Submitted to:

2nd International Workshop on Software Reusability
March 24-26, 1993 - Lucca, Italy

*Isolating Code Related to a Certain Functionality in a Reuse Re-engineering Process

Oreste Signore - Mario Loffredo

CNUCE - Institute of CNR - via S. Maria, 36 - 56126 Pisa (Italy)
Phone: +39 (50) 593201 - FAX: +39 (50) 589354
E.mail: oreste@icnucevm.cnuce.cnr.it

Keywords: Software re-engineering, Reusability, Program slicing

Abstract

Software re-engineering and reusability are two areas of growing interest in the last years.
However, while many researchers have focused their interest in the classification of reusable modules proposing meaningful examples of reusable components repositories and software packages libraries, the problem of potential reusable chunks searching and extracting is still opened.
In this paper, after a brief discussion of the re-engineering issues, we examine the problems related to the difficulty of identifying various subfunctionality in a module and isolating them.

* This work has been partially supported by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of C.N.R.

1 - Introduction

Software engineering is evolving towards the implementation of tools that could improve the maintainability of the existing software applications, possibly by their reconfiguration.

Un aspetto importante da considerare nell' opera di manutenzione è il riuso dei prodotti dello sviluppo di software ed in particolare del codice.
E' stato dimostrato che si tende a riusare componenti software e che le applicazioni sviluppate ex-novo sono poche.
Cerchiamo di tradurre in cifre la ridondanza dei sistemi software ([McClure91]):

- 40 - 60% di tutto il codice di un' applicazione può essere usato per un' altra;
- 60% del progetto di tutte le applicazioni d' azienda è riusabile;
- 75% delle funzioni di un programma sono comuni a numerosi programmi;
- 15% del codice di un programma è proprio di una specifica applicazione;

L' adesione alla filosofia della riusabilità consente, come vantaggi immediati, di ridurre il costo e il tempo di sviluppo (i costi di manutenzione sono stati ridotti del 90% da quando per sviluppare software sono stati adoperati componenti riusabili), migliorare la qualità del software, incrementare la produttività e condividere la conoscenza del sistema.
Le ricerche e le applicazioni sul tema della riusabilità procedono secondo due direttrici principali.
La prima direttrice si pone come obiettivo la ricerca di metodi, teorie e modelli che aiutino a progettare ed implementare software riusabile e si ricollega all' aspirazione di allestire laboratori e fabbriche di software su vasta scala.
La seconda direttrice si occupa di studiare metodi, teorie e modelli per il riuso del codice esistente ed è connessa all' obiettivo di minimizzare le perdite nell' attuale patrimonio software.
Quest' ultimo, infatti, costituisce una fonte naturale dalla quale ricavare astrazioni funzionali e astrazioni sui dati utili a definire moduli riusabili.
Il codice esistente può, quindi, essere recuperato attraverso l' allestimento di un processo di reengineering ([Chikofsky90]) o, per meglio dire, di un processo di ***reuse re-***

engineering: which is claimed to re-design a system reusing knowledge and design elements taken from the previous products.

In tale processo, il software deve essere sottoposto a cirteri di candidatura che siano di aiuto nell' individuazione delle componenti eleggibili a rappresentanti di certi tipi di funzionalità.
Successivamente, tali componenti devono essere fatte oggetto di processi di ripulitura e cristallizzazione delle funzionalità suddette.

2.2 - Related work

In spite of its novelty, some relevant work has been yet done by several authors, and we can distinguish two main approaches in the area of the Object Oriented Re-engineering.
In their paper, Jacobson and Lindström ([Jacobson91]) suggest that an object oriented development method can be used to gradually modernise an old system via a three steps process. The first step consists of a reverse engineering phase, which allows to identify how the components of the system relate to each other and then create a more abstract description of the system. In the second step, reasoning about the changes in functionalities is done at a more abstract level. Finally, in the third step, a forward engineering phase takes place, redesigning the system from the abstract representation to the concrete one.
In the whole process, the informal documentation (manuals, requirements specifications, etc.) is taken into account in order to reconstruct the knowledge about the system functionalities.
In this approach, it is supposed that it will be possible to migrate from a top-down design environment to an object-oriented one. This implies a hybrid Software Life Cycle model (Edwards90]), where we can map the Data Flow analysis models into Object Oriented Design techniques ([Alabiso88]).
Liu and Wilde ([Liu90]) concentrate their attention on the methodologies to aid in the design recovery of object-like features of a program written in a non object oriented

language. Two complementary methods are proposed, based on an analysis of global data or of data types.
As far as the approach proposed by Jacobson and Lindström is concerned, we notice that, even if in principle the informal documentation may be of relevant importance, in practice it might happen that it is lacking or incomplete or inconsistent and misleading. Therefore, information kept from the informal documentation should be carefully examined and validated.
Liu and Wilde themselves in their paper raise the question if their approach may produce "too big" objects. This is due to the intrinsic characteristics of the proposed methods. In fact they consider as strongly connected procedures and data structures if they are used together, and in this case they identify the set of the data structures as an object and the procedures as methods of this object.
Because of this, we completely agree with the authors about the fact that "a further stage of refinement will be necessary in which human intervention or heuristically guided search procedures improve the candidate objects".

2.3 - Problems

The basic concepts of the object oriented approach are objects, methods, encapsulation, inheritance. On the other hand, in conventional programs, all the knowledge that is contained in the definition of an object and its methods, namely the static and the dynamic constraints and the procedural knowledge, are dispersed in the software modules.
Therefore, re-engineering towards an object-oriented environment raises several problems, as the identification of the objects and their methods requires a semantic analysis of the code, and the identification of similarities, exceptions, rules, etc.. It is obvious that a human intervention is somehow required, however, the application of some general rules can provide a reasonable understanding of the existing software, and reliable suggestions about the identification of the right components.
In this paragraph, we will discuss some of these problems and will sketch some possible solutions.
First of all, we can face many difficulties when we intend to associate the concept of object to any component of a traditional system, developed according to the top-down design style.

In addition, we can consider two possible alternatives:

- all the data or part of them are to be considered as objects;
- a set of sub-functionalities acting on several data objects may be considered as a single "software object".

A third problem is constituted by the fact that the traditional systems have been designed on the basis of a functional decomposition. This implies that it is difficult to identify the functionalities that operates on shared data, while it is very frequent to encounter modules where several subfunctionalities are intermixed or fragmented.

Finally, it must be stressed that an object oriented design style is something more than objects and methods: fundamental properties like encapsulation, information hiding, inheritance, polymorphism and dynamic binding constitute the actual innovative aspect of the object oriented methodology, and can be implemented by the identification of class hierarchies. In the following, we will briefly discuss some potential ways these problems can be faced and solved.

2.3.1 - Identification of the objects

As far as the data structures are concerned, we can distinguish between the global and the local data structures.

As a first step, we can consider a reasonable decision to identify as *objects* the *global data structures*, while local data structures can be taken as data local to the methods.

In the traditional programming languages, global data structures are stored in main storage areas that can be accessed by every component of the system ([Edwards90]). A typical example is constituted by the COMMON blocks in Fortran, or the external variables in C. In the DBMS based applications, global data structures are constituted by the database items (relations, segments, etc.) whose declarations are normally embedded in the source code by some peculiar instructions, like special include, in the appropriate and easily identifiable section of the source program. The correct identification of these structures requires the access to the libraries, or a careful examination of the pre-processor output. At a higher level of abstraction, other elements peculiar to different design phases, like entities can obviously considered as likely "candidate objects".

However, local data structures can be shared by different methods. This fact can have influence on the program slicing phase.

2.3.2 - Identification of fragments

Object methods must be derived by the analysis of the functionalities embedded in the procedures that manipulate the data structures previously identified as "objects".
As it has been pointed out discussing the [Liu90] proposal, it is necessary to adopt a "fine granularity" method to identify modules that implement the various subfunctionalities and to split, as possible as we can, the fragments corresponding to them.
Slicing[1] ([Weiser84], [Jiang91]), may help in this task. Even if the slicing has been initially used in program debugging and testing, it can be used to determine the "low cohesion" modules ([Ott89]), i.e. modules that violates the Parnas' principle of encapsulation and hiding([Parnas72]).
Therefore, the identification of the methods can be accomplished in two steps:

- slicing of the functionalities of a single module;
- clustering of the functionalities acting on the same object.

After the completion of these two steps, we can obtain a detailed map of the different "code chunks" acting on the various global data structures.

2.3.3 - Identification of abstractions

Up to now, the identification of the functionalities of a piece of code has been performed analysing its control flow graph and isolating special strongly connected sub-graphs (primes) associated with an atomic subfunctionality. Afterwards, the comparison of these primes with a standard atom library ([Wills90]) lead to the comprehension of the program behaviour.
However, we think that this approach is in some way limited, as two programs with a different logical structure may in fact implement the same functionality. Therefore, it is necessary to understand the semantics of the program. The usage of pre and post conditions has been proposed ([Hausler90]) as a techniques for the precise semantic specification of the code.
Therefore, in order to identify the inheritance hierarchies, we have to solve essentially the following problems:

---

[1] The program slicing is a process that, once a subset of "program behaviours" has been defined (Slicing criterion), reduces the program itself to a "minimum form" such that it gurantees the same subset of selected behaviours.
For a more formal definition see [Weiser84].

Therefore, in order to identify the inheritance hierarchies, we have to solve essentially the following problems:

- identification of "program chunks" which are identical from the *structural* point of view;
- identification of "program chunks" which are identical from the *syntactical* point of view;
- definition of hierarchies of global data structures.

3 - The proposed approach

As it has been pointed out in the previous paragraphs, object oriented re-engineering requires the identification of the *objects*, the *methods* and the *inheritance hierarchies*. This task can be fulfilled partly automatically, partly supported by the human intervention. The knowledge about the system must be extracted either from formal sources, i.e. the code, either from informal documentation.

In the following, we will give an outline of the proposed approach, where, starting from the analysis of the code, we obtain a list of ......
The architecture of our approach is depicted in Fig.2.

The first step is to move from a "well structured" code, i.e. a code without GOTO written in C language. However, these assumptions do not reduce the generality of the approach, as code restructuring tools are available on the market, and the peculiarities of the C language will affect only a minimal part of the process. In fact, we are looking for data structures ("external variables") that are available in other programming languages, too, even if with different names.
Afterwards, we try to identify "similarities" of portions of code, building the nesting trees of the procedures and looking for the equal regular expressions that represent the code.
The third step is the program slicing, which is performed making use of construction of the Program Dependence Graph.

Finally, we can proceed to the identification of the functionalities. In this phase, a human intervention is required, to validate the choices. At this stage, the informal documentation may be taken into account.

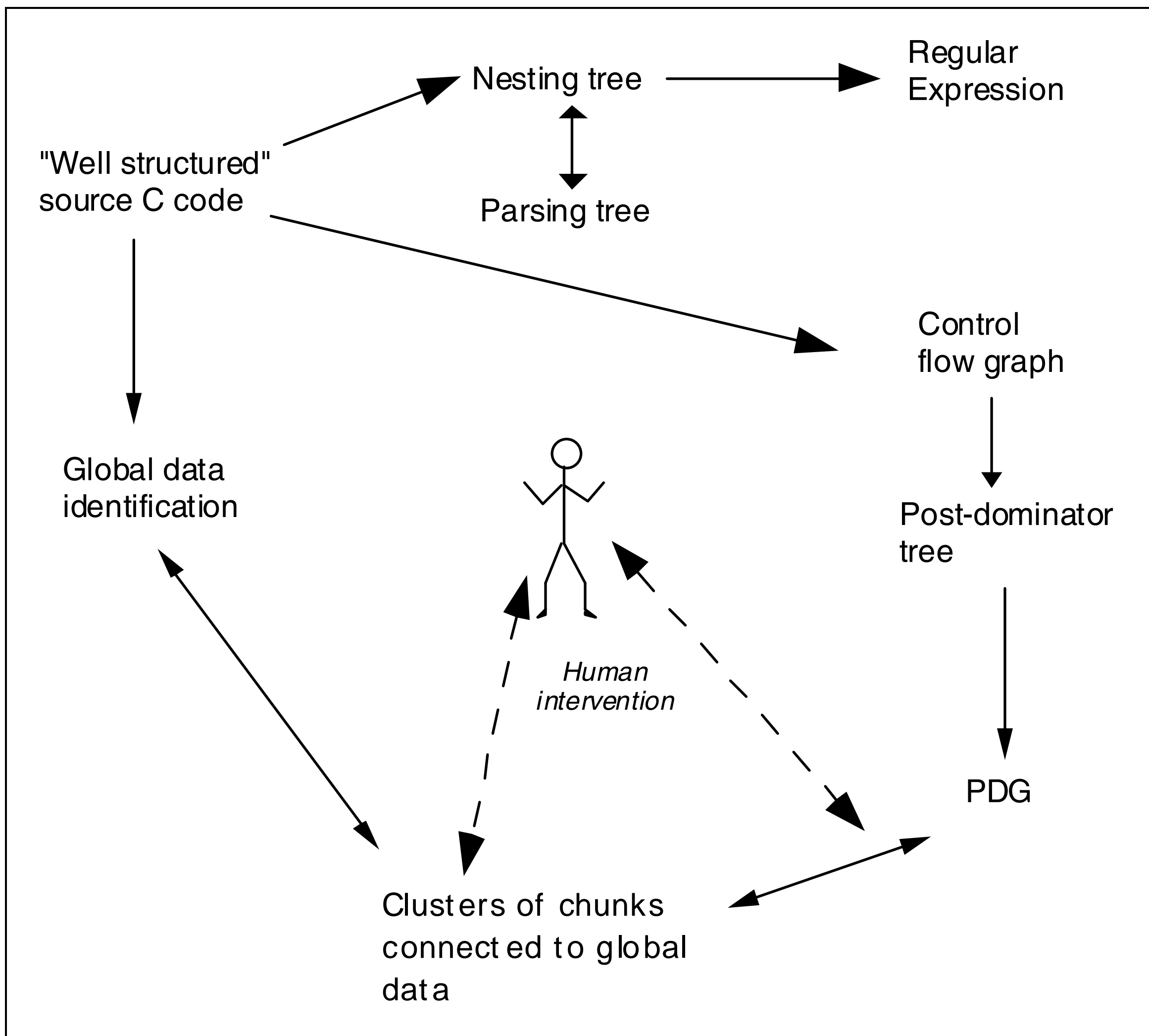

Fig. 2 - The general architecture

3.1 - The regular expression

We propose to use regular expression to represent the control logic of the code. This approach is also used by [Cimitile91] and [Wegman83] for different purposes. The great advantage of this kind of representation, is that we can reduce the identification of structurally identical subgraphs to the finding of identical substrings. In addition, we can take the advantages of the information retrieval approach (i.e. term significance) concentrating our attention on some general structural features of the code.
The grammar for the code “expressions” is reported in Appendix A.

In Appendix B we report an example of the identification of portions of code that belongs to two different procedures, but are identical from the structural point of view.

3.1 - The Program Dependence Graph

To solve the problems related to the search of candidate methods, we need a form of representation showing the relationship between sliced code and managed global data.
Therefore, such form of representation should have to show both data dependencies and control dependencies.
A form of representation matching these requirements is the **Program Dependence Graph** (PDG) ([Ferrante87]).
The PDG makes explicit both the essential data and control relationships without the unnecessary sequencing present in the control flow graph.
The PDG represents a program as a graph in which nodes are statements or predicate expressions and the edges incident to a node represent both the data flows and the conditions that control the execution of the operations.
In fact, there are two types of dependencies in a program.
First, a dependence exist between two statements each time a variable appearing in one statement may assume an incorrect value if the two statements are reversed.
For example, given

```
A = B * C                   (S1)
D = A* E + 1                (S2)
```

S2 depends on S1 because executing S2 before S1 an incorrect value for A would result in S2. Dependencies of this type are named *data dependencies*.
Another type of dependence exist between a statement and a predicate whose value immediately control the execution of the statement. For example, in the sequence

```
if ( A ) then               (S1)
   B = C * D                (S2)
endif
```

S2 depends on predicate A because the value of A determine if S2 is executed.
Dependencies of this type are named *control dependencies*.

3.1.1 - Control dependencies

First of all, we have to give some definitions.

Definition 1.

A *control flow graph* is a directed graph G augmented with an unique entry node START and a unique exit node STOP such that each node in the graph has at most two successors. We assume that nodes with two successors have attributes “T” (true) and “F” (false) associated with the outgoing edges in the usual way. We further assume that for any node N in G there exists a path from START to N and a path from N to STOP.

Definition 2.

A node V is *post-dominated* by a node W in G if every directed path from V to STOP (not including V) contains W.

Note that this definition of post-dominance does not include the initial node on the path. In particular, a node never post-dominates itself.

Definition 3.

Let G be a control flow graph. Let X and Y be nodes in G. Y is *control dependent* on X iff

(1) there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and
(2) X is not post-dominated by Y.

If Y is control dependent on X then X must have two exits. Following one of the exits from X always results in Y being executed, while taking the other exits may result in Y not being executed.

When applied to a loop in the control flow graph, our definition of control dependence determines a strongly connected region (SCR) of control dependencies whose nodes consist of predicates that determine the exits from the loop.

While in the control flow graph nested loops appear as nested SCRs, in the PDG they appear as distinct SCRs with a control dependence edge between the outer loop and each immediate inner loop. So the nesting hierarchy is most evident since loops at the same level appear as SCRs with a common ancestor.

How do we determine control dependencies?.

The first step consists of the construction of a post-dominator tree ([Lengauer79]) for the control flow graph augmented with a special predicate node ENTRY that has one edge labelled “T” going to START and the other edge labelled “F” going to STOP.
ENTRY corresponds to whatever external condition causing the beginning of program execution.

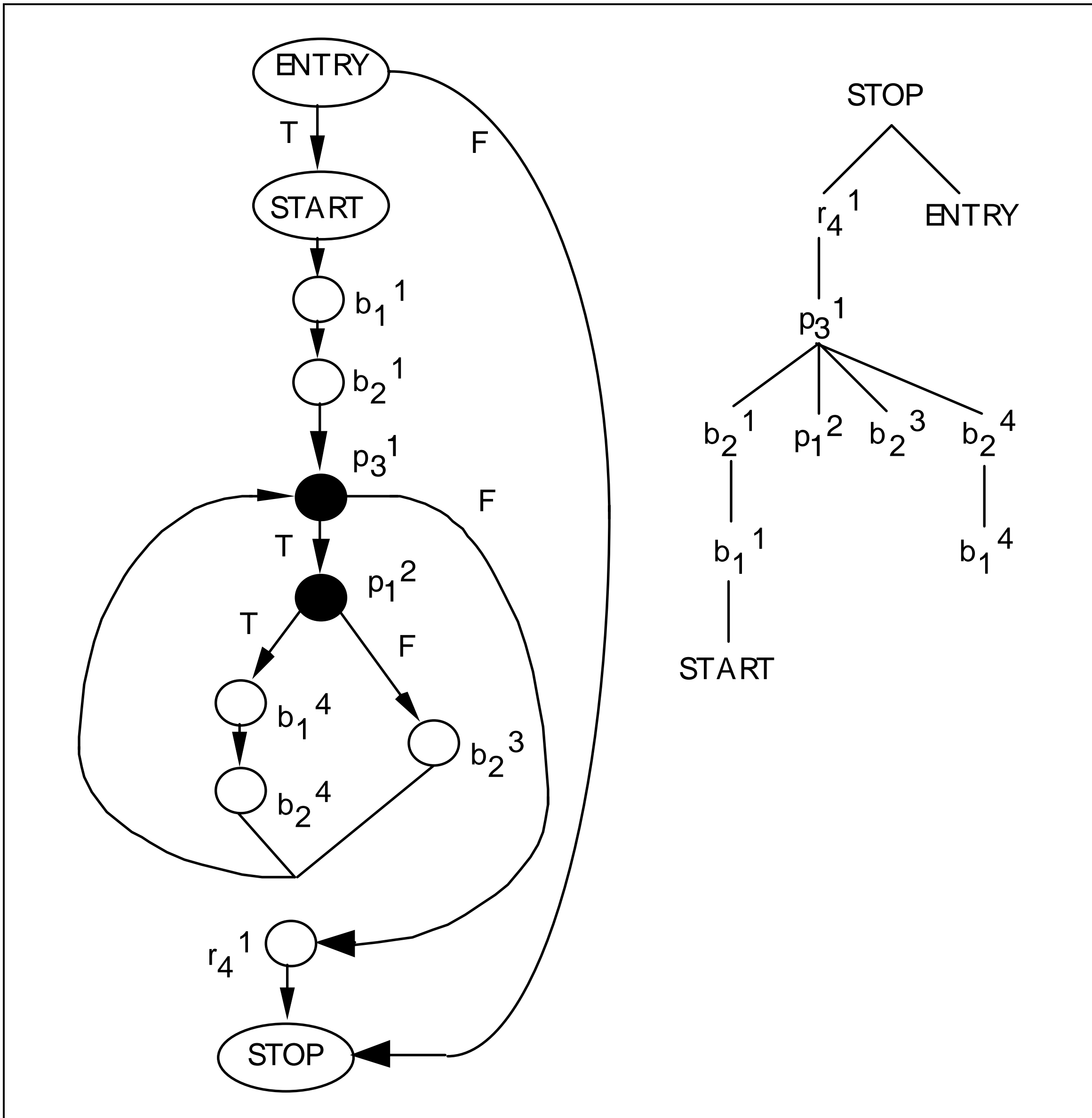

Fig. 3 - The augmented control flow graph of Proc1 and the related post-dominator tree

After determining the post-dominator tree, we can find control dependencies by examining certain control flow graph edges and annotating nodes on corresponding tree paths.

Lets take into account the augmented control flow graph in Fig. 3.
Given the post-dominator tree, we create the set S containing all the edges (A, B) of the control flow graph such that B is not an ancestor of A in the post-dominator tree (i.e. B is not post-dominated by A). Each of these arcs has an associated label “T” o “F”.
In our example, S={(ENTRY, START), (p_1^2, b_2^3), (p_3^1, p_1^2), (p_1^2, b_1^4)}.

The control dependence determination algorithm proceeds by taking into account each edge (A, B) in S. Let L the least common ancestor of A and B in the post-dominator tree.
By construction, we cannot have L equal B and we can obtain only two possible solutions:
L equal A or L is the parent of A in the tree.
Given an edge (A, B) in S and starting from B, we should traverse backwards from B in the post-dominator tree until reaching L marking all nodes that we will meet. The following table show the control dependencies found by examining each edge in the control flow graph of fig. 3.

Edge in S examined	Nodes marked	Control dependent on	Label
(ENTRY, START)	START, b_1^1, b_2^1, p_3^1, r_4^1	ENTRY	T
(p_1^2, b_2^3)	b_2^3	p_1^2	F
(p_3^1, p_1^2)	p_1^2	p_3^1	T
(p_1^2, b_1^4)	b_1^4 ,b_2^4	p_1^2	T

The last step for the construction of the control dependence subgraph consists of the addition of region nodes that summarise the set of control conditions for a node and group all nodes with the same set of control conditions together (Fig.4).

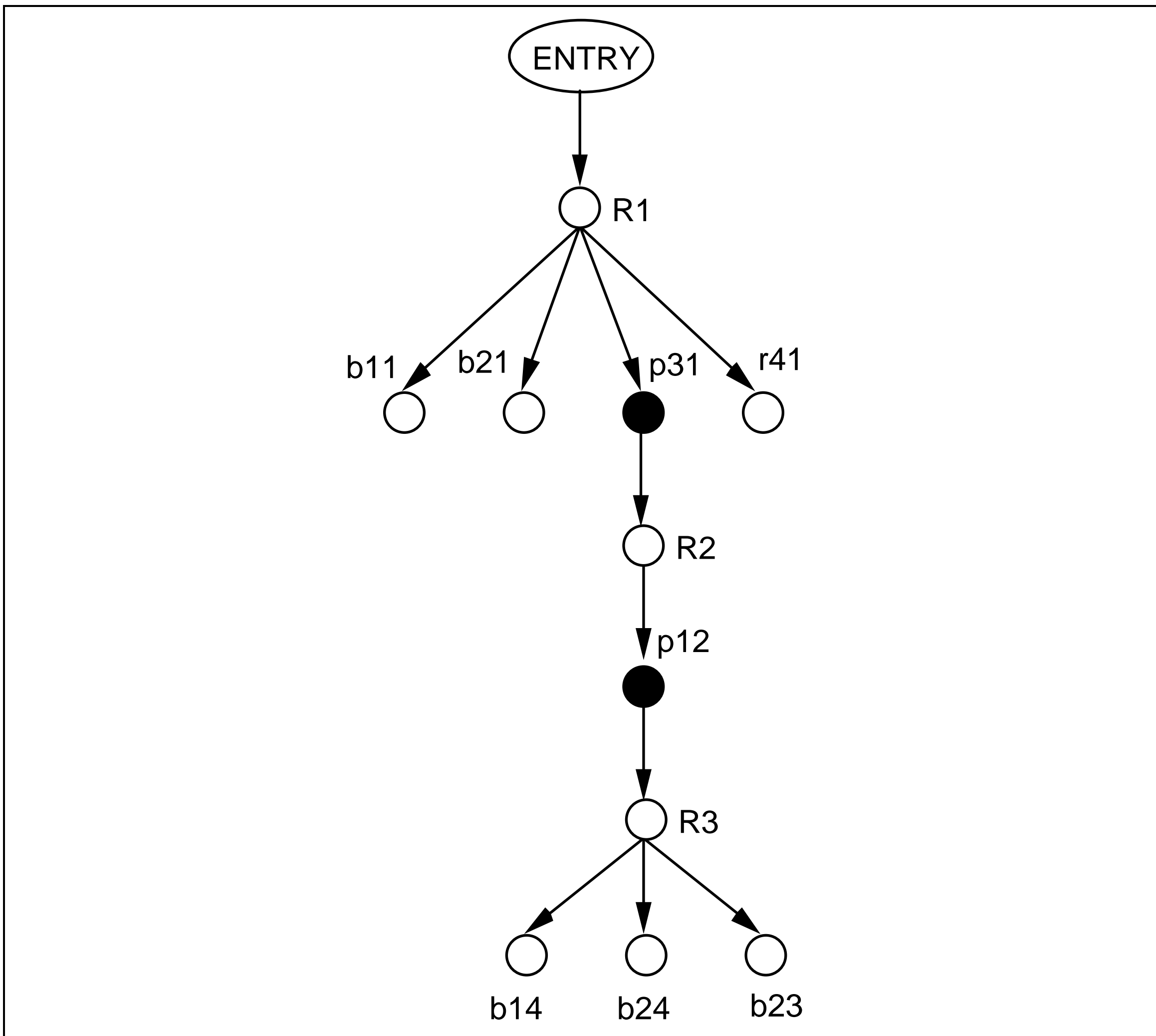

Fig. 4 - The control dependencies subgraph of Proc1

3.1.2 - Data dependencies

The construction of the data dependence subgraph whose nodes consist of statements and predicates have to face further problems caused by side-effects due to pointers, shared variables, or procedure calls with other than value parameters.
Directed acyclic graph (DAG) ([Aho77]) are constructed for each basic block during the initial parse of source program. Leaf node is labelled by unique identifiers, either variable names or constants which are initially assigned the value "undefined" at program entry. During computing the set of reaching definitions for each basic block, interior nodes are labelled by an operator symbol and are given an extra set of identifiers for labels.

Finally, the individual DAGs are connected to one another using the results of the data flow computation to make the definition-use chain explicit .

In this process, I/O operations are treated as operations on implicit file object so that the sequencing of operations is correctly represented.

In addition, the definite iteration statement (for) becomes a single operator whose operands are the initial, final and increment values and with two output values that are the index value stream and the predicate value stream.

A major problem in static flow analysis is that modification and reference to the elements of an array or elements specified by pointer are considered as references to the while object.

To face this problem, some "dummy" variables for each pointer variable are introduced; for example, dummy variables (1)p, (2)p will be introduced for pointer variable **p in C.

Annotation (i) represents the number of levels of indirect access through pointer variables (Fig. 5)

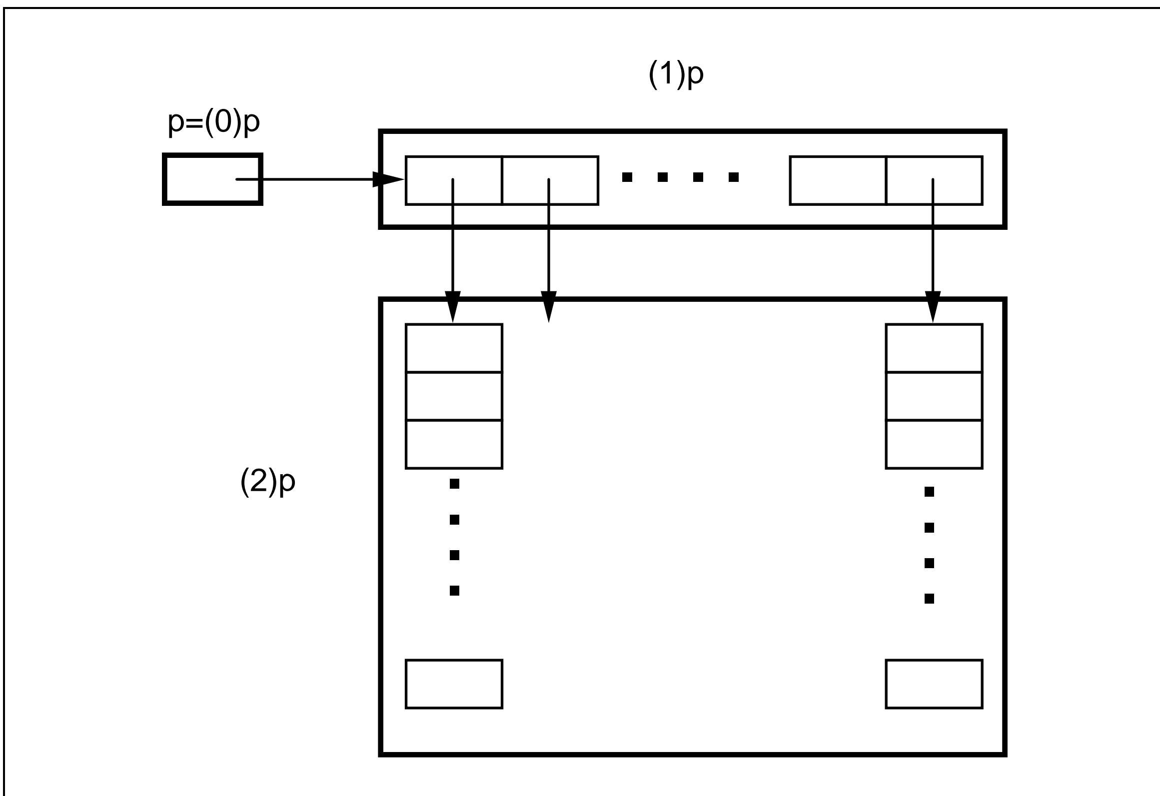

Fig. 5 - The "dummy" variables

To correctly propagate aliasing information based on assignment of the address of a variable to another variable, a dummy literal for each variable whose address is copied is introduced. If p=&x then this literal will be denoted by (-1)x.
The value of a dummy variable (i)p is either modified or used by a statement , whereas the values of pointer variable p and dummy variables (1)p, (2)p, ..., (i-1)p will always be used.
Aliasing and side-effects present obvious problems in accurately representing dependencies in the PDG.
Explicit aliasing of scalars is easily handled by treating aliases as synonyms implicit aliasing, induced by procedure parameter binding is detect using interprocedural data flow analysis ([Cooper85], [Wehil80]). Obviously interprocedural analysis must be performed before building the basic block DAGs.

3.1.3 - Slicing

The extraction of slices is based on data dependence even if control dependence is considered in the construction of slice as well.
A slice is directly obtained by a linear time walk backwards from some point in the graph visiting all predecessors.
Nodes must be annotated with references to the source code in order to permit the identification of the resulting slice.

4 - Conclusions

Object orientation is claimed to be the most suitable method that can be used to produce software systems which are robust, reliable and reusable. However,
On the other hand, the existing software patrimony and the related investments are so relevant that it goes without doubt that we have to recover as much as possible of the effort put in the development of the old software systems.
As a consequence, object oriented re-engineering appears a promising research area.
The main difficulty we are faced with when re-engineering old software towards an object oriented environment, is to understand the semantics of the existing code, so that it will be possible to identify objects, methods and inheritance hierarchies.

In this paper, we have presented a general framework for the implementation of a re-engineering cycle. Its main characteristics are the identification of “candidate objects” from the global data structures and of the methods via program slicing and clustering by the global data which are manipulated.
A human intervention is supposed to take place to solve ambiguous or non automatically decidible cases.

Aknowledgements

We have to thank A. Cimitile and U. De Carlini from University of Naples for useful discussions and suggestions.
We have also to thenk M. Gregori who participated in the early stages of this work.

References

[Aho77] Aho A., Ullman J.D.: *Principles of Compiler Design*, Addison-Wesley (1977)

[Alabiso88] Alabiso M.: *Transformation of Data Flow Analysis Models to Object Oriented Design*, Proceedings of OOPSLA’ 88, September 25-30 1988

[Chikofsky90] Chikofsky E.J., Cross II J.H.: *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software (January 1990)

[Cimitile91] Cimitile A., De Carlini U.: *Reverse Engineering: Algorithms for Program Graph Production*, Software Practice and Experience, vol. 21, n. 5 (May 1991)

[Cooper85] Cooper K.: *Analyzing aliases of reference formal parameters*, Conf. Record of 12th ACM Symposium on Principles of Programming Languages, New Orleans, Jannuary 14-16 1985

[Edwards90] Edwards J.M., Henderson-Sellers B.: *The Object-Oriented Systems Life Cycle*, ACM Communications, vol. 33, n. 9 (September 1990)

[Ferrante87] Ferrante J., Ottenstein K.J., Warren J.D.: *The Program Dependence Graph and its Use in Optimization*, ACM Transactions on programming Languages and Systems, vol. 9, n. 3 (July 1987)

[Hausler90] Hausler P.A., Pleazkoch M.G., Linger C.R., Hevner A.R.: *Using Function Abstraction to Understand Program Behaviour*, IEEE Software (January 1990)

[Jacobson91] Jacobson I., Lindström F.: *Re-engineering of old systems to an object-oriented architecture*, Proceedings of OOPSLA' 91

[Lengauer79] Lengauer T., Tarjan R.: *A Fast Algorithm for Finding Dominators in a Flowgraph*, ACM Transactions on programming Languages and Systems, vol. 1, n. 1 (July 1979)

[Liu90] Liu S-S., Wilde N.: *Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery*, IEEE Proceedings of IEEE Conference on Software Maintenance, San Diego, Nov. 26-29, 1990

[Meyer91] Meyer B.: *La produzione del software object oriented*, Ed. Jackson and Prentice Hall International (1991)

[Ott89] Ott L.M., Thuss J.J.: *The Relationship between Slices and Modul Cohesion*, Proceedings of 11th Int. Conference on Software Eng., May 15-18 1989 Pittsburgh

[Parnas72] Parnas D.L.: *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of ACM, Vol.15, N.12 (December 1972)

[Jiang91] Jiang J., Zhou X., Robson D.J.: *Program Slicing For C - The Problems In Implementation*, Proceedings of IEEE Conf. on Software Maintenance, Sorrento (Italy) 1991

[Signore92] Signore O., Loffredo M.: *Reverse, re-engineering e riuso: tre discipline connesse alla manutenzione*, Technical Report, Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo (in press)

[SYSTECH90] *Software Re-engineering Symposium*, Technical seminar organized by SYSTECH Systems Technology Institute (Roma, 12-14 February 1990)

[Wegman83] Wegman M.: *Summarizing Graphs by Regular Expressions*, Proc. of 10th Annual ACM Symposium of Programming Languages, Austin Texas, Jan. 24-26, 1983 pp.203-216

[Wehil80] Wehil W.E.: *Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables*, Conf. Record of

the 7th ACM Symposium on Principles of Programming Languages (January 1980)

[Weiser84] Weiser M.D.: *Program Slicing*, IEEE Transactions on Software Engineering, vol. SE-10, n. 4 (July 1984)

[Wills90] Wills L.M.: *Automated Program Recognition: A feseability Demonstration*, Artificial Intelligence, n. 45 (1990)

Appendix A: Grammar for the code "expressions"

Terminal symbols: $\{p_j^i, c_j^i, b_j^i, l_j^i, s_j^i, (,), /, ^*, +, \bullet, \phi_j^i\}$

Non terminal symbols: $\{SEQ_j^i, IF_j^i, IFT_j^i, CA(n)_j^i, W_j^i, R_j^i, F_j^i\}$

Initial symbol: $\{SEQ_j^i\}$

Productions:

Type of statement: $X_j^i = SEQ_j^i \mid IF_j^i \mid IFT_j^i \mid CA(n)_j^i \mid W_j^i \mid R_j^i \mid B_j^i \mid \phi_j^i$

Sequence of n statements: $SEQ_j^i = X_{j1}^{i+1}/......../X_{jn}^{i+1}$

If-then: $IF_j^i = p_j^i \bullet (X_{j1}^{i+1}+\phi_{j2}^{i+1})$

If-then-else: $IFT_j^i = p_j^i \bullet (X_{j1}^{i+1}+X_{j2}^{i+1})$

N-ways CASE: $CA(n)_j^i =$

$p_j^i \bullet (X_{j1}^{i+1}+CA(n-1)_{j2}^{i+1})$ if n>2

$p_j^i \bullet (X_{j1}^{i+1}+X_{j2}^{i+1})$ if n=2

While: $W_j^i = p_j^i \bullet ((X_{j1}^{i+1}/p_j^i)^*+\phi_{j2}^{i+1})$

Repeat: $R_j^i = X_{j1}^{i+1}/p_j^i \bullet ((X_{j1}^{i+1}/p_j^i)^*+\phi_{j2}^{i+1})$

For: $F_j^i = p_j^i \bullet ((SEQ_{j1}^i/b_{jn+1}^{i+1}/p_j^i)^*+\phi_{n+2}^{i+1})$ with $n = ||SEQ_{j1}^i||$

Simple statement: $B_j^i = c_j^i \mid b_j^i \mid l_j^i \mid s_j^i \mid r_j^i$

Predicate: p_j^i

Function call: c_j^i

Assignement: b_j^i

Read: l_j^i

Write: s_j^i

Return from function: r_j^i

Appendix B: An example

```
int Proc1(string, file_name)
char *string;
char *file_name;

   {
   int i;
   FILE *file;
   char buffer[80];

   file=fopen(file_name,"r");
   i=0;
   read(file, buffer,80);
   while (!eof(file))
   if (strstr(buffer,string)<>NULL)
      {
      i++;
      break;
      }
   else
      {
      read(file, buffer,80);
      i++;
      }
   return(i);
}
```

Proc1= Receives a string and a file name. Search for the first occurrence of the string in the file and returns the number of the row containing the string, otherwise returns "False".

```
int Proc2(d0, list)
double d0;
listdouble *list;

   {
   listdouble *scan;
   double d1, d2;
   int k;

   k=0;
   scan=list;
   scanf("%d %d", &d1, &d2);
   if (d1<d0)
      while (scan<>NULL && k==0)
         if (scan->doublenum==d1)
            {
            k=1;
            scan->doublenum=d2;
            }
         else
            scan=scan->next;
   else
      k=0;
```

```
   return(k);
}
```

Proc2= Receives a double d_0 and a pointer to a list of double elements. Reads two doubles: d_1 and d_2. If $d_1 < d_2$ then if d_1 is contained in the list, it is replaced by d_2, otherwise returns “False”.

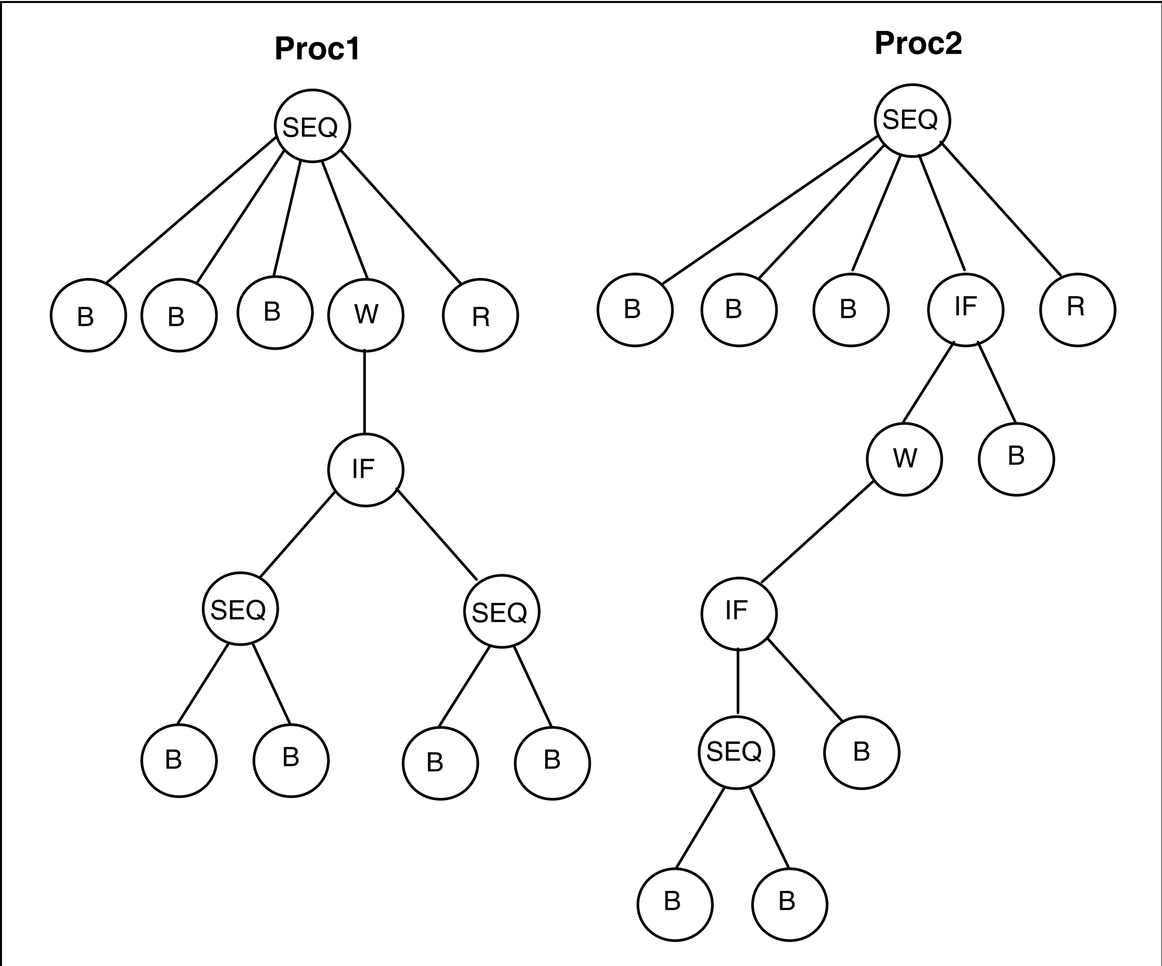

Fig. 6 - Nesting trees of the procedures Proc1 and Proc2

If we consider the nesting trees as parsing trees for the expressions, we can obtain the regular expressions corresponding to the procedures by applying repeatedly the grammar rules.

$F^0(\text{Proc1})=SEQ_1{}^0$

$F^1(\text{Proc1})=X_1{}^1/ X_2{}^1 / X_3{}^1 /X_4{}^1/ X_5{}^1$

$F^1(\text{Proc1})=B_1{}^1/ B_2{}^1/ B_3{}^1 / W_4{}^1 / R_5{}^1$

$F^2(\text{Proc1})=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet ((X_1{}^2/p_4{}^1)^*+\phi_2{}^2)/ r_5{}^1$

$F^2(\text{Proc1})=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet ((IFT_1{}^2/p_4{}^1)^*+\phi_2{}^2)/ r_5{}^1$

$F^3(Proc1)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet ((p_1{}^2 \bullet (X_1{}^3+X_2{}^3)/p_4{}^1)^*+\phi_2{}^2)/ r_5{}^1$

$F^3(Proc1)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet ((p_1{}^2 \bullet (SEQ_1{}^3+SEQ_2{}^3)/p_4{}^1)^*+\phi_2{}^2)/ r_5{}^1$

$F^4(Proc1)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet ((p_1{}^2 \bullet (X_1{}^4 / X_2{}^4+X_3{}^4 / X_4{}^4)/p_4{}^1)^*+\phi_2{}^2)/ r_5{}^1$

$F^4(Proc1)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet ((p_1{}^2 \bullet (B_1{}^4 / B_2{}^4+B_3{}^4 / B_4{}^4)/p_4{}^1)^*+\phi_2{}^2)/ r_5{}^1$

$F^4(Proc1)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet ((p_1{}^2 \bullet (b_1{}^4 / b_2{}^4+l_3{}^4 / b_4{}^4)/p_4{}^1)^*+\phi_2{}^2)/ r_5{}^1$

$$F(Proc1)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet ((p_1{}^2 \bullet (b_1{}^4 / b_2{}^4+l_3{}^4 / b_4{}^4)/p_4{}^1)^*+\phi_2{}^2)/ r_5{}^1$$

$F^0(Proc2)=SEQ_1{}^0$

$F^1(Proc2)=X_1{}^1/ X_2{}^1 / X_3{}^1 / X_4{}^1 / X_5{}^1$

$F^1(Proc2)=B_1{}^1/ B_2{}^1 / B_3{}^1 / IF_4{}^1 / R_5{}^1$

$F^2(Proc2)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet (X_1{}^2+X_2{}^2)/ r_5{}^1$

$F^2(Proc2)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet (W_1{}^2+b_2{}^2)/ r_5{}^1$

$F^3(Proc2)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet (p_1{}^2 \bullet ((X_1{}^3/p_1{}^2)^*+\phi_2{}^3) +b_2{}^2)/r_5{}^1$

$F^3(Proc2)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet (p_1{}^2 \bullet ((IFT_1{}^3/p_1{}^2)^*+\phi_2{}^3) +b_2{}^2)/r_5{}^1$

$F^4(Proc2)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet (p_1{}^2 \bullet ((p_1{}^3 \bullet (X_1{}^4+X_2{}^4)/p_1{}^2)^*+\phi_2{}^3) +b_2{}^2)/r_5{}^1$

$F^4(Proc2)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet (p_1{}^2 \bullet ((p_1{}^3 \bullet (SEQ_1{}^4+B_2{}^4)/p_1{}^2)^*+\phi_2{}^3) +b_2{}^2)/r_5{}^1$

$F^4(Proc2)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet (p_1{}^2 \bullet ((p_1{}^3 \bullet (SEQ_1{}^4+b_2{}^4)/p_1{}^2)^*+\phi_2{}^3) +b_2{}^2)/r_5{}^1$

$F^5(Proc2)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet (p_1{}^2 \bullet ((p_1{}^3 \bullet (X_1{}^5 / X_2{}^5+b_2{}^4)/p_1{}^2)^*+\phi_2{}^3) +b_2{}^2)/r_5{}^1$

$F^5(Proc2)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet (p_1{}^2 \bullet ((p_1{}^3 \bullet (B_1{}^5 / B_2{}^5+b_2{}^4)/p_1{}^2)^*+\phi_2{}^3) +b_2{}^2)/r_5{}^1$

$F^5(Proc2)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet (p_1{}^2 \bullet ((p_1{}^3 \bullet (b_1{}^5 / b_2{}^5+b_2{}^4)/p_1{}^2)^*+\phi_2{}^3) +b_2{}^2)/r_5{}^1$

$$F(Proc2)=b_1{}^1/ b_2{}^1 / l_3{}^1 / p_4{}^1 \bullet (p_1{}^2 \bullet ((p_1{}^3 \bullet (b_1{}^5 / b_2{}^5+b_2{}^4)/p_1{}^2)^*+\phi_2{}^3) +b_2{}^2)/r_5{}^1$$

If we consider only the terms, we can find that the expressions have some identical portions; the most relevant substrings are:

for F(Proc1) $p_4{}^1 \bullet ((IFT_1{}^2/p_4{}^1)^*+\phi_2{}^2)$

for F(Proc2) $p_1^2 \bullet ((IFT_1^3/p_1^2)^*+\phi_2^3)$

They corresponds to sequential structure scanning loop until a certain condition becomes true.

Furthermore, this representation form allow the identification of structural equivalent code portions at both abstract and detailed level.