AN EXTENDED SEMANTIC ANALYZER FOR ADA'95

by

Mehmet Özgür Karahan

BS. in C.M.P.E., Boğaziçi University, 1998

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science
in
Computer Engineering

112101

Boğaziçi University

2001

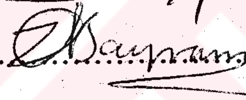
AN EXTENDED SEMANTIC ANALYZER FOR ADA'95

APPROVED BY:

Assoc. Prof. Can Özturan ....
(Thesis Supervisor)

Assoc. Prof. Levent Akın ....

Assoc. Prof. Zeki Bayram ....



T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

DATE OF APPROVAL 26.07.2007

ACKNOWLEDGEMENTS

I would like to express my appreciation to Assoc. Prof. Can Özturan for his helpful advice and guidance throughout this study.

I would also like to thank Assoc. Prof. Zeki Bayram for his comments and encouragement.

ABSTRACT

AN EXTENDED SEMANTIC ANALYZER FOR ADA'95

An *extended semantic rule* is a rule which we expect a program in a particular language to obey in addition to the standard semantic rules enforced by the compiler for the language. Such extended semantic rules may be necessary to ensure that software has certain software quality attributes. In this work, an *Extended Semantic Rule Set (ESRS)* that contains certain semantic restrictions for Ada'95 programs is defined and the effectiveness of that rule set to increase software quality attributes of Ada codes is examined. The work done in this thesis also describes the design and implementation of a software tool, a SemantiC Analyzer (SCA) that checks the compliance of a given Ada code to the Semantic Rule Set. The effectiveness of the Semantic Analyzer is discussed on sample input and its output is analyzed for a large set of previously compiled Ada code.

ÖZET

ADA'95 İÇİN GENİŞLETİLMİŞ ANLAM ANALİZÖRÜ

Genisletilmiş anlam kuralı, bir yazılım dilindeki standart anlam kurallarından farklı olarak, uygulandığı yazılımın kalitesini arttırmak için tasarlanmış bir anlam kuralıdır. Bu tezde Ada yazılım dili için bir anlam kuralları kümesi tasarlanmış ve bu kuralların yazılım kalitesi üzerindeki etkinliği tartışılmıştır. Tezde ayrıca, Ada yazılım kodlarının bu anlam kurallarına uygunluğunu tesbit edebilen bir yazılım aracının geliştirilmesi anlatılmıstır. Bu yazılım aracının etkinliği bir örnek üzerinde tartışılmış, ayrıca geniş bir Ada yazılımı kodu üzerinde denenmiştir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF SYMBOLS/ ABBREVIATIONS x
1. MOTIVATION 1
1.1. Syntax and Semantics 2
1.2. Problem Statement 3
1.3. Proposed Solution 3
1.4. Related Work 3
1.5. Contributions 5
2. SEMANTIC RULE SET 7
2.1. Extended Semantic Rule Set (ESRS) 8
3. IMPLEMENTATION OF THE SEMANTIC CHECKER 15
3.1. The Scanner 15
3.2. The Parser 16
3.3. The Semantics Rule Checker Functions 19
3.3.1. Rule 1 20
3.3.2. Rule 2 21
3.3.3. Rule 3 22
3.3.4. Rule 4 23
3.3.5. Rule 5 24
3.3.6. Rule 6 24
3.3.7. Rule 7 25
3.3.8. Rule 8 26
3.3.9. Rule 9 26
3.3.10. Rule 10 27
3.3.11. Rule 11 27
3.3.12. Rule 12 28

3.3.13. Rule 13 29
3.3.14. Rule 14 29
3.3.15. Rule 15 30
3.3.16. Rule 16 30
3.3.17. Rule 17 31
3.3.18. Rule 18 31
3.3.19. Rule 19 32
3.3.20. Rule 20 32
3.3.21. Rule 21 32
4. OUTPUT ANALYSIS OF THE SEMANTIC ANALYZER 33
4.1. A Sample Run 33
4.2. ATC Software Analysis by SCA 34
4.3. Resource Allocation and Time Analysis 37
4.3.1. Minimum Memory Requirement 37
4.4. Multi-Threaded SCA 43
5. CONCLUSION 45
APPENDIX A: THE REGULAR EXPRESSION SET OF LEX INPUT FILE 46
APPENDIX B: A SAMPLE FROM PARSER DESCRIPTION FILE 47
APPENDIX C: EXAMPLE INPUT FOR SCA 49
APPENDIX D: A SAMPLE OUTPUT FROM SCA 53
REFERENCES 55

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

LIST OF FIGURES

Figure 3.1. SCA component and process diagram 15

Figure 3.2. Syntax tree node type declaration 17

Figure 3.3. A CFG grammar rule representation for Yacc parser generator 18

Figure 3.4. 'If' statement rule for parser 19

Figure 3.5. The 'type' and 'subtype' declaration rules for parser 21

Figure 3.6. The 'subtype' declarations with and without range constraint 21

Figure 3.7. The nodes sequence up to the goal symbol 21

Figure 3. 8. The 'value' and 'comp_assoc' node structures....................................22

Figure 3. 9. 'Others' keyword usage in aggregate initializations.............................. 22

Figure 3.10. The node structure of aggregate initialization and 'others' usage..............23

Figure 3.11. The short circuit form and grammar link with expression.......................23

Figure 3.12. The expression and negation linkage on parser definition rules................ 25

Figure 3.13. The 'case' statement and 'others' keyword linkage in Ada'95 grammar......26

Figure 3.14. The 'exit' statement in Ada'95 grammar..26

Figure 3.15. Grammar definition of subprogram body in Ada'95............................ 27

Figure 3.16. The 'goto' statement 27

Figure 3.17. The nested control structure node definitions.............................. 28

Figure 3.18. The parameter modes in Ada'95... 28

Figure 3.19. The 'Subprog_spec' node...29

Figure 3.20. The with and use statements grammar rules in Ada'95 31

Figure 3.21. Raise statement and its grammar 31

Figure 3.22. Exception declaration rule and its grammar 32

Figure 3.23. 'Generic' declarations in subprograms 32

Figure 4.1. Error distribution per rule 34

Figure 4.2. Input file sizes 38

Figure 4.3. Generated token number from input file 38

Figure 4.4. Theoretical minimum and real syntax tree size 41

Figure 4.5. The ratio of syntax tree size and input code size 42

Figure 4.6. The ratio of node and token sizes 42

Figure 4.7. Execution time of SCA on different OS 43

Figure 4.8. Execution time of multi-threaded SCA on different OS 44

LIST OF SYMBOLS/ ABBREVIATIONS

F_{size}	File size in bytes
Kb	Kilo bytes
N_k	Number of nodes at level k
ATC	Air Traffic Control
CFG	Context Free Grammar
CPU	Central Processing Unit
DOS	Disk Operating System
ESRS	Extended Semantic Rule Set
KLOC	Kilo Lines of Code
OS	Operating System
POSIX	Portable Operating System Interface
SCA	SemantiC Analyzer

1. MOTIVATION

Ada programming language is proven to be an effective software tool [1] for various systems: Safety systems (such as air traffic control), security systems (to protect digital information) or financial systems (cash dispensers). As the complexity of these systems grows, so do the demands for improved techniques for their production. For critical systems, there is a need to ensure that they have certain properties, and this can only be achieved by the analysis of the software with static analysis and dynamic testing. Unfortunately, dynamic analysis of software, which is done by executing the software for every possible step, is prohibitively expensive. It is necessary to analyze every instruction in the program during dynamic analysis to ensure the integrity of the data or control flow.

Static analysis and dynamic testing are two complementary ways of comparing an implementation with its specification. Dynamic testing allows the most direct comparison between implementation and operational requirements. Testing every possible execution sequence of instructions exhaustively may not be possible, and testing in general cannot show the absence of errors. In contrast, the program source size rather than the number of its possible execution sequences determines the cost of static analysis. In principle static analysis can be complete in showing absence of errors of some classes depending on the specification of two main elements: A derived model of the source text, and a rule set that defines the errors during inspection of the derived model.

When the software life-cycle is considered, early detection of errors by static analysis in the Ada software codes decreases the amount of time, needed to produce the software system, by decreasing the cost of error repairs. The success of the software tool, which computerizes this process, depends on the design of the rule set and the derived model of the software code.

1.1. Syntax and Semantics

Error detection by static analysis starts with a code inspection process, in which the

syntax and semantic of the language is analyzed. A modern compiler performs that analysis in order to synthesize the machine-language equivalent of the source program. The steps taken by a compiler for this goal are scanning, parsing, semantic routines check, optimizing and machine code generation. In the first step, compiler views the source program as a stream of characters. It begins the analysis of the program by dividing the stream of characters up into 'tokens' (identifiers, integers, reserved words, delimiters, operators, etc.). Given a precise description of a programming language grammar using a 'context-free grammar' (CFG), the compiler uses a parser to read and group tokens into meaningful units as described in the CFG. In the process, the parser identifies, and may correct, syntax errors. As portions of the syntactic structure of the source program are recognized, the parser will call semantic routines to generate target code, or the parser will build a syntax tree which will be input to the semantic routines once the tree is constructed (the source program has been completely parsed).

The basis of every language is the language definition, which is composed of syntax and semantics. The 'syntax' of a language defines what sequences of symbols are legal and it is related with the spelling rules for the basic symbols (keywords, special symbols, identifiers, etc.). A common property of the rules defining syntax is that they do not affect the meaning of the program being represented.

The 'semantics' of a language describes the meaning of a program in terms of the basic concepts of the language. Static semantic rules are a set of restrictions that determine which syntactically legal programs are actually considered to be valid. For example, it might state that all variables must be declared before being used, the operands must be type-compatible, and so on. Dynamic semantics specifies what a program actually does or what it computes. Clearly this involves a specification of the meaning of data types (basic and composite), the operations that can be performed on them, the notion of type equivalence, scoping, when and how variables are bound, and so forth.

1.2. Problem Statement

For specific kinds of application areas, it is sometimes necessary to have a stricter set

of rules that programs need to conform to than those enforced by the Ada language. Safety-critical software, where the lives of people might depend on the correct functioning of the software, is an example of such an application area. Extended semantic rules can result in less error-prone and hence more reliable programs. Furthermore, just as the standard semantic rules are enforced automatically by Ada compilers, it would be better to ensure conformity of Ada programs to extended semantic restrictions automatically, without the need for manual code checking.

1.3. Proposed Solution

An extended rule set is defined and justified for safety critical applications written in the Ada'95 language. An extended semantic analyzer is also implemented that checks conformity of Ada programs to the extended set of rules.

1.4. Related Work

'Static analysis', is the function of a static analyzer to determine the behaviors of a program that are not intended by the programmer. Allen and Cocke [2] were among the first to study the problem in general and Gary Kildall [3] was the first to apply lattice theory to static analysis, bringing a mathematical basis to static analysis. But it was the ground breaking work of Cousot [4] and that finally provided a solid theoretical foundation for static analysis, which is called 'abstract interpretation'. Abstract interpretation is based on the idea of viewing the analysis of a program as an abstraction of the program's behavior. The generality of this view allows abstract interpretation to abstract the concept of analysis altogether, providing a single framework that encompasses all the standard analysis techniques.

The foundation on which abstract interpretation is built is a formal understanding of how a program should behave. This discipline of formality benefits from the programming language implementation and in general, as it precisely defines what every program should do. This, in turn, makes it possible to prove that an analysis is correct. As analyzers and the program properties they seek to determine become subtler and more complex to compute,

this proof of correctness will become increasingly important. Fortunately, the framework of abstract interpretation comes with a guarantee of correctness; only the individual analyses need to be proved.

Schaeffer [5] statically analyzed the Ada programs to propose a safer exception handling mechanism. Bacon and Sweeney [6] investigate the ability of static analysis to improve C++ programs by resolving virtual function calls, thereby reducing compiled code size and reducing program complexity so as to improve both human and automated program understanding and analysis. Sundaresan [7] analyze same problem for Java language. Steindl [8] implemented a static semantic analyzer for object-oriented language Oberon to implement program slicing methodology. Baer [9] is designed a semantic analyzer to resolve the layer structure of C programs to enforce a layered structure during forward engineering and the extraction of layered structure from pre-existing C language source.

Researchers and commercial groups implemented various automatic verification tools for coding convention compatibility of program sources based on static analysis. For C the 'CodeCheck' tool is implemented and uses a set of rules that focuses on the portability of C programs to other hardware platforms. This tool can verify the portability of a C source code between cross platforms like DOS, Unix, Microsoft Windows, and other 64-bit machines. Another commercial product for C++ is 'CodeFix' and it does same kind of analysis on C++ grammar and syntax for various platforms. For Java language, a tool called 'Jlint', which is implemented by Knizhnik [10] to verify the semantic verification of Java source for a set of Java coding convention rules.

Several researchers analyzed automatic generation of code scanners for various programming languages. Horspool and Levy [11] implemented 'Mkscan' and Mossenbock [12] designed 'Alex' to generate automatic scanners. Grosch's [13] 'Rex' and Nicol's [14] 'Flex' tools are other examples for automatic scanner generators. Parsing process can also be automatized by a parser generator depending on a CFG description of the programming language grammar. One example to the parser generator for CFG grammar is implemented by Bunke, and Haller [15]. Another example is 'Bison', which is implemented by

Donnelly, and Stallman [16], which generates a parser in C language. A functional parser generator is implemented by Uddeborg [17].

1.5. Contributions

In this thesis, a software tool is designed and implemented for Ada programming language, which is used to increase the software quality of Ada software systems. There is not any Ada Semantic analyzer in software world, which is based on the static semantic analysis of code. This is due to the fact that there is not any standardized version of semantic restrictions to vast semantic space of Ada language and there are lots of different compiler versions of Ada language.

The Semantic Rule Set of SCA contains semantic restrictions for Ada grammar and it is designed to decrease the error generation risk during program execution. Even this rule-set is not checked by a computerized tool, it can be a guideline to Ada programmers during coding. So ESRS includes software rule items that can help to programmers and functionality testers to detect and identify certain resource of errors.

The methodology used in SCA to identify the semantics of Ada can be used for other software languages such as C, C++ or Java. These languages are very popular among programmers and they are also used in different application areas. Large project codes, which are implemented with these languages, can be investigated with tools like SCA. It is also possible to use these languages with Ada and the program interfaces between different languages can be checked with SCA like tools in order to decrease the production time and cost.

The SCA rule set and its implementation is extendable to cover further semantic rule descriptions and needs. Depending on the requirements and circumstances, new rules can be added or excluded from the set to increase the elasticity of the tool. This might be possible especially when compiler vendors implement or apply new technologies to Ada compilers and previous application software might not be possible to be updated.

The SCA tool generated in this thesis can be used for Air Traffic Control (ATC) applications because most of the semantic restriction set targets the problems, which are usually detected on ATC applications. Ada is very popular among ATC software area and this tool increases the validity and safety of ATC programs.

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

2. SEMANTIC RULE SET

In this section, semantics rule set is described. The rules are designed to aim one or more particular software quality targets among five main categories, which are portability, readability, maintainability, safety and execution efficiency [18,19].

Portability is the ability of software to run on a new platform and/or compile with a new compiler. Dahlstrand [20] describes portability as "having the same meaning of software to compiler and hardware". Mooney [21] interprets portability as a measurable quantity and expresses it as the percentage of unchanged codes during porting of an application source code.

Readability [22,23] is the ability of software to be easily understood in functional behavior for an external user. Here external user means any user who has to do some new development with the code and who has no clear idea about the usage of the code.

Maintainability is the ability of software to be evolved easily in all circumstances such as additional functionality necessities and removal of some residual errors. Cheaito [24] defined maintainability as a measure of the ease with which a software system can be maintained.

Safety [25] is the ultimate goal for most of the software systems. First aim is having no errors during the software is run. For critical systems like ATC, this needs a further step, if an error occurs, system should be left in a non-life threatening status. Ada has additional facilities to control the exceptional cases and strong memory space disposal operations. Program coding habits sometimes become potential error source that threatens the safety or reliability of software system. The rules will prohibit certain user coding schemes that may be potentially dangerous.

Execution efficiency is another important criteria for software programs. Efficiency can be restored in a program in a way such that the desired functionality can be achieved

by optimum usage of system resources and processors.

2.1. Extended Semantic Rule Set (ESRS)

Rule 1: Usage of predefined types integer and float should be avoided. Definitions of floating point, fixed point or integer types should include the range constraint.

Ada language facilitates the usage of predefined types *float* and *integer*. Redefinition of the predefined types removes the machine or platform dependency on the representation of these numbers. For example the float type is defined by several attributes: The number of decimal digits in the mantissa, largest exponent value, the smallest positive value, the difference between number '1' and the next number above it (epsilon). These attributes should be redefined if the target compilation environment has a different byte size or order. Redefinition of the predefined types is possible in Ada and done by assigning new values to these attributes depending on the target hardware. Redefinition of the predefined type increases safety and the portability of the software system. The semantic analyzer **SCA** prints a warning message every time the predefined types are used in the code except in the redefinition statements of these types.

Rule 2: Named association should be preferred to positional association.

In the Ada function and procedure (subprogram) calls, named association usage increases the readability of the code for argument checking. Named association also increases safety by decreasing the risk of wrong argument and value usage because all argument names are listed before its value is entered. This kind of subprogram calling style informs the programmer about the argument usage and increases readability. All arguments in the subprogram calls without an associated argument name will be reported by the SCA to programmer.

Rule 3: Use explicit intervals instead of keyword "**others**" when initializing an aggregate.

This style of programming makes sure complex object is correctly initialized. A warning message will be printed for every usage of others keyword in an aggregate initialization and increases safety.

Rule 4: Use short circuit form of logical operations systematically in all cases where the operands may not be mandatory for proper execution of program.

This style of programming increases the execution efficiency during the logical operation value calculation. Short circuit formed logical expressions are evaluated rapidly if the rest of the operands are not necessary to evaluate the expression. Ada enables usage of shot circuit logical operators to skip the calculation of the unnecessary operand values. A warning message will be printed for every logical expression where the short circuit form is not used.

Rule 5: Ada garbage collection library procedure **Unchecked_Deallocation** usage is prohibited.

By using this function, any previously allocated memory is disposed and internal system is informed that this memory piece is ready for further use without a security check. This can lead to dangling reference and subtle program errors. Compiler should left all control to user to avoid in use memory disposal so usage of this procedure increases the erroneous execution risk. It is thus prohibited. A warning message will be printed for every usage of this procedure.

Rule 6: Expressions more than one negation should be avoided.

The programmer should easily understand the expression meaning and multiple negation usage distorts the understandability of the meaning. Programmer is warned for usage of multiple negated expressions.

Rule 7: When using a case control structure avoid using **others** keyword.

Case control structure handles multiple selection problems and it consists of a value and several choices. If an invalid value is checked by a case control that the program is not expecting, the 'others' field captures the unexpected value. So program security is decreased and a potential leak point for erroneous values is opened. A warning message will be printed for every 'others' field in a case control structure.

Rule 8: If used, exit statement should only be used with the name of the loop, which the exit is used for or with a 'when' statement.

First usage of 'exit' statement removes the risk of erroneous exit in a loop and increases the readability of the code. Second usage of 'exit' specifies explicit conditions for changing the execution sequence of the code and it is visibly specified for the program reader. A warning message will be printed for every 'exit' statement that is not used with the specified conditions.

Rule 9: 'Return' statement must be used only once in a function.

Multiple return points mean multiple exit points and this is not desirable if new code is added in the future. A warning message will be printed for every other return statement in a function.

Rule 10: 'Goto' statement must not be used.

'Goto' instruction usage leads to "Spaghetti code", which is very hard to trace and verify program correctness. An error message will be printed for every 'Goto' statement.

Rule 11: Nested structure of control should be limited to four levels.

Block structures in Ada are used for two reasons: To prevent other parts of the program from using particular objects (locally declared items) and to provide local exception handler. Nesting of blocking should be restricted in order to easily manage the specific part of the code. An upper level to the nested structuring eases the control and

increases maintainability of the software. A warning message will be printed for higher levels of nesting.

Rule 12: The formal parameter mode 'in' must explicitly be used. Therefore the default parameter mode is not allowed.

Ada has four argument modes at the procedure or function declarations. These are 'in', 'out', 'in out', and 'default' modes. If an argument is expected to be updated in the body of the subprogram that it is named as an out argument. If it is not updated then it is an 'in' argument. Usually if nothing is specified compiler takes the 'in' mode as default mode. Specifying explicitly the 'in' mode for arguments increases the portability of the code because compiler dependency on default mode interpretation is not removed. A warning message will be printed for all unspecified parameter modes in subprogram declarations.

Rule 13: All parameters of a subprogram with the 'out' mode must receive a value before returning to the caller.

The 'out' mode is used to define arguments, which are given permission to be updated in the body of a subprogram. Any 'out' mode argument that is not updated in the subprogram reflects a semantic error. Usage of a non-updated argument may cause fatal errors during execution. A warning message will be printed for all unchanged 'out' parameters in a subprogram. The semantic error removal increases the run time safety of the program.

Rule 14: Conventional meaning of overloaded operators should be preserved.

In Ada the functionality of all mathematical operators can be overloaded with new functionality. For example a plus sign indicates addition and the overloaded functionality should preserve this meaning to increase the readability of the program source code. A warning message will be printed for all overloaded operators.

Rule 15: Subprogram arguments should not be initialized in the header part.

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

The initialized argument values in a subprogram declaration are not visible to the programmer if the source code is located on a separate file. Ada compiler does not force programmers to locate an argument value for initialized arguments. So if a value is not specified for the initialized argument then compiler uses the value that is specified at the declaration. It is better to leave control to the programmer when the arguments are used during subprogram calls. A warning message will be printed for all initialized parameters at declaration.

Rule 16: An initialization function should be written to control Elaboration sequence.

Every execution of a main program in Ada activates elaboration of certain library units into the program at run-time. During compilation the library packages included into the code by usage of 'with' clauses and every package name specified in the 'with' clause is elaborated. But what happens if there are other packages to elaborate, in the elaborated packages. To control this in Ada, there is a compiler director command, which is called 'pragma elaborate'. A package name specified within the pragma is forced to be elaborated. An 'Init_Packagename' function can control the sequence of package elaboration for every package. A warning message will be printed for all packages without an Init_Packagename procedure.

Rule 17: All package names included with a 'with' clause should be supported with a 'use' clause to decrease the length of package names used in the code.

The packages elaborated with a 'with' clause can be referred by the absolute naming notation. This notation specifies all the library hierarchy down to the name of the referred unit: library.package.class.procedurename. If the library or the package absolute name is specified in a use clause, then the unit name can be referred in the program without the editing the part mentioned in the 'use' clause. The shorter unit names increase readability. A warning message will be printed for all packages with a 'with' clause but not supported with a 'use' clause.

Rule 18: Do not use predefined exceptions with raise statement.

Ada has five predefined exceptions: 'Constraint_error', 'numeric_error', 'program_error', 'storage_errror' and 'tasking_error'. It is not secure to raise predefined exceptions because it may not be possible to distinguish between these and real system exceptions during run time especially if when they are raised by the system because of another reason. Detecting the system-generated exceptions shortens the determination period of the run time errors and enables fast debugging to increase system safety and maintenance. A warning message will be printed for all raised predefined exceptions.

Rule 19: A subprogram declared on a specification package must only use exceptions in its body that are defined or renamed in its specification package.

This kind of exception usage avoids the exception contamination in the system and enables fast debugging for safety establishment during functional testing because the generated exceptions can easily be traced. A warning message will be printed for all exceptions that are raised and not declared in the package.

Rule 20: Numerical values should not be used in the code except constant or type definitions.

It is not easy to remember the numerical values and the value they represent. It is better to name every numerical value in the code to increase the readability and maintainability. Semantic analyzer print a warning message every time there is a numerical value in the code is it is not used in a type or constant declaration.

Rule 21: Generic units should not be declared in subprogram.

The instantiation of a generic body in a subprogram is done each time this subprogram is called and this may lead to a memory deficiency if it is executed several times. Declaration of generic unit should be moved to other parts of the code to increase

system safety. An error message will be printed for all generic type declarations in a subprogram.

3. IMPLEMENTATION OF THE SEMANTIC CHECKER

The Semantic Analyzer (SCA) for the defined semantic rule set is implemented with C programming language on Linux operating system platform. SCA has a scanner generated by Lex and has a parser created by Yacc. The general component structure of SCA is displayed on Figure 3.1, where the process sequence of the components is also visualized.

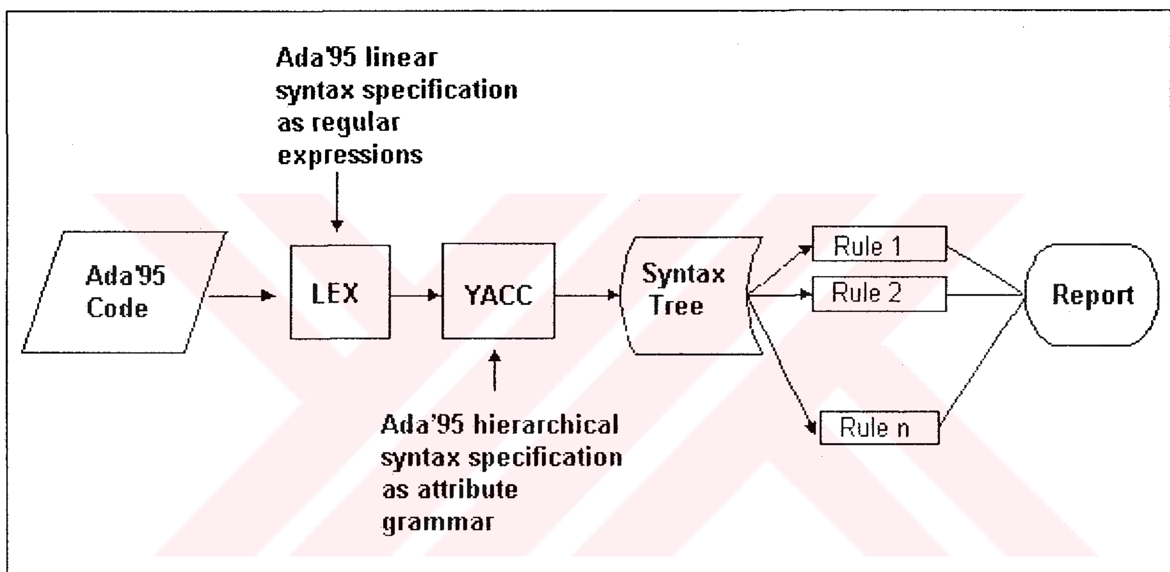


Figure 3.1. SCA component and process diagram

3.1. The Scanner

The scanner generation process initially requires a token description file and this file is analyzed by Lex to create the scanner code in C language. A sample from the Lex input file is listed in Appendix A and it should be compiled with other software component parts of the SCA. The regular expression set, which identifies the Ada tokens, is implemented by Ada Home software group [26].

The scanner is the first component of the SCA and the character stream of the input Ada file is decomposed into tokens, which are meaningful objects for Ada grammar. Since the scanner is the first processing stage, it initiates the calls that invoke the next processing

unit: Parser.

3.2. The Parser

As the scanner generates Ada tokens, it sends them to the parser by labeling them with the proper identifier. The tokens contain two types of data: First the type of the token and depending on this type, a related data. The reserved words of the Ada language is identified as an integer value, whereas the character strings that are used to name variables, functions, or other entities as four different types: 'char_lit', 'identifier', 'char_string', 'numeric_lit'. That group of tokens is entered by the user and they are not part of Ada reserved words. SCA needs the name of functions; type declarations and other data; such as the line numbers where the tokens are located in the code, to generate a meaningful and understandable report.

The Ada'95 grammar is edited in the parser definition file, in a way such that the parser generator can interpret them. There is an example code sample in Figure 3.3, and the grammar rules are written in CFG rules form with an additional C block that is executed upon reduction. Reduction happens when scanner finds the tokens specified on the right-hand side. During this reduction process the goal is to create an abstract syntax tree representation of the input source code by executing the C code. The Ada grammar set is retrieved from the Ada'95 grammar specification, which is defined by Ada Consortium [26]. An example section from the parser definition file is listed in Appendix B.

The syntax tree keeps the whole Ada program by keeping every token and reduced rule headers on the nodes and leaves of the syntax tree. The tree node is defined as a C structure, which also keeps some additional information about line number where the token is found in the input file or the value of the function name as a character string and so on, to generate a semantic error report.

In Figure 3.2, the declaration of the syntax tree node is listed. The 'type_of_node' field is used to keep the node type. This value is determined by the Ada grammar rule, which is reduced as this node is created. For example if this is the root node of the tree,

then the ‘type_of_node’ will be ‘goal_symbol’ to reflect that is it created when the goal symbol rule of the CFG grammar is executed. All node types used in the grammar are retrieved from the rule headers used in the Yacc parser definition file. The ‘char_val’ field of the node is used to keep the character string values used to represent or identify the user-defined parts of the code: a function name, identifier or type names.

```
1:                  typedef struct NODE {
2:                        int type_of_node;
3:                        char *char_val;
4:                        int number_of_rule;

5:                        int number_of_nodes;
6:                        int linenumber;
7:
8:                        struct NODE *nodes[MAX_NUMBER_OF_NODES];
9:                        struct NODE *up;
10:                 } NODE;
```

Figure 3.2. Syntax tree node type declaration

The CFG rules can have multiple choices when a node is created or reduced to other rules. In Figure 3.3, there is an example for this situation. The ‘decl’, which represents any declaration in Ada language, can be retrieved by reduction of ‘object_decl’, ‘number_decl’ or ‘type_decl’ nodes. These nodes are created if there is an object declaration, a number declaration or a type declaration exits in input Ada code, respectively. During the syntax tree generation, the number of the grammar rule, which is reduced for the creation of ‘decl’ node is understood by checking the ‘number_of_rule’ field of the node structure. In Figure 3.3, the ‘decl’ node has ‘number_of_rule’ field that is equal to 1, if it is retrieved from an object declaration, 2 for number declaration and 3 for type declaration and so on. These values are entered to the creator or reduced node by making a call to the ‘Create_Node’ function. This function accepts four arguments: First the type of node which is to be created, the string value associated with this node, the rule number that is used for reduction, the number of nodes that will be the children of new node on the tree. After a new node is created from the reduction of nodes, they will be connected to the new node as children nodes.

```
1:          decl   : object_decl   {
2:                       $$ = Create_Node   ( _DECL, "", 1, 1) ;
3:                       Add_One_Node ( $$, $1 );
4:                 }
5:
6:                 | number_decl {
7:                      $$ = Create_Node   ( _DECL, "", 1, 1) ;
8:                      Add_One_Node ( $$, $1 );
9:                 }
10:
11:                | type_decl    {
12:                      $$ = Create_Node   ( _DECL, "", 1, 1) ;
13:                     Add_One_Node ( $$, $1 );
14:                  }
```

Figure 3.3. A CFG grammar rule representation for Yacc parser generator

In Figure 3.4, there is an example CFG rule for node creation. Here, a new node is created to represent 'if_stmt', which corresponds to the 'if' statement in Ada. The 'Create_Node' function will update the 'type_of_node' field with an integer value to point that this node is an "if statement" node. The second field is an empty string but it may also be the function or a variable name. Third argument is the number of rule that is reduced, which is one. This value has no other option because there is only one rule for reduction. The final argument is number two, which shows that there will be two other children on the node. Although there are other tokens on the rule, which are 'if', 'end' and ';', these values are hidden and not included in the syntax tree. By looking at the 'type_of_node' and 'rule_number' fields it is possible to retrieve the information necessary to understand that there are other tokens such as keywords 'if', 'end', and ';'. The 'nodes' and 'up' fields of the structure definition are used to link other child and parent nodes of the new node. The up field is initially equal to Null and it is updated if it is linked to another node. So 'Add_Two_Nodes' function adds the existing nodes of 'cond_clause_s' and 'else_opt' to the 'if_stmt' node. And links the 'up' fields of 'cond_clause_s' and 'else_opt' to point to the 'if_stmt'. This sequence of reductions and linking of nodes continues up to the reduction of 'goal_symbol' node, which indicates that the parsing process is successful.

```
1:          if_stmt : IF cond_clause_s else_opt END IF ';'
2:               {
3:                    $$ = Create_Node   ( _IF_STMT, "", 1, 2) ;
4:                    Add_Two_Nodes   ( $$, $2,  $3 ) ;
5:               };
```

Figure 3.4. If statement rule for parser

The 'line_number' field is used to store the current line number that the token is located. The scanner keeps a line number data and during token generation it is incremented as the input stream of characters include a new line character. In order to report the errors, SCA needs a line number data and during code generation the 'Create_Node' function locates the line number information to this field.

3.3. The Semantics Rule Checker Functions

The input source code, which is to be checked by SCA, is now converted to a tree structure and it is ready to be used by the semantic rule checker functions. These functions generate the final report about the semantic rule violations and errors. The functions, which are responsible to check a semantic rule, accept the syntax tree as input, and the semantic analysis of the input code starts after the syntax tree is created. The functions are independent from each other in terms of data dependency and they do not update or change the content of the syntax tree. Every function traverses the tree and generates the messages as they detect a violation or error. A global variable, 'Global_Tree' points the syntax tree and it becomes ready after the parser creates the 'goal_symbol' node.

The mechanism used in the semantic checker functions depends on searching and detection of a target node. This target node is determined by the rule definition. If a rule restricts the usage of a certain statement, such as 'goto', then the corresponding function searches the 'Global_Tree' to find the instances of statements; particularly 'goto' statements. So target node search is implemented by every function individually. All functions search their target node by traversing the tree up and down. This traversal is possible by using the 'up' and 'nodes[n]' fields of the node, where '*n*' is the number of children on the sub-tree. In the following sections, the rule checking functions are discussed. The rule checker functions traverse the tree left-to-right and depth-first fashion.

3.3.1. Rule 1

The usage of predefined 'integer' and 'float' type is restricted by rule 1. Real numbers are defined in mathematics as consisting of all rational and irrational numbers, Rational numbers are those which can be expressed as the ratio of two integers, e.g. ½, 4/3, etc. Irrational numbers are those such as √2 which can not be defined in such a way. In Ada there is a "real type" to represent such numbers. However this real type can only represent a limited set of infinite set of real numbers, the so-called "model numbers". Real numbers can be defined by using floating point or fixed point notation. Thus there are two kinds of model numbers, one for each notation. Further more, model numbers can be represented in computer without error. The model numbers may either be predefined or defined by programmer. The model numbers of a predefined type, such as float or integer predefined types of Ada, can be different from compiler to compiler, so they are implementation dependent. Thus, a mathematical routine compiled by different compilers, has possibility to generate different results at the run time. By contrast, a programmer-defined type always contains the same set of numbers; it does not depend on the compiler. Therefore mathematical routines written using such types fully transportable between machines. If compiler can not meet the requirements of a type declaration it should flag up an error. Since the fixed point or floating point notations defines same set of numbers, then the programmer should be warned to use these notations to create new types instead of using predefined types.

SCA implements first rule, by first detecting all 'float' and 'integer' identifiers, which are created as tokens by lexer in the syntax tree. Then, the process continues by the checking of these tokens to find if they are located in a type declaration. The 'float' or "integer" tokens should be located on the type declaration subtree, which is created on the syntax tree of the input source code, to represent a type declaration. Since the syntax tree can be traversed upwards the type declaration or subtype declaration nodes can be searched. In Figure 3.5, the parser rules are displayed. If the predefined float and integer types are used in these rules, it is clear that they are used to create new types and by checking the range constraint definitions, SCA can determine whether the rule is violated or not. The 'subtype_ind' and 'type_completion' nodes exist if the type declaration

includes a constraint definition such as a range constraint as it is displayed in Figure 3.6.

```
1:      subtype YEAR_TYPE is INTEGER range 1900..2000;
2:
3:      subtype CENTURY is INTEGER;
```

Figure 3.5. The type and subtype declaration rules for parser

```
1:    type_decl    : TYPE identifier discrim_part_opt
2:                   type_completion ';'
3:
4:    subtype_decl : SUBTYPE identifier IS subtype_ind ';'
```

Figure 3.6. The subtype declarations with and without range constraint

3.3.2. Rule 2

According to rule two, name association should be used in procedures or functions calls. In Figure 3.7, an example Ada code is listed for named association usage in sub-program calls. The arguments of the procedure are written before the values they send into the program body. SCA searches all the sub-program calls by analyzing the syntax tree of the code and finds the arguments located in the sub-program headers.

```
1: EXAMPLE_FUNCTION    ( TAPE          => TAPE,
2:                       MODE          => MODE,
3:                       COMPRESS      => COMPRESS,
4:                  STATUS        => STATUS);
```

Figure 3.7. The nodes sequence up to the goal symbol

The 'index_comp' node is searched on the syntax tree. The 'indexed_comp' node has two children: a name node and a 'value_s' node. The 'name' node includes the name of the sub-program, and 'value_s' is the node of the sub-tree that keeps the arguments that have to be called with named association. The 'value_s' is composed of other 'value' nodes and it has the structure, which is listed in Figure 3.8. It may have an 'expression' node, a

'comp_assoc' node or a 'discrete_with_range' node. So if there is no 'com_assoc' node, this sub-program is called with other ways, which is not permitted. By using the 'line_number' field of 'comp_assoc' node, the line number is displayed to the programmer and the name association usage is suggested.

```
1:          value :  expression
2:                | comp_assoc
3:                | discrete_with_range   | error;
4:
5:          comp_assoc : choice_s RIGHT_SHAFT(=>)  expression
```

Figure 3. 8. The 'value' and 'comp_assoc' node structures

3.3.3. Rule 3

In Ada an aggregate is a basic operation that combines component values into a composite value. Ada permits an initialization method to initialize the composite elements easily and it is usually done by using 'others' keyword. If an aggregate is initialized with 'others' keyword, the value associated with 'others' keyword is assigned to all the elements of the composite element. The usage of 'others' is listed in Figure 3.9 with an array structure.

```
1:          Type MATRIX_ELEMENTS is array (1..5) of INTEGER;
2:          COEFFICIENTS : MATRIX_ELEMENTS;
.
.
n:          COEFFICIENTS := MATRIX ELEMENTS'(25, others => 0);
```

Figure 3. 9. Others keyword usage in aggregate initializations

The Ada grammar rules and their CFG descriptions are listed in Figure 3.10. SCA first finds all 'aggregate' nodes in the tree and then the choice nodes are searched on the subtree pointed by 'comp_assoc' node. The aggregate definitions in Ada grammar are identified by the existence of an 'aggregate' node. The 'others' keyword token is reduced to a 'choice' node whose 'rule_number' field is equal to three. This value is equal to three because it is the third rule and during the generation of syntax tree, number three is

assigned to 'rule_number' field of choice node. If SCA detects a node whose node type is choice node type and 'number_of_rule' field is equal to three, then it reports a warning message about the usage of 'others' keyword.

```
1:    aggregate: '(' comp_assoc ')'
2:          |'(' value_s_2 ')'
3:          |'(' expression WITH value_s ')'
4:          |'(' expression WITH NuLL RECORD ')';
5:
6:    comp_assoc  :choice_s RIGHT_SHAFT expression;
7:
8:    choice_s     :     choice
9:          | choice_s '|' choice
10:
11:   choice       :expression
12:         | discrete_with_range
13:         | OTHERS;
```

Figure 3.10. The node structure of aggregate initialization and 'others' usage

3.3.4. Rule 4

Short circuit usage in logical operations is recommended by rule four and every time logical operations are used in the code. In logical expressions, which are located in the subtree of 'expression' node, it is possible to detect the existence of short circuit usage by comparing the second and third reduction rules. In Figure 3.11, the linkage between expressions and relations is described. So by comparing the expression subtrees, the short circuit usage is controlled by SCA.

```
1:                short_circuit : AND THEN
2:                              | OR ELSE;
3:
4:                expression    : relation
5:                              | expression logical relation
6:                              | expression short_circuit relation;
```

Figure 3.11. The short circuit form and grammar link with expression

When an expression is detected during syntax tree traversal, the nodes, which are labeled with 'expression' type, are inspected for the existence of 'short_circuit' node. If

'logical' node is used instead of 'short_circuit' node, then it is reported as an error.

3.3.5. Rule 5

When memory deallocation is needed, the library function 'unchecked_deallocation' can be used to direct the system to free any memory content. During this disposal if the memory was in use, function do not raise an exception or warning message. So this function usage is prohibited by rule five.

SCA searches all function names used in the code and investigates if this function is used or not. This is possible if the function names are retrieved from the syntax tree. SCA finds the function names by checking all the 'simple_name' nodes in the body and compares the string content located in the 'char_va'l field of the node with 'Unchecked_Deallocation'.

3.3.6. Rule 6

Checking negated expressions more than one-negation starts finding the expressions in the syntax tree and counting the negations on the subtrees pointed by their children. The negated expressions can be detected by checking the 'rule_number' field of factor nodes. When a factor node is created by reduction of the primary nodes, if there is a 'not' keyword before the 'factor' node, then the 'number_of_rule' field becomes equal to two. And whenever SCA detects the number of factor typed nodes with 'number_of_rule' field is equal to 2, then a warning message is printed. In Figure 3.12, from the expression to factor, node-creating rules are listed. And in the Figure, the 'not' keyword is located on the second reduction rule of 'factor' node.

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

```
1:          term   :      factor
2:                 | term multiplying factor;
3:
4:          simple_expression : unary term
5:                 | term
6:                 | simple_expression adding term;
7:
8:          relation : simple_expression
9:                 | simple_expression relational
10:
12:         simple_expression :
13:                | simple_expression membership range
14:                | simple_expression membership name;
15:
16:         expression : relation
17:                | expression logical relation
18:                | expression short_circuit relation;
19:
20:         factor : primary
21:                | NOT primary
22:                | ABS primary
23:                | primary EXPON primary
24:
```

Figure 3.12. The expression and negation linkage on parser definition rules

3.3.7. Rule 7

Usage of 'others' keyword in case structure is also prohibited by seventh rule. The first step is finding case statements in the syntax tree. The case statements are created on the 'case_stmt' nodes in the syntax tree. SCA first detects the existence of all case statements by finding the choice nodes. SCA understands rule violation by checking the choice node's 'number_of_rule' field. If the 'others' keyword is used this field is equal to three, because it is the third reduction rule.

3.3.8. Rule 8

Exit statement usage is restricted to two cases: The statement must be followed by a 'loop' name, from which the statement will exit and there must be a 'when' statement. For all cases other than these two, SCA should display an error message. Exit statement is defined in the grammar as in Figure 3.14.

SCA displays an error message if both of the ‘name_opt or when_opt’ nodes are equal to null.

```
1:     case_stmt    : case_hdr pragma_s alternative_s END CASE ';'
2:
3:     alternative_s : { $$ = NULL; }
4:                   |alternative_s alternative;5
5:
6:     alternative :      WHEN choice_s RIGHT_SHAFT statement_s;
7:     choice_s    :      choice
8:                 |      choice_s '|' choice
9:
10:    choice       : expression
11:                 | discrete_with_range
12:                 | OTHERS
```

Figure 3.13. The ‘case’ statement and ‘others’ keyword linkage in Ada’95 grammar

```
1:     exit_stmt : EXIT name_opt when_opt ';'
```

Figure 3.14. The exit statement in Ada’95 grammar

3.3.9. Rule 9

In a subprogram body, there must be only one ‘return’ statement according to ninth semantic rule. So for all subprogram bodies, SCA should count and detect if there are multiple return statements. In Figure 3.15, the ‘subprog_body’ node is shown, and starting from the ‘block_body’ node, SCA counts the ‘return_stmt’ nodes. If it finds more than one ‘return_stmt’ node located on the procedure of function body, it displays an error message by giving the name of the function. The name of the sub-program is retrieved by an upward traversal until the ‘subprog_spec_is_push’ node is found. A search is performed here again to detect the function name. After the ‘subprog_spec_is_push’ node is found then ‘subproc_spec’ node is detected because the ‘compound_name’ node, which keeps the subprogram name, is located on the subtree of ‘compound_name’ node. When the subprogram name is retrieved for reporting, the line number of the node is also displayed and programmer becomes aware of multiple return statement usage.

```
1:    subprog_body:      subprog_spec_is_push decl_part
2:                       block_body END id_opt ';'
3:
4:    return_stmt : RETURN ';'
```

Figure 3.15. Grammar definition of subprogram body in Ada'95

3.3.10. Rule 10

SCA displays an error message, for all 'goto' statements that are used in the code. SCA finds this statement by investigating the existence of 'goto_stmt' node in the syntax tree. In Figure 3.16, the structure of goto statement node is written. When it is detected on the syntax tree, it is reported as an error.

```
1:                       goto_stmt : GOTO name ';'
```

Figure 3.16. The 'goto' statement

3.3.11. Rule 11

According to the semantic rule eleven, the nested structure of control is limited to four levels. This means, the number of any control statement, such as a case, if or loop structure can be written in other control structures with a limit of four times.

There are eight such statements in Ada'95 grammar and they are defined as different nodes in the Yacc parser generator rules. The basic structures are 'if', 'case', 'while' loop, 'basic loop' and 'select'. The node definitions of these functions are listed in Figure 3.17.

SCA starts searching these nodes from the beginning of syntax tree and it checks if the node type of the current node is any one of these nodes. If it detects that the node is a control structure, it increments the number of previously found control structure node number and displays an error whenever current number of control is exceeded the limit of four.

```
1:    if_stmt            : IF cond_clause_s else_opt END IF ';'
2:
3:    case_stmt          : case_hdr pragma_s alternative_s END CASE ';'
4:
5:    basic_loop   : LOOP statement_s END LOOP
6:
7:    accept_stmt  : accept_hdr ';'
8:                 | accept_hdr DO handled_stmt_s END id_opt ';'
9:
10:   select_wait  : SELECT guarded_select_alt or_select else_opt
11:   END SELECT ';'
12:
13:   async_select        : SELECT delay_or_entry_alt THEN ABORT
14:   statement_s END SELECT ';'
15:
16:   timed_entry_call : SELECT entry_call stmts_opt OR delay_stmt
17:   stmts_opt END SELECT ';'
18:
19:   cond_entry_call  : SELECT entry_call stmts_opt ELSE
20:   statement_s END SELECT ';'
```

Figure 3.17. The nested control structure node definitions

3.3.12. Rule 12

All parameters of a function or procedure declaration have four different modes in Ada'95 grammar. A parameter can be 'access', 'in', 'out' or 'in-out' mode and one of these modes is the defined as the default mode. Usually the default mode changes from compiler to compiler. And semantic rule twelve restricts the unspecified parameter mode usage during subprogram declaration. SCA finds the argument declarations by tracing the 'param' nodes in the syntax tree. Whenever it finds a 'param' node, it looks to the mode node and displays an error of it is not specified. In Figure 3.18, it is possible to see the 'mode' node and it is left null if programmer specifies nothing.

```
1:              mode :    { $$ = NULL; }
2:                     | IN
3:                     | OUT
4:                     | IN OUT
5:                     | ACCESS;
6:
7:              param : def_id_s ':' mode mark init_opt   ;
```

Figure 3.18. The parameter modes in Ada'95

3.3.13. Rule 13

In Ada'95, an argument declared in 'out' mode, should receive a new value in the body of the subprogram it is used. All the subprogram arguments with an 'out' mode, which is not updated, should be reported by SCA. The first strategy is to check assignments to find whether the parameters are updated or not. SCA keeps the out parameters and searches all assignments in the body of the subprogram. All value assigned parameters are excluded from the list of parameters and the rest of arguments are displayed with a warning message.

3.3.14. Rule 14

The conventional operators can be overloaded with new functional abilities in Ada'95 and semantic rule fourteen claims the programmer should be convinced that the operator keeps its conventional meaning with the new functionality. For example, the addition operator 'plus' sign can be used to name a function that adds an element to a list. SCA checks all function declarations to find if the conventional operator plus, minus, divide or multiply are overloaded or not. If they are overloaded then the programmer is warned to be sure that they are keeping their conventional meanings on the functionality. Detection of function declarations is possible by tracing the 'subprog_spec' nodes in the syntax tree. If the 'subprog_spec' has a 'compound_name' node, which carries the operator name, such as a plus or minus sign, a message warns the programmer for its existence. In Figure 3.19, the structure of 'subprog_spec' node and an overloaded function declaration is listed.

```
1:    subprog_spec : PROCEDURE compound_name formal_part_opt
2:           |FUNCTION designator formal_part_opt RETURN name
3:           | FUNCTION designator
4:
5:    function "+" (num: integer, num2:integer) return integer;
```

Figure 3.19. The Subprog_spec node

3.3.15. Rule 15

The sub-program arguments should not be initialized when they are declared. The initialized arguments of subprograms are checked on the 'param' nodes of the syntax tree and its definition in listed in Figure 3.18. If the param node has an 'init_opt' node, this reflects that it is initialized during the subprogram declaration. SCA prints a warning message for all initialized arguments of subprograms.

3.3.16. Rule 16

Elaboration of Ada program units is controlled by compiler and initiated by 'with' statement usage in the code. During the elaboration sequence, packages are initialized or declared to the software. In Ada, the elaboration sequence is compiler dependent and compilers do not perform elaboration in a unique fashion. In rule sixteen, the control of this elaboration sequence is left to user and it is controlled manually by a user-defined subroutine that is created for every package. Whenever a package is defined, there must be an initialization routine, which begins with the name of the package appended to 'Init' keyword. SCA checks existence of this function if the input file contains a package declaration. SCA checks all the declared subprogram names and by comparing their names with 'Init_package_name' it understands if there is such an elaboration control function to fulfill the task.

3.3.17. Rule 17

The 'with' and 'use' statements include other packages definitions in the package and they start the elaboration. When a package is included with a 'with' statement, the content of the package is referred by the absolute names, which are usually long. For example, if a variable is defined in the System package any other package uses this variable should refer this variable with 'System.variable_name'. This convention creates long variable and function names and to shorten these names for a readable code, the package can be 'use'd to call the variable without specifying the absolute path of the variable or function. According to the rule 17, all packages that are included by a 'with'

statement should be also be supported with a 'use' statement.

```
1:    with_clause :     WITH c_name_list ';'
2:
3:    use_clause  :     USE name_s ';
4:                |     USE TYPE name_s ';'
```

Figure 3.20. The with and use statements grammar rules in Ada'95

SCA finds the list of package names in 'with' statements and prints a warning message if their names are not used in a 'use' statement.

3.3.18. Rule 18

The programmer should not raise predefined exceptions intentionally. SCA finds all the raised exceptions, and it checks from a predefined exception list of Ada, whether the raised exception is in the list or not. As it is listed in Figure 3.21, the 'name_opt' node contains the name of the raised exception.

```
1:    raise_stmt : RAISE name_opt ';'
```

Figure 3.21. Raise statement and its grammar

3.3.19. Rule 19

A subprogram can only raise exceptions that are defined in its package. SCA finds all declared and raised exceptions. The grammar rule, which is used to declare exceptions, is listed in Figure 3.22. The node 'def_id_s' keeps the name of the declared exception. The declared exception names are compared with the raised exception names. SCA prints an error message if the raised exception is not declared in the current package.

```
1:     exception_decl : def_id_s ':' EXCEPTION ';'
```

Figure 3.22. Exception declaration rule and its grammar

3.3.20. Rule 20

SCA displays a warning message for all the numerical values found in the code except they are used in a type or constant declaration. SCA finds all the numerical values that are outside of the type or constant declarations and prints a message to warn programmer not to use them.

3.3.21. Rule 21

SCA should return a warning message for all generic type declarations in subprograms. This is necessary to avoid memory insufficiency because the declared type instantiated every time the procedure is called. In Ada, the declarations done between the subprogram header and subprogram body. SCA searches the syntax subtree of this area to find any generic type declaration. In Figure 3.23, the generic declaration node is displayed. The search starts from 'decl_part' node and SCA searches generic_formal_part nodes on the subtree of 'decl_part' node. If it is found, this shows that there is a generic type declaration on the subprogram and it is reported as an error.

```
1:     subprog_body : subprog_spec_is_push
2:                decl_part block_body END id_opt ';'
3:
4:     generic_formal_part : GENERIC
5:                | generic_formal_part generic_formal
```

Figure 3.23. Generic declarations in subprograms

4. OUTPUT ANALYSIS OF THE SEMANTIC ANALYZER

The semantic rule set has twenty-one rules for Ada grammar and SCA tool has a C routine per rule to inspect the source. In this section, the C routines that inspect the rule set are analyzed. First, the effectiveness of the SCA tool is justified on a sample Ada code. Then, previously compiled real world application Ada software is inspected with SCA. The output of the inspection is analyzed in terms of execution time and memory usage. Also a multi-threaded version of SCA tool is proposed to decrease the inspection time on multi-processor hardware.

4.1. A Sample Run

The SCA effectiveness is tested on the example code, which is listed in appendix C. This example code is not a part of any real Ada application. It just contains some semantic errors or coding parts that has violations to the ESRS. It is the control input and SCA must generate some messages concerning the compatibility of this code to the ESRS.

The example code contains a package declaration and the package body implementation, which has some function and procedure calls. It tests the float or integer type usage by using these predefined types in function or procedure bodies. Also in aggregate initializations and case statements, 'others' keyword is used. An array is initialized with 'others' and there is an 'others' choice in one of the case structures. There are examples of invalid 'exit' statement and 'goto' statement usage. Also a set of numerical values is entered both in type declarations and other parts of the code to check whether SCA has the ability detect them. Invalid subprogram arguments are also declared in the package specification according to the ESRS.

SCA runs and investigates the example code, and then generates the output in appendix D. It can be seen that SCA can detect the violations to the ESRS and can generate relevant information about the Ada code and its semantic content. The example was an artificial code and it was intentionally created to contain some semantic violations. In the following

section, a real-world example is analyzed and the report generated by SCA is discussed.

4.2. ATC Software Analysis by SCA

SCA is run on 41 different, previously compiled source code of a real Air Traffic Control application software. The total number of lines in these files is 56,7 KLOC (Lines of Code). SCA generated 8687 error or warning messages, after analyzing these Ada files. In Figure 4.1, the percentage of messages per rule is displayed.

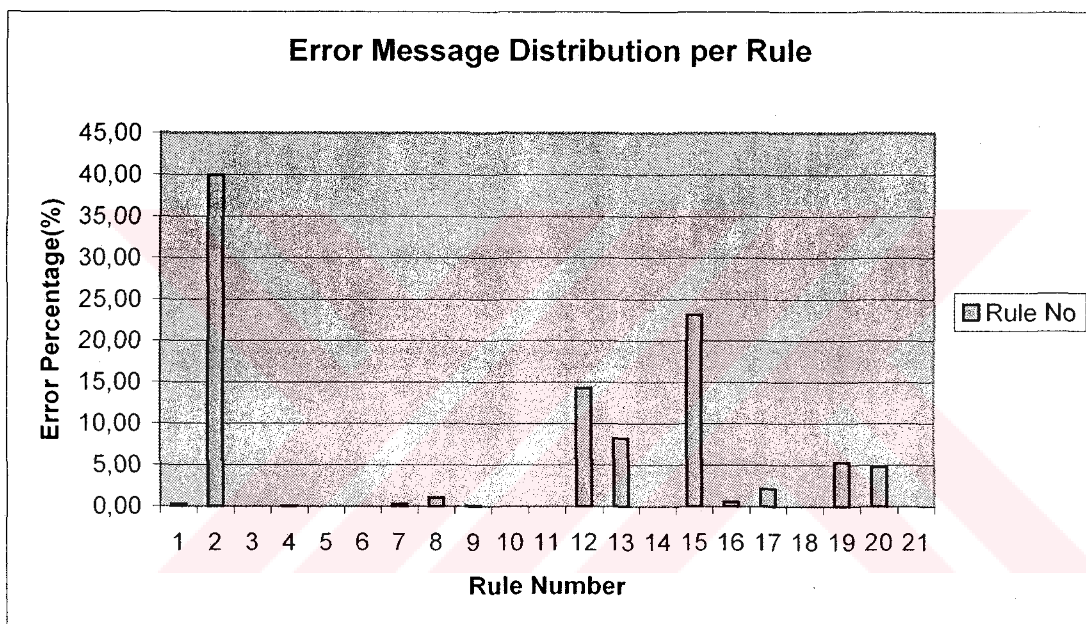


Figure 4.1. Error distribution per rule

The most violated rule is the second rule, which warns programmer to use name association when calling subprograms. The 39% of report messages are concerning the name association usage. This value is high because SCA generates a warning message for every argument in the subprogram call. Usually programmers tend not to use named association if the used subprogram is linked from another package specification or library where the function headers are not easily inspected for argument names. In order to increase readability, named association should be used because the argument names can give an idea about the usage of the function. Name association usage decreases the chance of invalid value entry to the subprogram calls because the association ensures that the

values are send correctly.

Initialization of subprogram parameters is another common violation, which is defined by rule fifteen. When variables are initialized at declaration, then it becomes not necessary to enter a value for initialized arguments for subprogram calls. This may cause an information deficiency about the usage of function for the code reader. Besides, the initialized value may cause execution errors because programmers may not take the value into account, when they build the code. The reason to use initialized variables upon subprogram declarations is to create a value even the programmer do not enter any value for the argument. But it is better to leave the control to the programmer and he is forced to enter the value intentionally.

Default parameter mode usage is the third most violated rule and the default parameter mode changes from compiler to compiler. Some compilers take *in* mode as the default mode and it assumes that it will not be modified in the body. Even it is modified, when the procedure is returned to its caller, the value change is lost. If the default mode is 'inout' mode, the value that modifies the argument is kept even the subprogram returns. These two cases have potential to create execution errors and it may take time to detect. Using SCA output to enter the parameter modes explicitly can eliminate this risk.

The 'out' mode parameters must receive a new value in the subprogram body. Verifying whether a parameter is modified or not, is done by checking if they are located on any left part of an assignment statement. If the parameter is updated via another *out* or 'inout' parameter call in a subprogram call, then this modification of the argument is not inspected by SCA. The programmer should assure that the argument is modified. About 7.5% of the messages are concerning about the usage of 'out' mode parameters which are not updated in the subprogram bodies.

Ada has predefined exception set and these exceptions are raised by the system when an unexpected condition happens. Divide by zero is a common example and programmers sometimes raise this exception intentionally. But semantic rule 19 restricts the manual or hard coded raise of predefined exceptions. With a 'raise' statement

programmers can raise all exceptions. But this kind of usage of raise statement creates a problem: System raised exceptions and program raised exceptions are not easily detectable and debugging takes time especially if exceptions are raised because of a system error. In ATC programs, SCA detected about 459 messages concerning rule 19 and this corresponds to 5% of total messages. Instead of raising system exceptions, the programmer should use his own exceptions and they should be defined for every created package. Raising other exceptions that are not defined in the current package should also be avoided, because using exceptions that are belonging to the current package can help to find the reason of the exception easily. About 5% of messages concern with the rule 20, which restricts usage of other package exceptions.

The 'with' statement should be supported with a 'use' statement to increase the readability of the program. Use statement directs compiler to replace the absolute path name of the identifier names. About 0.3% of messages are generated to warn the programmer to use a 'use' statement with 'with' statements.

Exit statement is used in Ada for breaking the loops and exiting program block when certain conditions happen. Usage of exit statement can be dangerous because it affects the execution sequence directly. The exit statement concerning messages is %5 of total messages and they reflect that exit statement is used without the loop name to exit or a 'when' condition. This kind of 'exit' statement usage is hard to maintain and if it is used without any condition or a loop name, then it is immediately executed and causes to change the current execution location to next program block. SCA detects all these invalid exit statement usage and warns programmer to take extra cautious steps to avoid unexpected execution errors.

The 'others' keyword usage in aggregate initialization is not detected in ATC applications and they are used in 'case-control' structures 24 times. The usage of 'others' keyword in case statements creates a leak point for the unexpected values. Detection of invalid values is important during the production process and 'others' choice can be used in case structures if everything about the program functionality is fulfilled and under control.

4.3. Resource Allocation and Time Analysis

Syntax tree generation is the main memory consumer process and memory is allocated for the created nodes. Memory requirement for the syntax tree generation depends on the number of tokens in the input file. The input ATC code used in the analysis is composed of 41 files and the file sizes are shown in Figure 4.2. During the semantic analysis, all dynamic memory allocations to generate the syntax tree are calculated to find the average memory consumption.

Required time for the report generation depends on the size of the tree. Since the semantic rules are checked by tree-traversing functions, as the number of nodes in the tree increases, the time needed to traverse all the nodes of the tree also increases. In the next subsections, the memory consumption of the analysis and the time needed to analyze the input codes on different platforms are compared to find if the semantic analysis is performed in a reasonable time and with a reasonable memory consumption.

4.3.1. Minimum Memory Requirement

For every tree node created by the parser, 80 bytes are allocated and the context-free grammar representation of Ada grammar allows five children to be connected to a node in the tree. As number of tokens in the files increases, the number of nodes created by the parser also increases. The number of tokens found by the scanner in the input files is drawn in Figure 4.3. In average, a token is composed of 4 characters and 1 Kb file has 250 Ada tokens in it.

When all of the nodes have five children, the tree size becomes minimum. This is because of the fact that minimum number of nodes is created to connect the input tokens as a tree structure.

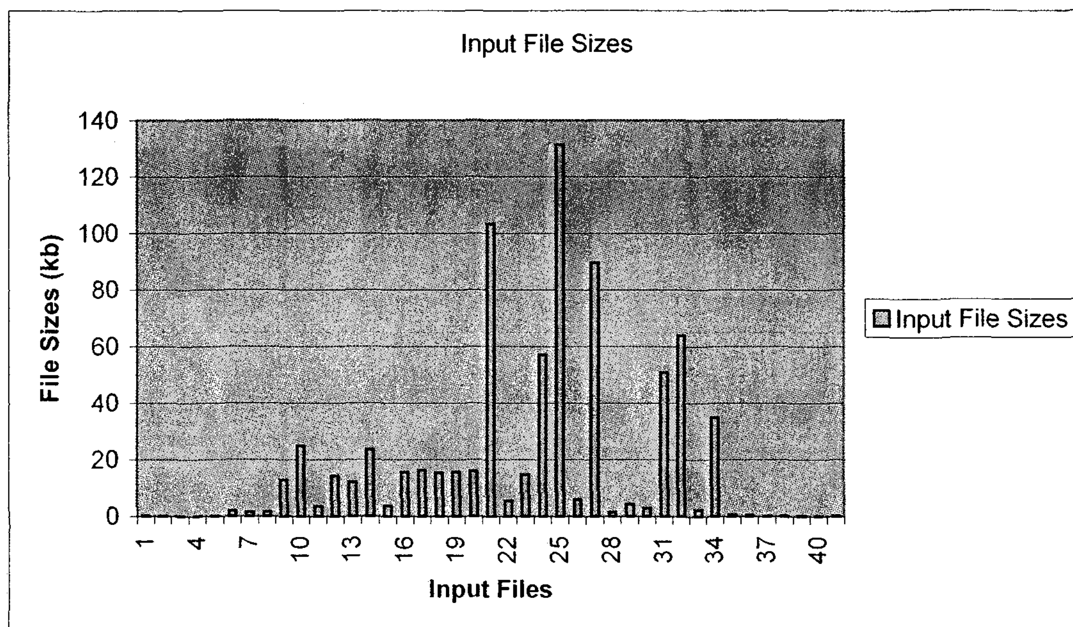


Figure 4.2. Input file sizes

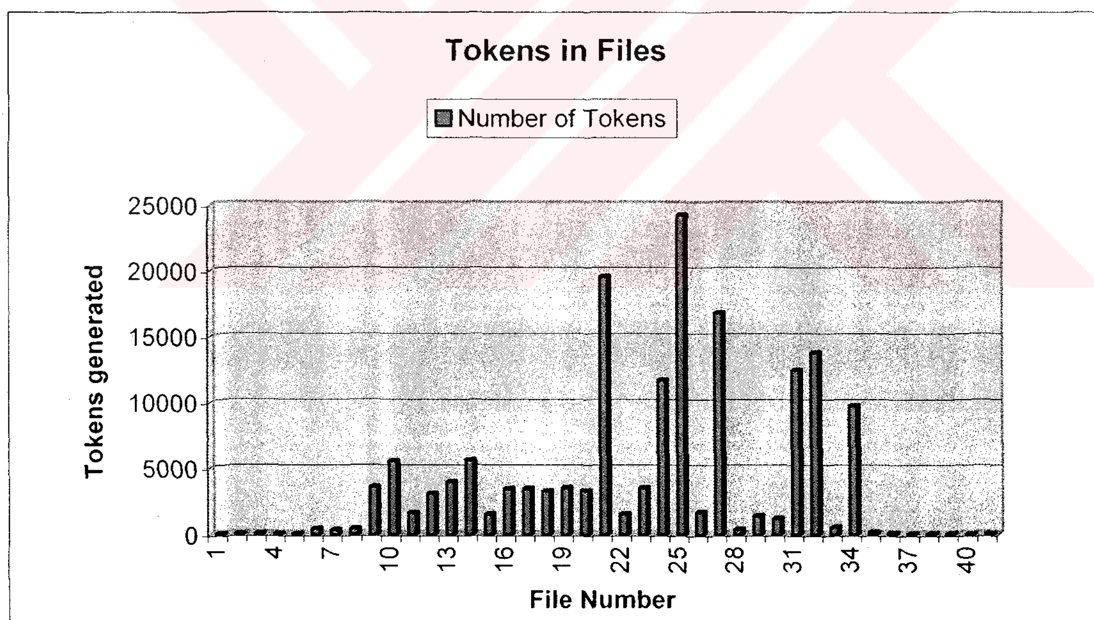


Figure 4.3. Generated token number from input file

The minimum tree can be constructed if the tree is a full-balanced tree where all inner nodes have five children. Starting from the root node, if we donate the number of nodes at level k as N_k than we can find the nodes at level k with formula 5^k:

The number of nodes at root level is:

$$N_{k=0} = 5^0 = 1 \text{ where level is zero.}$$

The number of nodes at level one is:

$$N_{k=1} = 5^1 = 5 \text{ where level is one.}$$

And if we continue to the lowest level of the tree where the level is equal to the height of the tree than:

$$N_{k=h} = 5^h \text{ where h is the height of tree}$$

Since the leaves of the tree are actually the tokens of the input file, than we can relate the tokens and tree height as:

$$\text{Number of Tokens} = 5^h \text{ where h: height of tree} \quad (4.1)$$

Total number of nodes from level 0 to the highest level, we can use Equation 4.1 to find the total number of nodes in the tree.

$$\text{Total Number of Nodes} = 5^0 + 5^1 + 5^2 + 5^3 + 5^4 + \ldots + 5^{h-1} \text{ where h: height of tree} \quad (4.2)$$

The right hand side of the Equation 4.2 is simplified and we can reach to:

$$\text{Total Number of Nodes} = 5^h - 1 / (5-1) \quad (4.3)$$

Writing 4.1 into 4.3 results roughly:

$$\text{Total Number of Nodes} = \tfrac{1}{4} \text{ (Number of Tokens)} \quad (4.4)$$

The size of the node structure to represent one node in the tree is 80 bytes and in

average 4 bytes are used to create one token. So if we know the file size then we can reach to the number of tokens in it by dividing the file size with four.

$$\text{Number of Tokens} = F_{size}/4 \qquad \text{where } F_{size} \text{ is the file size in bytes} \qquad (4.5)$$

So required memory for minimum tree is found by multiplying the number of created nodes by node size in bytes:

$$\text{Minimum Allocated Memory} = \text{Number of Nodes} * 80 \qquad (4.6)$$

Since we know the relation between tokens and created nodes, we can use Equation 4.4 in Equation 4.6 to find the relation between tokens and memory size.

$$\text{Minimum Allocated Memory} = \tfrac{1}{4}\,(\text{Number of Tokens}) * 80 \qquad (4.7)$$

By using Equation 4.5, we can find the relation between minimum tree size in memory and input file size.

$$\text{Minimum Allocated Memory} = (F_{size}/4) * 20 = 5 * F_{size} \qquad (4.8)$$

In Figure 4.4, the real tree size and calculated minimum syntax tree sizes of the files are compared. The minimum tree size is calculated with the real token numbers found in the files and the generally the real syntax tree size is found to be 6 times of theoretical minimum syntax tree size. In other words, in average:

$$\text{Real Tree Size} \cong 6 * \text{Theoretical Minimum Tree Size}$$

When the ratio of the real syntax tree size and file size is considered, the average of the ratio is found to be 15,5 when the average is taken from Figure 4.5. The minimum of this value is found as 4,5 from Equation 4.8.

$$\text{Real Tree Size} \cong 15{,}5 * \text{Input File Size}$$

The minimum required memory to analyze an input file is about 16 times of its size. The number of tokens in the files and generated number of nodes by the parser is drawn in Figure 4.6 and the average shows that for every token generated in the lexer, parser generates 1,28 node. This value shall be 0,25 from Equation 4.4 for minimum tree case and conforms to the fact that the real tree is about 6 times more of the theoretical tree size.

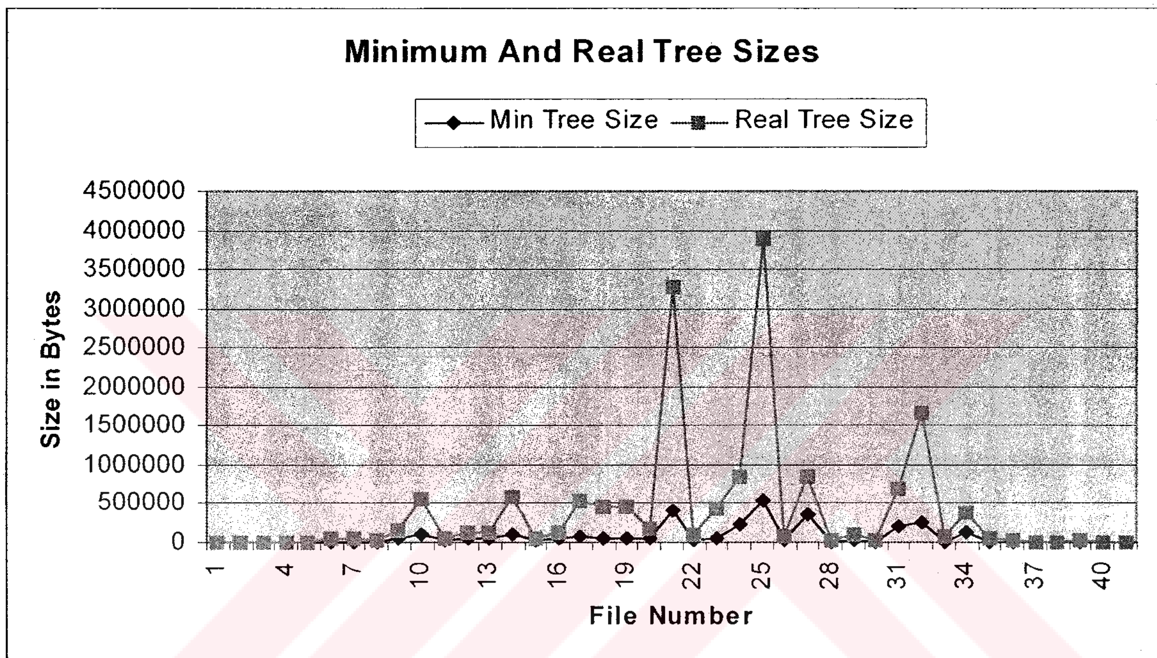


Figure 4.4. Theoretical minimum and real syntax tree size

During abstract syntax tree generation 34% of the code is embedded in the tree by using the 'char_val' pointer of the nodes. This part is the non-predefined set of tokens in Ada grammar, which are used to name identifiers, function names and so on. That information is used during the report generation.

After investigation of the memory requirement, we investigate analysis time of SCA on Ada'95 input codes on different platforms. We run the tool on two different hardware and software platforms: First on a personal computer, which has Linux OS and on a Digital Alpha Unix workstation.

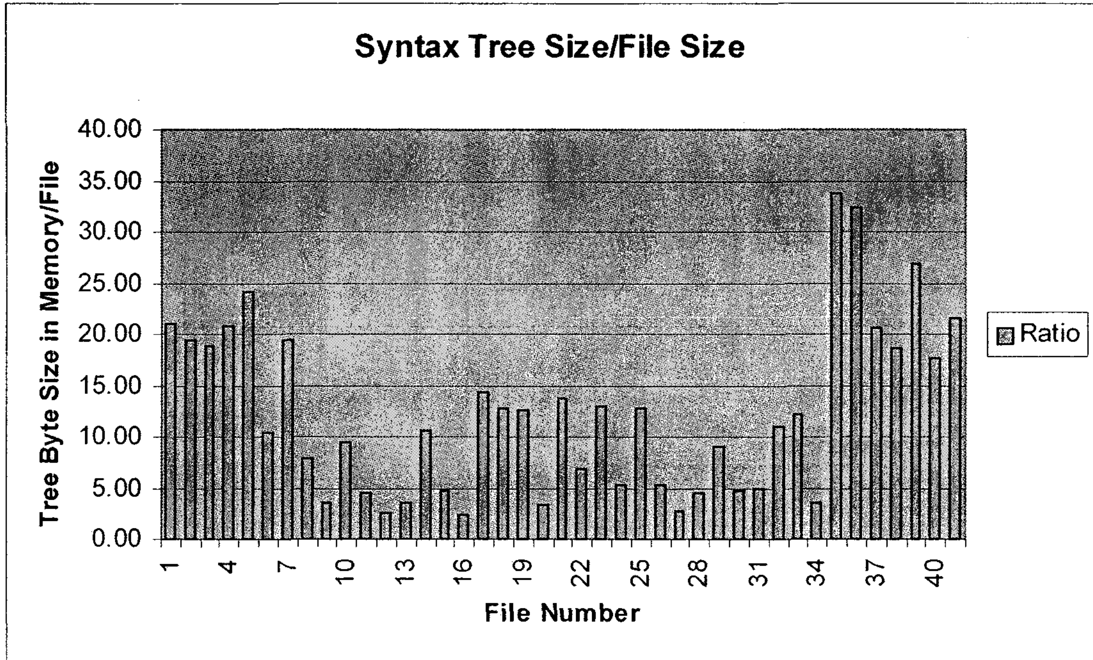


Figure 4.5. The ratio of syntax tree size and input code size

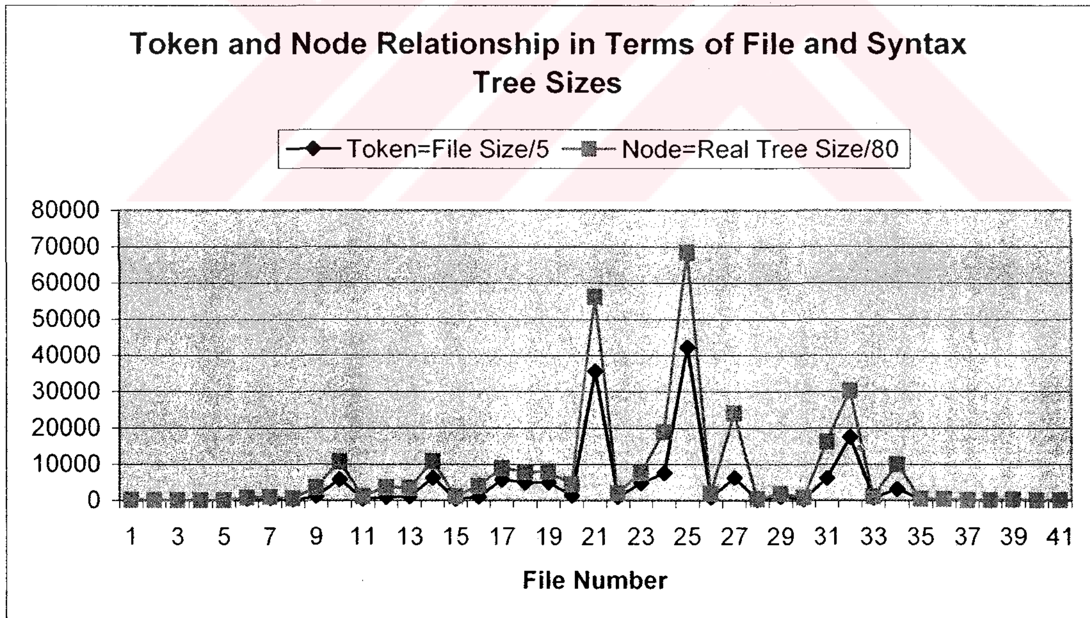


Figure 4.6. The ratio of node and token sizes

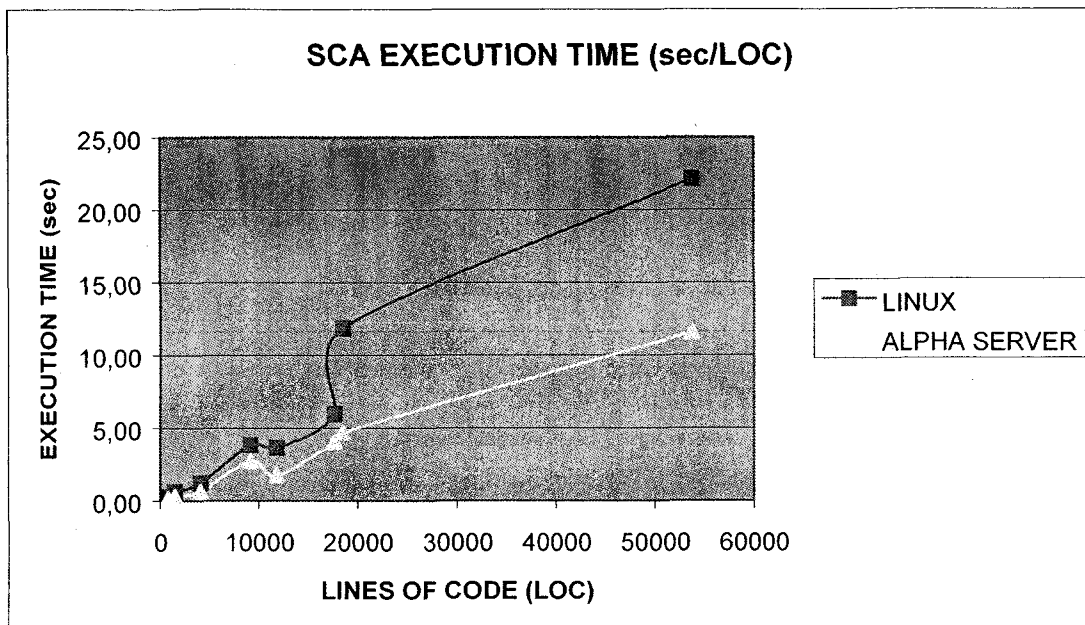


Figure 4.7. Execution time of SCA on different OS

The total time needed to check all of the files on an Alpha Unix station are 11,55 sec. When the software is run on Linux platform the required time becomes 20,13 sec. In Figure 4.7, the execution time versus file sizes is drawn.

4.4. Multi-Threaded SCA

The rule checker functions can be executed in any order and this is a potential parallelism in the execution sequence of SCA. The syntax tree data is not updated by any function so there is no data dependency. Creation of multiple threads in the parent process of SCA and performing analysis with several function threads result an improvement on the execution time especially on multi-processor hardware. Even on the uni-processor environments depending on the POSIX implementation, the CPU cycle allocation to threads result improvements on the execution time. In Figure 4.8, the multi-threaded version of SCA and its execution time is displayed. The threads are created with default attributes, such as default stack size and a slight improvement in the Linux machine is detected even it is run on a uni-processor hardware.

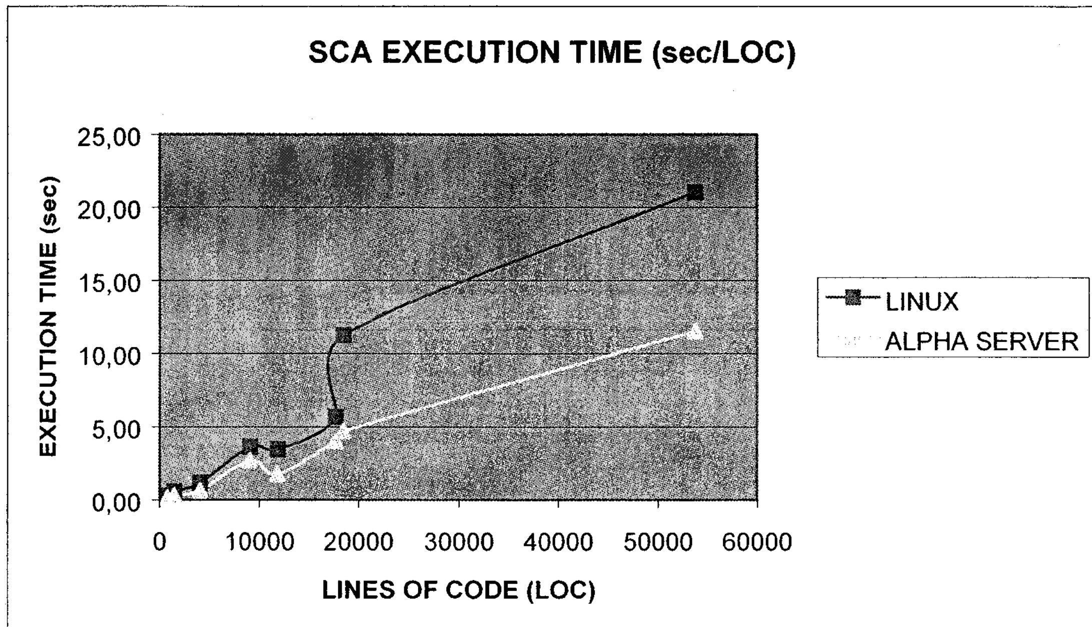


Figure 4.8. Execution time of multi-threaded SCA on different OS

5. CONCLUSION

The semantic rule set is proven to contain effective restrictions for Ada grammar semantics to increase software quality and decrease erroneous execution risk. Every rule in the set has one or more target quality attributes to effect. The rule set is tested on the ATC application, which needs a high level of safety and integrity and shown that even ATC applications might have semantic error risk.

The semantic rule set is shown to be analyzable by a software tool and its methodology, which is similar to an Ada compiler, is described. A scanner and a parser is designed and integrated to a C program to build a software tool, which can be used to inspect the Ada codes.

The SCA software tool is run on different UNIX platforms and its effectiveness is analyzed. The memory allocations of the tool are analyzed and minimum memory requirement is defined.

The potential parallelism of the SCA tool is identified and a multi-threaded version of SCA is implemented. The new version is analyzed on uni-processor environments and its execution time is compared with serial version.

As a result, a new semantic rule set for Ada software is defined and a tool to verify Ada source codes is implemented successfully. This new semantic rule set can be a guideline for Ada programmers and the tool can be used to inspect Ada project codes in order to retrieve high quality Ada projects.

APPENDIX A: THE REGULAR EXPRESSION SET OF LEX INPUT FILE

The following is a sample from the Lex file, which contains the Ada'95 token descriptions. On the left hand side, the token types are declared and on the right hand side they are described as regular expressions.

```
DIGIT                      [0-9]
EXTENDED_DIGIT             [0-9a-zA-Z]
INTEGER                    ({DIGIT}(_?{DIGIT})*)
EXPONENT                   ([eE](\+?|-){INTEGER})
DECIMAL_LITERAL            {INTEGER}(\.?{INTEGER})?{EXPONENT}?
BASE                       {INTEGER}
BASED_INTEGER              {EXTENDED_DIGIT}(_?{EXTENDED_DIGIT})*
BASED_LITERAL              {BASE}#{BASED_INTEGER} (\.{BASED_INTEGER})
                           ?#{EXPONENT}?
```

APPENDIX B: A SAMPLE FROM PARSER DESCRIPTION FILE

The following is the Yacc input file, which is used to generate the parser for the Ada'95 grammar. The rules have the form of context-free grammar and cover all Ada'95 grammar. This file is anayzed by Yacc and it generates a C file, which can be compiled with other C files.

```
%{
#include <stdio.h>
#include <string.h>
#include "types.h"
#include "functions.h"
%}

%union {
    char *ident;
    struct  NODE *Node;
}

%token TIC
%token DOT_DOT
%token LT_LT
%token BOX
%token LT_EQ
%token EXPON
%token NE
%token GT_GT
%token <ident> char_lit
%token <ident> identifier
%token <ident> char_string
%token <ident> numeric_lit

%type <Node> goal_symbol
%type <Node> pragma
%type <Node> pragma_arg_s
%type <Node> pragma_arg
%type <Node> pragma_s
%type <Node> decl
%type <Node> object_decl
%type <Node> def_id_s
%%
goal_symbol : compilation
    {
                    $$ = Create_Node   ( _GOAL_SYMBOL,"", 1, 1) ;
                    Add_One_Node ( $$, $1 );
                    Global_Tree = $$;

                };

pragma   : PRAGMA identifier ';'
    {
                    $$ = Create_Node   ( _PRAGMA,$2, 1, 1) ;
```

```
                }

        | PRAGMA simple_name '(' pragma_arg_s ')' ';'

                {
                    $$ = Create_Node  ( _PRAGMA,"", 2, 2) ;
                    Add_Two_Nodes ( $$, $2, $4 );
                };

pragma_arg_s : pragma_arg
     {
                   $$ = Create_Node  ( _PRAGMA_ARG_S, "", 1, 1) ;
                   Add_One_Node ( $$, $1 );
                }
        | pragma_arg_s ',' pragma_arg

                 {
                    $$ = Create_Node  ( _PRAGMA_ARG_S, "", 2, 2) ;
                    Add_Two_Nodes ( $$, $1, $3 );
                 };

pragma_arg : expression
      {
                    $$ = Create_Node  ( _PRAGMA_ARG, "", 1, 1) ;
                    Add_One_Node ( $$, $1 );
                 }

        | simple_name RIGHT_SHAFT expression

                 {
                    $$ = Create_Node  ( _PRAGMA_ARG, "", 2, 2) ;
                    Add_Two_Nodes ( $$, $1,  $3 );
                 };
```

APPENDIX C: EXAMPLE INPUT FOR SCA

The following is the example input file to test the capability of SCA tool. It is not a real application software code instead it is an artificially generated code to contain semantic errors.

```
1:with STANDARD_TYPES;
2:use STANDARD_TYPES;
3:with NEW_TYPES;
4:
5:
6:package AIRPORT_IO is
7:
8:
9:  type STATUS is (INCOMING, OUTGOING);
10:
11:  type    FLIGHT   is
12:          record
13:             Callsign     :  CALLSIGN;
14:             Start_Time   :  DATE;
15:             Start_Point  :  POINT;
16:             Stop_Time    :  DATE;
17:             Stop_Point   :  POINT;
18:             Speed        :  FLOAT;
19:             Distance     :  Float;
20:          end record;
21:
22:  type    FLIGHT_LIST;
23:  type    FLIGHT_POINTER is access FLIGHT_LIST;
24:
25:  type    FLIGHT_LIST    is
26:          record
27:            Flight         :  FLIGHT;
28:            Next           :  FLIGHT_POINTER;
29:          end record;
30:
31:  type    FLIGHT_MODE    is new NEW_TYPES.FLIGHT_MODE;
32:  type    COUNT          is new NEW_TYPES.COUNT;
33:  subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
34:
35:  procedure CREATE_NEW_FLIGHT(Flight_Name : out FLIGHT;
36:                             File : in out FILE;
37:                             Init : out INTEGER;
38:                             Clr  : out INTEGER;
39:                             Mode : FILE_MODE := NEW_TYPES.DEFAULT_MODE);
40:
41:  function "+" (Flight_1:in out FLIGHT_LIST;
42:                Flight_2:in FLIGHT_LIST) return FLIGHT_LIST;
43:
44:  procedure FREE is new UNCHECKED_DEALLOCATION(FLIGHT_LIST,
45:              FLIGHT_POINTER);
46:
```

```
47:  NO_FLIGHT_LIST    : exception;
48:  EMPTY_FLIGHT_LIST : exception;
49:
50:  SPEED : constant := 3.0;
51:  HALF_SPEED : constant := 1.5;
52:
53:end IOCALL_DIRECT_IO;
54:
55:---------------------------------------------------------------------
56:package body AIRPORT_IO is
57:
58:procedure CREATE_NEW_FLIGHT (Flight : out FLIGHT;
59:                             Status: STATUS;
60:                             Distance: in FLOAT;
61:                             Speed: in FLOAT;
62:                             Destination : LOCATION) is
63:begin
64:
65:
66:    if (Flight = NULL) then
67:       Flight := new FLIGHT;
68:    end if;
69:
70:
71:    Flight.Callsign := Get_Next_Callsign;
72:
73:    case Status is
74:
75:
76:         when OUTGOING =>
77:
78:         Flight.Start_Time  := Get_Current_Time;
79:         Flight.Start_Point := Get_Current_Location;
80:         Flight.Stop_Time   := Get_Current_Time + Distance/Speed;
81:         Flight.Stop_Point  := Destination;
82:
83:         when INCOMING  =>
84:
85:         Flight.Start_Time  := Find_Start_Time (Flight);
86:         Flight.Start_Point := Find_Start_Point (Flight);
87:         Flight.Stop_Time   := Flight.Start_Time + Distance/Speed;
88:         Flight.Stop_Point  := Current_Location;
89:         goto EXIT_PTR;
90:
91:         when STOPPED   =>
92:
93:         Flight.Start_Time  := Find_Start_Time (Flight);
94:         Flight.Start_Point := Find_Start_Point (Flight);
95:         Flight.Stop_Time   := Flight.Start_Time + Distance/Speed;
96:         Flight.Stop_Point  := Current_Location;
97:         <<EXIT_PTR>> exit;
98:
99:         when others =>
100:              raise Unknown_Status;
101:    end case;
102:
103:end CREATE_NEW_FLIGHT;
104:
105:
```

```
106:function "+" (Flight_1: in out FLIGHT_LIST; Flight_2: in
FLIGHT_LIST)
107:                                  return FLIGHT_LIST is
108:
109: List_Pointer : FLIGHT_POINTER;
110:
111:begin
112:        if ( Flight1 = NULL ) then
113:
114:             return FLIGHT_LIST;
115:        end if;
116:
117:
118:        if  ( Flight_2 = null ) then
119:
120:            return Flight_1;
121:        end if;
122:
123:
124:        if NOT ( Flight_1 = NULL) OR NOT ( Flight_2 =  NULL) then
125:
126:            return Merger_Lists(Flight_1, Flight_2);
127:        end if;
128:
129:end "+";
130:
131:
132:
133:function SQUARING(X :IN  ITEM) return ITEM is
134:    begin
135:     return X*X;
136:    end SQUARING;
137:
138:
139:procedure FIND_ENTRY (PARAM:INTEGER) is
140:
141:    UNKNOWN_FLIGHT_EXCEPTION : EXCEPTION;
142:    Internal_Flight_List : TABLE := (others => 0);
143:    generic
144:       type ITEM is private;
145:    function SQUARING(X : IN ITEM) return ITEM;
146:
147:begin
148:
149:
150:    Internal_Flight_List'(5, 8, 4, 1, others => 0);
151:
152:    PARAM :=  RETURN_FLIGHT("ASAP102011", Internal_Flight_List);
153:
154:    if ( PARAM < 0 AND PARAM > 200 ) then
155:
156:      raise UNKNOWN_FLIGHT_EXCEPTION;
157:
158:    end if;
159:
160:    if ( PARAM > 0) then
161:
162:       if ( PARAM > 20) then
163:
```

```
164:          return GREEN_RANGE;
165:          if ( PARAM > 40) then
166:
167:             return BLUE_RANGE;
168:             if ( PARAM > 60) then
169:
170:               return YELLOW_RANGE;
171:               if ( PARAM > 80) then
172:                  return RED_RANGE;
173:               end if;
174:             end if;
175:          end if;
176:        end if;
177:     end if;
178:
179:   return Get_Flight (PARAM);
180:
181:end FIND_ENTRY;
182:
183:end IOCALL_DIRECT_IO;
```

APPENDIX D: A SAMPLE OUTPUT FROM SCA

The following is the output of the SCA after it analyzed the input file, which is listed in the Appendix C.

```
1:No syntax errors detected
2: ADA-SEC TOOL
3:       All rights reserved
4:
5:
6:PLEASE WAIT ...........
7:----------------------------------------
8:
9:
10:
11:Generating Report for example.a:
12:---------
13:
14:AT LINE:     44 TRY TO USE  NAME  ASSOCIATION
15:AT LINE:     45 TRY TO USE  NAME  ASSOCIATION
16:AT LINE:     85 TRY TO USE  NAME  ASSOCIATION
17:AT LINE:     86 TRY TO USE  NAME  ASSOCIATION
18:AT LINE:     93 TRY TO USE  NAME  ASSOCIATION
19:AT LINE:     94 TRY TO USE  NAME  ASSOCIATION
20:AT LINE:    126 TRY TO USE  NAME  ASSOCIATION
21:AT LINE:    126 TRY TO USE  NAME  ASSOCIATION
22:AT LINE:    152 TRY TO USE  NAME  ASSOCIATION
23:AT LINE:    152 TRY TO USE  NAME  ASSOCIATION
24:AT LINE:    179 TRY TO USE  NAME  ASSOCIATION
25:AT LINE:    142 OTHERS NOT ALLOWED IN AGGREGATE INITIALIZATION
26:AT LINE:    150 OTHERS NOT ALLOWED IN AGGREGATE INITIALIZATION
27:AT LINE:    124 TRY TO USE  SHORT  CIRCUIT FORM
28:AT LINE:    154 TRY TO USE  SHORT  CIRCUIT FORM
29:AT LINE:    124 MULTIPLE NEGATION NOT ALLOWED
30:AT LINE:     99 OTHERS NOT ALLOWED IN CASE STMT
31:AT LINE:     97 INVALID EXIT STATEMENT USEAGE
32:AT LINE:    129 MULTIPLE RETURN NOT ALLOWED
33:AT LINE:    181 MULTIPLE RETURN NOT ALLOWED
34:AT LINE:     89 GOTO USAGE IS NOT ALLOWED
35:AT LINE:    173 NESTED STRUCTURE IS LIMITED TO FOUR LEVELS
36:Default parameter mode is not allowed :
37:PARAM: MODE               LINE:     39
38:Default parameter mode is not allowed :
39:PARAM: STATUS               LINE:     59
40:Default parameter mode is not allowed :
41:PARAM: DESTINATION                 LINE:     62
42:Default parameter mode is not allowed :
43:PARAM: PARAM                 LINE:    139
44:Preserve Conventional Meaning "+" operator     42
45:Preserve Conventional Meaning "+" operator    107
46:In procedure "+" arguments at line:    106
47:FLIGHT_1
48:Should receive a value
```

```
49:
50:Parameters must not initialized :
51:PARAM: MODE                LINE:     39
52:AT LINE:      3 LIBRARY NEW_TYPES SHOULD BE USED
53:AT LINE:    100 EXCEPTION UNKNOWN_STATUS IS NOT DEFINED IN THIS
PACKAGE
54:UNCHECKED DEALLOCATION USEGA IS PROHIBITED AT LINE:     44
55:INITIALIZATION FUNCTION INIT_AIRPORT_IO IS NOT DECLARED
56:AT LINE:    150 FIND A NUMERIC LITERAL: 5
57:AT LINE:    150 FIND A NUMERIC LITERAL: 8
58:AT LINE:    150 FIND A NUMERIC LITERAL: 4
59:AT LINE:    150 FIND A NUMERIC LITERAL: 1
60:AT LINE:    154 FIND A NUMERIC LITERAL: 200
61:AT LINE:    162 FIND A NUMERIC LITERAL: 20
62:AT LINE:    165 FIND A NUMERIC LITERAL: 40
63:AT LINE:    168 FIND A NUMERIC LITERAL: 60
64:AT LINE:    171 FIND A NUMERIC LITERAL: 80
65:AT LINE:    143 GENERIC DECLARATION NOT ALLOWED:
66:       Successfully Done
```

REFERENCES

1. Barnes, J., *High Integrity Ada: The SPARK Approach*, Addison-Wesley, New York, 1997.

2. Allen, F. E. and J. Cocke, "A catalogue of optimizing transformations", in R. Rustin, (Ed), *Design and Optimization of Compilers*, pp. 145-158, Prentice-Hall, Englewood Cliffs, 1972.

3. Kildall, G., "A unified approach to global program analysis", *ACM Principles of Programming Languages*, Vol. 56, pp. 194-206, January, 1973.

4. Cousot, P. and R. Cousot., "Comparing the Galois Connection and Widening Narrowing Approaches to Abstract Interpretation", in M. Bruynooghe and M. Wirsing (Eds.), *Proceedings of PLILP'92*, pp. 269-295, Springer-Verlag, Berlin, 1992.

5. Schaeffer, C. F. and G. N. Bundy, "Static Analysis of Exception Handling in Ada", *Software Practice and Experience*, Vol.23, pp.1157-1174, October 1993.

6. Bacon, F. and P. F. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls", *OOPSLA'96 Conference Proceedings*, San Jose, 1-12 October 1996, Vol. 1, pp. 23-40, 1996.

7. Sundaresan, V., L. Hendren, and C. Razafimahefa, *Practical Virtual Method Call Resolution for Java*, Sable Technical Report, No.1999-2, Sable Research Group, McGill University, April 1999.

8. Steindl, C., *Program Slicing for Oberon*, Technical Report 11, Institut für Praktische Informatik, JKU Linz, 1997.

9. Baer, J., *Static Layer Analysis for C Programs*, 1998, http://www.cs.washington.edu/homes/jbaer/papers/layers.html

10. Knizhnik J., *Java Program Checker*, 1999, http://www.ispras.ru/~knizhnik/jlint/ReadMe.html

11. Horspool, R.N., and M.R. Levy, "Mkscan: An Interactive Scanner Generator", *Software Practice and Experience*, Vol.17, No.6, pp. 369-378, June 1987.

12. Mossenbock, H., "Alex - A Simple and Efficient Scanner Generator", *SIGPLAN Notices*, Vol.21, No.12, pp.139-148, December 1986.

13. Grosch, J., *Rex - A Scanner Generator*, Compiler Generation Report Number 5, GMD Forschungsstelle University Karlsruhe, 1991.

14. Nicol, G.T., *Flex: The Lexical Scanner Generator*, Free Software Foundation Report 1.03, February 1993.

15. Bunke H. and B. Haller, "A Parser For Context Free Grammars", in M. Nagl (Ed), *Graph-Theoretic Concepts in Computer Science*, pp. 136-150, Plenum Publishing Company, Manchester, 1989.

16. Donnelly, C. and R. Stallman, *Bison: The YACC-compatible Parser Generator*, Free Software Foundation Report 1.25, November 1995.

17. Uddeborg, G., *A Functional Parser Generator*, Technical Report 43, Dept. of Computer Sciences, Chalmers University of Technology, Goteborg, 1988.

18. Narayan, P., *Portability and Performance: Applications on Diverse Architectures*, Report TR-92-22, Department of Computer Science, University of Virginia, July, 1992.

19. Tanenbaum, S., P. Klint, and W. Bohm, “Guidelines for Software Portability”, *Software-Practice and Experience*, Vol. 8, No. 6, pp. 681-698, 1978.

20. Dahlstrand, J., *Software Portability and Standards*, Ellis Horwood, Chichester, 1984.

21. Mooney, J.D., *Issues in the Specification and Measurement of Software Portability*, Report TR 93-6, West Virginia University, Morgantown WV, 1993.

22. Dale, E., J.S. Chall, “A Formula for Predicting Readability”, *Educational Research Bulletin*, Vol. 27, pp. 211-233, February 1988.

23. Barry, J.G., "Computerized Readability Levels", *IEEE Transactions on Professional Communication*, Vol. 23, No. 2, pp. 88-90, June 1980.

24. Cheaito, R., M. Frappier, S. Matwin, A. Mili, and D. Crabtree, *Defining and Measuring Maintainability*, Technical Report, Dept. of Computer Science, University of Ottawa, March 1995.

25. Leveson, N.G. and P.R. Harvey, “Analyzing Software Safety”, *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, pp.155-195, 1983.

26. Ada Home Design Team, *Ada Grammar and Parsing*, 1998, http://www.adahome.com/Resources/refs/grammar.html

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ