Research Journal of Applied Sciences 2 (3): 258-265, 2007

Decision Support Systems for JAVA Infrastructure Analysis and Control

Sundus Noory Shuker
College of Applied Sciences, University of Bahrain, Bahrain

Abstract: Programmers and managers involved in large software projects need insight into the overall structure of their programs and the relationships between components. This study describes a method of analyzing programs whereby cross-referential, dependency and other information can be abstracted automatically during compilation to support decision-making. These items of information support library administration, configuration management, version control, software reusability and software composition. The analysis is accomplished by transcribing language grammar rules directly into Prolog as predicates in first-order logic.

Key words: Dision, infrastructure, configration, soft ware reusability, JAVA

INTRODUCTION

A few modern programming languages are designed to support large software engineering projects in which a typical project would consist of perhaps several thousands compilation units. They embody many modern principles of software engineering such as modular development, separate compilation, separation of specifications from bodies, top-down and bottom-up development through the separation of stub implementations and context-clauses and reusable software modules. These good features, at the same time, make the components of software systems difficult to enumerate and control. In particular greater visibility of cross-referential and dependency information is needed.

Analysis and control of infrastructure of software systems is therefore desirable as some modern programming languages (such as JAVA) have potentially complex set of dependencies and cross-references that might be constructed in a particular application compared with previous high-level languages (For example, Algol 68 is a very complex language but it does not support modules and would not normally have a complex web of dependencies in a typical application). In addition, these languages are likely to be used for very large and difficult programming applications. These applications may have a large number of modules whose inter-relationships can be difficult to determine. These modules may be distributed over many files and documents (especially if specifications, implementations, or subunits are stored in separate files). In addition, modules written in mixed languages can be incorporated. These issues become more complicated when, as is essential for many software projects, versions or program families are supported.

A software systems programmer or manager, we believe, needs to have overall visibility of the infrastructure of a system and needs to have a clear picture of all the relationships between the various components of the system. This facility can be provided automatically by extracting the required information from the source code and the related software development phases. A software engineer or manager also needs to be able to establish quickly and efficiently the correspondence between software components and information from other phases of the life cycle. This information, therefore, can support and enhance decision making through the complete life cycle of a product.

The study is concerned with analysis and control as an aid to understanding program structures and module dependencies for the purpose of facilitating decision support mechanisms to manage large-scale programs. The analysis of software systems (information systems) is accomplished by transcribing the language grammar rules of directly into predicates in first-order logic. Although the techniques presented in this study are suitable for analyzing other computer languages, JAVA grammar rules are used throughout. The programming language JAVA has been chosen for the following reasons:

- It is a complex language with a rich selection of features. For example: Import, specification and subunit-of dependencies imply multiple, complex dependencies among source, object and interface files,
- JAVA is widely used for large complex systems and embedded applets within Web browsers where analysis of an application, we believe, is essential.
- There is a need for suitable automatic program examination facilities to be used as a basis for building knowledge-based environments. For example the PACT project has provided support, using PCTE interfaces, for C, Lisp and Prolog but not for JAVA.

Before describing the method of analysis developed some further background on compilation and definite-clause grammars is provided.

Previous work: Traditionally, program examination has been accomplished through printed listings or using interactive devices. One approach suggested by Weinberg (1971) in his Psychology of Computer Programming is code reading. Code reading has led to a number of methods such as walkthroughs, design reviews and inspections to find module interdependencies. In recent years, several attempts have been made to describe and analyze programs using different techniques. One of these techniques is the flowchart. It is used to design and describe program structure.

Flowcharts are not very helpful and do not enable a programmer to determine quickly and efficiently the overall structure of a program nor do they help in providing a clear picture of module dependency relationships. Charts can also become out-of-date. Several attempts have also been made to develop new conventions for information representation. The goals of these attempts were limited and cannot be used as a basis for controlling and managing objects of the life cycle of a software product. For example, Nassi and Shneiderman (1973) developed a chart that is more understandable and they also produced a new diagrammatical notation to represent the structure of programs. Their technique has been used by a number of researchers and developers for program documentation, design and construction.

Frei *et al.* (1978), Ng (1978) used Nassi-Shneiderman diagrams as a basis for program development systems for the language PL/1. Wagner (1979) has developed a system to produce diagrams for programs written in SPL, Pascal, FORTRAN, Assembler and Cobol. These diagrams were mainly concerned with inter-module relationships rather than internal structure of modules.

Ried (1983) has investigated the representation of the static structure of a program constructed from several separately compiled modules. This work was

developed to assist a programmer to establish quickly the structure of a program written in Ada. Most of the research of Ried's Technique was written in Ada using the York Ada workbench compiler. Ried's technique made it possible to represent a program structure on a screen using multiple colors.

For detailed analysis of program structures, there are two commercially available tools. The first one is the SPARK examiner, which gives a detailed analysis of a subset of Ada program structures. It uses data flow diagrams and assisting in maintaining consistency between specification and implementation (Jennings and Carre, 1989).

The second tool is MALPAS, advanced software package for software analysis and verification (MALPAS, 2005) which gives project managers and software engineers a unique tool for the comprehensive analysis of complex software. Because MALPAS analyses source-code without actually executing it (a process called static analysis), the tool does not require expensive test-rigs and is capable of giving 100% path coverage. By revealing errors simply and quickly, MALPAS leads not only to more reliable software but to reduce development and maintenance costs.

The MALPAS tool performs a number of forms of analysis. These are:

- Control flow analysis,
- Data flow analysis,
- Information flow analysis,
- Semantic analysis (symbolic execution) and
- Compliance analysis.

Most of the literature on program analysis is aimed at establishing consistency with specification. Compiler vendors are in the best position to produce tools incorporating many forms of the analysis discussed above, but they have concentrated their efforts on producing compilation systems to the exclusion of other tools.

As has been mentioned above, most of the available techniques for program analysis are not sufficiently developed to be used for more general analysis, abstraction, software configuration management and software reuse and construction. In addition, these techniques cannot be used for analyses and deductions about the software life cycle nor can they enhance the understanding and consistency of the relevant information.

Program analysis and control framework: A program can be defined is a collection of one or more compilation units submitted to a compiler at one or more compilations. There are four basic program units in the case of JAVA. These units are:

- Subprograms
- Packages
- Tasks
- Generics

Each unit may consist of two parts

- Specification: Which describes definitions or declarations that must be visible to other units,
- Body: Which describes the implementation details that need not be visible to other units.

A package can be defined as a collection of related classes. A class is used to create either a JAVA application or Applet. It is used to group a set of related operations; and it is used to allow users to create their own data types. A method, in JAVA, is a set of instructions designed to accomplish a specific task (Malik and Nair, 2003).

Modern practices in software applications encourage the decomposition of large software systems into manageable components that can be compiled separately and stored in a program library. These components (in case of the JAVA) can be library units or secondary units. A library unit is a subprogram or a package declaration compiled separately and stored in the program library. A secondary unit is a subprogram body, a package body, or a subunit.

Once a library unit has been compiled, it can be made visible to another compilation unit by means of an import-clause. Therefore, a compilation unit is a context clause (which declares its dependencies on other library units) followed by a library unit, or a context clause followed by a secondary unit. In the following section we show how Definite-Clause Grammars can be adopted to analyze language structures.

Definite-clause grammars: The fundamental principle of formal language theory is that a language can be described in terms of how its sentences are constructed. The definition is:

- A sentence is a string (a sequence) of symbols - rules for string
- A language is a set of sentences - rules for set.

According to the above definition, a grammar for a language can be defined
as: "A collection of rules for specifying what sequences of symbols are acceptable as sentences (statements) of that language."

Computer scientists have adapted the ideas of formal language theory to the study of programming languages, in the form of Context-Free Grammars (CFGs). In CFGs the basic symbols or words of the language, which they describe, are identified by terminal and non-terminal symbols. The terminal symbols are the basic constructs of the language. The non-terminal symbols describe categories of phrases of the language. A non-terminal symbol can be factorized into terminal and/or non-terminal symbols.

Colmerauer (1978), Kowalski (1974) describe a method to translate the special purpose formalism CFGs into a general one (Pereira and Warren, 1980) in the form of first-order predicate logic. The method is known as Definite-Clause Grammar (DCG). According to DCGs, rules of a grammar describe which strings of symbols are valid statements of the language.

Parsing a rule of DCGs, using Prolog, is accomplished by transforming it into a theory and trying to prove its validity by applying logical reasoning. The proof either fails or succeeds. Pereira and Warren (1980) explain the efficiency of DCGs compared with CFGs as follows: "If a CFG is expressed in definite clauses according to the Colmerauer-Kowalski method and executed as a Prolog program, the program behaves as an efficient top-down parser for the language the CFG describes. This fact becomes particularly significant when coupled with another discovery--that the technique for translating CFGs into definite clauses has a simple generalisation, resulting in a formalism far more powerful than CFGs, but equally amenable to execution by Prolog.

To express DCGs as logic clauses, one needs first to describe CFGs using the following notation:

head --> body

Where head is a non-terminal symbol and body is a sequence of one or more items separated by commas. An item is either a non-terminal or a sequence of terminal symbols. Prolog expresses non-terminal symbols as atoms or terms and terminal symbols as lists. The null-string is written as.
The CFGs for the method-claus is expressed in extended BNF as:

method-clause: = method-specification, method-body;

According to the above notation, method-clause can be arranged in the form:
method¯clause→ specifier, main, arguments, throws-clause, body, semicolon.

```
. . .
main → [main].
throws-clause → throws, exception-handler
 throws-clause → []
semicolon → [';'].
 comma → [','].
```

Each rule of a CFG is translated into definite clauses of logic by associating with each non-terminal symbol a predicate with two arguments. The arguments of the predicate represent the input list and the output (remainder) list. The first three rules in translate into:

```
Method_clause (In, Out) :- specifier (In, Temp1),
    main (Temp1, Temp2),
    arguments (Temp2, Temp3),
    throws-clause (Temp3, Temp4),
    body (Temp4, Temp5),
    semicolon (Temp5, Out).
```

We can read the first clause as the input list In has a valid method clause at the front (returning a left over list of words called Out) if In starts with public (leaving a temporary list Temp1) and Temp1 starts with specifier (leaving a temporary list Temp 2) and Temp 2 starts with main (leaving a temporary list Temp 3) and Temp 3 starts with argument (leaving the remainder list Temp 4) and so on.

This approach provides a number of extra mechanisms including:

- Inclusion of context information
- Imposing of conditions and constraints
- Building of structure trees
- Provision of partial semantics

The context information can be obtained from the grammar rule by which arguments can be associated with non-terminal symbols. These arguments are used to carry information. In addition, they can be used to produce parse trees in which a relationship between elements of a statement can be established. For example, to return information of the method-clause statement below:

```
method_clause ([ Head | Tail])--> public,
unit_simple_name( Head), rest_names( Tail), semicolon.
```

an argument is added to the non-terminal symbols and a list of values is passed from the body of the above rule to its head. For example, when a list such as:
IN = [main, 'Nag_Library',comma, maths_library, comma, 'Complex_Numbers', semicolon, eof]

is used as input to the head:
mainh_clause(Information, IN, OUT).
The following structures are returned:
Information = ['Nag_Library', maths_library, 'Complex_Numbers'].
OUT= [eof].

The infrastructure can contain explicit procedure calls in the body of the rule that are not part of the grammar itself but that are executed whenever a statement of the language is parsed. Such procedure calls are placed inside curly brackets to restrict the constituents accepted. A programmer or manager, for example, may be only interested in certain components of a 'method_clause' statements. In this case, he or she may test for such components. This can be done by calling the test requirement from within the
body of the rule as shown below:

```
methodh_clause ([Head | Tail]) --> method,
unit_simple_name (Head), { test( Head) }, rest_names
(Tail), semicolon.
```

where 'test (Head)' is any condition or constraint imposed on.

The infrastructure can also be used to build structure trees automatically from the parsed compilation. This structure tree can be organized in any suitable form. One can return, for example, the following structure tree when a subunit is parsed:

```
sub_unit( subprogram_body( 'Rosenbrock_Function'))
```

Or can return a structure tree in the form:

```
parent_unit (package_body ('Complex_Numbers'),
subprogram_body ('Rosenbrock_Function').
```

when the compilation unit 'Complex_Numbers', which has 'Rosenbrock_Function' as one of its stubs, is parsed. It is also possible to use any grammar rule to build structure trees. For example, the pragma rule:

```
pragma--> pragma, identifier, argument_list, semicolon.
```

can be arranged in the following form:

```
pragma (I) --> pragma, identifier (I), argument_list (List),
                {assert( pragma( I, List) ) }, semicolon.
```

To insert a fact in the knowledge base connecting the pragma name and its parameter's names. This piece of information is particularly important for the program library for supporting foreign language bodies.

The infrastructure can incorporate semantics relates to the meaning or interpretation of a word, phrase or sentence, which often necessitates a knowledge of the context.

The infrastructure can incorporate this possibility to infer such semantics from the grammar rules of a language. Procedure calls can be used within the body of

a rule to relate certain entities to others in order to make some deductions about the meaning of a statement of a programming language. The procedure calls are not considered part of the body of a grammar rule but they are executed when the associated code is parsed. Consider for example the generic specification below which is extracted from
JAVA Reference Manual [ARM83]:

```
generic
   type ITEM is private;
   type VECTOR is array (POSITIVE range <>) of ITEM;
   with function SUM (X, Y: ITEM) return ITEM;
package ON_VECTORS is
   function SUM (A, B: VECTOR) return VECTOR;
   function SIGMA (A : VECTOR) return ITEM;
   LENGTH_ERROR : exception;
end;
```

The package can be instantiated as follows:

package INT_VECTORS is new ON_VECTORS (INTEGER, TABLE, "+");

A partial meaning of the parameters can be deduced by relating the parameters of the instantiated package 'INT_VECTORS' to the meaning of the formal parameters as:

The package 'INT_VECTORS' is calling the package 'ON_VECTORS' which imports three classes:
"ITEM" of type private,
"VECTOR" of type array of ITEM,
"SUM" is a function with two parameters of type ITEM and it returns a value of type ITEM. More descriptions can also be made to associate with program units through the use of facts.

Parsing is a process of discovering whether a sequence of input characters, symbols, items, or tokens constitutes an executable program i.e it defines which symbols, strings, or words are valid sentences according to the grammar rules defining that language. The grammar generally gives some kind of analysis of the sentence into a structure which makes its meaning more explicit. Parsing a compilation needs two main tools:

A lexical analyzer: Which identifies patterns of characters in an input stream and produces a stream of words or tokens (i.e. a list of words successors, or comments). The effect of a program depends only on the particular sequences of lexical elements that form its compilation, excluding the comments, if any. the parser itself: which recognizes syntactic objects in a list of words. The input to the parser is some top-level syntactic object (a compilation) in a tokenized form which is a list of words constituting a language compilation. The parser succeeds if it finds a list of words forming the required syntactic object on the front of the list of words; otherwise it fails.

Building structures from a compilation: A compilation is a collection of one or more compilation units. According to the JAVA Reference Manual, its grammar rule is written as:

```
compilation : = {compilation_units}
```

The above grammar rule may be interpreted as: A compilation is defined as a collection of one or more compilation units. The specification of a compilation grammar rule can be translated into Prolog code recursively as follows:

```
compilation( X, Y, P, S, In, Out) :-

  compilation_unit( X1, Y1, P1, S1, In, Temp),
   compilation( X2, Y2, P2, S2, Temp, Out),
  !.% to stop backtracking.
```

Where X, Y, P and S can be expressed as lists in the forms:

```
 X = [ X1 | X2],
Y = [ Y1 | Y2],
P = [ P1 | P2] and
S = [ S1 | S2].
```

DCGs implement top-down, left-to-right recognisers or parsers. To stop left recursion, a condition must be imposed on compilations. A recursion will be stopped when each of X, Y, P and S becomes empty. In DCGs, a non-terminal symbol is expressed in terms of smaller non-terminal symbols. For example, the above compilation is expressed in terms of compilation units. A compilation unit is then expressed in terms of smaller non-terminal symbols as shown below:

```
compilation_unit( X, Y, S, In, Out) :-
          context_clause( Y, In, Temp),
          library_unit( X, S, Temp, Out).
compilation_unit( X, Y, P, S, In, Out) :-
          context_clause( Y, In, Temp),
          secondary_unit( X, P, S, Temp, Out)
```

The smaller non-terminal symbols such as context_clause, library_unit and secondary_unit are themselves decomposed into goals to find smaller non-terminals symbols and so the process continues until the lexical element level is reached. This is a top-down behaviour, as no account is taken of the input list 'In' until terminal symbols of the language are considered. As Prolog takes charge of the input and output list the clauses for compilation and compilation units will appear in the implementation as follows:

```
compilation( [X| Xs], [Y| Ys], [P| Ps], [S| Ss]) -->
        compilation_unit( X, Y, P, S),
    compilation( Xs, Ys, Ps, Ss),
        !.% to stop backtracking.

compilation( [], [], [], [])          --> [].
compilation_unit( X, Y, S )              -->
                        context_clause( Y),
                        library_unit( X, S).

compilation_unit( X, Y, P, S)-->
                        context_clause( Y),
                        secondary_unit( X, P, S)\fP.
```

The predicate compilation above has four arguments, each of type "list". The first one the compilation units defined within the compilation. The kind descriptions of those compilation units are specified for every compilation processed.

The second list represents with-clause dependencies that belong to every compilation units defined in the processed compilation.. The third argument lists parent units for each compilation unit. The fourth argument contains program unit identifications, pragmas, overloading, renaming, procedure calls that were defined in a compilation unit. In addition other entities could be extracted from a compilation unit by including the user's own specific requirements in the body of the relevant grammar rule.

The above items of information extracted from a compilation, we feel, are sufficient for building the required knowledge base for a programming language, since other information can be derived from this knowledge base. The four lists in the compilation clause will have different lengths in most practical applications. As an alternative, they could be represented as a list of quadruples, however this would offer no obvious advantages.

The analyzer also specifies the constituents of each compilation unit with a partial semantic description of each constituent.

A full analysis of compilation units is established in the following section. Analysis of Compilations and Software Configuration Management Analysis of compilations has a great effect on easing the understanding of software. It will provide the programmer or manager with valuable information concerning the entities of a compilation.

The kind of information required depends upon the needs of the manager or programmer. He or she may need to summaries several packages by tabulating the numbers of types, subprograms, exceptions and inner packages in each. Alternatively a list of all the type names in a set of packages, each with a cross-reference to the package containing it may be needed. Another user may want a tabulation of subprograms from their parameter types, sorted by type, disregarding order of parameters. However, we are interested in extracting the necessary information for program library, software configuration management and software construction purposes. Such information, particularly relating to module dependencies, is important for sound configuration management.

The information, which has been extracted from a source code, is represented in the knowledge base as facts. These facts are best represented as follows:

Unit (UnitName, Description, VersionNumber, Type, List of Subunits).

Other information is also extracted from a compilation such as name of procedures and functions with their type of parameters, filenames, pragma, code which does not comply with the language grammar rules and so on.

Error behavior of the implemented framework: The analyzer checks that each statement of the submitted compilation is syntactically correct; this means a compilation is successfully analyzed when it is coded according to the language grammar rules. When a compilation has syntactic errors the analyzer stops and reports a failure.

The disadvantages of the implemented parser, particularly when other tools call it, is that it does not show the place the error occurred. When the analyzer is invoked directly in a way similar to invoking a compiler it shows the place where the error occurred and the code which does not comply to the language rules is returned as output through the fact 'end-of-file'. Consequently, the analysis process is cut and a failure is reported.

Program analysis and decision support: The information extracted from the source code of a language can be used for a number of purposes. These may include:

- Data query
- Abstraction
- Rogram evolution and maintenance

Program analysis and its advantages for decision support mechanism are discussed in turn in the following sections.

Data query: Many facts are extracted by the analyzer and inserted into the knowledge base that acts as a pool of information. By means of facts and rules many queries can be performed to assist a programmer or a manager determining and reasoning about many objects of the knowledge base. Data query assists in reasoning about a

program text, source code, constituents, type of parameter, . . ., etc. This type of reasoning helps not just in understanding a structure of a software system but it can demonstrate its reliability (i.e., its correctness with respect to a specification). Relying on module testing and system testing alone will not prove the absent of bugs as Dijkstra reports (1976): "program testing can be used to show the presence of bugs, but never to show their absence!". By means of rules and the above facts queries such as the following can be answered:

- How many versions does 'MATHS_LIBRARY' have?
- What components does 'MATHS_LIBRARY' depend on?
- What versions of 'MATHS_LIBRARY' were created before April 30, 2005 and where are they stored?
- What is the component that contains a subprogram function which has one formal parameter of type complex and it also contains a function that returns a value of type real?

The last query is probably useful for classifying components for re-usability.

Abstraction: Abstraction is an important principle of modern software engineering. It is a necessary element in software management and software reusability in which the essential information that is relevant to a particular purpose can be exposed. Abstraction is an indication for productivity. That is, raising the level of abstraction may increase productivity.

The program analyzer abstracts the internal structure of a module including information about renaming and overloading. In addition, program units
are represented in a form indicating to which category each belongs, i.e., package, procedure, function or task. It is possible to have a full description of the formal parameter types of a program unit but this is abstracted through the visible part of a compilation unit.

Program development and maintenance: Other useful information, for example, for program maintenance and evolution, can also be represented or deduced. This includes: Stubs of compilation units as shown below:

```
stubs_of( 'Complex_Relations8936091245',
    package_body('Complex_Relations'),
    [subprogram_body( read_in),
    subprogram_body( calculateimp),

    . . . ] ).
```

Where the first argument 'Complex _ Relations 8936091245' of the fact stubs_of/3 represents a version of the component 'Complex_Relations'. 'Complex _ Relations' is a package body represented by the second argument of stubs_of/3. This version specifies its own list of stubs as shown by the third argument of stubs_of/3 (i.e., [subprogram_body (read_in), subprogram_body (calculateimp), . . .]). components that a compilation unit depends upon:

```
 method_clauses( direct_io8921181443, [io_exceptions,
'SYSTEM'], generic_specification( direct_io) ).
```

where 'direct_io8921181443' is the version number of the generic specification 'direct_io' (i.e., generic_specification(direct_io)) which depends on the units shown by the second argument of the fact with_clauses/3 (i.e. [io_exceptions, 'SYSTEM']). This facility enable a software developer to specify dependencies of each version. components that depend upon a compilation unit: this can be found by querying the knowledge base i.e

```
listall_units_madeoutofdate( decl89310114321, List).
List = [ 'Complex_Relations',
     'Manipulate_Vectors',
     'Rotate_Axes',
     . . . ,
     optimise].
```

Where 'decl89310114321' is a version number of a component that can easily be traced. There are a number of software components that depend upon the above version. These components are computed using the predicate listall_units_madeoutofdate/2. The list will be ['Complex_Relations', 'Manipulate_Vectors', 'Rotate_Axes', . . . , optimise].

The above types of automatic extraction would expect to reduce errors compared with other systems based on makefile where dependencies are recorded manually. This degree of automation provides a developer with information that can be used to check consistent and inconsistent compositions and then can significantly increases the probability of producing a consistent and complete configuration with enough information on each compilation unit. It also assists in testing modules and integration.

The above information, particularly that concerned with the interior structure of modules and their relationships, is necessary for the design and management of re-usable components. It is also useful for software construction as well as enhancing understandability. The proposed Program library and SCM system rely heavily on the correctness of the dependency and consistency information supplied by the

parser. Information recorded during the course of parsing is also useful for classification in software re-use and automatic (re)generation of products. It is also valuable in identifying and accessing individual objects of compilations such as procedures, functions, stubs, exception, renaming, type of parameter, etc).

The results of manipulation data queries are presented as relationship tables where a range of operations is provided including selection, projection, union and other operations provided by relational data bases (see for instance: Oracle, Ingres manuals).

CONCLUSION

I have proposed, in the study, a way of analyzing programs using a program analyze written in Prolog. Prolog is particularly suitable for both prototyping and for writing natural and programming language parsers. The JAVA analyse described above encompasses two main activities. These are:

- Translation of JAVA source code into its lexalized elements and
- Applying the JAVA syntax rules to the lexalized version (the analyzer).

The original motivation for building the analyzer was to provide better visibility of JAVA programs and as a basis for building portable JAVA program libraries and to enhance decision support mechanisms for configuration management and version control tools, software construction and software reusability.

REFERENCES

Colmerauer, A., Les Grammaires de Matamorphose, 1975. Groupe d'Intelligence Artificielle, University' de Marseille-Luminy. Appears as Metamorphosis Grammars. L. Bolc (Ed.), Natural Language Communication with Computers, Berlin, 1978.

Dijkstra, E.W., 1976. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, NJ.

Frei, H.P., D.L. Weller and R. Williams, 1978. A Graphical-based Programming-Support System, SIGGRAPH-ACM Vol. 12.

Jennings T.J. and B.A. Carr'e, 1989. A Subset of Ada for Formal Verification (SPARK), Proceedings of the 7th JAVA UK Conference York (AdaUser, Supplement), pp: 121-126.

Kowalski, R.A., 1974. Predicate Logic as Programming Language, Proc. IFIP 74, Stockholm.

Malik, D.S. and P.S. Nair, 2003. JAVA Programming: From Problem Analysis to Program Design, Thompson Publisher.

MALPAS, 2005. http://www.advantage-business.co.uk/products-malpas.asp, accessed.

Nassi, I. and B. Shneiderman, 1973. Flowchart Techniques for Structured Programming, SIGPLAN Notices of the ACM, pp: 12-26.

Ng, N., 1978. A Graphical Editor for Programming Using Structured Charts, IBM Research Report RJ2344 (31476) 9/19/1978, IBM Research Laboratory, San Jose, California.

Pereira, F.C.N. and D.H.D. Warren, 1980. Definite Clause Grammars for Language Analysis --- A survey of Formalism and a Comparison with Augmented Transition Networks, Artificial Intelligence, 13: 231-278.

Reid, P., 1983. The Use of diagrams and colours in the display of Ada programs, Ph.D Thesis, computer science department, York University, UK.

Wagner, H., Visualization of Structures and Traces of Software Systems (Tool AURUM), in Practice in Software Javaption and Maintenance, R. Ebert *et al.* (Ed.), North-Holland, pp: 167-180.

Weinberg, G., 1971. The Psychology of Computer Programming, Van Nostrand Reinhold, New York.