

# A Concept Formation Based Approach to Object Identification in Procedural Code<sup>\*</sup>

Houari A. Sahraoui<sup>1</sup>, Hakim Lounis<sup>1</sup>, Walcélcio Melo<sup>2</sup> and Hafedh Mili<sup>3</sup>

<sup>1</sup>*Centre de Recherche Informatique de Montréal, {hsahraou, hlounis}@crim.ca*

<sup>2</sup>*Oracle do Brasil and Universidade Catolica de Brasilia, Dept. de Ciência da Computação, wmelo@br.oracle.com*

<sup>3</sup>*Université du Québec 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

---

<sup>\*</sup> 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 domain models guide such tools, 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 of their limitations and proposes a five step domain independent approach that takes into account the profile of the application to migrate. The object identification step is presented in detail for one of the two algorithms we used. 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:

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

where "Reverse engineering" is the activity of defining an abstract representation of the system, "Changes" is 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:

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

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 ( $\underline{\underline{P(x_1) \cap P(x_2) \neq \emptyset}}$ ). Figure 1-a shows the reference relations between the routines  $f_i$ 's and the global data  $d_i$ 's of a program. The  $t_i$ 's represent the global data types. Figure 1-b shows 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.

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

Figure 1.a Reference relation between routines and global data

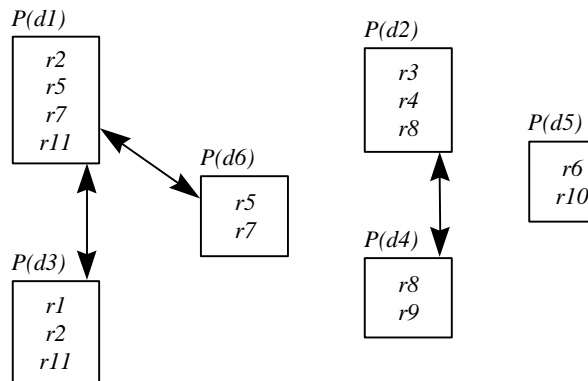


Figure 1.b Routine interdependence graph of the relation of figure1.a.

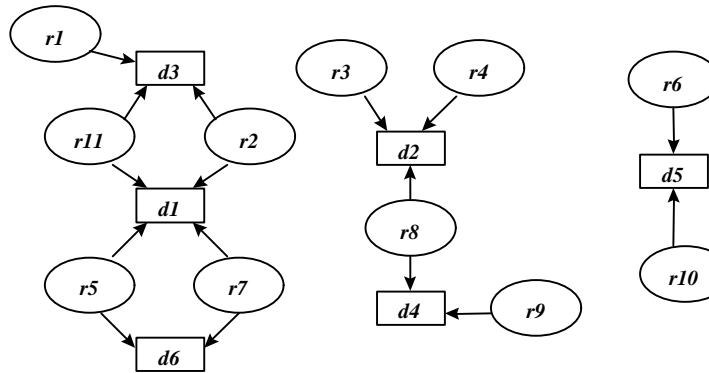


Figure 2. Reference graph of the relation in figure 1.a.

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 shows a partial type visibility graph based on the relation in figure 1-a. This kind of graph helps to identify directly the classes of objects rather than objects. This identification is done using isolated sub-graphs.

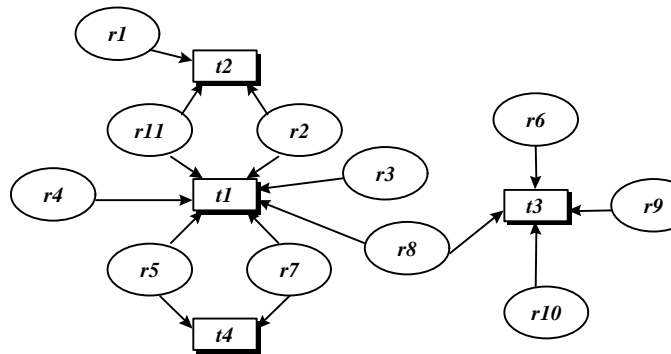


Figure 3. Type visibility graph of the relation of figure 1.a.

Note that routine interdependence graphs and reference graphs represent the same basic information, while type visibility graphs represent an 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.

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 object. Later other works ([Ogando et al., 94], [Livadas & Roy, 92], [Harris et al., 95], and [Sward & Hartrum, 97]) proposed some heuristics 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 (*DIC*) of the reference graph. At the beginning, each routine defines a sub-graph (with a *DIC*). 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 *DIC* is greater than a threshold value. *Slice*, consists of slicing a routine to dissociate two sub-graphs. This occurs when the *DIC* 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 modeling*, 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 modeling 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. Module 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.

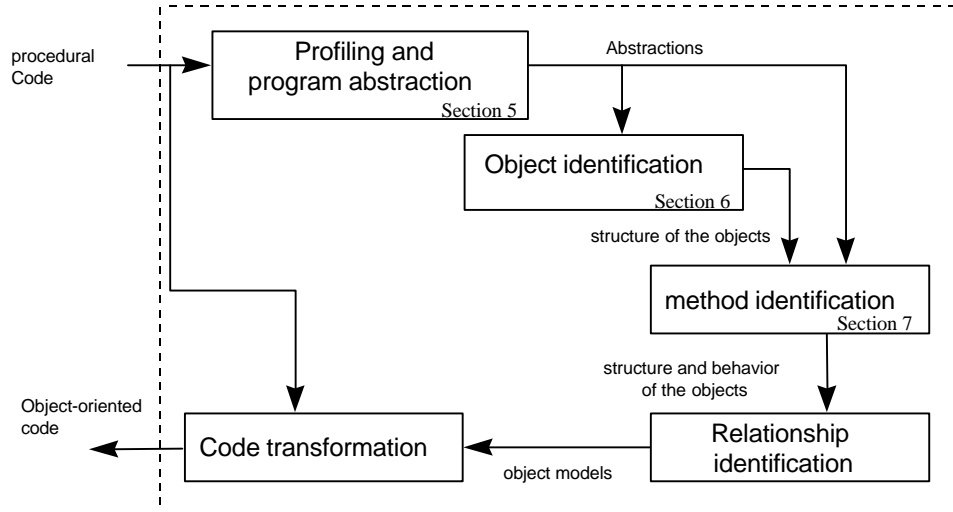


Figure 4. Overview of the object identification approach

*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-type 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, or 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<sup>1</sup>” routines per global variables, might suggest that the reference graph wouldn’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 does not concern a wide range of projects to be significant and to allow us, to validate such rules.

---

<sup>1</sup> A routine is considered useful for object clustering purposes if it accesses at least one global variable.

Name	Description	Formula	Concerned abstraction
<i>VUR</i>	Proportion of global variable useful routines	NRV/NR	Reference graph
<i>PFU</i>	Proportion of user type useful routines	NRT/NR	Type visibility graph
<i>ARV</i>	average routines per global variable	NR/NV	Reference graph
<i>ART</i>	average routines per user type	NR/NT	Type visibility graph
<i>AURV</i>	average useful routines per global variable	NRV/NV	Reference graph
<i>AURT</i>	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  $\pi$  be a binary relation between pairs of subsets  $\langle X, X' \rangle$ , where  $X \subseteq S$  and  $X' \subseteq S'$ , such that  $\langle X, X' \rangle \pi \langle Y, Y' \rangle$  if and only if  $X \subseteq Y$ , and  $Y' \subseteq X'$ . Let  $\langle X, X' \rangle$  be a pair of sets. We say that  $\langle X, X' \rangle$  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.  $X' = f(X) = \{x' \in S' \mid \forall x \in X, xRx'\}$
2.  $X$  is the set of all the common antecedents of the elements of  $X'$  by relation  $R$ , i.e.  $X = f'(X')$  where  $f'(X') = \{x \in S \mid \forall x' \in X', xRx'\}$

The Galois lattice  $G$  is the set of complete elements  $\langle X, X' \rangle$ , where  $X \in P(S)$  and  $X' \in P(S')$ .

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

		<i>S'</i>								
<i>R</i>		<b>a</b>	<b>B</b>	<b>c</b>	<b>d</b>	<b>e</b>	<b>f</b>	<b>g</b>	<b>h</b>	<b>i</b>
<i>S</i>	<b>1</b>	1		1			1		1	
	<b>2</b>	1		1				1		1
	<b>3</b>	1			1			1		1
	<b>4</b>		1	1			1		1	
	<b>5</b>		1			1		1		

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  $\langle X, X' \rangle$  where  $X$  is the set of rows and  $X'$  is the set of columns.

		<i>S'</i>								
<i>R</i>		<b>f</b>	<b>h</b>	<b>c</b>	<b>a</b>	<b>i</b>	<b>g</b>	<b>d</b>	<b>b</b>	<b>e</b>
<i>S</i>	<b>4</b>	1	1	1					1	
	<b>1</b>	1	1	1	1					
	<b>2</b>			1	1	1	1			
	<b>3</b>				1	1	1	1		
	<b>5</b>						1		1	1

Figure 6-b. A partial geometric representation of relation R

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

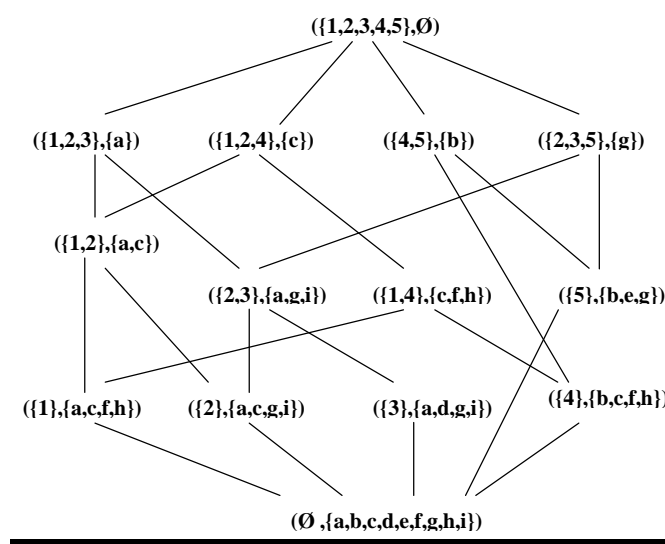


Figure 6-c. Galois lattice for relation R.

## 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 items and a set of operations that manipulate those data items. 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 the 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  $(C, \pi)$  where a pair  $\langle X, X' \rangle \in C$  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  $\langle X, X' \rangle$  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  $\pi$  relationship between two pairs  $c_1 = \langle X_1, X'_1 \rangle$  and  $c_2 = \langle X_2, X'_2 \rangle$ , 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 an *extension*,

whereby the “subclass” ( $c_1$ ) defines new attributes ( $X_1 - X_2$ ), and *specializes/redefines* some functions ( $X'$ ) 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 Not yet Selected 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  $\langle X, X' \rangle$ ) that introduces the most *new* data items (i.e. the pair  $\langle X, X' \rangle$  such that  $X \cap X'$  has the biggest cardinality). This last criterion accommodates the third objective above. Finally, a node with a single data element that have a basic type (e.g. int, char, etc.) is ignored.

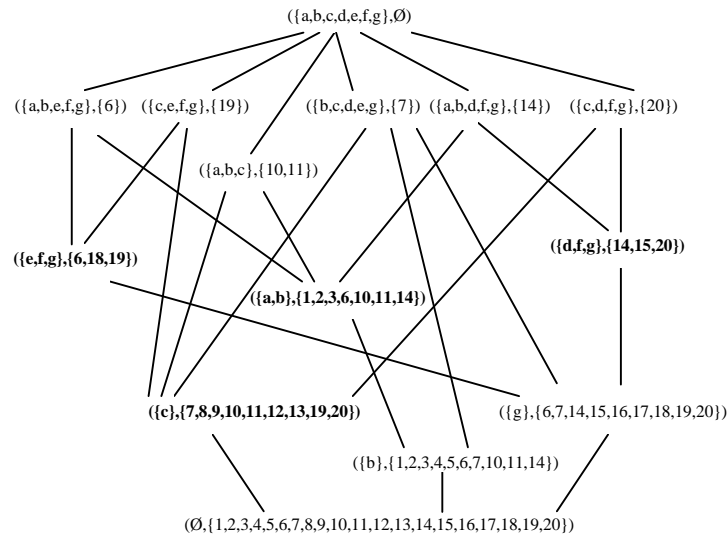


Figure 8. Galois lattice for the reference relation (*collections* program).

The application of this algorithm to the example of figure 8 is described by the following steps :

- $NS = \{a,b,c,d,e,f,g\}$ . The node  $(\{b\}, \{1,2,3,4,5,6,7,10,11,14\})$  has the biggest set of routines (10). But as  $b$  is an integer (`stack_point`), the node is ignored.
- Nodes  $(\{c\}, \{7,8,9,10,11,12,13,19,20\})$  and  $(\{g\}, \{6,7,14,15,16,17,18,19,20\})$  have the biggest sets of routines (9). They have the same structure size (1) and introduce the same number of new data (1).  $(\{g\}, \{6,7,14,15,16,17,18,19,20\})$  is considered randomly and ignored as  $g$  is an integer (`queue_num_elem`).
- $(\{c\}, \{7,8,9,10,11,12,13,19,20\})$  is then considered and a new candidate object `col = {c}` is identified ( $c$  has the complex type “list”).  $NS = \{a,b,d,e,f,g\}$ .

- $(\{a,b\},\{1,2,3,6,10,11,14\})$  is the next node with the biggest set of routines (7). This node is chosen as a second candidate object  $co2 = \{a,b\}$ .  $NS = \{d,e,f,g\}$ .
- Nodes  $(\{e,f,g\},\{6,18,19\})$  and  $(\{d,f,g\},\{14,15,20\})$  have the biggest sets of routines (3). They have the same structure size (3) and introduce the same number of new data (3).  $(\{e,f,g\},\{6,18,19\})$  is considered randomly and a new candidate object  $co3 = \{e,f,g\}$  is identified.  $NS = \{d\}$ .
- The node  $(\{d,f,g\},\{14,15,20\})$  is then considered. This node introduces a single new data element which is d. A new candidate object  $co4 = \{d,f,g\}$  is identified.  $NS = \emptyset$ . The algorithm stops.

The algorithm gives then 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  $\langle X, X' \rangle$ ) 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  $\langle X = \{co_1, \dots, co_i\}, X' \rangle$  be a node of this lattice. We have:

$$X' = \bigcap_{co_j \text{ contains}(co_j)}$$

The decision to merge the candidate objects in a given node  $\langle X = \{co_1, \dots, co_i\}, X' \rangle$  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  $co4$  share most of their data items, i.e. two out of three  $(\{f,g\})$