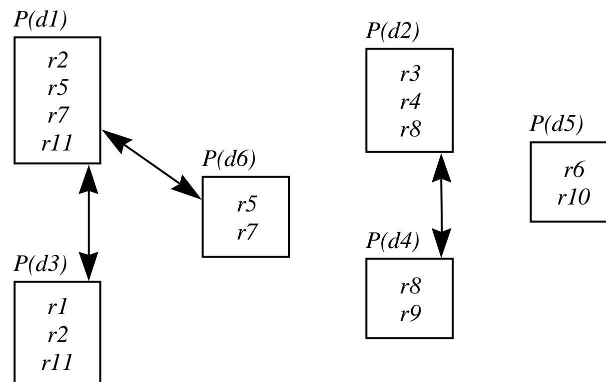

Figure 1.b Routine interdependence graph of the relation of figure1.a.

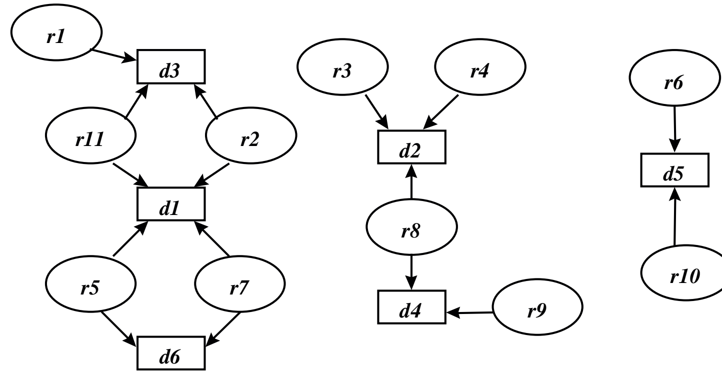

Figure 2. Reference graph of the relation in figure 1.a.

Introduced by Yeh, type visibility graphs represent the visibility relation between the routines and the data structures (or types) of a program [Yeh et al., 95]. A type *t* is said to be visible by a routine *r*, if *r* uses a global variable of type *t*, or if *r* has a formal parameter of type *t*, or if *r* has a local variable of type *t*. Figure 3 gives a partial type visibility graph based on the relation in figure 1.a. This kind of graphs helps to identify directly the classes of objects rather than objects. This identification is done using isolated sub_graphs.

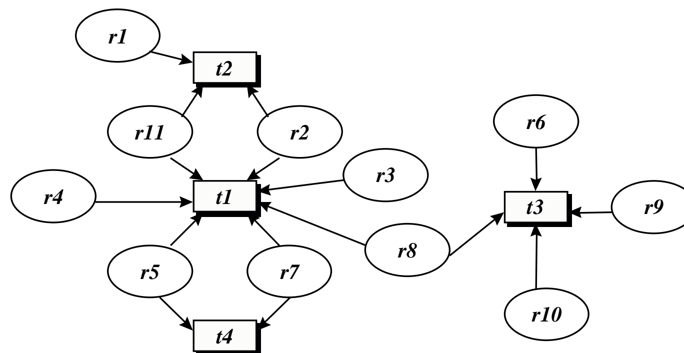

Figure 3. Type visibility graph of the relation of figure 1.a.

Note that routine interdependence graphs and reference graphs represent the same basic information, while type visibility graphs represent and overlapping but complementary information. First, whereas the first two record relations involving *individual variables*, type visibility graphs are interested in *types*. Second, whereas the first two are interested only in global variables, type visibility graphs also record relations to types of *local variables* and *routine parameters*.

2.2 Object Identification

Procedural code does not contain an explicit representation of objects. It contains only global variables, data structures (records) and routines (functions and procedures). However, the conscientious designer often restricts the access and modification of a data structure to a limited number of routines in order to enhance the modularity of the system. The identification of such a grouping of routines and records is the intuition behind many of the object identification techniques. The other type of grouping involves routines and global variables that are not structured types[WM1].

In [Liu & Wilde, 90], Liu and Wilde have proposed two algorithms, one to group data structures with routines that use them as parameters or return values, and the other to group the global variables with routines. The latter uses the routines interdependence graph (c.f. 2.1). Each strongly connected sub-graph is identified as an objet. Later other works ([Ogando et al., 94], [Livadas & Roy, 92], [Harris et al., 95], and [Sward & Hartrum, 97]) proposed some heuristics [WM2]to enhance Liu & Wilde's work. Yeh & al combine data structures with global variables in order to form groups of routines, data structures and global variables [Yeh et al., 95]. Girard et al. propose an approach called similarity clustering to identify both objects (called abstract state Encapsulations) and classes (or abstract data types) [Girard et al., 97]. This approach is based on the similarity metric introduced by [Schwanke, 91].

Other algorithms use reference graphs as introduced in [Dunn & Knight, 93] (c.f. 2.1). Canfora & al propose an algorithm that transforms a reference graph into a set of strongly connected and disjoint sub-graphs [Canfora et al., 96], where each sub-graph represents a candidate object. This transformation is based on the notion of variation of the internal connectivity of sub-graphs (ΔIC) of the reference graph. At the beginning, each routine defines a sub-graph (with a ΔIC). Two primitives are used to transform the graph (in an iterative way) : *Merge* and *Slice*. *Merge* clusters all the data of a sub-graph into a single data node. This is done when the ΔIC is greater than a threshold value. *Slice*, consists of slicing a routine to dissociate two sub-graphs. This occurs when the ΔIC is less than the same threshold value. The major weakness of this algorithm is the way the threshold value is calculated. The proposed approach is based on the programming style, which is very difficult to assess.

Based on the hypothesis that source code does not contain enough information to identify objects, other methods use additional domain knowledge. One such method is the COREM method [Gall & Klösch, 95]. In COREM, the migration to object technology is seen as a four step process. The first is *design recovery* which consists of extracting various low-level design documents (i.e. structure charts, data-flow diagrams) from the source code. These documents lead to the generation of an entity-relationship diagram (ERD). The ERD is transformed into an object-oriented application model (called RooAM). *Application modelling,* which is the second step of the migration process, consists of creating another object-oriented application model (called FooAM) based on the requirements analysis of the procedural input program. The object-oriented application modelling process is done by a human expert who is either experienced in

the application domain or who participated in the development of the program under consideration. In the third step of the migration process (called *Object Mapping*) the elements of the RooAM are mapped to the elements of the FooAM, resulting in a target application model (target ooAM). The target ooAM represents the desired object-oriented architecture and is defined as the synthesis of the FooAM and the RooAM. It incorporates all the elements that can be mapped between the two application models. The final step (called *source-code adaptation*) completes the program transformation process at the source-code level and is based upon the results of the previous steps, especially the target ooAM. This kind of methods relies heavily on the domain analysis of the application to migrate. In addition to the cost of this analysis, which is usually very high, most of legacy applications are not documented and the domain expertise is not always available, making this method of little practical use.

Finally, concept formation methods have been applied in software engineering for remodularization (see [Siff & Reps, 1997] and [Lindig & Snelting, 1997]). In these two projects, Galois (concept) lattices are used to identify modules in legacy code. Modules identification is different from object identification. In the first case, the identification is driven by the routines (how to group routines that share common data). Inversely, in the second case, the identification is based on the data (how to cluster data manipulated by a common sets of routines).

3. Overview of the proposed approach

The object identification approach we propose in this paper is based on the relationship between data and routines. It consists of five steps (see figure 4). First, we compute some metrics to determine the profile of the application at hand. This profile allows us to choose the appropriate program abstraction that we can use to identify objects. Then, we identify objects using different algorithms. Third, we identify the methods of these objects. The fourth step consists of identifying the relationships between the objects (generalization, aggregation, or more generally, associations). Finally, the source code is transformed using the so-derived object model. In this paper we limit ourselves to the first three steps. We have just started work on the last two. In the remainder of this section, we briefly introduce these three first steps. A detailed description is given in sections 5 to 7.

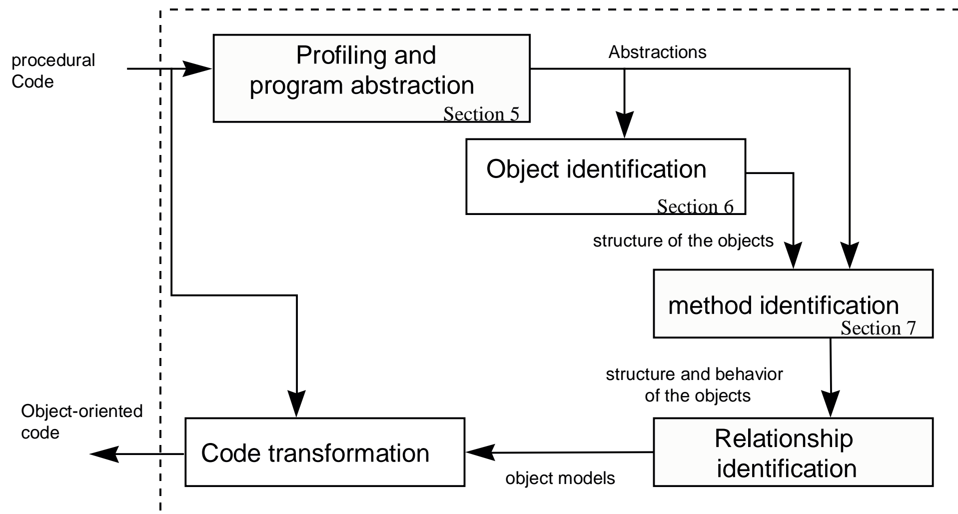

Figure 4. Overview of the object identification approach

Application profiling and program abstraction: We have adapted/defined three graphs that describe the relationship between routines (procedures/functions) and data in an application: (1) reference graphs, (2) user types visibility graphs and (3) data visibility graphs [Sahraoui et al., 97]. To be efficient, an object identification algorithm should use the appropriate program abstraction. For a given application, the topology of the different graphs may depend on the application domain and on the programming style, and different graphs may carry more or less significant information. For example, when migrating a library of functions, we should not use a reference graph, since such libraries rarely use global variables. A set of metrics can help in choosing the appropriate graph (e.g. average number of routines per global variable, or average number of routines per user type, etc.).

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 of operations that manipulates the data. Most of graph based approaches to object identification group data with the routines that use them. In our approach, we use two algorithms: (1) graph decomposition (a variant of [Canfora et al., 1996]) and (2) automatic concept formation using Galois lattices.

Method identification: The identification of methods is based on the relation between routines and data. The technique used here is independent from the algorithm used to identify objects. The only difference is that in the case of the graph decomposition algorithm, the identifications of objects and methods are done in the same time.

4. An example

To illustrate our approach and the related algorithms, let us take the well-known example introduced in [Canfora et al., 96]—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 routine, the body is replaced by a comment that indicates the list of data used by the routine. 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 */}
```

5. Application profiling and program abstractions

In this section, we present the three program abstractions we use in our approach, and the metrics that can help us choosing the right abstraction depending on the application domain and on the programming style. The three abstractions express most of the important relationships between routines (procedures/functions) and data (or types) in an application: (1) reference graph, (2) user types visibility graph, and (3) data visibility graph.

In the case of reference graphs, the relationship between a routine and a global variable simply indicates that the routine uses the variable. In our case, the way in which the routine uses the variable is important. We define three modes: modification or write mode (m) when the routine modifies the value of the variable, access or read mode (a) when it accesses its value to compute something else, and predicate mode (p) when the variable is used to control the execution of the routine (in a predicate). This classification is based on [Offutt et al., 93] and [Lounis & Melo, 97]'s work on module coupling. This improvement can help us for two reasons:

- 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).
- When we identify methods, the mode can be considered in conflict situations.

We extend the type visibility graph too with the attribute mode. Another attribute expresses the kind of visibility (formal parameter, global variable, local variable).

Finally, we define a third graph called *data visibility graph*. Like reference graphs, this one has two types of nodes (data and routines) and a single type of edge, meaning that a routine refers to data. The difference is that the data consist of both global variables and the local variables that are transmitted as parameters to others routines. A variable *v* is visible to the routine *r* in which it is declared and to all the routines that receive it as a parameter, either directly from *r*, or via other routines recursively.

As stated before, applications are different depending on the domain and on the programming style. To define the profile of an application, we have to apply a set of metrics. Depending on the measured values, we can choose the appropriate abstraction

for identifying objects. We have started working on the definition of a set of metrics. In this paper, we present some examples of metrics.

Let *NR* be the number of routines, *NV* the number of global variables, NRV the number of routines that access at least one global variable, NRT the number of routines that reference at least one user type, and finally NT the number of user types. These basic metrics allow us to compute other metrics. Figure 5 gives some examples and the abstraction concerned by each example.

The idea here is to define ranges for these metrics that would suggest the appropriateness of one program abstraction versus another. For example, a value of 200 for the metric *AURV*, meaning that there are, on the average, two hundred "useful[1]" routines per global variables, might suggest that the reference graph won't be of much help here.

To do so, we exploit the power of machine learning algorithms, in capturing knowledge that deduce a dependent variable, starting from a set of independent variables. In our case, the independent variables are profiling metrics that, in our sense, suggest the appropriateness of one program abstraction versus another. The dependent variable is the concerned abstraction.

Thanks to an initial restricted set of projects, we have started to extract and compute from their source code the profiling metrics. We have also selected the best abstraction for each project and consequently the dependent variable. By applying C4.5 [Quinlan, 93], a machine learning algorithm, we have generated rules that predict the dependent variable (i.e., the program abstraction) from a combination of relevant profiling metrics. However, presently the study do not concern a wide range of projects to be significant and to allow us, to validate such rules.

[1] A routine is considered useful for object clustering purposes if it accesses at least one global variable.

Name	Description	Formula	Concerned abstraction
VUR	Proportion of global variable useful routines	NRV/NR	Reference graph
PFU	Proportion of user type useful routines	NRT/NR	Type visibility graph
ARV	average routines per global variable	NR/NV	Reference graph
ART	average routines per user type	NR/NT	Type visibility graph
AURV	average useful routines per global variable	NRV/NV	Reference graph
AURT	average useful routines per user type	NRT/NT	Type visibility graph

Figure 5. Examples of application profiling metrics.

For all kinds of graphs, 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 syntactic pattern recognition and transformation program to extract the necessary information. The result of this process (a file) is a set of facts. For the reference graph facts are of type refers_to(f, v, t, m) where f is a routine, v is a global variable, t is the type of v, and m is the usage mode. For example, in the collections program (section 4), refers_to ("stack_push", "stack_point", "int", "m") means that the procedure stack_push uses the variable stack_point which is an integer in modification mode.

6. Object identification (Concept formation algorithm)

This step consists of identifying objects (their structure) using one of the three abstractions discussed earlier. To this end, we use two algorithms, a graph decomposition algorithm [Dumont, 98], and our own algorithm, which uses concept formation with Galois lattices. For the purposes of this paper, we limit our discussion to the second algorithm.

6.1 Principle of Galois lattice

Our approach relies heavily on an automatic concept formation method based on *Galois Lattices* (see e.g. [Godin et al. 95a]) that uses information extracted directly from code. In this section we present the basic definitions for Galois lattices, proposed by Godin in [Godin et al. 95a]. A better coverage of this subject can be found in [Davey & Priestly 92]. Algorithms based on this method are described in [Godin et al. 95b].

Consider two finite sets S and S' and a binary relationship R between them, and let *P(S)* and *P(S')* be the powersets S and S', respectively. Let π be a binary relation between pairs of subsets <X,X’>, where $X \subseteq S$ and $X' \subseteq S'$, such that <X,X’> π <Y, Y’> if and only if $X \subseteq Y$, and $Y' \subseteq X'$. Let <X,X’> be a pair of sets. We say that <X,X’> is complete with respect to the binary relation R if and only if:

1. X’ is the set of all the common images of the elements of X by relation R, i.e. $X' = f(X) = \{x' \in E' \mid \forall x \in X, xRx'\}$
2. X is the set of all the common antecedents of the elements of X’ by relation R, i.e. $X = f'(X')$ where $f'(X') = \{x \in E \mid \forall x' \in X', xRx'\}$

Consider the boolean matrix representation of the relation R. Graphically, provided a reordering of the rows and columns, complete pairs <X,X’> appear as rectangles full of 1’s that are of *maximal size*, i.e. there are no more full rows or columns.

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

(Column header: ***E'***)

Figure 6-a. Representation of binary relation R

Figure 6-b shows the same binary relation R after we have arranged the rows and columns. The rectangles show some of the complete pairs <X,X’> where X is the set of rows and X’ is the set of columns.

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

(Column header: ***E'***)

5			
	1	1	1

Two intersecting rectangles are in a π relationship if one bounds the other along one of the two dimensions. For example <{1,4},{f,h,c}> π <{1,4,2},{c}>, and <{1,2},{c,a}> π <{1,4,2},{c}>, but <{1,4},{f,h,c}> and <{1,2},{c,a}> are not comparable. Figure 6-c shows the graphical representation of the Galois lattice.

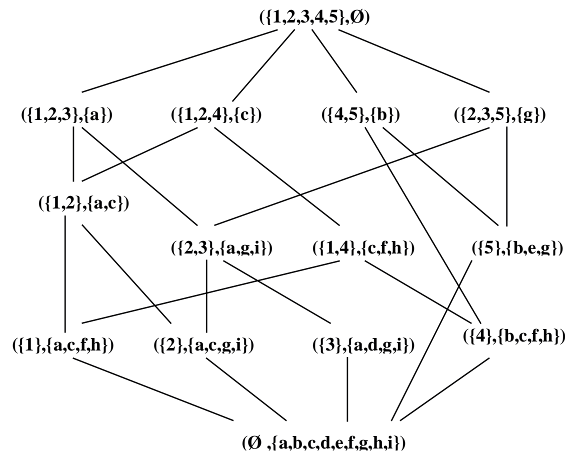

Figure 6-c. Galois lattice for relation R.

6.2 Applicability to object identification

In an OO design, an application is modeled by a set of objects where each object consists of a set of data and a set operations that manipulate this data. Most of the graph based approaches to object identification group data with the routines that manipulate them. Let *R* be the binary relation between data items and routines such that $(d,r) \in R$, or *d R r*, if the data item *d* is somehow manipulated by routine *r*. Let *S* (c.f. 6.1) be the set of data items, and *S'* the set of routines, then we can define a Galois lattice (C, π) where a pair <X,X'> ∈ C represents the association between a set of data items (X) and a set of routines (X') such that:

1. There are no other routines, besides the ones already in X', that manipulate all of the data items in X, and
2. There are no other data items, besides the ones already in X, that are manipulated by each and every routine in X'.

In other words, a complete pair <X,X'> represents a cohesive set of data items along with the routines that manipulate them, and the pair may be considered as a candidate object. The π relationship between two pairs $c_1 = <X_1, X'_1>$ and $c_2 = <X_2, X'_2>$, which

means that the set of data items X_1 *contains* the set of data items X_2 and the set of routines X'_1 is *included* in the set of routines X_2, can be interpreted as both an *extension*, whereby the "subclass" (c_1) defines new attributes (X_1 - X_2) and *specializes/redefines* some functions (X'_1) of the "superclass" (c_1).

6.3 The steps of the algorithm

The algorithm consists of three major steps:

1. Building the Galois lattice for the def-use graph,
2. Identifying an initial set of candidate objects,
3. Applying additional heuristic rules to filter and reorganize the initial set of candidate objects.

We illustrate the three steps using the *collections* program mentioned earlier.

6.3.1 Building the Galois lattice

Let S be the set of data items, and S' the set of routines, and let R be the relation which states that a data item $d \in S$ is used by the routine $r \in S'$. Figure 7 shows the matrix representation of R^{-1} (for readability purposes) of the *collections* program. For the sake of readability, names of routines and data items are replaced by codes (number for a routine and letter for data) when building the Galois lattice. The Galois lattice constructed from R identifies all the *potentially* significant groups of data items (see Figure 8).

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

Figure 7. Matrix representation of the reference graph for the *collections* program.

6.3.2 Candidate object identification

The goal of this step is to identify candidate objects out of all the possible groupings identified by the Galois lattice. For instance, the number of nodes within a lattice can be fairly large (see section 6.4), and not only aren't all such nodes "interesting", but there is a lot of redundance between them. Accordingly, we developed a set of heuristics to filter out the set of candidate objects. The heuristics we developed are meant to favor, all other things equal:

1. Groupings of data that share the most behavior,
2. Small size objects,
3. Disjoint objects (i.e. favor a partitioning of the data items).

To accommodate the first and second objectives, we start our evaluation of the groupings from the bottom of the lattice, i.e. with the nodes with the smallest number of data items and the greatest number of functions. From there we branch upwards, evaluating

candidate nodes, as long as there are data items not yet accounted for; let *NS* be the set of *N*ot yet *S*elected data items. Initially, $NS = D$, and the process stops when $NS = \emptyset$.

The algorithm proceeds in a branch and bound fashion, each time expanding the most recently retained/selected node. Out of all the terminal nodes, we select the one with the bigger set of routines (first objective above). In case of a tie, we select the node with smallest set of data items (objective two). In case of a tie, we select the node (a pair <X,X'>) that introduces the most *new* data items (i.e. the pair <X,X'> such that $X \cap NS$ has the biggest cardinality). This last criterion accommodates the third objective above.

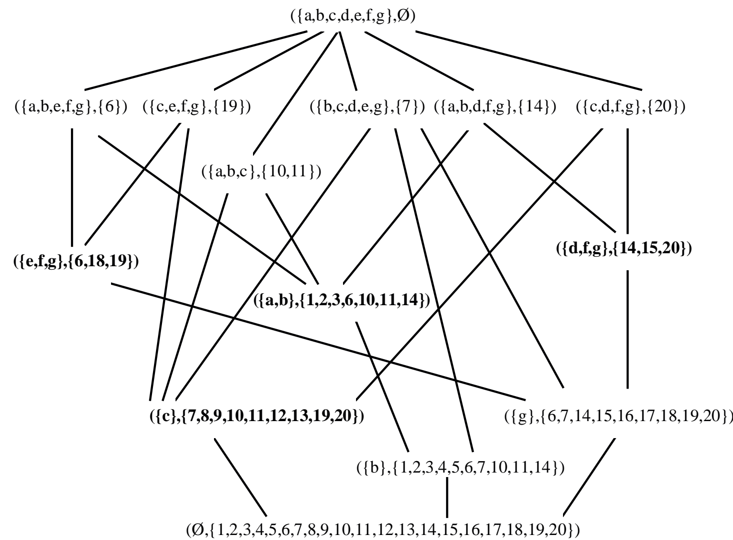

Figure 8. Galois lattice for the reference relation (*collections* program).

The application of this algorithm to the example of figure 8 gives the following four candidate objects/data item sets:

co1 = {c} = {list}

co2 = {a,b} = {stack_struct, stack_point}

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

co4 = {d,f,g} = {queue_tail, queue_struct, queue_num_elem}

Note that we describe the candidate objects by focusing on the data part, for two reasons. First, the set of data items (the X in the pair <X,X'>) unambiguously determines the set of corresponding functions. Second, while routines are used to identify the cohesiveness of data sets, their potential assignment to several groupings can be problematic, as explained in section 7, and a separate step is needed to identify methods.

6.3.3 Final object identification

If we consider the candidate objects co3 and co4, we note that they share two data items out of three. Such situations motivate the introduction of a new step that automatically merges these two objects. To detect these situations, we built a Galois lattice based on the *contains* relationship between candidate objects and the data items they contain. Referring to co2 above, we have co2 *contains* stack_struct and co2 *contains* stack_point.

Figure 9 shows the resulting Galois lattice. Let <X={$co_1,\ldots,co_i$},X'> be a node of this lattice. We have:

$$X' = \cap_{coj}\ contains(co_j)$$

The decision to merge the candidate objects in a given node <X={$co_1,\ldots,co_i$},X'> is based on the number of data items they have in common (cardinality of X'), relative to the number of data items they don't share. In the *collections* example, the candidate objects co3 and co4share most of their data items, i.e. two out of three ({f, g}). This motivated the following rule:

Merging rule:

Given a node <{$co_1,\ldots,co_i$},X'= $\cap_{coj}$ *contains*(co_j)> of the Galois lattice. The candidate objects { $co_1,\ldots,co_i$} are candidates for merging if and only if:

$$\underset{i}{Max}(|\,contains(co_j) - \bigcap_{co_k} contains(co_k)\,|) \leq |\bigcap_{co_k} contains(co_k)\,|$$

In other words, the candidate objects should be merged if "what they have in common is more than what differentiates them".

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

o1 = co1 = {c} = {list}

o2 = co2 = {a,b} = {stack_struct, stack_point}

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

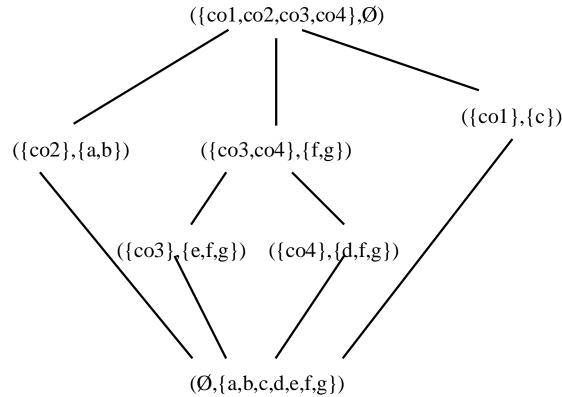

Figure 9. Galois lattice for the *contains* relationship between candidate objects and their data items for the *collections* program.

In our prototype an expert can amend the decisions of the system to merge candidate objects that weren't considered by the program, or override the program's decision to merge two objects.

6.4 Complexity

Let n be the cardinality of the set S (c.f. 6.1) and R be the binary relation, and assume that there is a finite upper bound K on the cardinality of R(s) for any $s \in S$, i.e.

$$K = \mathrm{Max}(\{\mathrm{Cardinality}(f(x)) \mid x \in E\}),$$

Godin et al. show that in this case, the worst case complexity of the Galois lattice (number of nodes *nl*) is linear with respect to n [Godin et al., 95a]:

$$nl \leq 2^{K} n.$$

At the same time, it is proven that the relations R and R' give 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 data 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 data

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.

In conclusion, the size of the lattice is linear with respect to the size of the program (number of routine). This show that our algorithm is applicable to large programs.

7. 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 routines. 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 [Mili, 96].

Let *O* be the set of identified objects, *F* the set of routines in the legacy code, and *D* the set of data. For each routine *r*, we define two sets *ref(r)* and *modif(r)* as follows:

$\forall\, r \in F,$

$ref(r) = \{o_i \in O \mid \exists\, d_j \in D \text{ and } d_j \text{ in } o_i \text{ and } d_i R r\}$ where *R* denotes the relation *is used by*.

$modif(r) = \{o_i \in O \mid \exists\, d_j \in D \text{ and } d_j \text{ in } o_i \text{ and } d_i M r\}$ 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 section 5).

There are three possible cases :

1. cardinality of *ref(r)* = 1

2. cardinality of *ref(r)* > 1 and cardinality of *modif(r)* = 1

3. cardinality of *modif(r)* > 1

For each case we define a rule.

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

The first case is trivial. For example in *collections*, *ref(stack_full)* = $\{o_2\}$. *stack_full* becomes a method of o_2.

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

This rule is motivated by the fact that conceptually we consider a routine as a behavior of an object if it modifies its state. For example, ref(stack_to_list) = $\{o_1, o_2\}$ and modif(stack_to_list) = $\{o_1\}$, stack_to_list becomes a method of o_1. stack_to_list is a conversion routine. In object oriented programming, there are two possibilities to

convert an object o_1 into another object o_2 : (1) ask o_1 to become o_2 (e.g. in smalltalk, method asPolyline in Circle class which convert a circle into a polyline), and (2) create o_2 from o_1 (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 routine r, if cardinality of ref(r) > 1 and cardinality of modif(r) > 1, then r must be sliced when possible to create a method for each object in modif(r).*

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*.

Program slicing is a family of program decomposition techniques based on selecting statements relevant to a computation, even if they are scattered throughout the program (or the routine in our case). Slicing has a very clear semantic based on projection of behavior of the program being sliced ; it can be used to isolate code fragments implementing a functional abstraction like a cohesive class method.

Slicing was originally defined by [Weiser, 84] and is based on static data flow and control flow analysis on the flow graph of the program. There are some other algorithms, such as, transformation slicing for program reuse, conditioned slicing for program understanding, and specification driven program slicing for identifying and isolating reusable routines. Generally speaking, we can define a slicing criterion for each identified object. It is the combination of a program point s and a set of relevant data *D* at that point. In our context, the program point is the end of the routine we want to slice, and the relevant data are those that constitute the object, for which we are building a method :

$$\text{Criterion}_{i} = \langle s, D \rangle, \text{ where } s : \text{end of the target routine, and } D = \{d_j \mid d_j \in o_i\}$$

Thus, we can extend the original definition of Weiser's program slice to an "encapsulated" slice, one that include statements that contribute directly or indirectly on the computation of the data dj of the object oi. Thanks to the def/use graph of the concerned routine, the slicing algorithm calculates each executable node from s back to the first one. On the other hand, it could be necessary to apply the same slicing procedure on other routines. It is the case when the slice of the routine includes a statement which calls or is called by other routines.

However, the main problem in applying such a technique is to preserve the logical control flow of the initial program. This is the present topic we are working on in this approach.

On the other side, it is not often possible to break a routine into cohesive methods. Other solutions can be used depending on the target OO language. In C++ for example, it is possible to define a routine 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(r)*, and put *r* as a method in that object.

8. 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 [Revault et al., 1995]) and a pattern recognition extractor generator. Figure 11 shows a graphical editor of our prototype, which allows to display and manipulate Galois lattices.

We used the prototype on five C applications ranging in size between 3,000 and 47,000 lines of code. We applied both the concept formation algorithm (CF) and the graph decomposition algorithm (GD). Experts assessed, on a scale of 0 to 2, their level of agreement for each identified object (0 for inappropriate, 1 for appropriate with some changes, and 2 for appropriate as is). Let P0, P1, and P2, represent the percentage of identified objects to which the experts assigned agreement levels 0, 1, and 2, respectively. For each combination (algorithm i, application j, abstraction k), we obtain one set of values (percentages) for the variables P0, P1 and P2—call them P0ijk, P1ijk, and P2ijk. From these values, we can identify, for each combination (algorithm i, application j), the abstraction k that yielded the highest agreement with the experts' assessment. Symbolically :

$$k = x \mid P2_{ijx} = \max_{k}(P2_{ijk})$$

We do that because we do not define yet the threshold values for the abstraction selection metrics (c.f. 5). We obtain then for each combination (algorithm, application) three values $P0_{ij}$, $P1_{ij}$ and $P2_{ij}$. Finally for each algorithm, we calculate for each level the average of the five values corresponding to the five applications (see figure 10):

$$P0_i = \mathrm{Avg}_j(P0_{ij}),\ P1_i = \mathrm{Avg}_j(P1_{ij}) \text{ and } P2_i = \mathrm{Avg}_j(P2_{ij})$$

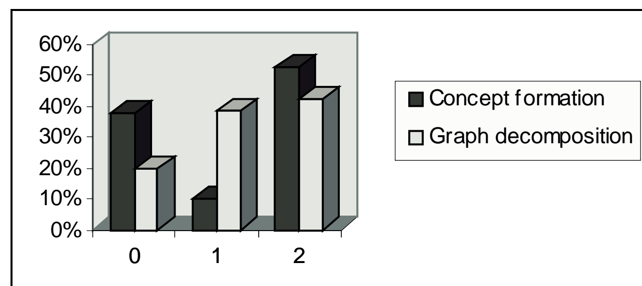

Figure 10. Validation results

First, we note the relatively high number of rejected objects (37.5% for CF and 19.56% for GD). By looking more closely at the experimental data, we identified two

factors that might have contributed to this number. First, all of the applications use external libraries, and some of the data nodes in the abstraction graphs correspond to global variables used within (or for) those libraries, and hence, should not have been considered by the algorithm. For example, in the application Proverbe (the largest one), a system for education record processing, we found that a number of data are related to the windows interface library (e.g. DLL handles), and our algorithms identified objects that were composed of both domain data and library data. The library data should not have been included in the identified objects for two reasons: (1) the code of the library is not available, and (2) as a general design principle, application code should be kept separate from user interface code. One solution would be to remove from the abstraction graphs all data nodes related to external libraries, since the goal is not to migrate libraries, but the applications that use them. When we did that for the Proverbe application, the identified objects were more meaningful. Figure 11 shows the Galois lattices before and after removing the undesirable data. In addition to illustrating the dramatic reduction in the complexity of Galois lattice, it also gives an idea about the size of the Galois lattices with all the data nodes.

The main lesson we learned from this problem was 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 a domain-independent library, and which components belong 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.

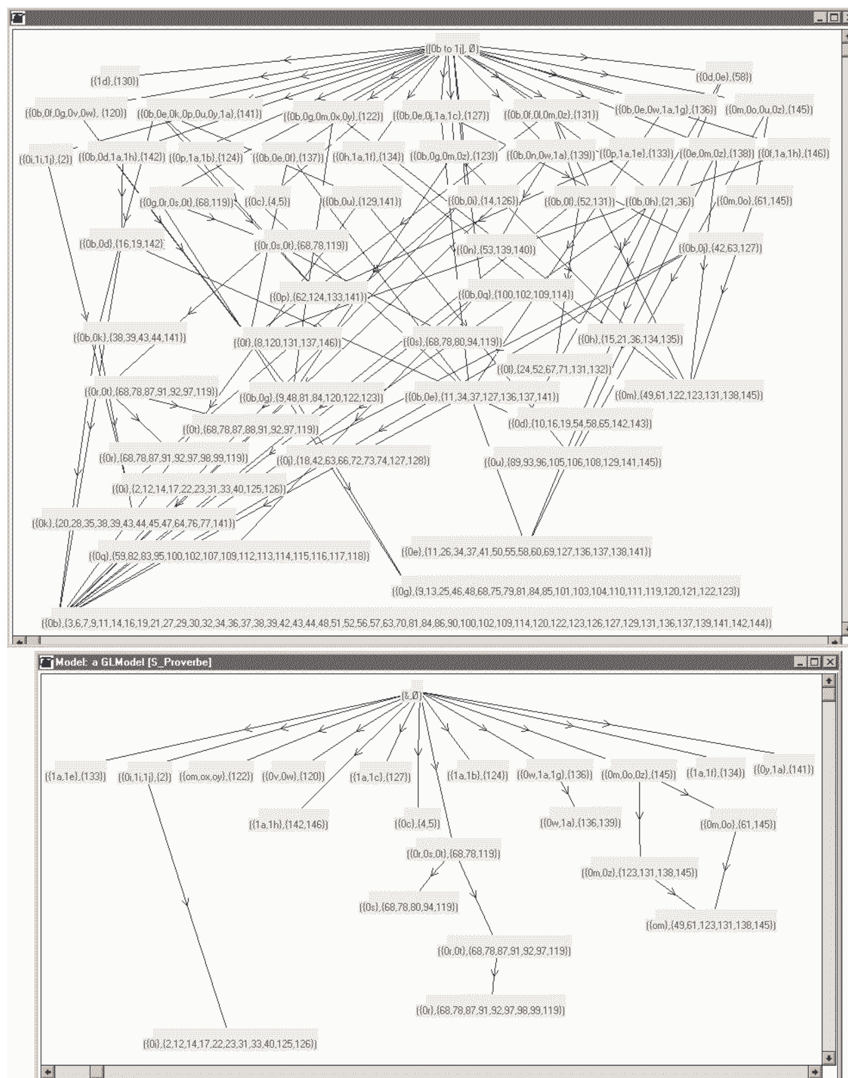

Figure 11. An overview of Galois lattice before and after removing the undesirable data for Proverbe application

The second contributing factor to the high level of rejected objects was that even within the domain-specific part of the application, not all of the data are relevant. Some of them are used as temporary variables or flags and do not represent domain objects. This was particularly true for the application SBC2 (6kLOC), a library for geometric shape recognition. The programmer of this application used a large number of flags and working variables for temporary storage. The same solution above can be applied to this situation.

When we compare the two algorithms, we notice that CF has more rejected objects. This can be explained by the fact that GD decomposes a graph into isolated sub-graphs and a data node is in a single sub-graph, corresponding to an identified object. In CF, the same data node can appear in more than one identified object. Combined with irrelevant data, the same variable can give several objects and will be rejected several times.

For the appropriate objects, CF gives better results (52.5% instead of 42% for GD). We expected this result because the decision criteria in GD are more subjective (calculation of the step value c.f. 2.2).

9. Conclusion and future work

In this paper, we described an approach aimed at 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 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 (routines-data/types) 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 [Godin et al. 95b]).

The same technique (Galois lattices) can also be applied to identify relationships between objects. Godin and Mili used Galois lattices to build inheritance hierarchies based on class signatures [Godin & Mili, 93]. Their algorithm can be used in the fourth step of our approach (c.f. 3). We have already started work on the actual migration of code (last step of the approach). We are currently implementing slicing algorithms (see [Weiser, 84], [Gallagher & Lyle, 91], [Lanubile & Visaggio, 93], and [Canfora et al., 94]) which allows us to generate two or more methods from a routine based on the results of the method identification step.

REFERENCES

[Canfora et al., 94] 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, 1994.

[Canfora et al., 96] G. Canfora, A.Cimitile, and M.Munro, An Improved Algorithm for Identifying Objects in Code, *Software Practice and Experience*, vol. 26, N°1, pp. 25-48, 1996.

[Corbi, 89] T.A. Corbi. Program understanding: Challenge for the 1990s, *IBM System Journal*, vol. 28 N°2, pp. 294–306, 1989.

[Davey & Priestly 92] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*, Cambridge: Cambridge University Press, 1992.

[Dumont, 98] F. Dumont, Extracting object from procedural code using graph decomposition, Master thesis, University of Sherbrooke, Quebec, 1998 (in french).

[Dunn & Knight, 93] 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, IEEE Computer Society Press, 1993.

[Gall et al., 95] H. C. Gall, R. R. Klösch and R. T. Mittermeir, Architectural Transformation of Legacy Systems, Workshop on Program Transformation for Software Evolution, ICSE, 1995.

[Gall & Klösch, 95] H. C. Gall and R. R. Klösch, Finding objects in procedural programs, *Second Working Conference on Reverse Engineering*, pp. 208–217, IEEE Computer Society Press, 1995.

[Gallagher & Lyle, 91] K.B. Gallagher & J.R. Lyle, Using Program Slicing in Software Maintenance, *IEEE Transactions on Software Engineering*, vol. 17 N°8 pp. 751-761, 1991.

[Girard et al., 97] J-F. Girard, R. Koschke, and G. Schied, A Metric-based Approach to Detect Abstract Data Types and State Encapsulations, *Proc. of IEEE Automated Software Engineering Conference*, pp. 82-89, 1997.

[Godin & Mili, 93] R. Godin and H. Mili, Building and Maintaining Analysis-Level Class Hierarchies using Galois Lattices, *In Proceedings of OOPSLA*, pp. 394-410, 1993.

[Godin et al., 95a] 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*, vol. 5, N°1, pp. 119-142, 1995.

[Godin et al., 95b] R. Godin, R. Missaoui and H. Alaoui, Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices, *Computational Intelligence*, vol. 11N°2, pp. 246-267, 1995.

[Harris et al., 95] D.Harris, H.Reubenstein, and A.S. Yeh, Recognizers for extracting architectural features from source code, *Second Working Conference on Reverse Engineering,* pp. 252–261, IEEE Computer Society Press, 1995.

[Jacobson & Lindstrom, 1991] I Jacobson and F. Lindstrom, Re-engineering of Old Systems to an Object Oriented Architecture, *Proceedings of OOPSLA*, pp. 340-350, 1991.

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

[Lehman & Belady, 85] M. M. Lehman and L. A. Belady, *Program evolution*, Academic Press, New York, 1985.

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

[Livadas & Roy, 92] P.E. Livadas and P.K. Roy, Program dependence analysis, *In Conference on Software Maintenance*, pp 356–365, 1992.

[Liu and Wilde, 90] 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, 1990.

[Lounis & Melo, 97] H. Lounis, W. Melo, Identifying and Measuring Coupling in Modular Systems, *8th International Conference on Software Technology ICST'97*, 1997.

[Mili, 96] H. Mili, On Behavioral Description in Object-Oriented Modeling, *The Journal of Systems and Software*, vol. 34, N°2, pp. 105-121, 1996.

[Offutt et al., 93] J. Offutt, M. J. Harrold and P. Kolte, A Software Metric System for Module Coupling, *The Journal of Systems and Software*, vol.20, N°3, pp. 295-308, 1993.

[Ogando et al., 94] R.M. Ogando, S.S. Yau, and N.Wilde. An object finder for program structure understanding, *In Journal of Software Maintenance*, vol. 6, N°5, pp. 261–83, 1994.

[Pressman, 87] R.Pressman, *Software Engineering: a Practioner's approach*, McGraw-Hill, second edition, 1987.

[Revault et al., 95] N. Revault, H.A. Sahraoui, G. Blain and J.F. Perrot, A Metamodeling technique: The METAGEN system, *Proceedings of TOOLS 16*, pp. 127-139, 1995.

[Sahraoui et al., 97] H.A. Sahraoui, W. Melo, H. Lounis, and F. Dumont, Applying Concept Formation Methods to Object Identification in Procedural Code, *Proc. of IEEE Automated Software Engineering Conference*, pp. 210-218, 1997.

[Siff & Reps, 1997] M. Siff, T. Reps, Identifying Modules via Concept Analysis, In *Proc. of ICSM'97,* 1997

[Sommerville, 92] I.Sommerville, *Software Engineering*, Addison Wesley, fourth edition, 1992.

[Sward & Hartrum, 97] R. E. Sward and T. C. Hartrum, Extracting Objects from Legacy Imperative Code, *Proc. of IEEE Automated Software Engineering Conference*, pp. 98-106, 1997.

[Schwanke, 91] R. W. Schwanke, An Intelligent tool for re-engineering software modularity. In *Proc. of International Conference on Software engineering*, p.83-92, 1991.

[Yeh et al., 95] A. S. Yeh, D. R. Harris, and H. B. Reubenstein, Recovering Abstract Data Types and Object Instances from a Conventional Procedural language, *Second Working Conference on Reverse Engineering*, pp. 252–261, IEEE Computer Society Press, 1995.

[Weiser, 84] M. Weiser, Program Slicing, *IEEE Transactions on Software Engineering*, vol. 10, N°4, pp. 352-357, 1984.