Building a Tool for Software Code Analysis A Machine Learning Approach

Gilles Fouqué[1]
B 004
EDF-DER
1, av du Gal de Gaulle
92141 Clamart Cedex
Tel : (33) 1-47-65-57-03
e-mail : gfouque@csi.uottawa.ca

Christel Vrain[2]
LIFO
Université d'Orléans
Rue de Chartres
45067 Orleans Cedex
Tel : (33) 38-41-70-00
e-mail : cv@univorl.univ-orleans.fr

Abstract. This article presents the application of a machine learning technique to a software code analysis tool. This tool builds a base of elementary and relevant program structures, acquired from the analysis of a primitive set of programs of good programming style. These program structures are compared to new programs to determine the quality of the latter. In this paper we stress the framework of the tool and discuss the critical details of its modules.
The learning technique has been developed to use intensively a specific knowledge base in order to acquire the base of relevant program structures. We present problems that arise due to the necessity of using knowledge which is non-monotonic in nature.
Particular issues will be highlighted by the analysis of requests written in the SQL language.

1. Introduction

Software engineering has to provide tools and techniques to improve software quality. To this end, the automatization of the analysis and validation process is an important issue and several tools have been developed [4, 15]. Usually, these tools are classified in two groups : dynamic and static tools. The former concludes with a diagnosis of the quality of the code from observations of its execution. The latter is based on the study of the data flow and the control flow of the code to check if it satisfies a set of known rules for identifying good programming style code.

In our point of view, neither of these approaches concludes with a satisfactory diagnosis. In most cases, dynamic approaches are intractable and are used for very specific domains such as autonomous or safety systems. Static approaches use metrics to measure the quality of the analyzed code and they then compute an abstract view of this code which involves an important expertise step. Aware of these limitations, we kept in mind the behavior of a programmer during his analysis task, to specify our tool. The programmer compares the elementary program structures of new code to similar structures within his experience. He is thus able to give a precise semantic meaning to those structures. At the end of this review process the analyzed program is a mosaic of known and unknown structures. From a detailed understanding of the latter the programmer will have enriched

[1] G. Fouqué is currently pursuing post-doctoral studies at the University of Ottawa with the Ottawa Machine Learning Group

[2] a part of this research has been made in the Inference and Learning Group, LRI, Université Paris XI

his knowledge of elementary program structures or will have modified them. As the strategy of the programmer is adaptative we were interested in a machine learning approach which could resolve the problems previously presented.

The paper is organised as follows. The next section presents previous approaches for software code analysis. We then detail our approach and its advantages, and go on to describe each one of its modules. The last sections are devoted to the description of the developments to the learning technique and to the experiments performed on several kinds of data.

2. Previous static code analysis approaches

The main task of software engineering researchers deals with decreasing the weight of the maintenance step in the software life cycle. As this maintenance is related to the understanding of the code, the first approaches have been to standardize the program writing task using program standards and macros. These approaches are not commonly used because they are inflexible, difficult to perform and cannot be used to understand existing programs.

More elaborate approaches [10, 16] compare new programs with a fixed and hand-build base of structures called *clichés*. This comparison process is performed using a knowledge base including properties of the domain. These approaches have to face two problems : the acquisition of the clichés and their recognition. Firstly, there are too many clichés to generate a complete base of them. Secondly, construction of a complete and consistent knowledge base for manipulating these clichés is also intractable if we consider the extraordinary complexity of the domain.

Our tool uses a machine learning techniques in order to have an incremental acquisition of the program structures - the clichés - and a powerful matching process which uses intensively a knowledge base. Our approach allows us to build a tool having a dynamic form of management as the base of clichés grows during the analysis of new structures. Moreover, our specific approach does not need a complete knowledge base as we detail below. The idea to use machine learning for a software engineering task is not a new one [2]. Most research in this domain deals with using analogy [1, 5] often combined with an EBL approach [3]. These projects involve important research in both the fields of, software engineering - to specify a design language or to generate some code - and machine learning - most of this research uses a case-based reasoning approach and has to resolve indexing and retrieving problems. Here, we limit our approach to the analysis of final code using the grammar of the programming language as the main source of knowledge in the domain.

3. An adaptative acquisition of programming structures

3.1. A global schema

Our tool is composed of two modules, the acquisition and the recognition module. Figure 1 presents a global schema of the tool.

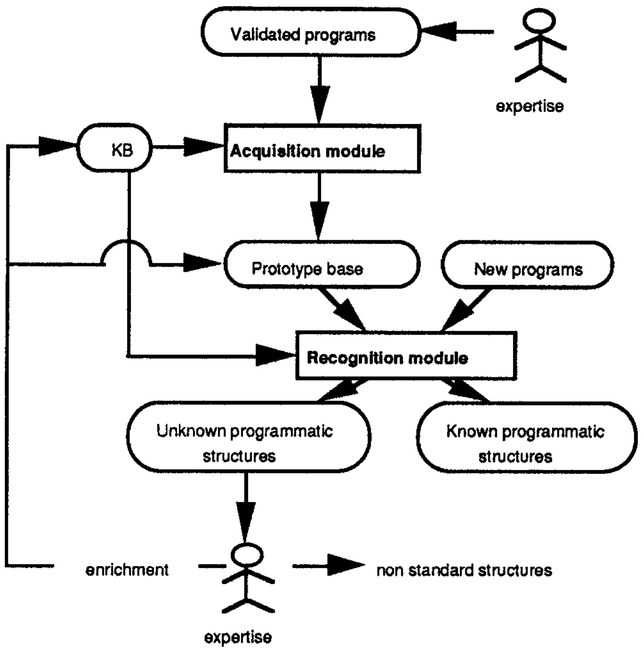

Figure 1 : Schema of the analysis tool

The acquisition module analyses a primitive set of validated programs and builds a base of program structures. A validated program is a program having certain standards of good programming style. These standards do not have to be explicit, they are concluded from a by-hand analysis of the programs. This module uses a two steps process. First, it defines the program structures for each program. Second, it builds a base of prototypes from these structures using its knowledge about the specific programming language. So, the primitive set of program structures is subsumed within the base of prototypes considered by the knowledge base.

The recognition module compares new programs to the base of prototypes. It specifies the program structures it is able to recognize and those it considers as new structures.

Most of our work deals with the acquisition module, the recognition one has not yet been implemented but should be based on the same learning techniques as the former in order to compare the prototypes to new structures.

The previous schema presents the use of our tool for a validation task. The base of prototypes is an extended representation of the rules of good programming style used to write the set of validated programs. The acquisition module removes the features of each implementation to keep the syntactic characteristics shared by several programs. Applied to a new application, the recognition module highlight new structures which do not contain these characteristics in order to help the programer understand this application. A similar approach could be used to look at bugs in a new application where the acquisition module would have to analyze programs with common bugs.

The acquisition module could be used in another software engineering task : the reuse of code. Currently, the moderate success of reuse is the lack of software libraries as the choice of the good components for such a library is difficult to perform [8, 11]. This choice is presently the task of the acquisition module which determines the program structures which are commonly used in an application. The automation in the choice of the components would be an important feature in the building of an evolutionary library. Nevertheless, our approach does not deal with the whole range of the reuse process such as the indexing or the documentation of the components.

3.2. The choice of a learning technique useful for a software engineering task

Our approach implies a learning from example strategy [9]. To resolve the problem of using inductive learning to handle structured data and make use of a knowledge base of domain properties, we chose the structural matching technique [6]. This technique has the advantage to use a two step algorithm which limits the uncertainty brought by an inductive approach. The first step uses intensively the domain properties according to a deductive process. The second step is an inductive one, it removes the remaining features of analyzed examples so that they match structurally and then builds a generalization for these structures. In our application, the examples are program structures and the domain properties are transformation rules used in order to get the analyzed structures more similar in a syntactic meaning.

So, the aim of the first step is to use a knowledge base to transform the program structures so that the second step is able to induce the most specific prototype considering this knowledge base. Expressed in formal terms, the structural matching definition is :

> Let $\{e_1, e_2, \ldots, e_n\}$ be a set of n examples. These examples match structurally iff there exists a formula F and n substitutions σ_i such that, for each i, $\sigma_i(F) = e_i$.

F is the recognition function or the generalization of the examples and its "quality" depends on the knowledge base. In our approach, an example is a program structure and a generalization is a prototype. The primitive structural matching technique needs several developments to be used for this specific application. These developments rely on improving the efficiency and the ability of this technique to handle structured data.

4. The Acquisition module

This section describes the acquisition module which includes : the data structure, the definition of the program structures, a clustering process based on syntactic features and a learning process to build the set of prototypes.

4.1. The data structure

Our tool uses a first order logic representation of the programs in order to easily manipulated them considering our knowledge based approach. This representation is based on the recognition of a finite set of sub-trees in the parse tree of each program. During a parsing, a specific predicate name is given to each sub-tree of the parsing tree and the arguments of this predicate are the values of the sub-tree nodes.

4.2. Definition of a program structure

We did not yet define what we mean by a program structure which is the elementary data structure used by our tool. Such structures must have a syntactic and semantic integrity in order to be manipulated independently of the rest of the program. A program structure has a syntactic integrity if it is composed of a set of contiguous instructions. Moreover, this structure has a semantic integrity if the variables of its instructions are independent of the instructions of the other structures. These specific structures are built using an analysis of the data flow of each program to define its connected components. This process is based on the work used in the LAURA system [7] for the analysis of PASCAL programs. The result of this analysis is a set of program structures defined as a block of instructions with specific properties. An important property is the possibility to perform transformations on the instructions of a block without looking at the other blocks. We will not describe this process further as most of our developments are dealing with the analysis of SQL requests. These requests have, by nature, this semantic and this semantic integrity because the way that each request performs its operations is independent of the other requests of the application.

4.3. The clustering process

The aim of this step is to cluster the program structures which are similar in a syntactic meaning. These clusters will be the inputs of the learning process which has to build a prototype for each of them.

We chose to include two processes in the acquisition module - clustering and learning - rather than to get a single classifier based on an inductive process or on a data analysis technique because :
- The application domain has a strong structure which allows efficient clustering using a standard data analysis process. Nevertheless, such an approach requires a posteriori analysis of its results as it deals with numeric criteria.
- The main purpose of the learning process was to use intensively a knowledge base specialized in the manipulation of derivation trees. Learning-based classifiers are inappropriate for this task.

Considering these characteristics, the aim of this combination of a numeric and a symbolic approach was to build first drafts of the program structures with the clustering process and then to refine them with the machine learning one.

We are using an ascending clustering process [12] to compare the syntactic derivation trees of several program structures. This process repeatedly clusters the two most similar structures using a distance function D. We defined D in order to perform a breadth first parse of the parse trees during its comparisons because the main features of the program structures are in the higher nodes of there trees. Considering two program trees Pk and Pl, represented as two ordered vectors of length n, the distance function we built is :

$$D(Pk, Pl) = \Sigma_{j=1,n}\ c_j\ |Pk_j - Pl_j| = \Sigma_j\ d_j(k,l)$$

where Pk_j and Pl_j are the j^{th} elements of the Pk and Pl vectors and c_j is a decreasing factor expressing the decreasing meaning of the analyzed elements.

The choice of the clustering method and of the distance function factors has been made in order to cluster SQL requests. For another programming language, the factor c needs to be modified considering the link between the depth of the tree and the semantic meaning of its nodes. The results of this clustering for the SQL requests are, for example, to cluster select requests with monotable queries and a single condition, select requests with a single joint order, etc...

4.4. Learning recognition functions

Now, each class has to be analyzed in order to build a relevant prototype. We chose the system OGUST [14] based on the principle of structural matching and we developed it to deal with our specific application. OGUST learns a recognition function of a concept from a set of examples of this concept and knowledge about the application domain. According to the structural matching technique, OGUST's learning process is divided in two steps. The deductive step chooses a constant in each example, replaces all its occurrences by a generalization variable and tries to make all its discriminating occurrences disappear using the knowledge base, i.e. the aim of the learning process is to use the knowledge base in order to have the biggest set of constants sharing the same properties. Schematically, an occurrence of a variable x, argument of a predicate P, is a discriminating one if x is not an argument of the same predicate P in the other examples.

These operations are performed until no constant remains. The inductive step , then, builds a generalization of these examples keeping their common properties.

An example :: we present here the general learning process applied to 3 requests :

S1 :
```
Select Name from person
 where not (date_exam is null) and age > 18 and age < 25 and status = 'student'
  order by no_d_inscription
```

S2 :
```
Select Name from person
 where age > 30 and age < 45 and status = 'professor' and salary is not null
  order by no_order
```

S3 :
```
Select Name from person P1
 where status = 'visiting professor' and age between 25 and 35
    and exists (select job from person P2
                  where P1.name = P2.name)
  order by no_order
```

If we consider the knowledge base :

Rg1 : P1 and P2 ⇔ P2 and P1
Rg2 : P1 between B1 and B2 ⇔ P1≥ B1 and P1 ≤ B2
Rg3 : P1 is not null ⇔ not (P1 is null)
Rg4 :
```
Select R1 from T1        ⇔    Select R1 from T1 C1
   where P1 is not null        where exists (select  R2 from T1 C2
                                                 where C1.P1 = C2.P2)
```

During the deductive process :
the rule Rg3 is applied to S1 to transform : *not (date_exam is null)* into *date_exam is not null,*
the rule Rg1 is applied three times to S2 to transform : *age > 30 and age < 45 and status = 'professor' and salary is not null* into *salary is not null and age > 30 and age < 45 and status = 'professor'*
the rule Rg4 is applied to S3 to transform : *Select Name from person P1 where status = 'visiting professor' and age between 25 and 35 and exists (select job from person P2 where P1.name = P2.name)* into *Select Name from person where status = 'visiting professor' and age between 25 and 35 and name is not null*
then, the rule Rg2 and five times the rule Rg1 are applied to transform : *status = 'visiting professor' and age between 25 and 35 and name is not null* into *name is not null and age > 25 and age < 35 and status = 'visiting professor'*

Then the inductive process builds the generalization of the three examples keeping their common properties, using the "dropping conjunction rule" for the remaining discriminating variables :

```
G : Select Name from person
    where col1 is not null and age > n1 and age < n2 and status = ch1
      order by ch2
```

where n1, n2 are numbers, ch1, ch2 are strings and col1 is a column-name.

This example shows the strategy of the OGUST algorithm. One of the main problems of this algorithm is the choice of constants in predicates. Different choices led to several recognition functions which could be irrelevant or incomparable. For an effective application of this algorithm to software code analysis information specific to the domain must be applied, as will be described below.

5. The recognition module

In this section, we present our specifications of the recognition module which has not yet been implemented. The acquisition module produced a base of *m* prototypes from the set of *n* initial programs. Then, the recognition module compares a new program structure *e* to this base and concludes about its recognition or its non-recognition. *e* is recognized by a prototype G_i if G_i covers *e* :

> *Gi* covers e iff *Gi* is more general than *e*
> *Gi* is more general than *e* iff $\exists\ \sigma_i$, a substitution and F_i a function such that :
> $\sigma i(Fi) = e$ and, for the set X of F variables, $Fi(X) \Rightarrow Gi(X)$

There are three outputs of the recognition module :

- $\exists!\ G_i$: the program structure *e* has been recognized as a known prototype. It inherits of the properties which could have been linked to G_i.
- $\exists\ G_{i1}, \ldots, G_{ik}$, *k generalizations such that* $\cup_{i=i1,\ldots,ik} G_i = e$: *e* is a complex program structure composed of several concepts independently learned in several prototypes. *e* inherits of a combination of the properties linked to each prototype.

- *others* : the recognition module has not been able to recognize *e* and a programmer needs to diagnose about :
  - the validity of *e*,
  - the validity of the knowledge base,
  - the validity of the prototype base.

The explanations of the failures in matching help the programmer to choose between two possibilities. In the first case, *e* is not considered as valid and cannot be efficiently transformed to make it similar to a known prototype. In the other case, the concept described by the example has already been met but the knowledge base does not include the useful transformation rules. If these rules are easy to acquire the programmer completes the knowledge base, otherwise he completes the prototype base. The programmer does the latter choice too if *e* describes a new concept.

6. Developments of the learning technique

Previous sections described the core of the learning process. This section presents the developments of OGUST which enable it to analyze software code. These developments involve the program structures and the knowledge base.

6.1. Problems to be solved

Constants choice. A priori, the matching strategy of OGUST is a good one because it takes care of the big nodes of the parse trees, i.e. the nodes which have many sons. A skeleton of the prototype is quickly built. Nevertheless, as this strategy is a syntactic approach rather than a semantic one, some unimportant nodes can be chosen. Then the result of the matching process will be an useless set of disjoint sub-trees rather than a consistent parse tree. Another problem with the primitive strategy of OGUST is its exhaustive comparison of constants. To compare *n* examples with an average of *m* constants, the complexity formula is : $O(m^n)$. This complexity is intractable considering the amount of data to be analyzed in a software code analysis approach.

The use of the knowledge base. The rules Rg1, Rg2, Rg3 and Rg4 described for the analysis of SQL requests are non monotonic in nature. OGUST uses these rules to transform the derivation trees - to drop branches and to add new ones. Previously, OGUST used idempotency to apply its rules - taxonomies or implications - adding new predicates without dropping any information. Using non-monotonic rules - named equivalences - involves taking care of the long-range consequences of the transformations they perform on the program structures.

6.2. Solutions

Two kinds of solutions have been implemented : using biases to decrease the complexity formula and an improvement of the resolving process.

Using biases. The quality of the result of the matching process cannot be ensure because it performs syntactic feature comparisons. Thus, we introduced semantic features using biases [13] deduced from the grammar. These biases are ordering typed constants and incremental learning.

Each constant can be typed using the transition of the production it is derived from. The similarity is only computed between constants of the same type. Moreover, it is interesting to favour constants derived from the first steps of the syntactic analysis because they get the main semantic features. So we built a hierarchy of types according to the call graph of the program.

During the matching process, the constants of the higher type are analyzed followed by constants of the lower type, etc... This choice simulates a breadth first parse of the parse tree. This strategy has already been used during the clustering process to build ordered vectors. It is useful for the recognition module too as it speeds up the retrieval process and avoids an indexing step.

The previous biases on types change the factor *m* in the complexity formula, less constants are compared during an iteration of the OGUST algorithm. Nevertheless this formula is still an exponential one. To transform the factor *n* into a constant we use an incremental approach. In fact, we use two kinds of incrementality, a constant-based and an example-based incrementality :

- *constant-based incrementality* : as the primitive set of examples of the acquisition module is known and finite, it can be ordered according to the information that each example gets (this information is computed using the hierarchy of types). Then, we compare couples of vectors rather than to compare *n* vectors of constants.
Expressed in a formal way :

Rather than to look at a set $(C_{1i_1}, \dots , C_{ni_n})$ such that :
$\text{Sim}\ (C_{1i_1}, \dots , C_{ni_n}) = \text{Max}\ (\text{Sim}\ (c))$
$\forall\, c \in \mathfrak{C}$
Where $\mathfrak{C} = \{(C_{11}, \dots, C_{1m_1}) \otimes \dots \otimes (C_{n1}, \dots, C_{nm_n})\}$
and $\otimes$ represents the cartesian product of two sets and C_{ij} is the jth constant of example i,

The constant-based incrementality strategy looked at $(C_{1i_1}, \dots , C_{ni_n})$ such that :
$\text{Sim}(C_{1i_1}, C_{2i_2}) = \text{Max}\ (\text{Sim}(c1)), \forall\, c_1 \in \mathfrak{C}_1$
$\text{Sim}\ (C_{1i_1}, C_{2i_2}, C_{3i_3}) = \text{Max}\ (\text{Sim}\ (c_2)), \forall\, c_2 \in \mathfrak{C}_2$
...
$\text{Sim}\ (C_{1i_1}, \dots , C_{ni_n}) = \text{Max}\ (\text{Sim}\ (c_{n-1})), \forall\, c_{n-1} \in \mathfrak{C}_{n-1}$
where
$\mathfrak{C}_1 = \{(C_{11}, \dots, C_{1m_1}) \otimes (C_{21}, \dots, C_{2m_1})\}$
$\mathfrak{C}_2 = \{(C_{1i_1}) \otimes (C_{2i_2}) \otimes (C_{31}, \dots, C_{3m_3})\}$
...
$\mathfrak{C}_{n-1} = \{(C_{1i_1}) \otimes \dots \otimes (C_{n-1i_{n-1}}) \otimes (C_{n1}, \dots, C_{nm_n})\}$

- example-based incrementality : when the recognition module concludes that the new analyzed example *e* inherits of the properties of several prototypes, a new prototype is stored in the base of prototypes as the generalization of *e* and each prototype. To be able to build the most specific generalization we should have to compare *e* with the examples covered by each prototype because the sets of rules used to build a generalization can be different if we compare an example with an abstraction of several examples (Gen(G, e)) rather than with other examples (Gen(e_1, e_2,..., e_n, e)). For the software code analysis process, there is no gap of meaning between a generalization and an example because both are derivation trees expressing a consistent program structure. So, using previous biases we do not loose information if we directly generalize a new example with the generalization, so : Gen(G, e_{n+1}) = G(e_1, ..., e_{n+1})

Theorems using. The characteristics of the knowledge base used for software code analysis are : a finite set of predicate names, many rules and complex equivalence rules. OGUST uses a mixed resolution strategy which led to loops during the backward resolution. We do not describe here biases we used to manage the resolver but we detail the main bias developed to use efficiently the equivalence rules. This bias, called "spreading", allows looking at the long-range consequences of the application of an equivalence rule. Consider that we have to analyze two examples *E1* and *E2* and that we match *T* typed constants. For each constant *C* of type *T* of an example *E* we define its "surrounding" *S(C,E)* as the set of constants belonging to the same predicates than *C* but having a lower type than *C* type in the hierarchy of nodes. Then, to compute the similarity of two constants *Sim(C1, C2)* we compute the similarity between their "surroundings" *Sim(S(C1, E1), S(C2, E2))*. A forward saturation is performed to the predicate of the analyzed constants in order to take care of all their properties, explicit or implicit ones. This first saturation is an artificial one as it is used to find the best constants to be matched, then the primitive resolving process is applied. This spreading strategy improves the quality of the results and the efficiency of the tool too as OGUST does not match separate nodes but sub-trees.

7. Initial results

Several steps of our tool have been validated analyzing SQL requests used in real applications :

- *Clustering :* we used two sets of requests (70 and 54 requests) to define the distance function. The first set has been clustered by hand by a SQL programmer. This first clustering has been compared to several automatic clustering techniques to define the parameters of the distance function. These parameters have, then, been validated by an a posteriori analysis of the clustering of the second set. From the application of the same process to larger sets of requests we concluded that the number of classes with a functional meaning is stabilizing and that abnormal requests are the only element of their own class.

- *The acquisition of the recognition functions :* different kinds of examples have been used to check the knowledge base :
    * *Examples deduced from the rules :* we deduced artificial examples from the knowledge base building an example from the condition and an example from the conclusion of some rules. These examples were useful to test the "spreading" process. Currently, this process is statically limited in its depth and the user has to choose between several matching constants when the spreading process has not been able to do it.
    * *Real examples :* these examples does not need an intensive use of the knowledge base to be matched as they are well-formed and so have a regular structure. This approach was useful to test the incremental approach which

allows to analyze an important set of SQL requests (to 25 real requests) without using a knowledge base.

* *Toy-examples :* which need a complex use of the knowledge base. These examples have been useful to test the behavior of the motor engine at the limit, specifically to resolve the problems of loops during the backward chaining.

8. Conclusion

In this paper we have presented the advantages of adopting a strategy based on a static analysis of code in order to improve the maintenance of software. Then, we have described our approach based on the application of a complex machine learning technique. We have discussed the extensions needed to existing machine learning techniques in order to apply them to our specific software engineering task. These extensions used features of the domain to increase the quality of the results and the efficiency of the system. Currently our tool resolves most of the problems we have presented for specific requests, a couple of developments have to be done in order to merge all those solutions to deal with real requests. These developments included :

- to adapt the limits of the surrounding of constants during the spreading process, according to the amount of effort needs to compute the similarity between two constants,
- to reuse the saturation of the examples by the knowledge base during the spreading process to improve the resolving strategy of OGUST during the matching process,
- to be able to perform several generalizations rather than only one when several constant choices are possible,
- to improve the efficiency of the implementation of OGUST.

An interesting application of our tool will be to analyze classical programming languages. Tests have been performed with subsets of grammars such as declarations in FORTRAN programs or input/output orders of PASCAL programs. Nevertheless, the analysis of programs of these languages using their whole grammar is not a simple extension of the approach presented here as their grammars are bigger and more complex than the SQL one. Currently, we are interested to use data flow and control flow graphs of structured programs. Those graphs should be helpful to choose the productions of the grammar which are useful to build the data. Then, the traditional code optimization technics based on those graphs should be adapted in order to build the knowledge base.

References

1 Bailin, S.C. & Gattis, R.H. & Truszkowski, W. 1991. A learning software engineering environment. *6th knowledge-based software engineering conference.* pp 251-263.

2 Dershowitz, N. 1986. Programming by analogy. *Machine learning 2 : an artificial intelligence approach. Michalski R.S., Carbonel J.G., Mitchell T.M., Morgan Kaufmann eds*, pp 395-423.

3 Geldrez, C. & Matwin, S. & Morin, J. & Probert, R.L. 1990. An application of EBL to protocol conformance testing. *IEEE Expert.* October 1990. pp 45-60.

4 Halstead, M. 1978. Elements of software engineering. *Elsevier eds.*

5 Harandi, M.T. & Lee, H. 1991. Acquiring software design schemas : a machine learning perspective. *6th knowledge-based software engineering conference.* pp 239-250.

6 Kodratoff, Y. & Ganascia, J.G. 1986. Improving the generalization step in learning. *Machine learning 2 : an artificial intelligence approach. Michalski R.S., Carbonel J.G., Mitchell T.M., Morgan Kaufmann eds,* pp 215-244.

7 Laurent, J.P. & Fouet, J.M. 1982. Outillage de manipulation de programmes fondé sur une représentation arborescente. *Premier colloque de génie logiciel.* pp 105-118.

8 Maarek, Y.S. & Berry, D.M. & Kaiser, G.E. 1991. *An Information Retrieval Approach For Automatically Constructing Software Libraries.* IEEE Transactions on Software Engineering. Vol 17, N 8. pp 800-813.

9 Michalski, R.S. 1983. A theory and methodology of inductive learning. *Machine learning 1 : an artificial intelligence approach. Michalski R.S., Carbonel J.G., Mitchell T.M., Morgan Kaufmann eds,* pp 1-19.

10 Ning, J. & Harandi, M.J. 1989. An experiment in automatic program analysis. *Proc AAAI symp artificial intelligence and software engineering. AAAI Press eds.* pp 51-55.

11 Prieto-Diaz, R. 1991. *Implementing Faceted Classification for Software Reuse. Communication of the ACM.* Vol 34, N 5. pp 88-97.

12 Saporta, G. 1990. Probabilités, analyse des données et statistiques. *Editions technip.*

13 Utgoff, P.E. 1986. Shift of bias of inductive concept learning. *Machine learning 2 : an artificial intelligence approach. Michalski R.S., Carbonel J.G., Mitchell T.M., Morgan Kaufmann eds,* pp 107-148.

14 Vrain, C. 1990. OGUST : a system that learns using domain properties expressed as theorems. *Machine learning 3, an artificial intelligence approach. Kodratoff, Y. & Michalski, R.S. Morgan Kaufmann eds.* pp 360-382.

15 Webb, J. 1988. Static analysis : an introduction and example. *Journées Internationales : le génie logiciel et ses applications.* pp 523-539.

16 Wills, L.M. 1990. Automated program recognition : a feasibility demonstration. *Artificial intelligence N°45.* pp 113-171.