An Evolutional Knowledge-Based Framework for Reverse and Forward Engineering

Stefan Trausan-Matu

Department of Computer Science
"Politehnica" University of Bucharest
313, Splaiul Independentei
77206 Bucharest
ROMANIA

Abstract: The paper describes a knowledge-based framework for the development of program reverse and forward engineering. The substratum for this framework is an object-oriented environment for knowledge-based applications (XRL) written in Common Lisp.
The approach starts from the idea that forward and reverse engineering are evolutive, knowledge intensive activities. Regarding the development of knowledge-based systems, knowledge acquisition is considered as a modelling activity that implies not only the evolution of the knowledge base of the system but also the evolution of the mental model that the human experts actually use. Therefore, an integrated collection of tools and techniques that support the construction, evolution, and usage of knowledge-based models of programming concepts and constructs, is provided. An evolutive taxonomy of re-usable program components related to programming concepts is also viewed as essential to the approach.
One of the main applications written in the knowledge-based framework is the development of an intelligent reverse engineering system of FORTRAN programs. After obtaining a high level description of an analysed program, this description may be refined to C or pseudocode.

Stefan Trausan-Matu received the engineer degree in Computers from the Polytechnical Institute of Bucharest in 1983. In 1994 he took his Ph D degree from the same higher education institution. Between 1983 and 1985 he worked in the field of CAD of integrated circuits at "Microelectronica" Factory in Bucharest. Since 1985 he has worked in the domain of Knowledge Representation and Processing at the Research Institute for Informatics in Bucharest. He has been head of Expert Systems Laboratory at the institute. Last October he left the institute and took an academic position at the "POLITEHNICA" University of Bucharest. He teaches courses on Artificial Intelligence, Data Structures and Algorithms, Advanced Programming Languages. His research interests are knowledge representation, constraint processing, object-oriented systems, expert systems, and artificial intelligence applications in software engineering.

1. Introduction

Reverse engineering systems are following backward the path normally taken by software development methodologies (in forward engineering systems). The goal of such systems is obtaining a high level description (a specification) of a program written in a usual language (e.g. FORTRAN or COBOL). This high level description may be used towards attaining several goals as re-writing the program in other language, maintenance, validation or restructuring of the program.

Usually, reverse engineering systems start with an analysis similar to that of compiler front-ends. The result of this first phase is an intermediate form (a simpler language and/or the control flow and data flow graphs) which is the starting point for a second, abstraction phase. This second phase is usually completed by a set of abstraction transformations of the intermediate form towards obtaining a higher level description.

Intelligent, knowledge-based reverse engineering systems extend the capabilities of automatic program abstraction by the usage of general software engineering and specific problem domain knowledge (a good discussion about the usage of knowledge-based techniques in software engineering is [2]). Such knowledge may suggest the suitability of a particular abstraction, dramatically reducing the amount of search through the space of possible abstractions. A knowledge-based approach is motivated by the complexity and by the knowledge intensive character of program understanding. One goal of such systems is obtaining at least similar results as human experts' in program understanding.

A remarkable characteristic of knowledge-based reverse engineering systems is their potential for evolution. One of the main starting points of this approach is the fact that knowledge-based reverse engineering systems may be improved after each run. The improvement results from knowledge

acquisition or restructuring (e.g. the acquisition of new heuristics about selecting a particular abstraction). The indication that some knowledge must be acquired or/and previous knowledge must be restructured comes after the validation of the results obtained by the system vs. the understanding of some human expert regarding a sample program. A very important remark here is that the knowledge acquisition process may extend and/or restructure not only the system's knowledge base but also the knowledge of the human expert. Considering knowledge acquisition as a modelling activity, yields such a remark [3,7,21].

This paper presents a knowledge-based framework for developing program understanding and refinement applications. This framework has been developed in the XRL object-oriented environment for knowledge-based applications [4,5,7,19]. The main concern of the framework is the provision of tools for the representation and processing of the knowledge categories involved in program transformations. It considers three categories of knowledge: a taxonomy of programming concepts, a collection of abstraction and refinement transformations, and several rule sets that include program understanding heuristics. One of the main applications written in this framework is an intelligent reverse engineering system for understanding and re-writing FORTRAN (IV) programs in C.

The next section of the paper introduces the XRL environment. The knowledge-based framework for program understanding and refinement is presented in the third section. The knowledge-based reverse engineering system for FORTRAN programs is described in the fourth section.

2. The XRL Object-Oriented Environment for Knowledge-Based Applications

Object-oriented programming (OOP) is now largely looked upon as a powerful programming paradigm, able to cope with software change and re-use. OOP provides an anthropomorphic metaphor (the object) that is more cognitively ergonomical for complex systems decomposition. An object has a number of components (slots) and can respond in a specific way to a number of messages. An inheritance relation is defined among objects: When an object A inherits from another object B, it can be considered that A has all the components of B (unless they are redefined in A).

Knowledge-based systems explicitly represent and process knowledge from the application domain. The most powerful knowledge-based programming environments (e.g. KEE [12]) integrate several knowledge representation paradigms in order to cope with the various knowledge types of an application. Such environments are usually written in Lisp around an OOP language. In addition to common OOP characteristics, the kernel language offers some facilities for the so-called frame knowledge representation paradigm [12]. As a consequence, complex structuring of objects is usual, a great emphasis is placed on multiple inheritance with method combination, and other AI knowledge representation and control mechanisms are integrated: rules, logic programming, demons, and constraints.

XRL is an object-oriented environment for knowledge-based applications, that integrates structured objects, production rules, and constraint representation with several processing mechanisms [4,5]. The environment has been developed in Lisp and has the layered architecture depicted in Figure 1.

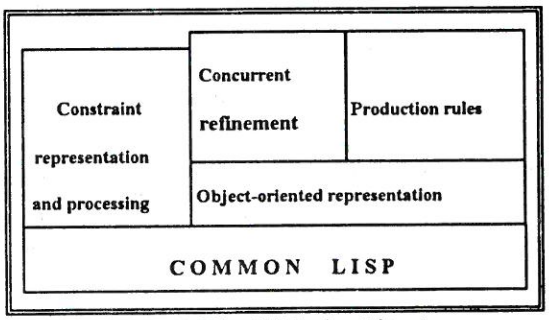

Figure 1. The XRL Architecture

XRL is a so-called prototype-based object-oriented language. In such languages there is no separation among class and instance objects. An individual (the equivalent of an instance) object is created by a clone (copy) operation of its

prototype. This individual object may be a prototype to another individual object a.s.o. Therefore, several (dynamic) successive refinements of an object may be performed. This is not the case with class-instance OOP languages that do not allow an instance to be further refined to another instance.

An XRL object is defined with the "unit" construct. The object's meta-description is declared in the self slot. The supers sub-slot of the meta-description declares the list of objects which the current object may inherit slots and methods from. The assignment of methods to messages for the object is described as selector-method pairs in the same self slot. Slots may be meta-described in a similar way to objects (see the "AddPositive" object described at the end of this section).

Below, four XRL objects are presented. They are used in the reverse engineering system described in the fourth section. The "selfAssign" object inherits properties from "assign" and is the source of inheritance for "inc". All of these objects have a specific attached method (describing their own behaviour) for the "write_in_C" message. "inc32" is a particular object that is a clone of the generic "inc" object.

```
(unit assign
 self (a statement
        write_in_C Cassign)
 variable undf
 expression undf)

(unit selfAssign
 self (a statement supers (assign)
        write_in_C Cselfassign))

(unit inc
 self (a statement supers (selfassign)
        write_in_C Cinc)
 expression 1)
(unit inc32 (a inc variable x1))
```

C code for the "inc32" statement is obtained by sending it the "write_to_C" message: (msg 'write_to_C 'inc32). This message is processed by the "Cinc" associated method:

```
(defmethod Cinc (selector_name target_object)
... Lisp code for printing the inc statement)
```

The description of a program in the reverse engineering system presented in this paper consists in a network of XRL objects. Objects in this network contain explicit links (slots filled with other object/s) for control flow, for data flow, for object decomposition, and for abstraction and refinement to other objects.

Production rules are the most popular knowledge representation techniques. They are very suitable to represent heuristic knowledge as situation-action pairs [8,10]. XRL rules are implemented as objects. This implies that rules may be related in inheritance hierarchies similar to concept taxonomies. For example, the set of rules that detects which are the usages of an array in a program, has the following structure:

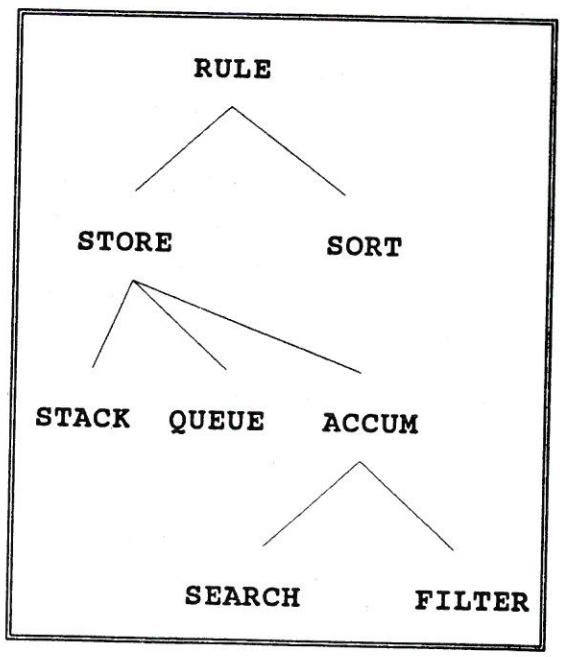

Figure 2. A Rule Taxonomy

An example of an XRL rule is:

```
(unit store
 self (a rule_behavior)
 arguments (array)
 if (notInterchanges singleIndex dynamics)
 notInterchanges (not (fslot 'interchanges array))
 singleIndex (= (fslot 'nr_of_indexes array) 1)
  dynamics (member (fslot 'dynamics (eval (car
(fslot 'indexes array))))
                              (onlyincrementation
onlydecrementation))
 then ((pslot 'usage array (cons 'store (fslot 'usage
array)))))
```

The integration of constraint processing facilities in an OOP system enhances the power of representation and processing. Relations among objects or among components of an object, transformations of the state of objects, can be very elegantly described using constraints. Constraint-based programming may be considered as a new programming paradigm that is useful in a wide area of applications. In artificial intelligence, constraints attracted great interest due to their very suitable representation for describing complex problems involving search. Constraints are relations between variables named cells of the constraint. Each variable may have values in a particular set, finite or not. Constraint networks are created by sharing cells of a set of constraints. Constraint processing consists in the assignment and/or change of values in cells so that, finally, all constraints are satisfied [20].

An example of a generically, parametrically, and hierarchically defined constraint in XRL, for a series connection of N electrical resistors, is:

```
(:DeclareConstraint NSeriesResistors
  :cells (u1 u2 i1 i2 r)
  :GroupsOfCells ((t1 (u1 i1)) (t2 (u2 i2)))
  :parameters (n)
  :InnerConstraints ((sum :isa (n_adder :n n))
                     (res :isa (:set n :of resistor
:atleast 2)))
  :connections ((:map 'r :to 's :of sum)
                (:group-map 't1 :to 't1 :of (res 1))
                (:group-map 't2 :to 't2 :of (res n))
                (do ((i 1 (1- n))) ((= i n))
                    (:group-connect 't2 :of (res i)
:to 't1 :of (res (1+ i)))
                (:connect 'r :of (res i) :to (term
i) :of 'sum))
                (:connect 'r :of (res n) :to (term
n) :of 'sum)))
```

The concurrent refinement module [6] offers a blackboard-based [8] problem-solving architecture for instantiating (considered as refinement) XRL objects. Associated with this architecture, a refinement language is defined. For example, the generic object description below is a declarative specification of a procedure for the addition of the positive elements in a collection. The same description also contains annotations for the instantiation of the object, as a whole, and of its components. For example, the object "AddPositive" and the "description" slots will be refined by "expand"-ation. The "inputs" slot will be "anchor"-ed to an existing description, and the "outputs" slot will be left "as-it-is".

```
(unit AddPositive
   self (a ProgrDescr modeR 'expand)
   inputs (a Collection elem-type Real)
      (a Slot ModeR 'anchor
         to-anchor ExecChooseCol)
   outputs (a RealVar)
         (a slot ModeR 'as-it-is)
   description (a DataFlowSequence
                  description
                        (allof
                           (a generate gen (the
iterator from inputs))
                           (a TestIfPositive)
                           (a SelfAdd))
                     (a Slot ModeR 'expand)
                  start (the inputs)
```

finish (the outputs))
(a slot ModeR 'expand))

The concurrent refinement and constraint processing modules were not used in the first version of the reverse engineering system described in the fourth section. Nevertheless, they will be used in the next version due to their potential for describing complex problem- solving regimes (for example, in the co-operation of multiple knowledge sources in program understanding). The concurrent refinement system has been used for a program specification and refinement application [21].

3. A Knowledge-Based Framework for Program Understanding and Refinement

The XRL environment has been the substratum for developing a complex framework for forward and reverse engineering of programs represented as object networks. This framework has three parts that correspond to three kinds of knowledge involved in software engineering:

1. A taxonomy of programming concepts and constructs;
2. An abstraction and refinement mechanism based on a programmed graph grammar formalism;
3. Mini expert systems containing heuristics for program understanding and refinement.

3.1 The Taxonomy of Program Concepts and Constructs

The taxonomy of programming concepts and constructs is organized as an extensible hierarchy of XRL objects along the inheritance relation. The objects in the taxonomy are prototypes for all the individual objects used for program descriptions at various levels of abstraction. Each object is at the same time a construct that may be used in a program description and it also defines a programming concept.

In the taxonomy there are three main groups of objects corresponding to data, control, and procedural abstractions. The data abstraction objects are generic descriptions of data structuring as container, table, set, stack, a.s.o. Control abstraction objects include control structures and various kinds of statements or high level processings on data objects. Procedural abstraction objects describe classes of procedures.

A very important facility of the framework is the possibility of extending the taxonomy by the inclusion of new objects. For example, for a particular application, the domain specific objects may be included in this taxonomy. These specific objects may, in some cases, extend even the general programming taxonomy of objects by introducing new programming concepts or by restricting the existing taxonomy (as discussed in [21]). This is one of the main ideas of the approach presented in this paper. In fact, the taxonomy of programming concepts is a knowledge base that may be considered as a part of a programming theory. This theory evolves as a result of developing new applications (for understanding and refinement of programs in various domains) that need the addition of new objects and/or the revision of the existing ones. New concepts may be suggested after a training process that consists in automatically understanding a program with the reverse engineering system, comparing this understanding with the one of a human (eventually the author of the program), and finding the missing concepts that determine the existing differences.

Figure 3 presents a part of the taxonomy of programming concepts and constructs that are now in the framework.

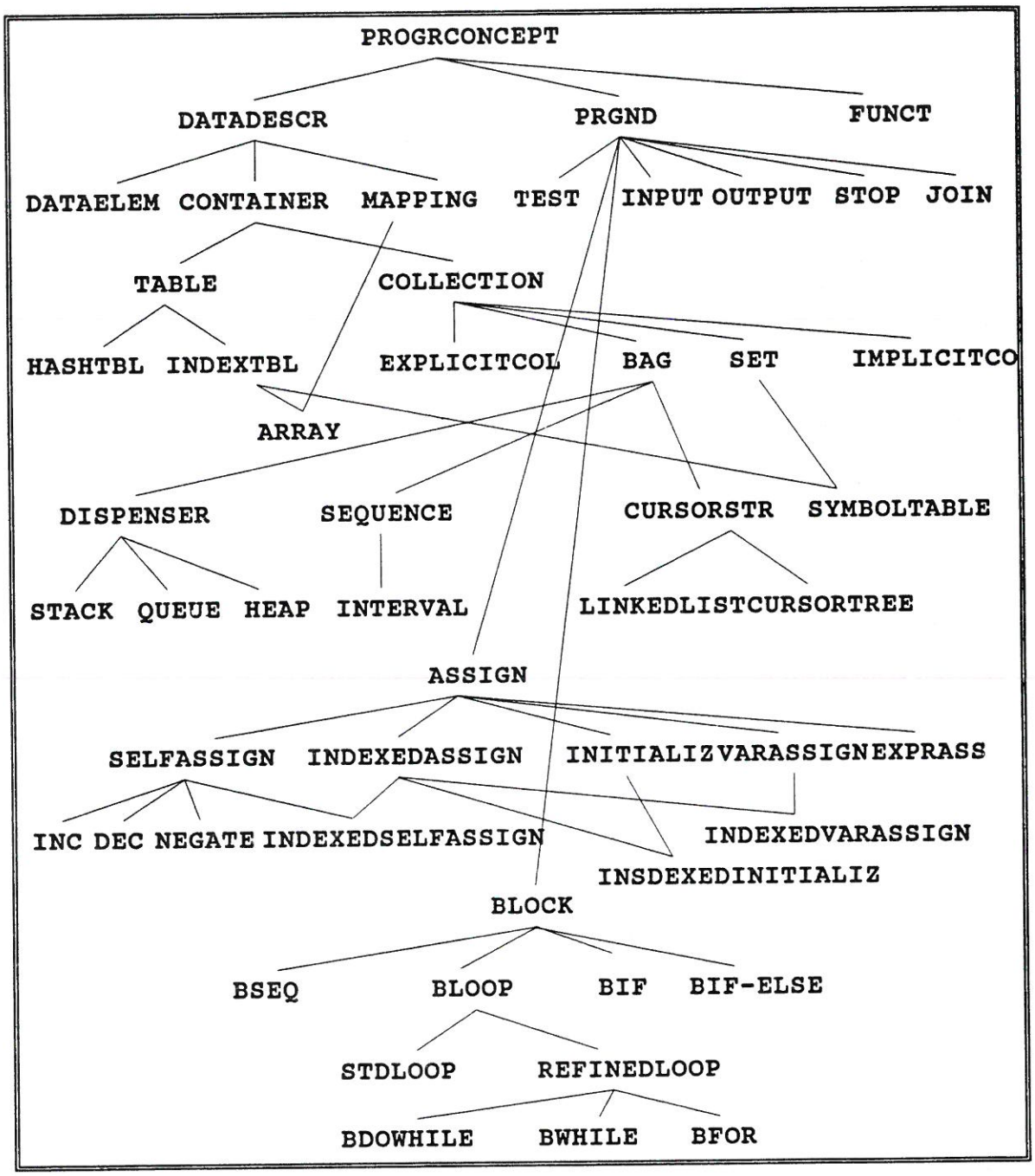

Figure 3. A Taxonomy of Programming Concepts and Constructs

3.2 Abstraction and Refinement Transformations

The description of a program in the framework presented here consists of a network of objects that are clones of generic objects from the taxonomy above discussed. Program abstraction and refinement are obtained by transformations of the network of objects. For the description and manipulation of such transformations, a specialized module has been developed. In this

module, each possible transformation is represented as a production in a programmed graph grammar [11]. In Figure 4, an example of a production that transforms a four-node subgraph into a new node is presented. In this example, T is the so-called "embedding" transformation of arcs [11] and P is the production applicability predicate.

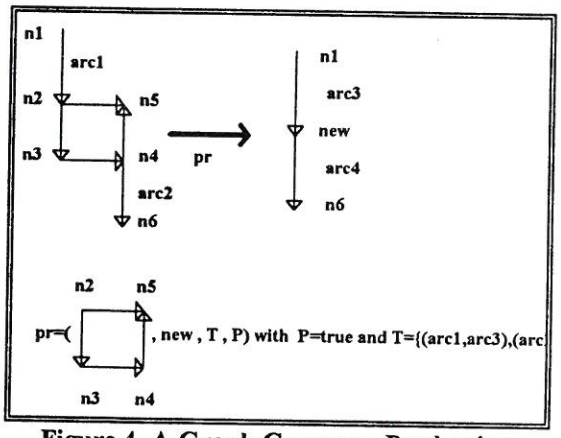

Figure 4. A Graph Grammar Production

3.3 Mini Expert Systems Containing Heuristics for Program Understanding and Refinement

The concept taxonomy reflects the structuring of the programming domain. The graph grammar transformation is a formalism for the description of knowledge about abstraction and refinement among groups of concepts. In addition to these two knowledge categories, heuristic knowledge is usually involved in program understanding and refinement. For example, if in a procedure the components of an array are updated only by interchanging them in a loop, the procedure might be a sorting one. Such heuristic knowledge is commonly used by humans when understanding a program. For the representation and processing of such knowledge, a production rule paradigm [10] is adequate.

As discussed in Section 2, XRL rules are represented and organized as objects. This gives the possibility of a hierarchical organization of rules. Another advantage of the object implementation is the uniformity of representation that facilitates the integration of heuristic knowledge into taxonomically organized concepts. Several related rules are grouped in a so-called rule set. A rule set is an object that may be viewed and used as a mini expert system that provides information about some possible abstractions or refinements.

4. A Reverse Engineering System for FORTRAN Programs

The reverse engineering system presented in this section is an application that combines the facilities provided by the knowledge-based-framework described in the last section with compilation specific techniques. The goal of the system is abstracting a high level description of a FORTRAN program. The result of program abstraction is the starting point for the generation of a C or pseudocode version of the initial FORTRAN program. The processings made by the system are illustrated in Figure 5. These processings go through three phases: the source program analysis and intermediate code generation, the generation of the network of objects' representation, the abstraction phase, and (if desired) generation of code in another language (e.g. C).

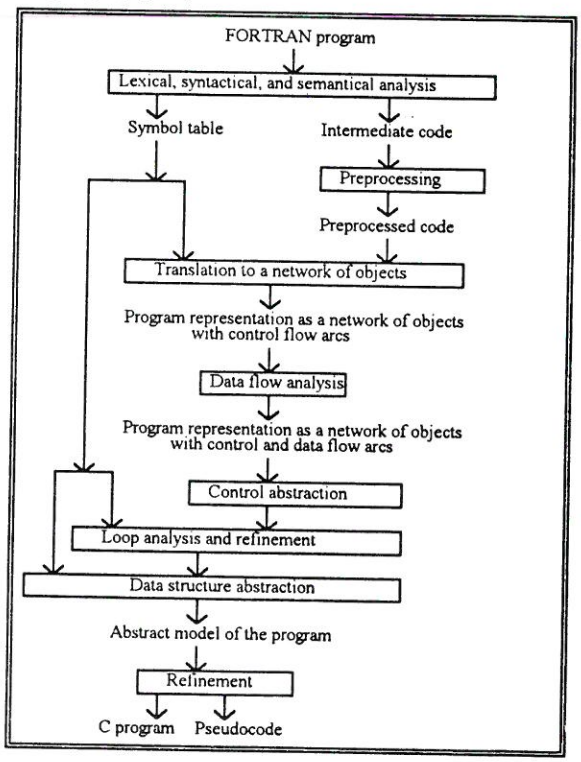

Figure 5. Processings in the Reverse Engineering System

4.1 Source Code Analysis and Intermediate Code Generation

The FORTRAN program is, first of all, lexically, syntactically, and semantically analysed and translated into an intermediate code language. A symbol table is also generated during this phase. The intermediate code is further preprocessed for some simplifications that reduce the number of statements and control structures. For example, a FORTRAN program, its intermediate and preprocessed codes are below listed, emphasizing the statements' correspondence in the three languages. In this example, preprocessing reduces logical assignment, test, and, eventually, a branch statement, to "IF" or "IFNOT" statements. The "LOOP" statement (that corresponds to FORTRAN "DO") in the intermediate language is translated into assignments, tests, and branches.

FORTRAN program	Intermediate code	Preprocessed code
I=1	(EXPR %%VAR0 1)	(EXPR %%VAR0 1)
S=0	(EXPR %%VAR1 0)	(EXPR %%VAR1 0)
2 IF(I.GT.10)GOTO1	2	2
	(LEXPR %%VAR2 %%VAR0 ***GT*** 10)	(IF (%%VAR0 ***GT*** 10) 1)
	(IFNOT %%VAR2 -2)	
	(GOTO 1)	
	-2	
I=I+1	(EXPR %%VAR0 %%VAR0 + 1)	(EXPR %%VAR0 %%VAR0 + 1)
S=S+I*I	(EXPR %%VAR1 %%VAR1 + %%VAR0 * %%VAR0)	(EXPR %%VAR1 %%VAR1 + %%VAR0 * %%VAR0)
WRITE(*,3)I	(OUTPUT %%VAR0)	(OUTPUT %%VAR0)
GOTO2	(GOTO 2)	(GOTO 2)
1 IF(S.GT.10)WRITE(*,3)S	1	1
	(LEXPR %%VAR3 %%VAR1 ***GT***)	
	(IFNOT %%VAR3 -3)	(IFNOT (%%VAR1 ***GT*** 10) -3)
	(OUTPUT %%VAR1)	(OUTPUT %%VAR1)
	-3	3
3 FORMAT(I2)	(INFO 3 FORMAT(I2))	(INFO 3 FORMAT(I2))
DO 4 I=1,10	(LOOP 4 %%VAR0 1 10 0)	(EXPR %%VAR0 1)
		1.1
		(CONTINUE)
4 WRITE(*,3)I	4	4
	(OUTPUT %%VAR0)	(OUTPUT %%VAR0)
		(EXPR %%VAR0 % 1)
		(IFNOT (%%VAR0 ***GT*** 10) 1.1)
STOP	(STOP)	(STOP)
END		

The symbol table obtained after parsing is:

```
((I %%VAR0 INTVAR 0)
 (S %%VAR1 REALVAR 0)
 (%%VAR2 %%VAR2 VAR 0)
 (%%VAR3 %%VAR3 VAR 0))
```

4.2 Generation of the Program Representation as a Network of Objects

The next processing phase is concerned with the creation of objects for each preprocessed code statement. Links that reflect control flow and data flow between the generated objects are also set up.

A classification of statements according to the hierarchy of programming concepts is made when generating objects. For example, "EXPR" assignments in the preprocessed code are translated into specific clones of "INITIALIZ," "SELFASSIGN," "INC," and other descendants of the "ASSIGN" object from the taxonomy of programming constructs. Control flow links are also generated at the moment of each creation of a new object.

A first control abstraction of sequences of statements is also obtained during this phase. As a result of this abstraction, basic blocks [1,15] are generated as clones of the object, that describe the sequence control abstraction. Between objects describing statements and the surrounding basic blocks, a link is maintained.

The network of objects is further enhanced by data flow information computed by an algorithm similar to those used in some compilers [1,15]. The description of the program is now a network of objects (for statements in the preprocessed code and for basic blocks) with links reflecting the control and data flow. This description is the starting point for several types of abstractions.

Preprocessed code	Generated object	Basic block
(EXPR %%VAR0 1)	INITIALIZ-82	BSEQ-850
(EXPR %%VAR1 0)	INITIALIZ-830	
2	JOIN-831	BSEQ-854
(IF (%%VAR0 ***GT*** 10) 1)	IF-832	
(EXPR %%VAR0 %%VAR0 + 1)	INC-833	BSEQ-857
(EXPR %%VAR1 %%VAR1 + %%VAR0 * %%VAR0)	SELFASSIGN-834	
(OUTPUT %%VAR0)	OUTPUT-835	
(GOTO 2)	GOTO-836	
1	JOIN-837	BSEQ-853
(IFNOT (%%VAR1 ***GT*** 10) -3)	IFNOT-838	
(OUTPUT %%VAR1)	OUTPUT-839	BSEQ-856
-3	JOIN-840	BSEQ-852
(INFO 3 format(i2))	INFO-841	
(EXPR %%VAR0 1)	INITIALIZ-842	
1.1	JOIN-843	BSEQ-851
(CONTINUE)	CONTINUE-844	
4	JOIN-845	
(OUTPUT %%VAR0)	OUTPUT-846	
(EXPR %%VAR0 %%VAR0 + 1)	INC-847	
(IFNOT (%%VAR0 ***GT*** 10) 1.1)	IFNOT-848	
(STOP)	STOP-849	BSEQ-855

4.3 Abstractions

All of the abstractions in this phase are obtained by means of the graph grammar parsing module described in the previous section. New abstractions may be included by adding new productions in the grammar. Abstraction processes may be guided by heuristics represented

as XRL production rules. Usually the result of an abstraction is a new object, clone of an object from the taxonomy of programming concepts and constructs.

Due to the unstructured character of FORTRAN programs, control abstractions are the first transformations meant for recognizing the basic control structures (sequence, decision, and loop) in the program described as a network of objects. For each abstraction, a new object, clone of the object describing the abstraction (from the taxonomy of programming constructs) is generated. This new object contains the objects that have already been abstracted, and may be considered for other abstractions.

The control abstraction process is carried out by means of a grammar similar to that in [14]. Figure 6 illustrates the application of an "If" and a "Loop" transformation to the control flow graph of the program discussed in this section.

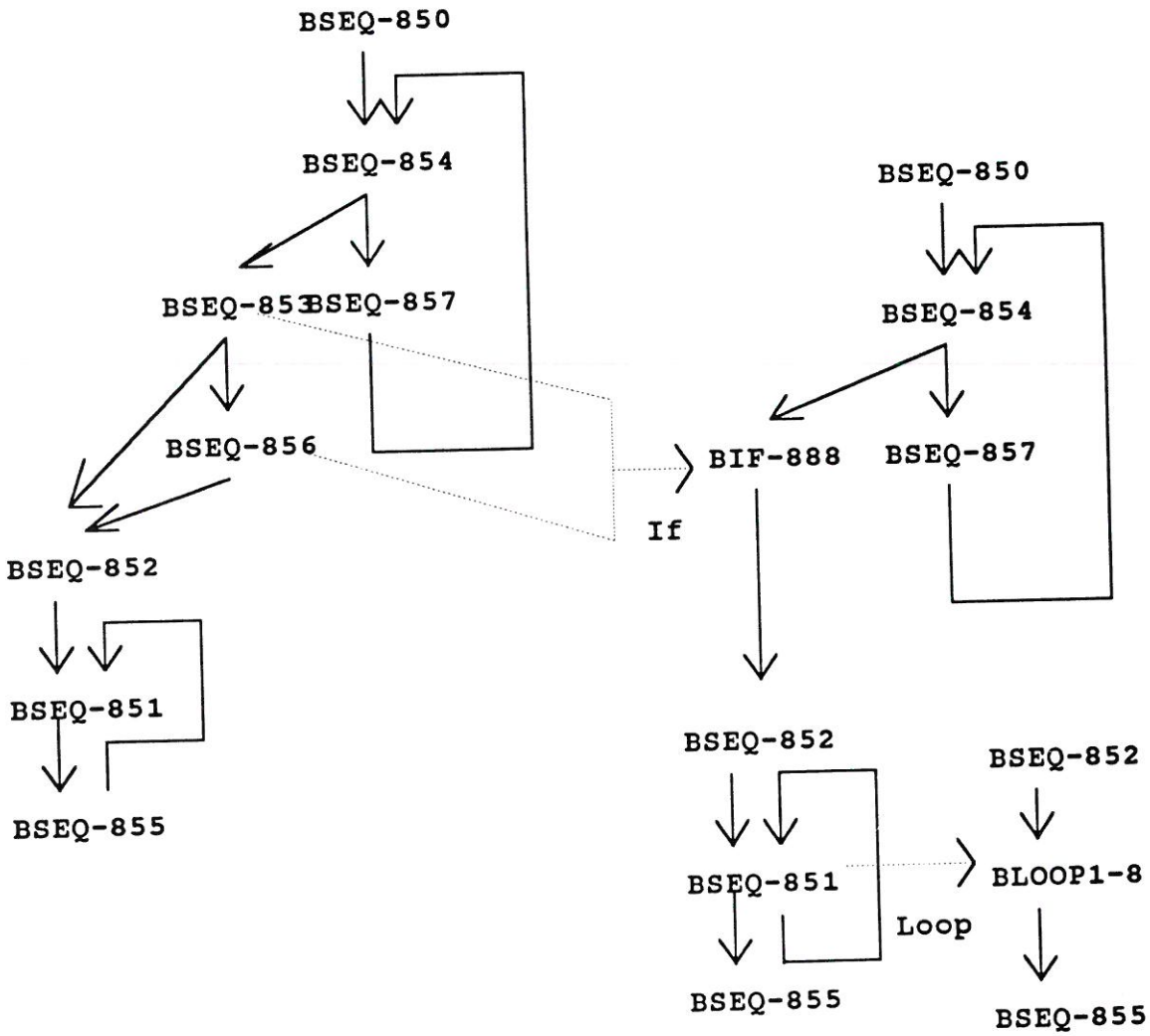

Figure 6. An Example of "If" and "Loop" Control Abstraction Transformations in the Program Above Considered

The loop and data structure abstractions also consist in the application of some specific transformations, represented as productions in the graph grammar. The decision on applying a transformation takes into account information (including, for example, the notion of cobweb in [25]) and heuristics (represented as production rules) about variables in the program. A special emphasis is paid to variables (and especially to induction variables [1]) in loops. Techniques specific to loop optimizations in compilers [1] are combined with loop understanding heuristics for loop abstraction. In the next version of the system, loop analysis will be extended to covering loop understanding presented in [22].

Array analysis is another major concern of the reverse engineering system presented in this paper. The usage of heuristics permits some guesses about the possible usage of an array, suggesting possible abstractions of each array (associated with its indices) to a higher level data structure (e.g. a stack or a queue). In [21], an experiment for abstracting a stack from a FORTRAN program, is presented.

4.4 Code Generation

After the abstraction phases above presented, a network of objects describing the initial program at a higher level, is obtained. This description can be further abstracted at an even higher level or it may be the starting point for forward engineering with a view at obtaining code in a specific programming language. For example, in the system described here, the abstract description resulted from the reverse engineering of FORTRAN programs can easily be translated into C or pseudocode. This forward engineering phase is completed by (C and pseudocode) generation methods attached to each object. The FORTRAN program which the examples in this section referred to, has been translated into C as:

```
void main()
{int I;
 float S;
 I=1;
 S=0;
 while (!(I>10))
        {
         I++;
         S=S+I*I;
         printf("\n I=%d",I);
        }
 if(S>10)
        {
         printf("\n S=%f",S);
        }
 for(I=1; !(I>10); I++)
        {
         printf("\n I=%d",I);
        }
 exit(0);
}
```

Loop and array analysis is very important for the translation of the analysed programs into other languages. For example, loop analysis provides information for the refinement of each loop to a specific loop statement in C. In the above example, two loops have been refined to a "while", respectively a "for" loop.

Array analysis is also important for the generation of C code. For example, C arrays start from 0 and not from 1 as in FORTRAN. This fact brings about some decisions on specific refinements. These decisions are due to array usage analysis that may be considered a mini expert system. An example is the program below:

```
dimension x(4)
x(1)=10
x(2)=20
x(3)=30
x(4)=40
w=1
```

```
 s=0
 do 1 i=1,4
 s=s+x(i)
1 w=w+s*x(i)
 write(1,1)w
 stop
 end
```

has been translated into:

```
void main()
{
 int I;
 float W,S;
 float* X;
 X[0]=10;
 X[1]=20;
 X[2]=30;
 X[3]=40;
 W=1;
 S=0;
 for (I=1-1; !((I+1)>4); I++)
        {
         S=S+X[I];
         W=W+S*X[I];
        }
 printf("\n W=%f",W);
 exit(0);
 }
```

but,

```
   dimension a(8)
   ....
```

```
 do 4 i=1,n
4 q=q*t+(n+1-i)*a(i)
 ...
 stop
 end
```

has been translated into:

```
void main()
{
 int N,M,I;
 float T,R,P,Q,X;
 float* A;
 ...
 I=1;
 do
        {
         Q=Q*T+(N+1-(I-1))*A[(I-1)];
         I++;
        }
 ...
 }
```

4.4 Results and Comparison with Other Approaches

The reverse engineering system described in this paper was developed for a subset of the FORTRAN IV language and tested on some tens of routines. There were not considered statements like computed or assigned goto, encode, decode, common, and equivalence. Format statements were parsed as comments. A run of the reverse engineering system is considering one procedure. Nevertheless, the resulted object network can be saved and its information used in the analysis of related procedures. Due to the evolutionary character of the system, (and to the possibility of knowledge acquisition) the system can be extended to covering untreated features and statements.

The approach made in this paper is, in several aspects, similar to that of the "Programmer's Apprentice" (PA) project at MIT [17]. XRL has a similar functionality with the first layers of the system CAKE [16], in which the latest version of PA has been implemented. XRL has been developed as a general-purpose environment for knowledge-based applications. Therefore, it does not have an equivalent of the plan calculus layer from CAKE. Nevertheless, XRL has some other important facilities as production rules, advanced constraint representation and processing, and the concurrent refinement modules that enhance the power of knowledge representation and processing. Programmed graph grammar parsing in the present approach is also related to the program understanding approach from the same "Programmer's Apprentice" [23,24] but extends parsing by taking into account heuristic knowledge. XRL also offers some of the facilities of the V language [18].

One conceptual difference between PA and the approach in this paper is the metaphor taken into account. As emphasized in [21], the current system is not directed towards being an apprentice of an experimented programmer. It is rather considered as an environment that encourages and facilitates the cognitive modelling processes for the development of a theory of programming activities represented as an articulated knowledge corpus. This idea is related to the point of view that sees knowledge acquisition as a modelling activity [3]. Therefore, knowledge bases are not only libraries of programming "cliches." It is desired that they would rather contain an evolving model of programming concepts and constructs. From this perspective, program understanding and development are very useful activities for the evolution of the system's knowledge about programming. In this evolution it is very important also the enhancement of human knowledge as a result of getting new insights as side effect of the development of programming knowledge.

The idea of considering global information about data is not recent. It is also used in [25]. They determine variable "cobwebs," an extension of def-use pairs' idea from compilers [1] and use this information for program restructuring. Data structuring as used in [13] is for generating an object-oriented structuring.

Control abstraction has been considered for many years as structuring unstructured FORTRAN programs [14]. The present approach includes a similar technique, one important difference being the fact that the restructuring of the program is only a first abstraction phase in the spectrum of abstraction possibilities. The versatility of the programmed graph grammar parsing also gives the possibility of various restructurings, according to any control abstraction.

The idea of developing hierarchies of programming concepts and (eventually) re-usable constructs is not a new one. Barstow's PECOS system might be an example [9]. The up-to-date libraries of re-usable objects for several OOP languages (e.g. C++) are in fact developed along such hierarchies. The approach in this paper tries to make aware the fact that such hierarchies must converge to a structuring reflecting an evolving theory of programming. It is also very important to facilitate and attract the contribution of human experts to the development of such hierarchies.

5. Conclusions

The approach made in this paper starts from the idea that knowledge acquisition is a modelling activity. One consequence is that building programming apprentices is only a perspective of the possible integration of knowledge-based techniques in software engineering. A more adequate point of view is focused on the human expert which must have the most "ergonomic" tools for modelling, experimentation (a point of view also related with fast prototyping), and, in connection with the first two activities, knowledge representation. In this context, the OOP paradigm (extended towards frame knowledge representation), is viewed as naturally supporting at the same time the modelling process, re-usability, and evolution in all the phases of program development.

The development of the framework presented in the paper and the experiments performed are sustaining these ideas. For example, the

taxonomy of programming concepts and rules and, at the same time, the model that the author had had about programming concepts and activities evolved after some of experiments. Now, the definition and integration of new objects in the taxonomy is intended (e.g. algorithm construction schemes as "divide & conquer" and "greedy", high level loop abstractions suggested by [22]). After having included new objects, complex restructurings are planned in the idea of maintaining a sound "theory" of programming represented by the taxonomy of concepts.

REFERENCES

1. AHO, A., SETHI, R. and ULLMAN, J., **Compilers. Principles, Techniques, and Tools**, ADDISON-WESLEY, 1986.
2. BALZER, R., **A 15 Year Perspective on Automatic Programming**, *IEEE TRANS. ON SOFTWARE ENGINEERING*, Vol. SE-11, No. 11, November 1985, pp. 1257-1268.
3. BARBUCEANU, M., **Knowledge Acquisition, Modelling, and Operationalization**, PhD Thesis, "Politehnica" University of Bucharest, 1993.
4. BARBUCEANU, M. and TRAUSAN-MATU, ST., **Integrating Declarative Knowledge Programming Styles and Tools into a Structured Object Environment**, in J. McDermott (Ed.) Proceedings of 10-th International Joint Conference on Artificial Intelligence IJCAI'87, Milan, Italy, MORGAN KAUFMANN, 1987.
5. BARBUCEANU, M. and TRAUSAN-MATU, ST., **XRL: A Layered Knowledge Processing Architecture Able To Enhance Itself**, STUDIES AND RESEARCHES IN COMPUTERS AND INFORMATICS, Vol. 1, No. 1, Bucharest, 1989, pp. 76-106.
6. BARBUCEANU, M., TRAUSAN-MATU, ST. and MOLNAR,B., **Concurrent Refinement: A Model and Shell for Hierarchical Problem-solving**, in J.C. Rault (Ed.) Proceedings of 10th Workshop on Expert Systems and Their Applications, Avignon, 1990.
7. BARBUCEANU, M. and TRAUSAN-MATU, ST., **MODELS: Towards a Language Construction Approach to Expert System Design and Enhancement**, STUDIES AND RESEARCHES IN COMPUTERS AND INFORMATICS, Vol.1, No. 2, Bucharest, 1990, pp. 57-76.
8. A. Barr and E.A. Feigenbaum (Eds.) **The Handbook of Artificial Intelligence**, Vol.2, MORGAN KAUFMANN, 1982.
9. BARSTOW, D., **Knowledge-Based Program Construction**, NORTH-HOLLAND, 1979.
10. BROWNSTON, L., FARRELL, R., KANT, E. and MARTIN, N., **Programming Expert Systems in OPS5. An Introduction to Rule-Based Programming**, ADDISON-WESLEY, 1985.
11. BUNKE, H., **Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation**, IEEE TRANS. PATT. ANAL. AND MACH. INTELL., Vol. 4, No. 6, 1982.
12. FIKES, R. and KEHLER, T., **The Role of Frame-Based Representation in Reasoning**, COMMUNICATIONS OF THE ACM, Vol.28, No.9,1985 , pp. 904-920.
13. LANO, K., BREUER, P. T. and HAUGHTON, H., **Reverse-Engineering COBOL via Formal Methods**, SOFTWARE MAINTENANCE RESEARCH AND PRACTICE , 5 , 1993, pp. 13-35.
14. LICHTBLAU, V., **Decompilation of Control Structures By Means of Graph Transformations**, in G. Goos and J. Hartamanis (Eds.) Mathematical Foundations of Software Development, Lecture Notes in Computer Science No. 185, SPRINGER-VERLAG, 1985.
15. MUCHNICK, A. and JONES, N.D., **Program Flow Analysis: Theory and Applications**, PRENTICE-HALL, 1981.
16. RICH, C., **The Layered Architecture of a System for Reasoning About Programs**, *Proceedings IJCAI'85*, MORGAN KAUFMANN, 1985, pp. 540-548.

17. RICH, C. and WATERS, R.C., **The Programmer's Apprentice**, ACM PRESS, ADDISON-WESLEY, 1990.

18. SMITH, D., **Research on Knowledge-Based Software Environments at Kestrel Institute**, IEEE TRANS. ON SOFTWARE ENGINEERING, Vol. SE-11, No. 11, 1985, pp. 1278-1295.

19. TRAUSAN-MATU, ST., **Micro-XRL: An Object-Oriented Programming Language for Microcomputers**, Research Report, Technical Institute of Cybernetics , Slovak Academy of Sciences, Bratislava, 1989.

20. TRAUSAN-MATU, ST., GHICULETE, GH. and BARBUCEANU, M., **The Integration of Powerful and Flexible Constraint Processing into an Object-Oriented Programming Environment**, in J.C.Rault (Ed.) Representation par Objets, La Grande Motte, France, June 1992.

21. TRAUSAN-MATU, ST., **Computer Aided Software Engineering; Program Re-design and Re-use by Reverse Engineering** (in Romanian), Ph.D Thesis, "Politehnica" University of Bucharest, 1993.

22. WATERS, R., **A Method for Analyzing Loop Programs**, IEEE TRANS. ON SOFTWARE ENGINEERING , Vol. 5, No. 3, 1979, pp. 237-247.

23. WILLS, L.M., **Automated Program Recognition: A Feasibility Demonstration**, *ARTIFICIAL INTELLIGENCE*, 45, 1990, pp. 113-171.

24. WILLS, L.M., **Automated Program Recognition by Graph Parsing**, PhD Thesis (also AI-TR 1358), MIT, 1992.

25. ZIMMER, **Restructuring for Style**, SOFTWARE PRACTICE AND EXPERIENCE, Vol. 20, No. 4, 1991, pp. 365-389.