Contract Programming Checker

A study for making an automated test tool using a parser

HamidReza Yazdani Najafabadi
Abstract

Contract Programming Checker

HamidReza Yazdani Najafabadi

Thanks to computer software development, the communication industry has evolved a lot during the last few years. Software and hardware integration made it possible to exploit the best out of available equipments.

One of the important issues in software development process is to avoid bugs or detect them in the early stage of development phase. Experiments have shown that most of the bugs are usually coming from the small fraction of the code. If this part of the code can be detected in advance then it is possible to benefit the cost of software production in great amount of time and money. Development teams have to make sure that they deliver a verified code to next team and that is why they obliged to use a concept called “contract programming”. It means expecting each module which is working with other modules to respect to some kind of contract. As long as the contract is respected in all module interactions, valid output will be guaranteed.

Several problems will remain in this approach. First issue is to make sure all necessary contracts have been embedded to the code. On the other hand, contracts are memory and time consuming to check so try doing over protection results in weaker performance.

Considering the scalability problem, there is an urgent need for an automatic tool which is capable of checking against all possible defects to tell the programmer exactly where the contract is needed without performing any under or over protection.

This thesis tries to address this problem by generating a parser using UNIX tools, Lex (lexical analyzer) and Yacc (parser generator), to detect or warn about the possible cause of defects. General built-in functions with different algorithms have also been implemented in C language to perform different level of code analysis. The outcome of this thesis is a parser which fulfills three different requirements.

Firstly, checking all protected required places to check if they have been protected by their proper contracts.

Secondly, notifying the extra contracts in places where they are not needed. It is done by parser which analyzes the calling graph of different functions to verify if the contracts are actually needed.

The last but not least requirement is to find the least protection required areas. It means places where protection should be kept even if all internal computations are guaranteed to be correct. This facility will be used when the code wants to be delivered to other teams and the internal integration of the code is already verified.

The tool is also capable of performing statistical analysis to give an exact percentage of protection in each function block and the software unit as a whole.

The Developed tool has successfully passed all of the exhaustive tests for furnishing these requirements.
## Contents

Background ..................................................................................................................................................1

1 Problem Description ................................................................................................................................1

1.1 Defensive programming Versus Contract programming .................................................................2

1.1.1 Defensive programming .............................................................................................................2

1.1.2 Contract programming ...............................................................................................................3

1.2 Research questions ..........................................................................................................................7

2 Requirement specification .......................................................................................................................7

2.1 Functional requirement .......................................................................................................................7

2.1.1 Warning for unprotected areas ....................................................................................................7

2.1.2 Removing extra protections .........................................................................................................7

2.1.3 Determining least protection required areas ...............................................................................8

2.2 Non-functional requirement ..............................................................................................................8

3 Solutions/Methods ..................................................................................................................................9

3.1 Tool description ................................................................................................................................9

3.1.1 Current similar tools ..................................................................................................................9

3.1.2 User Interface ...........................................................................................................................10

3.2 Development approach ....................................................................................................................12

3.2.1 Defining the problem ................................................................................................................12

3.2.2 Alternative Solutions .................................................................................................................12

3.2.3 LEX ........................................................................................................................................12

3.2.4 YACC .....................................................................................................................................13

3.2.5 Flex/Bison ............................................................................................................................16

3.2.6 Solution ...................................................................................................................................16

3.3 Developing parser with Flex/Bison .................................................................................................17

3.3.1 Gathering required information ...............................................................................................19

3.3.2 Different data structures .........................................................................................................28

3.3.3 Different stack implementations ..............................................................................................29

3.3.4 Config file ................................................................................................................................32

3.3.5 Find-file module .......................................................................................................................34

3.3.6 yywrap() routine .....................................................................................................................35

3.3.7 Different functions implementation in Lib.c ........................................................................39

3.3.8 Fulfilling the requirements ......................................................................................................47

4 Extra features .......................................................................................................................................52

4.1 Disable warnings .............................................................................................................................52

5 Some shots from the program output ....................................................................................................53

5.1 First requirement ..............................................................................................................................53

5.2 Second requirement ........................................................................................................................56

5.3 Third requirement ............................................................................................................................57

5.4 Function call tree ............................................................................................................................59

6 Solution analysis ..................................................................................................................................60

7 Critical analysis .....................................................................................................................................60

8 Comparative study ...............................................................................................................................60

9 Literature search and techniques .........................................................................................................61

10 Delimitations ......................................................................................................................................61

11 References ..........................................................................................................................................61
Table of figures

Figure 1: Sample program showing defensive programming concept ................................................. 2
Figure 2: Sample program showing design by contract concept ........................................................ 5
Figure 3: Development cycle of one software unit consisting of four modules .................................. 6
Figure 4: The current user interface of the tool. It can be seen by using the help option (-h) ............... 10
Figure 5: Illustrating the concept shift/reduce conflict in a sample grammar .................................. 14
Figure 6: This figure illustrates reduce/reduce conflict in a sample grammar ................................... 15
Figure 7: Different start state embedded for catching divisions in the input code ................................ 21
Figure 8: Different start state embedded for catching array indexes in the input code ....................... 22
Figure 9: A sample program which is given as an input to the tool .................................................. 23
Figure 10: The pseudo code for the function check_type_extra() for detecting functions name ........ 24
Figure 11: The pseudo code for check_type_extra_extra() for detecting different functions arguments 26
Figure 12: Available data structures which is used in the tool for storing different information ....... 28
Figure 13: A sample program used to show how stack implementations works .............................. 30
Figure 14: This figure shows how the stack implementation works .................................................. 31
Figure 15: This figure shows involved modules in parsing one or several files ................................. 32
Figure 16: General structure of the config file .................................................................................... 33
Figure 17: The implementation of the function yywrap() .................................................................. 37
Figure 18: This figure shows involved modules in parsing with config file ..................................... 38
Figure 19: The pseudo code of the function search_recursive_function_called() ......................... 40
Figure 20: The pseudo code of the function search_function_called() .......................................... 41
Figure 21: The pseudo code of the function check_extra_attention_required_aux ......................... 42
Figure 22: The pseudo code of the function check_extra_attention_required ................................. 44
Figure 23: The pseudo code of the function search_in_out_function() ............................................ 46
Figure 24: A sample code showing how the second requirement is furnished .................................. 49
Figure 25: The function call tree ........................................................................................................ 51
Figure 26: A sample input file test.c ................................................................................................. 53
Figure 27: The result of the tool using the first requirement in non verbose mode ......................... 54
Figure 28: The result of the tool using the first requirement in more verbose mode ....................... 54
Figure 29: The output result of the tool using the protection statistic option -x .............................. 55
Figure 30: Function call tree of the program test.c ........................................................................... 56
Figure 31: The output of the tool using the second requirement ...................................................... 57
Figure 32: Showing the least protection required area by the tool to fulfill the third requirement ...... 58
Figure 33: Function call tree of the file test.c containing four function blocks ............................. 59
Background

It is necessary to make sure the integrated modules communicate correctly with each other to assure designing robust and correct code. In order for these properties to hold, the cause of errors should be prevented before it actually happens. One of the source of errors is value passing between different functions in the program. This master thesis studies the concept in C programming language which is our target language in the current thesis.

Due to lack of strong type checking in C, the need of mechanisms capable of stronger checking is more obvious. There are different methods to check against such kind of pitfalls. One of them is defensive programming in which the designer tends to defense the program against different errors by different kinds of checking which may be performed in several ways such as “if statements”. The problem with this approach is ending up doing over protection instead of actual programming. This may cause to further errors, therefore the need of substitute techniques is obvious and recommended. One of the alternatives is the concept of contract programming. In Contract Programming, designer protects the code with predefined macros which will take care of input and output validity. Theses Macros in our studied code are DBC_PRECOND, DBC_POSTCOND and DBC_ASSERT. The designer has an option to turn on or turn off these macros in the compile time. The important thing is to make sure that final version of the code should not include any of such macros.

1 Problem Description

Software reliability is one of the most important features of software. Reliability in software industry means a system’s ability to perform its operation according to specification (correctness) and to handle abnormal situations (robustness). In the other word, reliability is the absence of bugs. How can we build reliable software? The answer has several components. For example static typing is a major help for catching inconsistencies before they get a chance to turn to future bugs. [3,4]

The general goal of this project is to check massive amount of code against different defects. The tool is used to verify the whole project code. The backbone of the tool is a parser which is designed to scan the code which is written in C language and prompting different sources of error to the code designer. The designer will use the tool against different codes which has been protected by different types of contracts.

One of the challenges in this thesis is detecting the circular function calls and terminating the DBC checker in an efficient way in the second requirement.
1.1 Defensive programming Versus Contract programming

Defensive programming and contract programming have the same final goal which is to guarantee the software quality and reliability, the only difference is how these goals should be achieved. To understand the difference between these approaches, it is necessary to know their definition.

1.1.1 Defensive programming

Defensive programming means to make as little assumption as possible about the behavior of the input data. Consider this simple C toy example in order to explain the issue:

```
#include <stdio.h>
#include <math.h>

double mysqrt(double num);
int main()
{
    double d;
    float input;
    scanf("%f", &input);
    d = mysqrt(input);
    return(1);
}

double mysqrt(double num)
{
    double d1;
    if (num >= 0)
    {
        d1 = sqrt(num);
        return d1;
    }
    else
    {
        printf("you cannot square a negative value");
        return(-1);
    }
}
```

In the above example, programmer protects the program from invalid entry by using proper defensive check against negative values. If this check is not done, the program might crash without any proper warning or it may goes to undefined state, but if the protection is done with proper check against negative values, this problem will be eliminated. Note that the supplier module, “mysqrt”, will not assume anything about the input data so it needs to check the input against negative values. The customer module which is “Main” will not assume anything about the “mysqrt” return value but since the square root of any value cannot be negative it is needed to be checked to guarantee a meaningful
result. The same might apply for many similar cases such as array indexes or null pointers depending on program implementation.

In tandem with benefits, disadvantage of defensive programming needs to be clarified. If the concept of defensive programming is being overused, the readability of the code might suffer strongly. Complex defensive piece of code might be used which is not easy to understand why the code should be there in the first place thus complexity of the code might increase exponentially.

The other problem which might arise is the legacy problem of the software. It means if the defensive code is adopted due to the certain expected usage of that code in some environment, these assumptions might not be valid in other circumstances especially if the code was reused in other software units. For example if the program is designed due to the fact that it works with integers and then the code is reused with real values the embedded conditions might not be enough in some cases.

### 1.1.2 Contract programming

Under Contract programming theory, a software system is viewed as a set of communication components whose interaction is based on precisely defined specifications of the mutual obligations which is called contract.

In this approach, the supplier module will require some condition to respect from the customer module and it will guarantee some post condition for the return value but only if the client already respected the precondition of the supplier.

There are three main different kinds of contracts in our studied module, which are DBC_PRECOND, DBC_POSTCOND and DBC_ASSERT. The definition of these three contracts can be found in the following:

#### 1.1.2.1 Precondition

A precondition is a part of the contract which determines what condition should be met initially to make a function guarantee some specific results.

There is no control inside the function to verify that the precondition is met. The function assumes that a check has already been done before the function is called.

If the precondition is not complying with the requirements, the function does not take any responsibility for the result and the conclusion can be undefined operations and even termination.

A precondition can be written in clear text or as an executable expression.

A precondition can be expressed to tell which range of values for different parameter is allowed or in which state the object needs to be.
Using preconditions in this way will result in partial function, which is a function that is not defined for all types of inputs.

### 1.1.2.2 Post condition

A post condition defines what a function is obliged to deliver when a correct function call is made. A correct function call means the preconditions met the requirement when a call is made to “it”. In this case, the function is responsible for the result and the client can trust that the post condition is hold.

A post condition could be written both as clear text and in code format. When a post condition is written in the code format it describes what the function does, not that it should be executed.

The post condition should not be connected to the implementation of the function. It will only guarantee the results. The result could be a return value or changing of a state.

### 1.1.2.3 Assert condition

An assert condition defines the expected result of a function or variable, it will check the input values of the other functions which can be outside the current function module. It can also perform the boundary check.

If the assert condition fails, the program might terminated with a proper massage or goes to undefined state.

Let’s consider the previous example but implemented with contract programming principals this time.
In the above example, the customer module will guarantee not to call the supplier module with negative values. In the other words the supplier module will expect the customer module to respect its precondition. If the precondition holds the customer module will guarantee the post condition which is to return a positive value. If the customer module violates the supplier precondition, depending on the implementation of the pre_cond and post_cond macros, an exception will be raised.

This approach which requires some kind of contract between supplier and customer modules is called contract programming. It will reduce lots of disadvantage of defensive programming in which it is always needed to check the input data of the supplier module and therefore it can reduce the complexity by reliving the supplier from extra checks. It will also solve the legacy problem of the supplier module. It does not have any implicit expectation from the input data, it can be used with any kind of valid data for any environment by simply changing the previous contracts.
Consider the following example:

![Diagram of software unit development cycle](image)

**Figure 3: Development cycle of one software unit consisting four modules. The red contracts can be removed after development phase. The black contracts should remain for delivery phase.**

In the above example, four modules are integrated in one software unit. Each arrow shows one interaction between two different functions. In this figure, if function “A” calls function “B”, there is an arrow from A to B in order to show this relation. DBC checks inside the protected zone should be used in development process in order to make sure that interrelated module’s parts are interacting correctly with each other and with other modules. These contracts can be turned off after testing and debugging phase of this software unit. The important thing is to make sure the external contracts are always on.

The external contracts are those which protect the software unit interaction with inputs or outputs. As this example suggests, the output might be related to direct hardware access. The other important issue is to consider that interaction with library functions does not need any contract since library calls are always trusted. If an error happens inside the library function, the program will be terminated with informative massage. The black DBC checks in the picture above are contracts which protect the least
protection required areas. As you can see later using the current tool, they will be detected automatically. The interested readers can refer Code craft by Pete Goodliffe[1]

1.2 Research questions

1-Is it possible to extract enough information to decide whether the studied function is protected or not and also to find out by how many percentage it is protected?

2-What is the relation between different characteristics of the code (for example: size of the code) and other interesting issues such as level of code protection?

2 Requirement specification

The brief survey of the requirement is already done in introduction but still the first thing to consider is to define the complete set of requirements which the tool suppose to fulfill after development process. The next step is to design some test cases to check if the tool is really covering those requirements.

The following will describe the requirement specification of the project which should be met at the end of the project. The most of the job had based on these requirements.

2.1 Functional requirement

2.1.1 Warning for unprotected areas

The tool should search the whole input module to determine which places need different kind of protections which are DBC_PRECOND, DBC_POSTCOND, and DBC_ASSERT. All of the input arguments should have been protected with precondition. All of the return value should have been protected with post condition. There are also several places which assert condition is needed. They are array indexes which should be checked to be in bound. Functions which return pointer should be checked against NULL pointer and divisions which should be protected against division by zero.

2.1.2 Removing extra protections

Since contract programming macros can take lots of memory, the tool should be capable of detecting different kind of macros which are not necessary. There are two different cases which this case can be seen.
2.1 Static functions:

In the case of static functions, since the scope of static functions are only limited to one module, it only needs to be protected by its calling function inside that module. It means that if the calling function has already protected the specific argument of the called static function, it is not needed to over protect the same argument inside that static function.

2.2 Double protection:

After extracting the nested function call graph from the input module, the tool traverses the graph until it reaches to endpoints. In this definition, the end points are functions which do not call any other function but they might have been called by the other functions. If these end points have some kind of protection for passed value to upper levels, then all of the protections in the upper levels seem to be unnecessary and therefore it is possible to remove all of them up to the top function.

2.1.3 Determining least protection required areas

After the development phase of software unit, it seems unnecessary to keep all contracts when it is needed to deliverer the code to be used in the whole architecture. Since these macros take lots of memory, it is beneficial to know where the places which are least protection required areas. In order to detect these places, it is needed to specify a root function. The root function is the function which the operation of the software unit begins from that point and therefore it is required to keep protection on its input argument. The other places which the protection is still require are places which take the input from outside of the current software Unit, because it is not guaranteed if inputs have been protected or not, therefore it is needed to keep protection in those places.

2.2 Non-functional requirement

REQ 1.1 Read a config file where:

1.1.a Different files names are specified
1.1.b The library functions are specified, these functions would be the endpoint of checking in our Parse tree
1.1.c The type definitions are specified, Since in our code there are lots of these cases where a short integer for example, is assigned an alias such as U16
1.1.d The root function should be specified in the config file, this is going to be used for requirement number 3 of the tool
1.1.e Scan a directory path specified in the config file and parse all targeted files. The tool should also find the files in the subdirectories of predetermined path
REQ 2. Identify all local and global functions in all files
REQ 3. Build a call graph tree with all included files

REQ 4. check contracts in one file and
   4.1 make a warning if no contract has been defined
   4.2 make suggestion of the required contract if possible
REQ 5. Check contract between several files
   5.1: To see if we have protection
   5.2: To identify the starting point of the contract

REQ 6. Detecting the circular function calls

3 Solutions/Methods

3.1 Tool description

The project will tentatively use the tools LEX and YACC which are the parser generators in C, in order to produce required parser for this project. Most of the programming part will be performed in C.

3.1.1 Current similar tools

There is no similar tool right now in the market which can cover exactly the same set of requirements; Since the DBC Macros are implemented inside Ericsson Company therefore the need for the specific tool to analyses effects of using these macros is necessary. The similar prototype has been developed by the supervisor of this thesis which does not cover lots of requirements. The goal is to make a more complete tool which covers all of requirements. That tool has been used as a prototype for developing the current tool
3.1.2 User Interface

According to our usability problem, I together with my supervisor decided on the friendly user interface which is implemented with getopt() function in C. The interface is much similar to most of the non-graphical tools available in the market in the way that it uses some kind of input options to interact with user. The user can be informed about different available options through the command –h. The help menu looks like this:

```
Usage: cchecker [OPTIONS] [FILE(S)]
DBC Contract checker developed by HAMIDREZA YAZDANI NAJAFABADI
for contract checking

Options:
  -e  showing the extra contract check in the module
  -c  to specify the config file, aware of it's special format
  -f  use this option if you only need to check one file
  -m  use this option if you only need to check several files
  -h  report different available options
  -t  report different available options, it should be use before other options
  -l  specifying the least required protection areas
  -v  specify the more informative verbose mode

Information:
Note: you need to specify a config.cfg file in special format and declare the path
to make -c option to work. you also need to use all options with either -f,-m,-c
```

Figure 4: User interface of the tool. It can be seen by using the help option (-h)

Different options available in the current tool are explained here:

- **-e:**
This option will be used to extract extra contract checks in current module. This has been done by the use of several recursive functions

- **-c:**
This option will be used to specify the config file which is one of the main features of this tool. Many tool facilities could not be used if the config file was not used. For example, the library functions, special definitions, and root function(s) for furnishing requirement three must be declared in the config file. If the user wants to determine a directory and it is required to check the entire C files in that directory and its subdirectories then it is necessary to use the config file option.

-\textbf{f} and \textbf{-m}:

These are the options which one should use to give the specific file location to the tool, if the user wants to check only one file, it is needed to use \textbf{-f} option, otherwise if the tool is used to check several files, \textbf{-m} option should be used.

-\textbf{t}:

This option will be used to plot function call tree outside the program. It should be used with one of the options -c or -f or -m. There is a nice feature in output tree which can show all functions calls inside one function. It will also show whether if the called argument of that function is already protected with contract or not. The developer is capable of seeing the whole protected areas.

-\textbf{l}:

This is the implementation of requirement three of the tool which suppose to determine least protection require areas in a module. In order to do so, the root function should be specified inside the config file.

-\textbf{v}:

This option will be use to specify more informative verbose mode. In this mode, the output will determine the exact place and exact argument which suppose to be protected by contracts. If this option is not used, the tool will only show how many protection of the different kind \textbf{PRE}, \textbf{POST} or \textbf{ASSERT} have been missed in different files.

-\textbf{x}:

There is a hidden option \textbf{“-x”} which shows the percentage of protection for all functions inside the parsed Software unit. I have been asked to keep this option hidden due to some reasons which is not in the scope of this report.
3.2 Development approach

3.2.1 Defining the problem

The tool should be capable of meeting the defined requirements. It should be also user friendly and easy to work. It should pass lots of test cases in order to ensure fulfilling the requirements. The tool should be capable of parsing huge amount of code and storing required information for further analysis, it should also have the ability to infer from the collected information where the protections can be removed or where they should be kept when the software unit is delivered to other teams.

Make a tool that satisfies all the requirements which is approved by the Supervisor of this project in Ericsson. Intensive test of this tool on different supplied data which are the real programs, developed in Ericsson

3.2.2 Alternative Solutions

By studying different scenarios in using Lex and Yacc, it has been tried to figure out what is the best way to make a parser capable of extracting the required information from the actual code. For discussion about different available methods, it is required to know different options available on when using the LEX/YACC compiler tools.

3.2.3 LEX

LEX [5], is in fact, a lexical analyzer which is capable of extracting predefined tokens and passing them to the YACC. It is important to mention that the lexer can be used individually to parse different parts of the code and extract the required information without using YACC but then it is not capable to perform strong syntax checking.

The main idea behind Lex implementation is regular expressions. It will work by translating these regular expressions to something understandable by C language. The output file from the Lex is not executable by itself. In fact the LEX will translate this code to be used as implementation of the routine yylex(). Different customer functions should call this routine in order to use Lex. The return value of this routine are tokens which are results of matching input string against different regular expressions.

Regular expression are using meta language, which is a set of characters defined as standard ASCI character set defined in UNIX and MS-DOS. The list of complete characters which form this meta language are obtainable in different sources but it is important to mention the inherited rules of Lex when it tries to match input stream against different regular expressions.
1-The Lex will only match a given input data against any regular expression once. It will also only match input data for one regular expression but not more.

2- The selected regular expression is the one which can make the longest possible match. If two regular expressions have the same equal length for specific input pattern, then it will use the one which is defined first in regular expression rules.

The matched string or character is accessible inside the LEX through “yytext” variable which is the pointer to the first element of the matched string therefore different process can be made on the result of this pattern matching.

3.2.4 YACC.

YACC [5] works with series of grammatical rules which together form a language. The essence of the compiler is automata machine which works with different grammatical rules. In each automata machine, there are a series of tokens, which act like alphabets of each language, then there are grammatical rules which determine how these alphabets combine together and forming a semantic phrase in that special language.

The C language has its own tokens and grammatical rules. If the programmer tries to violate these syntactical rules a syntax error will be encountered in the compile time. In fact what the compiler does is to extract tokens from the code and use them by applying different predefined rules and then translate them to a language which is understandable for Computers. It also takes some other steps in order to optimize the I/O operations and improving the speed of execution which is not in our interest in this master thesis.

If the compiler managed to consume all tokens and end up in one of the final states, it means the code has compiled successfully, otherwise there are syntax errors.

The grammatical rules are statements which consist of two different parts, called the left hand side and the right hand side of each grammatical rule. The left hand side of the rule always consists of one non-terminal token while the right hand side consists of one or more terminal or non-terminal tokens.

The example can be seen in the following:

Sentence → subject+VERB+object | subject+verb
subject → NOUN
object → NOUN

In the first approach, it had been tried to implement our own grammar for C programming language which is capable of analyzing different codes and present required information to the program. If it was successful, then presented information could be stored in different storage implementation, such as link
list or Marcove chain or hash table, and then continue to analyze that data from that step.

It took lots of efforts to end up with conclusion that making a grammar which is capable of parsing different C codes is not possible in the available time. The problem was the shift/reduce and reduce/reduce conflicts[3]. It is needed to give a short description about these problems and about their relation with grammatical rules.

**Shift/reduce conflict:**

Shift/reduce conflicts occur when there are two possible ways to parse an input stream, one of them will complete a rule (reduce) and one dose not (shift).

If we consider the following grammar, it has one shift/reduce conflict

\[
\begin{align*}
\text{e: 'x'} \\
| \text{e '}+' \text{ e} \\
; \\
\end{align*}
\]

Consider the input string “x+x+x”, as illustrated here, there are two possibilities to parse this code:

<table>
<thead>
<tr>
<th>x+x+x →</th>
<th>x+x+x →</th>
</tr>
</thead>
<tbody>
<tr>
<td>e+x+x →</td>
<td>x+e+x →</td>
</tr>
<tr>
<td>(e+e)+x →</td>
<td>x+ (e+e) →</td>
</tr>
<tr>
<td>e+x →</td>
<td>x+e →</td>
</tr>
<tr>
<td>e+e →</td>
<td>e+e →</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

*Figure 5: Shift/reduce conflict in a sample grammar.*

**Reduce/Reduce conflict:**
This conflict will happen when same token may be consumed by two available rules. A simple example can be seen below:

```
e: a | b | e+e
a: x|z
b: x|y
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x+y</td>
<td>x+y</td>
</tr>
<tr>
<td>a+y</td>
<td>b+y</td>
</tr>
<tr>
<td>a+b</td>
<td>OR</td>
</tr>
<tr>
<td>e+b</td>
<td>e+b</td>
</tr>
<tr>
<td>e+e</td>
<td>e+e</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

*Figure 6: Reduce/reduce conflict in a sample grammar.*

The program is incapable of parsing strings like x+y since it does not know if it should first reduce the ‘x’ token to ‘a’ or if it should reduce it first to ‘b’ and then reduce ‘y’ to b in order to be consumed by the rule determined for ‘e’.

One solution which has been proposed for solving this kind of conflicts is to define some kind of precedence for different tokens so that rules consuming tokens with higher precedence should be used first. The complete description of this approach can be found at (R.levine october, 1992)[6]

The problem might seem easy to avoid but in fact, it will be very difficult to avoid when the size and domain of grammatical language expands. The trial produced grammar includes more than 1000 shift/reduce conflicts and 500 reduce/reduce conflicts at the best cases.

So another approach should has been followed which use available written C grammar for ANSI C to parse the entire code.

The available Grammar was posted on one of the Compilers community by Jeff Lee In net.sources. It is adopted to complete ANSI C standard. It is capable of compiling ANSI C 88, which is much different from common C grammar standards like 99 and GCC. The limitation with this grammar is that it does not allow different decelerations in middle of the code. For example, It is not also possible to define a variable in middle of a “for” loop.

The implemented parser extracts required information from the code by using Lex. The lex will parse the whole code and match the code against different embedded regular expressions.
3.2.5 Flex/Bison

The Flex/Bison tools are GNU open source software [2] which has been used in this project since Lex/Yacc is licensed under Unix SVR_x and therefore it was not possible to use them free. In most of the cases, it is possible to work with flex/Bison instead of lex/yacc pair except some very special cases, such as if it is needed to modify the character stream in FLEX, while Lex will allows to define your own code for character stream, FLEX will not do the same”. From now on, the terms lex/flex and Yacc/Bison has been used interchangeably till the end of this report.

3.2.6 Solution

There are many options when making a parser which is capable of parsing different kind of input source codes. Different parser generators can be used except LEX and YACC such as ANTLR, Coco/R, GOLD ,JavaCC. It is also possible to use different languages for implementing the parser such as Java or C. Parsers are also different depending on different kind of languages they can cover. They can be divided in several main groups which are precedence parser, BC parsers, GLR parser and CYK parser. Our focus is on special type of LR parser which is LALR parser, capable of parsing the context free grammars.[4] The YACC and BISON are assumed as LALR parsers.

There are different scenarios available in order to extract required information from input module, but two main options which are focused in this master thesis are:

1-To store and extract the main information required for further analysis from the YACC part (which must be used together with Lex)

2- To store and extract the main information required for further analysis from the LEX part (even without presence of YACC)

Each approach has its own advantage and disadvantage. There are some benefits for the first approach. First of all, interesting information is obtainable from the source code. Different combinatorial statements in the code can be kept track off in much easier way in comparison with second approach. Proper actions can be put in front of each consumed rule to perform various operations, such as storing the value of each token or performing different operation on these values in order to analyze them later.

The disadvantage with this approach is that it is time consuming to develop a suitable grammar which is capable of parsing the whole code, because it should not have any reduce/reduce or shift/reduce conflict or any other kind of ambiguity. The other problem is that by switching around different version of the same programming language, the whole grammar might get useless or very difficult to fix. In one example, when the predefined grammar for ANSI C89 written by Jeff Lee has been tried in the development, since our targeted codes were written in ANSI C99, the difference between these standards makes it a little bit hard to evolve the previous grammar in order to parse ANSI C99 . For example one of the differences is that it is not possible to declare a variable in middle scope of a function block in versions older than ANSI C99. In order to do so, it is required to change lots of places in the grammar which itself may cause many shift/reduce or reduce/reduce conflicts.

The second approach has an advantage that working with regular expressions itself will not be very difficult, If regular expressions can be designed accurately. It is still possible to have some overlaps
when the parser reads the input stream for matching against different regular expressions. In this case, LEXER will automatically chose the longest possible match or if the match length is equal for all matched regular expressions, it will chose the earlier regular expression. Except this difficulty, working with Lex is quite straightforward. There are lots of predefined actions such as REJECT, which can be used to ease the process of parsing. There are also some facilities such as start states which will be used in many places to ease the developing process.

The disadvantage with this approach is extracting complex information from an input file will be more difficult in comparison with the first approach, sometimes it is needed to implement your own special stack and different flags to keep track of different states which parser gets into when parsing the input file.

Both of the previous approaches have been used in this master thesis, but the second approach was found to give more successful results in the available time. The current tool is the result of following the second approach.

The process which was followed in order to implement the tool with the first approach is explained here. It is also discussed why the second approach was not successful in given scheduled time although the tool which could be developed with this approach is capable of doing more strong parsing.

### 3.3 Developing parser with Flex/Bison

There are few things to consider when the goal is to develop a parser with flex/bison. The first step is to specify tokens which pass between flex and bison. They are elements of language, specified by the grammatical rules. The tokens should be specified in a file where the grammatical rules are located. This file is launched by Bison or Yacc for making the parser. Since return value of the function yylex() is an integer and Lex is actually returning different tokens each time, it is needed to give an integer value to each token specified in Yacc/Bison file. By compiling this file with -d option, Bison/Yacc will automatically generate a file called “y.tab.h” which is included in the lex file in order to give an integer value to each of the predefined tokens to perform token passing between lex and yacc.

The next step is to define proper regular expressions in Flex/Lex for tokenizing the input stream properly and then sending the obtained tokens to Yacc. The action part in Lex and Yacc can be used to perform different operations on the input tokens. If values of that token is required to obtain then Yacc should pass the value of that token as well. Otherwise sending the token is enough.

If passing the value of that token is also necessary, it must be done before returning the actual token to the parser. It is done by assign the value to “yylval”. Programmer can assign any value to this variable to serve as the value of the actual token but in most cases the value of yytext should be assigned to yylval. If the token can get several different types of values, such as character or integer, these types must be specified as a union type in the Yacc file. The programmer must express the value of each
token by using types specified in the union. In that case, it must be specified exactly which kind of value is being passed to Yacc by adding a dot (.) to the end of yylval.

The C language is selected for implementing the tool since the speed of execution is one of the key points when parsing lots of files and doing further analysis on the collected information.

Design solution is divided into several different steps. Different modules are explained on the following discussion:

Makefile

Make file is generated according to GNU C Make file rules to make the whole module. More information about how one should write a make file can be obtained through lots of different resources.

main.c

Main function handles the input arguments. It will read these arguments from the command line and call the proper function(s) depending on the specified argument.

It will also take care of user interface so that user has a convenient way through calling help option to interact with the tool.

Except calling different functions in the program, the main module will perform two more jobs. First of all, it opens the file specified by one of the options -M,-F or -C and passing the file pointer to yyin as the input source of FLEX. If we do not specify the “yyin” it will read the input from “Stdin” by default.

The main function is also responsible for calculating protection percentage of a input software code, If the hidden option -X is selected, it will print out all percentage protection related to each function on the screen and gives a report for the mean value of all protections, otherwise it will only show the whole protection percentage without specifying each module protection.

definitions.h

The different data structures which have been used in the project are defined in this file, all of the Macros for defining maximum array indexes are also defined in this file.

grammar.l

The lexical analyzer with its designed regular expression is located in this file. The main idea come from the same post by Jeff Lee about ANSI C standard grammar, but large modifications have been made to adopt it for our specific case.
This module is responsible for most of functionalities such as storing require information or making the final analysis from the gathered data and so on. The list of complete module and their functionality can be found in appendix A.

### 3.3.1 Gathering required information

It is now required to explain the relationship between different modules, their interaction and their special role in the whole program. 

In order to explain the solution for these issues, it is necessary to understand the structure of the Lex file first. The Lex file includes different regular expressions for finding different tokens. Whenever the lexer reads the input stream, it will match the input against one of these regular expressions to find the proper token. If it fails to match the input to any of these regular expressions, it will ignore it through a general catch all rule which is `.' .The tokens are divided to different groups. The first group is tokens defined as normal C keyword language. The examples of such tokens are “for”, “if”, “while” and etc.. 

The second group is special characters and operations such as '',';','&','+' ... and the third group which is one of the most important groups to extract is IDENTIFIERS, which is basically everything that is not included in previous groups. 

Here, in most cases, it is only important to store the values of these identifiers. These identifiers can have wide range of roles from a function name or function argument or different variables which must be stored. 

The main job is to differ between different identifiers. After this step, it is possible to store them in the proper data structures such as link list or hash table and then extract them in the analyzing step. 

The other regular expressions will be used to determine in which specific state the parser is located in right now. They are also used to count the number of parenthesis or brackets. The number of brackets will help us to determine begin and end of function scope. Whenever the parser is in the state in which the number of brackets is zero and it already detected a function call then it can be assured that the parser is in the beginning of a function block. 

The number of parenthesis is also playing an important rule for detecting different states. If the number of parentheses is zero, then it is guaranteed that the parser is not inside any function call, otherwise if the parser is inside a function call then it is possible to keep track of which depth of function call the parser is located in right now. For example, consider the function foo1(foo2(i),j); by counting the number of parenthesis, it is possible to keep track of the places where the parser is in right now. 

It is needed to explain more about regular expressions In order to understand how they work. Consider the following regular expression:

```
{[a-zA-Z\_\-\.]({[a-zA-Z\_\-\.]|[0-9]}|->))*
```

This regular expression catches all identifiers in the input text. The first part of this regular expression is for catching all input data which consist of small alphabet “a” to the last small alphabet ‘z’, the
“-” character between ‘a’ and ‘z’ inside the bracket means all regular expressions from letter ‘a’ to letter ‘z’. The same applies for “A-Z”, the” \_ “ and “\.” means that letters might include underline or dot(‘.’) as well. This is because there are lots of cases where there are variables like “a->size_of_array” or “a.size_of_array” especially when it is needed to point to specific member of a structure. The beginning of the second part is exactly the same as the first part. The vertical bar between different sections of the second part means exclusive OR, so it will match any of the specified patterns. The second part tells Lex that our regular expressions can include different combination of numbers from 0 to 9 and the arrow sign (“->”) as well. The “*” character at the end of the second part means to match zero or more occurrence of the second pattern. The Lex gives this facility to user to substitute different patterns with arbitrary variable names to increase the readability and abstraction of regular expressions, so in the current regular expression, the alphabet pattern has named with “L” and digit pattern with “D” which results the following regular expression:

\{L\}(\{L\}|\{D\}|(\-\))*

Other regular expression which has been used in the lex file is fairly easy to understand without any further explanations.

The other concept which has been used a lot in the Lex file is the start state concept. The Lexer can switch between different states when it is parsing the input stream. It can remain in those states until the end of processing and then jump back to the initial state. This is a very useful feature of Lex which makes it easy to perform different parsing simply with defining a separate state. Consider that we are interested to extract all numbers in an input file but only when they are followed by alphabets. If the same definition of numbers and alphabets as above is used then the following code can be written in Lex file:

\%x state_1

\{L\}* \{printf("matched");BEGIN strart_1\}

\<start_1\> \{D\}* \{printf("Number found after alphabet");ECHO;BEGIN 0;\}

\<start_1\>. \{BEGIN 0;\}

In the above example, the symbol “%x” has been used intentionally to show an exclusive start state. There is also another kind of start state which is defined by “%s” symbol. It is different from previous symbol in the sense that when program triggers the normal start state (“%s”), regular expressions in the initial state can still be matched, so the program with normal start state will match all letters and alphabets when alphabet string has been seen.

In the project code, only exclusive start states have been used to determine different states of parsing the input stream. One of the start states usages in this project is to extract different patterns in divisions and array indexes, since the common rule for identifiers is not enough for these problems. Suppose that it is needed to store the (a+b) in the input string (1/ (a+b)) in order to check against division by zero. The general identifier rule will divide the expression (a+b) to two different tokens ‘a’ and ‘b’ and recognize the plus operation as common C syntax, so it is not possible to embed some functions in the action part of regular expression related to identifiers to store the whole expression easily. The solution is to introduce different start state which the function will get whenever it encounters a division followed by a parenthesis, since it is where the whole expression needs to be stored instead of its constructing elements. Number of open parenthesis is also needed to be considered. By storing this
number, it is possible to make sure that the parser is at the end of current parsed expression whenever the number of open parenthesis will get equal to the number of closed parenthesis at the end of a regular expression.

```
"/" { 
    BEGIN division; nr_parenthesis2++; }

<division>"(; {nr_parenthesis2++; }
<division>"){nr_parenthesis2--; 
if (nr_parenthesis2==0) 
    { 
        BEGIN 0; 
    }
}
<division>\n {line_nr++; }
<division>[^\n]* { 
    count(); 
    if (add_id(global_id)->main_function) 
        { 
            set_protect_required(global_id, yytext, line_nr); 
        }
}
<division>"/" ;
<division>. ;
```

Figure 7: Different start state embedded for catching divisions in the input code.

The same approach followed for array indexes, the only difference was to go to the different start states whenever the parser encounters the “[“ sign which is an indicator for starting of an array index.
One of the important Lex routines is yywrap(). It is responsible for opening different files and passing the file pointer to Lex. After finishing processing a file, each time Lex calls yywrap() to obtain a pointer to next file which should be parsed. If there are still more files, yywrap() will pass the file pointer to yyin (the Lex input file) and return zero to indicate that there is a file ready to parse, otherwise it will return one and Lex will terminate the parsing step.

The default yywrap() routine is only suitable for parsing one file. It will only return one after Lex encounters EOF (end of file) and calls yywrap() to get the next file pointer. The default definition of yywrap() is like following:

```c
int yywrap(void)
{
    return (1);
}
```
In order to parse several files, it is needed to redefine the default definition and define our own version of yywrap() routine. The specific yywrap() definition, which has been used in this thesis will be explained in later parts.

Different functions are called in specific places in Lex to store required information in order to perform further analysis. For example in the following code, these different processing steps are visible.

```c
File1.c

Foo ( int a, int b, (data)*d )
{
    DBC_PRECOND(a != 0 && b>0);
    int c;
    char *e;
    e=foo3(d->word);
    c=foo1(a);
    d->value=c;
    c=c+foo2(b);

    DBC_POSTCOND(c);
    return(c);
}
```

*Figure 9: A sample program which is given as an input to the tool*

First “yyin” variable which is a file pointer will be directed to read the File1.c. If the file successfully has opened, the parser will begin parsing from first line of this file, it will try to match FOO against different regular expressions. “Foo” will match the general IDENTIFIER rule since it does not match with any upper regular expressions, in identifier actions there are lots of different functions for taking care of required operations.

One of the function is check_type_extra() function, which will detect all function calls. It will scan input stream till it get to first open parenthesis. If it encounters an open parenthesis, it return one to show the current identifier is actually name of a function, otherwise it return zero. The definition of the routine check_type_extra() is as follow:
Check_type_extra()
{
    DO just read the input character one by one
    WHILE the input character is space (' ') or tab('	') to make the parse ignore all white spaces
    IF the last read character is open parenthesis '('
        Unput the last read character which was not space or tab into input stream again
        RETURN true to indicate the last identifier is a function name
    ELSE
        Unput the last read character which was not space or tab into input stream again
        RETURN false to indicate the last identifier is not a function name
    END IF
}

Figure 10: Function check_type_extra() pseudo code for detecting functions name

After detecting FOO is a function name, it should be possible to discriminate if the current function call is actually a function call inside a function block or if it is beginning of the function block. In order to do so, it is needed to keep track of number of open brackets. The parser will hold an internal variable which will show current number of the brackets, each time the parser encounters a close bracket it subtract one from this number. If the parser detects a function call and the number of open brackets is zero then two options are possible. Either the current function name is a function prototype at the beginning of the file in definitions section, or it is beginning of a function block definition and it is needed to be saved as function block name.

After detecting a function call, the solution for parser to be able discriminating between these different scenarios is to read all input strings first and then it will check if it encounters any open bracket before end of statement which is determined with semicolon. In that case it can be ensured that current function call is actually a name of a function block, otherwise it is only a function prototype and since it is not useful information for tool to be stored, it simply disregard the name by using the function delete_id which will delete the information regarding the module determine by its unique id.

But in previous example, the parser will detect that FOO is the name of function block, it will then read open parenthesis and increase the number of open parenthesis by one. The next step is to read “int” which will be treated as regular C definition so it will be ignored in this case. The next line of this example shows defining a pointer to characters. There is a special mechanism for dealing with pointer definitions which is to store the name of all pointers declared in one function scope. These pointers are needed to detect functions which return some pointer to some defined variable inside the function scope, otherwise detecting such functions would be much harder. The first thing to find out is to check if the function is a library function or not by consulting the library function list since the return value of library functions does not need to be protected. After storing the pointer name, each time a function assigns some value to some variable the parser will check if the assigned variable is a pointer. In that
case, the value of that variable should be protected by the mean of some DBC_ASSERT macro to ensure that the variable is not null. This process is exactly what should be done for the next line in our sample code where the function “foo3” return some pointer to the variable “e”, so “e” should be stored to check against the list of variable protected by assertion inside the function scope.

All other identifiers which are inside function parenthesis should be considered as the argument of the current function. The approach has been used to identify to check if current parser position is inside a function argument area is the same approach for detecting beginning of a function block except that the parser will maintain a Boolean flag to know if the last read identifier was a function call or not. If the parser read a function and number of open parenthesis is greater than zero it can be assured that the parser is in argument definition area. But decision making is not finished yet, the parser has to discriminate between real arguments and their types which can be assumed as identifiers themselves. For example, “d” argument in the upper example is qualified with data structure of type data but since data is not regular type definition (such as integer or float) the tool has to detect that data is not actual argument. Thesis kind of detections is done by built-in function which uses predefined actions in Lex/Flex such as input, unput();

The mechanism is to read input stream till the parser gets to a semicolon or a close parentheses. If the parser gets to the semicolon, it can be assured that current identifier is a argument otherwise if the parser gets to a parenthesis, it has to look at the next character to discriminate between casting inside the argument area or end parenthesis of the argument area. For instance, consider the end parenthesis after casting the variable ‘d’ to type called data in the previous example. If the parser do not separate these two different states from each other, it may end up assume data as argument of the function “foo”. The solution is to read the next character to see if it is one of the characters ‘,’ or ‘{‘ or ‘;’ then it can be assured that current parenthesis is the end parenthesis of argument area and the current identifier is an argument, otherwise that is probably a casting which should be ignored in this case. The definition of the function check_type_extra_extra() is shown in the following:
When the parser gets to DBC_PRE_COND macro, since exact regular expression for extracting this macro is embedded in Lex file in upper levels of the parser rules, the parser will detect this macro as a separate token and it will not consider the general rule of identifier for them. Afterward it is needed to store all variables protected by this macros in the identifier section (since they are actually identifiers).
This has been done by flag mechanism in the same way as the argument case. There is a general flag variable which will be changed by different parts of parser. This variable will be checked in identifier section to find out in which one of the implemented link lists the current variable should be stored. In this case, the parser changes the flag to one which will signal identifier section that coming identifier should be stored in variables link list which shows variables which are protected by precondition. It is necessary to store these variables in order to perform further analysis later.

The next line is defining an integer variable “c” which is not of our interest. The next line is a function call which will return some value to the variable “c”. Since it is needed to keep track of function calls, the parser will store the function calls names as functions which has been called by the function FOO. It uses the same mechanism as mentioned before to store the list of called argument of this function. The only different is whenever it detects an argument inside the scope of current function call, it will traverses the list of protected variables with precondition and sets the corresponding protected array of this function call for this argument to one in the case that the argument has been protected. The protection array ha integer type which has been considered for all function calls in order to store the list of their protected variables. Considering “foo” function in above example, if predefined size of protection array is assigned to be five, it also show the possible maximum number of arguments in an input software unit. In previous example, the corresponding protection array of different functions will look like this:

<table>
<thead>
<tr>
<th>Function name</th>
<th>Protection array</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foo</td>
<td>11000</td>
<td>The first and second argument has been protected.</td>
</tr>
<tr>
<td>foo1</td>
<td>10000</td>
<td>The first argument has been protected.</td>
</tr>
<tr>
<td>foo2</td>
<td>10000</td>
<td>The first argument has been protected</td>
</tr>
</tbody>
</table>

The next line is also an assignment which is not of our interest in this case. Afterward there is a function call which is treated as it has been explained. The following line is declaration of post condition which should be treated exactly the same as precondition. The last line is a return declaration. The parser should store all returned variables to perform further analysis. It should also check if they have been protected by the post condition.

In order to perform further analysis, the parser needs to store all functions names which return some value to some variable as well. In these cases, name of that variable together with that function name will be stored. They are used for fulfilling the second requirement which is detecting extra contract checks, since if the code already protected by putting some post condition on return value in the called function, protecting this variable inside our current function scope is unnecessary and it needs to be detected. The exact explanation of why it is needed to store this information will be explained in later parts of this report.
Totally, operations in identifier rule action part can be described briefly as follow, first the parser will check if the input identifier is a type name. If so, it will just set a flag to one to know that it saw a type name and exit the identifier rule, then it will check the other flag which is set by the equality rule action part to know if the current identifier is the next identifier to equal sign or not. In that case, it will store the current identifier inside. The next necessary step is to store protection required variables which are pointers assigned to return value of function calls. The parser will check if those assigned pointers are protected with DBC_ASSERT macros to check against NULL pointers. Afterward it will check the global variable flag which is set by other part of Lexer to know what kind of identifier has been read and what the parser should do with this identifier. Depending on flag value, it will store the identifier in proper sections.

### 3.3.2 Different data structures

There are several different data structures which some of them are needed to be explain here

![Data structures](Figure 12: Data structures used in the tool for storing different information)
3.3.3 Different stack implementations

In order to keep track of different states, it is needed to implement different kind of stacks. Consider the following nested function call.

```
Foo(e+foo1(d+foo2(a,(void*)b)));
```

The stack will be used to determine the exact arrangement of nested function calls and their arguments. In this example, first FOO will be parsed and since it is a function, it will be pushed on the function stack and then the variable “e” will be parsed. The parser then will consult the stack and will find out that the last function call was FOO, so the variable “e” should be considered as an argument of the function FOO. It is also needed to keep the parenthesis trace after each function call to know where the area of that function call has been ended to pop that function from our function lists in the stack.

The next function is foo1, FOO is already in the function stack and “#FP” is in our parentheses stack which stands for function parenthesis.

“foo1” will be set a function call inside the function which is on top of the function stack right now, it means “FOO”, then “foo1” itself will be pushed on the function stack and one more function parenthesis or “#FP” will be pushed on top of the function parenthesis stack, the next level is to assign the “d” as a argument of last function which is on the top of the stack, it means “foo1”, then the parser come to next function foo2, it will consult the function stack and assign foo2 as a called argument of function foo1 and add one more “#FP” to the function parenthesis trace. The parser will read “a” and do exactly the same until it comes to the casting of variable ‘b’ to (void), now since the parser did not detect any function call before the parenthesis of this casting, it will push normal parenthesis or “#NP” on the stack top.

By encountering the close parenthesis after void it will pop the “#NP” from parenthesis stack and then add ‘b’ as argument of the last function on top of the stack, which is foo2. Now, the reverse trend of the previous actions should be followed. For each close parenthesis, the parse will consult the stack parenthesis and will find out that the current close parenthesis is related to a function parenthesis, it will then pop a function from function stack each time it sees a close parenthesis until the closed parentheses are finished and there is no more function in the stack.

The other stack implemented for the current tool is a special stack for tracing “if” and “for” statements. Suppose that we have the following code as input module.
Void Foo(a,b,c)
{
    Int i;
    Extern int Max_Index;
    Extern float A[Max_Index];
    For (i=0;i<= Max_Index; i++)
    {
        If (a !=0 & & b != 0)
        {
            If (c != 0)
            {
                A[i] = (a/b+b/a)* 1/c;
            }
        }
        Else
        {
            A[i]= = (a/b+b/a);
        }
    }
}

Figure 13: A sample program used to show how stack implementations works

In the above code, none of the variables need to be protected. The reason is the variable “b” is already protected by if statement and the index variable of the array “A” is already assigned some boundaries within the FOR statement. Although there is still a chance that the boundaries will not be assigned a suitable variable in the range but the decision was to trust the programmer in this case.

The parser will store each variable protected by IF and FOR statement inside one level of this stack. The Stack has been implemented as a array which each of its elements points to the beginning of variable list protected by the specific IF or FOR statement, whenever the parser gets to some variable which is needed to be protected, it consults this special stack and if the variable is not already protected with IF statement or if it is not inside the “for” statement, it will then add it to the list of protected required variables. For searching inside this stack, it is needed to search all levels and all variables to detect if a specific variable is already protected or not. Mixing the stack implementation with link lists makes it easy to omit the required variable together with only one “pop” operation when the IF or FOR
statement is not in effect any more. The parser will pop each level of this stack when it encounters a close bracket regarding an open bracket of IF or FOR statement until the stack gets empty. The structure of this stack is illustrated in the following:

![Figure 14: Stack implementation](image)

As you can see, each level of the stack is a pointer to the beginning of a list, containing different variables protected with IF or FOR statement, the above example can show four nested IF statement, each one protecting their special variables.

The other issue in collecting the required information is how to discriminate between different definitions and actual variables which needs to be protected, for example in external definition of integer array A, Max_index should not be treated as an array index which needs to be protected. If the definition appears somewhere before the function block, it is quite straight forward to detect it by reading the value stored in number of bracket variable but if the parser is inside the function scope the problem needs to be solved in other ways. In order to detect this, the parser maintain a global flag called last_type_name which will be set to one, whenever it sees a type specifier such as int, float ... And store the array index for example only when the type specifier is not presented in beginning of this variable.

But types can also be specified in other ways. The programmer can define any arbitrary type by typedef command and changes the enum or struct definition names to other names which can be used as a type specifier. This can be done at the beginning of the code or somewhere else in the included headers. In the first case, the parser will keep track of all type definitions with “struct” or “enum” and store the list of predefined types in this way, in the second case, the parser should get the list of predefined types in the config file, the specification of the config file can be found in later parts of this report. After extracting these kinds of type’s definition, whenever the parser read an identifier, if the name of that identifier was between one of our predefined type definitions, it simply changes the type_flag to one, otherwise the flag will remain zero. The parser do this checking by the use of function search_type_name(char *s)
The structure of the parser with one or several input files can be illustrated as follow:

![Figure 15: Involved modules in parsing one or several files](image)

As above picture suggests, the main module will pass the pointer to input file(s) to the Lex, then it will parse the whole file to obtain the specified tokens and passing them to their proper C functions in LIB.C. The value of tokens and their information will be stored in their proper symbol tables. Lex will call “yywrap()” routine to know if there are more files to process, If there is more than one file, yywrap() will open the next file and pass the file pointer to LEX and return zero to show the next file is ready to process, otherwise It will return 1 and parsing job will be finished. The analysis part is performed by the implemented C functions through their interaction with stored tokens in the symbol tables. The final result of analysis is printed out to the user.

### 3.3.4 Config file:

The parser needs a config file in order to fulfill different requirements of the tool. It is also useful to ease the process of giving required information to the parser. The config file consist of different sections which are:

The first section of the config file is dedicated to library functions, all of the library functions should be specified in this section. The return value of the library functions need not to be protected since they assume to be protected internally (if some error happens in the library function, the program will ended up with error massage);

The next part is for special definitions. All of the special types which are defined in other parts of the code and has not been in the scope of the current parsing should be specified here. The parser will check the list of those specified types each time to know if the current identifier is actually a type name or not.
The following part is the root function definition. In order to fulfill the last requirement, which was finding the least protected required areas; it is needed to know about the root function. The input module might have one or several root function.

The next section is the only part in the config file which needs to be filled. The directory includes files which should be parsed by parser. The parser will send this directory to the other module call, find-file, to find all C files in that specified directory with its subdirectories.

Figure 16: Config file general structure. The required information should be filled in the area indicated by striped arrows.
3.3.5 Find-file module

This is the module which should find different files in the directory specified in the config file. It has been implemented using different C libraries which are <dirent.h> for using the function readdir(), <sys/stat.h> which includes the function stat() and <unistd.h> which includes the function getcwd(); It has also used lots of other regular C libraries like <string.h> and <stdio.h> and etc..... These libraries and their functionalities can be found on different links on the web [some link], so it is not needed to explain more about their functionality here.

The find-file module includes three different functions, the first function is findfile_main(char* dir_path, char* pattern) which is the main function. The input arguments of this function are the specified paths in the config file and the required pattern which is .C files in our case. The flexibility in the pattern section can be used for parsing the other input files like c++ files (.Cpp files). The main function will begin checking if the specified path in the config file is actually a directory. After this check, the recursive function findfile(char* pattern) will be called, this function will search for all C files in current directory and store them in a common variable which is an array of strings(pointer to character) and do this recursively until all of the subdirectories has been traversed once. The recursive function find-file will detect the type of each file by the use of function is_typed (char *name,char type) which is the third function in the find-file module.

After finding all of the C files, the result which is an array of characters pointer (beginning of each string) will be passed to the Lex for beginning the parsing process by opening files one by one.
3.3.6 **yywrap() routine**

As mentioned before, the default yywrap() routine is only useful for opening one file so specific implementation of yywrap() is used in this thesis, in order to achieve required goals of the project. The yywrap which is used in our implementation has the following definition:

```c
yywrap()
{
    fclose(yyin); // close the previous opened file
    yyin=NULL; // set the pointer to initial NULL value

    FILE *file; // define new file variable

    switch (service_id) // test service_id for possible values
    {
        case 0: // service_id zero belongs to config file mode
            if(max_configg2>=0) // test to see if there is a positive number of files to parse
                             remaining
            {
                assert(lib3[max_configg2]); // check if the current file pointer points to not NULL location

                // printing name of the file
                printf("\n open file %s %d",lib3[max_configg2],max_configg2);
                file_namee=strndup(lib3[max_configg2]); // setting global variable file_name to name of the current file. This variable is used to pass the file name to different called function in the parser
                initialize_lexer(0); // initialize the status of lexer to zero
                file=fopen(lib3[max_configg2],"r");
                max_configg2--; // reducing total remaining number of files
                if (file !=NULL)
```
\{  
  \texttt{yyrestart(file); // this function is a built-in function in lex \& yacc to restart the file pointer}  
  yyin=file;  
  if (max_configg2<-1) // check to see if still there is any file to parse  
  {  
    \texttt{return(1); //return one to indicate that there is not more file to parse}  
  }  
  \texttt{return(0); //return zero to indicate that the next file is ready to parse}  
\}\n
\texttt{fprintf(stderr,"cann't open a file %s \n",lib3[max_configg2]);}\n\texttt{return(1);}\n\texttt{}\n\texttt{return(1);}\nbreak;\n
\texttt{case 1: \textit{service id one belongs to multiple file mode}}\nwhile (filelist[currentfile] !=(char *) 0) //do this till the last file pointer\n{\n  \texttt{printf("\n searching for the files ");}\n  file = fopen(filelist[currentfile++],"r");\n  assert(file != NULL); //check to see if the file pointer is valid\n  if (file !=NULL) // the following loop work the same as config mode\n  {\n    \texttt{yyrestart(file);}\n    yyin=file;\n    \texttt{printf("opening file %s \n",filelist[currentfile-1]);}\n  }\n}
As it can be observed, it will first free the previous file pointer. It will then make switch statement on the specific service_id which is defined by user through the interaction with main function. There are three different cases regarding the specific service requested by the user. If the user needs to do parsing through the config file (Service_id zero), yywrap will check if there are still more files to parse and open the next file through the share array of pointers to file paths, it will then return zero to Lex, if there is no more files or if the current file can not be opened, yywrap() will return one and the parsing will be terminated.

If the user chose to open multiple file(service_id one), it will do the same thing except that the shared storage array of file names is a different array.

Finally, if the user wants to open only one file (service_id three), yywrap will just simply return one to show that the parsing has been finished.
- **Initialize lexer() function**

As it can be seen in the previous section, the yywrap routine call a routine call initialize_lexer() each time it wants to start passing the next file. As it is mentioned before, there are lots of different status flags for keeping track of different parser positions. Some of these values need to be initialized after each function scope and some of them need to be initialized only after finishing each file parsing. Depending on these two different states, the parser will call initialize_lexer() routine with input argument zero for the first state to initialize some flags partially after each parsed function scope or with one to initialize all flags and also emptied different stacks contents after finishing each parsed file scope.

The structure of the program with config file can be illustrated like following:

![Figure 18: Involved modules in parsing with config file.](image)

First the config file will be fetched into Lex. Lex then will parse each line of config file with different start states embedded in the lex to parse the config file. Lex will extract the required information such as special types and library functions. It will then looks for path directories. After finding the directories, where the target files are located in, these directories will be passed to find-file module in order to find different C files in those directories. Find-file will store file names and the path to those files in a global array and pass this array to Lex, the yywrap() routine in LEX will open the first file and pass the file pointer to LEX. LEX will then continue parsing the file same as previous case without config file until it finish parsing the first file. After this step, it will again call yywrap() to obtain the pointer to the next file until yywrap return one showing that there is no more input. The parser then will go to analysis step and will print out requested information to the output.
3.3.7 Different functions implementation in Lib.c

The most part of the C codes, which are used to collect and perform the further analysis are embedded in the module called Lib.c explanation of some of the function in this module can be found in following

**Recursive Auxiliary functions and their main functions**

There are several recursive functions in Lib.c to perform different functionalities. They mostly used as auxiliary functions for other defined functions to help them for performing their specific tasks.

- Search_recursive_function_called
  
  This function is used to locate a function inside the nested function call of one module, it begins by search the function name inside the first level of function call in the pointed module, if it finds the function name, it will return the pointer to that function call structure otherwise it will look to the functions called by the called function and so on until it traverses the whole nested function calls. If it is unable to find the function call, it will return a NULL pointer to indicate that. The code of this function can be seen in below:
search_recursive_function_called (parameters: function id number, pointer to name of the searched function, pointer to the struct function call)
{
SET an internal counter and initialize it to zero
Increment the counter once and check if it is not bigger than some predefined big value, it is done to avoid infinite loop
IF the searched function name and address of function call structure are valid
    FOR all the called function names inside the address of function call structure and their called function
        IF the searched function name and the function name of function call struct are similar
            RETURN pointer to that function call struct
        ELSE
            RETURN search_recursive_function_called(arguments: same function id number, pointer to the name of the searched function, pointer to the next function call struct)
        END IF
    END FOR
ELSE
RETURN a null pointer
ENDIF
}

Figure 19: Function search_recursive_function_called() pseudo code for locating different function calls inside one function block.

- **Search_in_function_called**

This function is used to find a function identified by its name in a structure which is identified by its unique id. It will use the function search_in_function_called_aux as a auxiliary function to search that specific function. If It managed to find the function It will then return a pointer to specific module which that function called, otherwise it will return NULL.
search_in_function_called(parameters: function id number, name of the function which should be searched)
{
  Find the pointer to the module identified by its unique id
  FOR all the functions which had been called inside the current function
    IF the name of the called function is equal to the name of required function
      RETURN the pointer to that specific function
    ELSE
      IF this function has called another function
        GET the retrun value of the function
        search_recursive_function_called(arguments: function id number, name of the function)
      END IF
      IF the return value points to the valid function name
        RETURN the function name which has been found
      END IF
    ENDIF
  RETURN FOR
RETURN NULL
}

Figure 20: Function search_function_called() pseudo code for locating different function calls inside one function block with the use of search_recursive_function_called()

- **check_extra_attention_required_aux**

  This recursive function is used by check_extra_attention_required as auxiliary function and will determine if there is any duplicate check in the nested function call tree, if it detects duplicate checks, it will return 1 and set a predefined global pointer to the name of module which cause this double protection, otherwise if it could not find double protection in the current node it will go on step deeper but with the condition that the current node get its return value from the other function which is actually the next node. it will be called recursively until it reach the leaf of the tree. if the leaf is protected by post condition, the protection in all the nodes up to the beginning node is extra and can be removed. To indicate this, the function will return one and set the global pointer to this leaf of the tree. If it could not find such a place, it will return 0.It is needed to explain check_extra_attention_required function which is the main function which calls this function as auxiliary function. The implementation of this function can be seen below:

41
check_extra_attention_required_aux(parameters: pointer to struct module which is needed to analys)
{
    IF the pointer to the current module struct is NULL
        RETURN 0
    ELSE
        FOR all function calls inside the parsed code which is not visited yet
            FOR all return parameters inside the parsed code
                IF the function returns some value which is obtained from the above function call
                    IF the current function has post condition
                        FOR all variables which has pos condition in the current function
                            IF the post condition is on the same variable which is returned by this function
                                SET some pointer to the current module which is caused double protection in upper modules
                                SET the current module as a visited module
                                RETURN 1
                            ELSE
                                IF the called function of the current function is defined in the parsed file(s)
                                    SET the current module as a visited module
                                    RETURN check_extra_attention_required_aux (argument: a pointer to the called function module)
                        END IF
                    END IF
                END IF
            END FOR
        END IF
    END IF
END IF
END FOR
IF the current function does not have post condition
    RETURN 0
END IF
IF the current function dose have post condition
    SET some pointer to the current module which is caused double protection in upper module
    RETURN 1
END IF
END IF
}

Figure 21: Function check_extra_attention_required_aux() pseudo code for fulfilling the second requirement
• check_extra_attention_required

This function is used to print out the list of all duplicated checks. It will print out if there is double assertion check or post condition check by traversing the call tree of the modules until it reaches the leaf of the tree or find a double protection area.
check_extra_attention_required()
{
FOR all modules inside the current parsed file(s)
    FOR all of the function calls inside the current module set the visited flag to false  
    END FOR  
END FOR  
FOR all modules inside the current parsed (s)
    FOR all function calls inside the current module  
        FOR all return values of the current module  
            IF the name of the called function is equal to the name of the return value  
                FOR all variables which have post condition in the current module  
                    IF the name of that variable is equal to the name of return value  
                        IF the return value of the function  
                            check_extra_attention_required(arguments: pointer to the called function which the current function gets the value from that) is true  
                                Print “the post condition of the current module is duplicated because of post condition indicated by the passed global pointer from the function check_extra_attention_required_aux”  
                            END IF  
                        END IF  
                    END IF  
                END FOR  
            END IF  
        END FOR  
    END IF  
END FOR  
END FOR  
FOR all modules inside the current parsed (s)
    FOR all function calls inside the current module  
        Set the visited flag to false  
    END FOR  
END FOR  
}  

Figure 22: Function check_extra_attention_required() pseudo code for fulfilling the second
Analytical functions:

- **pre_cond_check**:  
  This function is used to check if all input arguments of a function are protected by the precondition macro. It will check the list of input argument against the list of variable protected by precondition in that module and if it found any unprotected parameter, it will worn the programmer.

- **post_cond_check**:  
  This function is used to check all return variables to find out if they are protected by post condition. It will compare the list of return values against the list of protected variables in one module.

- **Assert_cond_check**:  
  This function is used to check if all protection required variables such as array indexes or divisions or pointers to the return value of functions are protected with assertion. It will perform this in exactly the same way as two previous functions.

- **search_inout_functions**:  
  This function is used to determine the least protection required areas in order to fulfill the requirement 3. It will notify all the variables which are assigned to the return value of some external function call. It will also search in the called modules to find the root functions and express the demand to keep the variable of this root function within protection.
Search_inout_function()
{
FOR all defined modules in the code
    FOR all variables which get some value from a function
        IF the function which return some value to that variable is not inside the defined functions which means that it is defined somewhere outside the current SWU
            IF the function is not a library function which the output should be automatically trusted
                SET a pointer to the structure of type protection required contains that variable, it’s function and line number
                SET the obtained pointer in previous step in the list of other obtained pointers (if any) in the last execution of the loop
            END IF
        END IF
    END IF
END FOR
PRINT the list of variables which has been obtained in the loop with their location including file name, Line number and their module name.
FOR all root functions found in config file
    FOR all modules inside the parsed code
        IF the name of the root function is equal to name of the module
            FOR all of the arguments of this found root function in the parsed modules
                PRINT the list of all argument of this root function which should remain within protection.
            END FOR
        END IF
    END FOR
END FOR
}

Figure 23: Function search_in_out_function() pseudo code for fulfilling the third requirement.
3.3.8 Fulfilling the requirements

Requirement 1

The analysis part will begin after gathering required information from different parts of the code. The first requirement is to check if all protection required places are actually protected with proper macros.

This has been done by going to different lists which the parser maintains for three types of macros. Each list contains name of variable which has been protected by one of the Macros, DBC_PRECOND, DBC_ASSERT, DBC_POSTCOND. The parser matches them against their corresponding lists which are the list which should be protected by these different macros. If the parser found a variable which already exists in the second list but was not protected with its required macro in that module, it will alert the user about that. The modules are separated from each other by assigning a separate unique id for all modules. Each module points to the different data structure of type module_info which has been explained before.

The MAX_MODULE in this data structure determine the maximum number of the modules in the input source code. The parse store different information about each function in its predefined module structure.

It only remains to sort out the output. Since each argument has been stored with the line number in which that specific argument has been seen, it is needed to report the output arranged by their line number to the output screen. It is implemented using the sort link list algorithm which will sort the link list containing “the Struct word” elements by their line number.

Requirement 2

Since the DBC macros, takes lots of memory and decreases the execution time, it is very useful to exclude the extra contract checks in the whole module.

The parse uses graph traversing algorithms in order to fulfill the requirement number 2. It will first go through the function call graph and parse each element of this function call. There are two cases that protection might be detected as not needed.

1-In order to explain the requirement number 3, It might be required to explain some data structures. The first one is called function_call_list. Each function call inside the function scope which returns some value to a variable is stored in the structure called function call.
typedef struct function_call{
    char *word_name;
    word *called_argument;
    int protection_array[max_index];
    int function_number;
    struct function_call *called_function;
    struct function_call *next;
} function_call;

The word_name is the name of the function which has been called. The arguments are stored in the link list of called_argument. The protection array is used to detect which variables has already been protected. Protection array is used to keep track of variables which have been protected in the current module. The called_function inside this structure will be used to determine the next nested function inside the current function call and the function_number will show which argument of the current function was actually a function itself. The next pointer points to the next such kind of structure.

If the input arguments of a static function, which is called by another function, have already been protected by precondition in its calling function, the extra protection in that static function is not needed.

It might be tricky to detect different variable names since the name of the arguments of the called static function might be different by the names which has been used in defining that static function in the first step.
Consider the following example:

```c
void Foo(int a, int b)
{
    DBC_PRECOND(a !=0 && b >0);
    float c;
    C=foo1(a,b)
    DBC_ASSERT(c >0);
}
Static Float foo1(int x, int y)
{
    float z;
    DBC_PRECOND(x !=0 && y >0);
    z=1/x +sqrt(y);
    DBC_POSTCOND(z>0);
    Return z;
}
```

Figure 24: A sample code showing how the second requirement is furnished.

The variable ‘a’ and ‘b’ has been protected by the function FOO, but they have been named differently in the called function to ‘x’ and ‘y’. In order to check these cases, the parser exploits the use of protection array for each function and called functions inside that function. For example If the maximum size of a protection array has been considered five, which means the maximum number of allowed arguments, the protection array of each function can be seen in the following.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Protection array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foo</td>
<td>110000</td>
</tr>
<tr>
<td>foo1(which is called inside the function foo)</td>
<td>110000</td>
</tr>
<tr>
<td>foo1(The foo1 function block definition )</td>
<td>110000</td>
</tr>
</tbody>
</table>
After detecting that “foo1” is a static function, the parser compares the protection array of the called function and the actual defined C function. For each variable which has the same value in the protection array, it will report the extra contract check.

2- The second scenario is when the return value of a called function is stored in some variable which is protected by post_condition or Assert condition, which can be removed since the called function has already protected its return value by other post condition. There might be other cases that return value comes from other functions which is called by the called function itself. If the third function has already protected its return value, the post_condition in the second function and the DBC_ASSERT or DBC_POST in the initial calling function is not needed.

For explaining the implementation of this requirement, it is needed to explain some data structures used namely as attention required structure:

```c
typedef struct attention
{
    int is_visited;
    int line_number;
    char *called_function;
    char *assignee;
    struct attention *next;
} attention;
```

Each node of the function call graph is actually an element of the above type so it is possible to traverse the whole graph by traversing from an arbitrary node. The first element is a flag for keeping track of traversing the graph when looking for extra contract checks. If the parser has already visited the current function call, is_visited flag will be set to one in order to avoid visiting the same node twice. The called_function will be assigned to name of the function which has been called inside the function scope. The assignee will be the variable which is assigned to the called function return value. The line_number will be set to the line number in which this assignment has been made. “next” is a pointer to the next structure in the link list.

The algorithms begins with parsing from an arbitrary node and continue traversing unvisited nodes which means tracing the whole graph to find out where the value is actually computed until it get to one of the end points. The end points are places where return some value to the upper levels of this graph. It also has this property that they do not call any other function. If this end point is protected its return value with post condition, all post condition and assert condition up to the first function is unneeded otherwise the parser cannot decide what to do so this will simply be ignored. It can be illustrated with the figure below:
Each arrow in this picture shows a function call from the calling function to the called function. As this picture suggests, the blue nodes in the bottom of the graph are end points since they do not call any other function. The root function is shown with black color which is where “function call” graph starts. The analysis algorithm can begin from any of these functions and traverse the whole graph until it gets to the endpoints.

Consider the previous example, since the function “foo1” has been already protected its return value with post condition, the assert check in function “foo” is not necessary. The same case might be applied if there was DBC_POSTCOND in the “foo” function as well. It is also the same if ”foo1” was not a static function.

**Requirement 3**

The requirement number 3 needs to be applied when the developer teams has finished with developing process and they need to deliver the code to the other teams. In this case, it is useful to have the DBC macros as less as possible due to their memory and time consumption. In order to detect where places with least protection required areas are located, the user needs to specify a root function in the config file.

The input arguments of the root function(s) are defiantly inside the area which should remains within protection.
If a variable gets some value from a function outside the current parsed software unit, it is also needed to remain these areas within protection.

To detect these variables, the parser goes to all function calls and their assigned variable. It will check if they have been declared inside current software unit otherwise they should remain within the protection.

4 Extra features

4.1 Disable warnings

Disable warnings feature is implemented in order to provide a convenient way for users to communicate with the tool. The user can disable or enable contract warnings with predefined symbols embedded in the comment part of the code.

The tool will ignore all requested warning after encountering these predefined symbols until the end of the current file. The predefined symbols are:

- **CCHECKER_NO_CONTRACT**
  
  By embedding this micro, the tool will ignore all the contract warnings from this symbol till the end of the file.

- **CCHECKER_NO_PRECOND**
  
  By embedding this micro, all precondition warnings from this symbol till the end of current file will be disabled.

- **CCHECKER_NO_ASSERT**
  
  By embedding this micro, all assert warnings from this symbol till the end of the current file will be disabled.

- **CCHECKER_NO_POSTCOND**
  
  By embedding this micro, all post condition warnings from this symbol till the end of the current file will be disabled.
5 Some shots from the program output

It worth here to include some output shots of the program when it is given a sample software unit containing two functions as follow.

![Code Snippet]

Figure 26: A sample input file  test.c to illustrate different functionality of the tool

5.1 First requirement

If we run the tool to fulfill toward fulfilling the first requirement on the example code above, the result will be like this:
Specifying the filename
Opening file test.c
in file test.c

2 DBC_PRECOND missing

Total protection is 75.000000

Figure 27. Results of the tool using the first requirement in non verbose mode.

It shows that in file containing the above functions which is named test.c, there are two preconditions missing totally. The resulting protection percentage of the whole file is 75%.

But if it is needed to know exactly where those missing contracts located we have to specify the more verbose mode in the input options of the tool. The result looks like this:

Specify the more informative verbos mode
Specifying the filename
Opening file test.c

********************************************************************************

file test.c notifications********************************************************************************

function foo1 notifications

------------------------
The argument d in line# 14 is not protected by precondition

function main notifications

------------------------
The argument b in line# 23 is not protected by precondition

Total protection is 75.000000

Figure 28: Results of the tool using the first requirement in more verbose mode
If it is required to know each function protected by how much percentage then we have to use the option –x which will show each function with its protection percentage. The result will look like this:

```
Specifying the metric measurement
Specifying the filename
Opening file test.c

In file test.c

    2 DBC_PRECOND missing

<table>
<thead>
<tr>
<th>function</th>
<th>protection percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo3</td>
<td>100.000000</td>
</tr>
<tr>
<td>foo2</td>
<td>100.000000</td>
</tr>
<tr>
<td>foo1</td>
<td>50.000000</td>
</tr>
<tr>
<td>main</td>
<td>50.000000</td>
</tr>
</tbody>
</table>

Total protection is 75.000000
```

Figure 29: Results of the tool using the protection statistic option -x
In the second requirement, it is required to locate the places where there are extra protections which can be omitted. In the above example, it is expected for the tool to notify the extra protection on variable ‘e’ in function foo1 since it is already protected with post condition in function foo2. The same applies for the protection on variable ‘c’ in the main function which is already protected by the post condition in function foo1. If we remove the extra protection if foo1, the protection on variable ‘c’ in the main function is still redundant since the final value still comes from function foo2 which is already protected.
There is also another extra protection in static function foo3, since the input argument of this static function is already protected by main() function, there is not any need to put precondition in foo3.

The second requirement can be used with option “-e” in the command line. The output looks like this:

<table>
<thead>
<tr>
<th>Specifying the extra DBC check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifying the filename</td>
</tr>
<tr>
<td>Opening file test.c</td>
</tr>
</tbody>
</table>

******************** file test.c notifications**************************

Function main notifications

The assert check on c in function main in file test.c line# 23 is duplicated because of post in foo1 in file test.c line# 14

Function foo1 notifications

The post condition on e line# 14 is duplicated because of post in foo2 in file test.c line# 5

Function foo3 notifications

The precondition in this static function, line# 7 is duplicated because of precondition in function main in file test.c line # 24 on called argument number 1 which is ‘a’

Figure 31: Output of the tool using the second requirement.

5.3 Third requirement

In order to detect least protection required areas, it is needed to recognize places in the input software unit where protection must remain even after the development process is finished. For fulfilling this requirement, it is needed to detect places where input values come from a source out of the current software unit. By determining the main function as the root function in the above example, the input argument of main should be consider as one of the input values to this software unit. The other output value comes from the function square() which should remain within protection as well. The output message of tool is:
specifying the least required protection areas
specifying the filename
opening file test.c

*************** file test.c notifications***************

Function foo2 notifications

------------------------
The variable g  line# 8 must remain within protection

Function main notifications

------------------------
The root function argument a line 23 must remain within protection
The root function argument b line 23 must remain within protection

*Figure 32: Showing the least protection required area by the tool to fulfill the third requirement.*
5.4 Function call tree

There is another feature in the tool which helps programmer to see the entire function call tree and their protection status. If the used variable is protected with contracts, the tool indicates this by one otherwise by zero. Running the tool on previous example gives an output like this:

Consider the function main as an example. It has two agreements, ‘a’ and ‘b’ which the only protected argument is ‘a’ so 01 stands for that. It calls “printf” function with only protected variable ‘a’, so the related number is 001 and the same applies for foo1 with two arguments which results in number 01 and also for foo2 which results the value 1.
6 Solution analysis

The tool has successfully met all of its predefined requirements. The only problem was regarding developing my own grammatical rule for the Yacc due to time shortage. If it was solved successfully, then it had been possible to perform much further statistical analysis on the code. In fact many of the advanced analysis are hard or even impossible to implement without using Yacc.

The execution time of the program falls in the reasonable bounders. Different functionality of the tool had been tested intensively on modules with more than 300 files each of which had at least 100 lines of code.

7 Critical analysis

The work has been done with several advantage and disadvantages. One of the advantages was execution time of the code. The other one is the ability to parse the whole function call graph from any arbitrary node due to algorithm implementation. The next advantage is the use of config file to give the ability to the user for defining as much information as needed for the tool to work correctly. The last but not least is user friendly user interface which ease the process of human computer interaction.

Different verbose mode in the program gives the ability to the user to detect defects and warning in a short time by just reviewing the non verbose mode information. The user can go to the details in more verbose mode after some parts of the messages seems interesting.

In tandem with advantages, several disadvantages can be recognized. [1] First of all, using “memwatch” (program for checking memory allocation and memory leak), it can be detected that the size of memory allocation, which cannot be free, grow exponentially with the size of input module. The reason is the necessity to keep required information of the whole code in order to do further analysis.

One of the existing solutions for this problem is writing the parsed information inside a file once in a while and free its address space until the whole module has been parsed, then in analysis part the parser can pick up the information it needs each time to perform required analysis. This solution has not been implemented due to time shortage.

8 Comparative study

As mentioned in the beginning of the project, there is not any similar tool developed for the same purpose in the market. The only similar tool, that we are aware of, is the simple prototype which has been done by the supervisor of this thesis in Ericsson Company.
9 Literature search and techniques

Most of the used literature for defining design rules are from the Ericsson internal design rule papers which unfortunately all of them are confidential and cannot be discuss very openly. There are other source and references which are mostly programming references.

10 Delimitations

Several issues are not going to be addressed in this thesis

- Producing an automatic generated code by the tool to test against the possible source of error
- The non-static checking of the source code

11 References

[5]. John Levine ,lex & yacc ,O'Reilly, October, 1992, page xx (lex & yacc)
[6]. John Levine ,lex & yacc ,O'Reilly, October, 1992,page 60

12 Bibliography


13 Acknowledgment

I would like to thank Mr. Kenneth Andersson who helped me to produce this document and I would like to appreciate his great advises and also to my kind and lovely reviewer Ivan Christoff