MSc Distributed Computer Systems Engineering

Department of Electronic & Computer Engineering

Brunel University

Coverage Analysis of Existing White Box Tests Sets

Markus Schuler

September 2000

A Dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science

MSc Distributed Computer Systems Engineering

Department of Electronic & Computer Engineering

Brunel University

Coverage Analysis of Existing White Box Tests Sets

Markus Schuler

Dr. I. D. Dear

September 2000

A Dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science

Abstract

In the dissertation a test coverage tool is analysed and developed. For that a certain background knowledge is necessary. Therefore several phases of a compiler are considered and how a parser can be created with help of the Unix tools lex and yacc at first. After that the different test methods and the structural testing are presented. Furthermore the requirements for such a tool are analysed and a design is created. Finally a description follows how the developed test coverage tool was tested. The last chapter summary and conclusion is a short survey of the dissertation and gives advice for improving such a test tool.

Contents

1 Foreword

When I was responsible for the evaluation of a test tool for 6 months I touched the test tools as user only. However my interest grew to see how such a tool works. In the master thesis I saw the best possibility to examine how the tool works and to create a prototype.

I want to say thank you to Mr Dr. I. D. Dear that I could interest him for my master thesis and that he looked after me as supervisor.

Finally I want to thank my wife, who supported me actively.

Stuttgart, 30th September 2000

2 Introduction

Software quality is determined by several factors. Important among them are correctness, reliability and usability. To fulfil the property correctness the software must be investigated for the existence of faults. At this a key factor is efficient software testing. In a typical software project about 50 percent of the total time and more than 50 percent of the total costs are needed to test the program or system [MYERS, 1991].

As time and money has to be saved by the companies there must be a way how to reduce the stated figures. One possibility which makes software testing more effective is to develop a tool proving which code is executed. By means of the results it can be decided if the existing test cases are sufficient or if further test cases have to be generated to achieve a higher coverage [NTAFOS, 1988].

The aim of this Master Dissertation is to create such a tool which in the following is called code coverage tool. This tool has to be used during different phases of the software development with different functionality. Hence it is split into the following two tools.

The first tool should read the source code, written in the programming language C, and should carry out static analysis. After analysing the tool should give a report on the results. To determine the coverage of the test cases the tool should instrument the source code in order to store information of the tests executed in a database. This database contains information about functions, blocks and paths of a program. While executing the test cases the information which functions, blocks or paths are executed is inserted into the database by the instrumented statements. In the following this tool is called Instrumentation-Tool.

With another tool the database should be queried and coverage analysis should be executed. Looking at the results the coverage analysis the developer can determine how much and which source code was executed by the test cases. Based on these figures he can decide whether additional test cases are necessary to execute further parts of the source code in order to achieve a higher coverage. In the following this tool is called Evaluation-Tool.

The Instrumentation-Tool and the Evaluation-Tool together constitute the envisaged code coverage tool. The phases of the code coverage tool and their outcomes are shown in Figure 2-1.

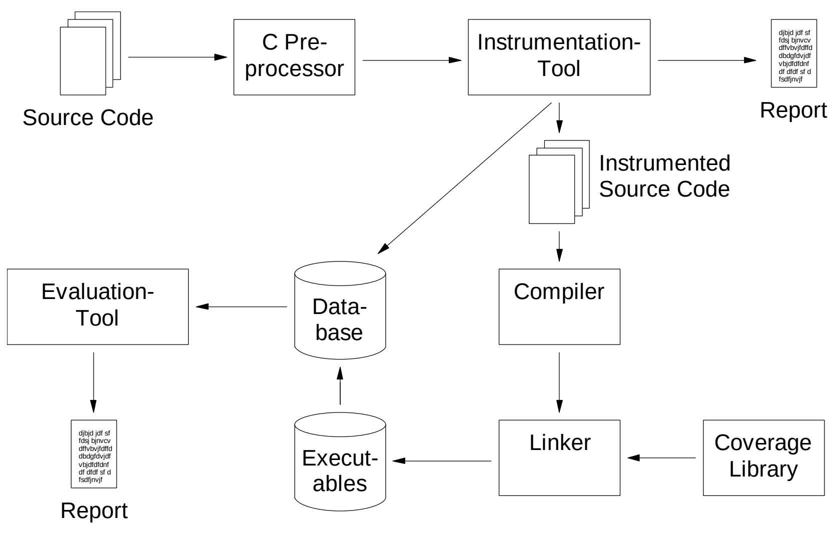

Figure 2-1 Phases of the code coverage tool

With code coverage tools it has always to be noticed that the source code is changed and therefore the results of compiling the instrumented code is different from compiling the original source code. This effects on one hand the runtime behaviour and on the other hand the possibility how compilers can optimise the code.

3 Background

The project is based on two theoretical areas. The first one deals with the theory of building a compiler and the second one with the methods of testing. These two areas are investigated in more detail in the following.

Additionally the Unix-Tools lex and yacc, which are used for the realisation of a parser, are dealt with.

3.1 Principles of Compiler Design

In this project the functionality of a complete compiler is not required, but several parts of it.

A compiler has the function to read a program written in the source language and to translate it into an equivalent program of the target language. In doing so the function of the compiler is divided into 6 phases [AHO, 1988].

The first three phase analyses the source program according to lexical, syntactical and semantical properties of the source language.

Then the intermediate code generation follows. Up to now the compiler phases are independent from the kind of target language used. In the following 5th phase the intermediate code is optimised before code is generated in the target language in the 6th phase.

During the 6th phases the compiler generates and administrates a symbol table and has in each phase the function of error handling. All the phases of a compiler are shown in Figure 3-1 below.

In the following chapters the function of each phase is dealt more deeply.

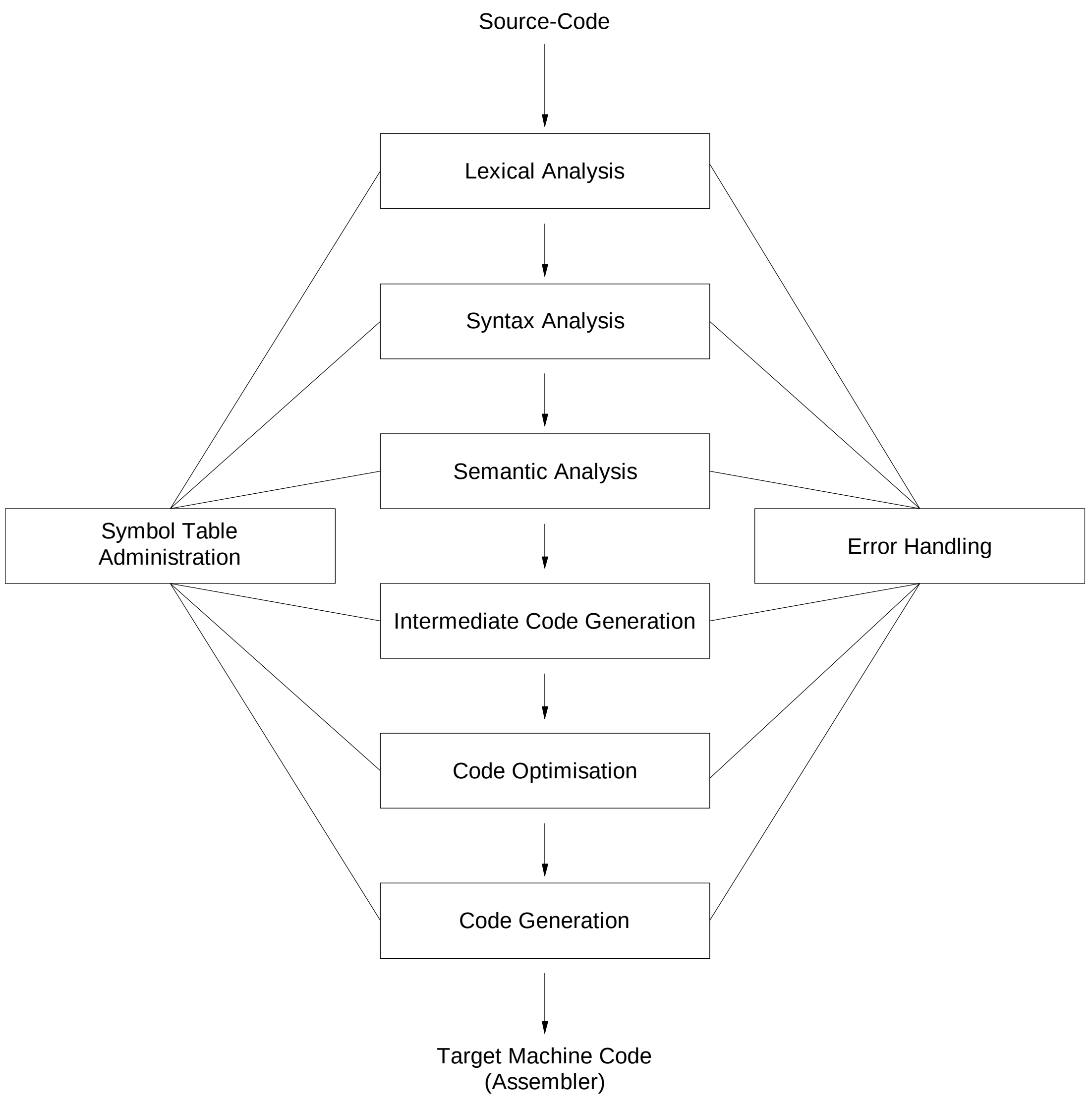

Figure 3-1 Typical phases of a compiler

3.1.1 Lexical Analysis

The lexical analysis is executed with the help of a scanner.

At the same time the source code is decomposed into tokens or symbols by the scanner. Tokens can be for example `identifier` for alphanumeric characters or `string-literal` for strings, which are written in quotation marks, or reserved words like `if` or `while`. The recognised tokens are passed to the parser which executes the syntax analysis.

For the implementation of a scanner the Unix-Tool lex can be used. The lexical rules can be listed in a kind of table, where in the left column regular expressions or patterns are written and in the right one actions can be specified, which are to be executed when the corresponding pattern is recognised in the input stream.

This tool will be used in this project to build a scanner which is able to identify the tokens of the programming language C in input source codes. It is described in chapter 3.2.1.

3.1.2 Syntax Analysis

Syntax analysis is executed with help of a parser. The parser gets the tokens identified by the scanner and has to check whether the sequence of these tokens forms a correct sentence of language. The rules of a language describing how sentences can be built constitute the grammar of that language. For the usual programming language the form of a so called context-free grammar is used and the rules are described in a notation called Backus-Naur-Form (BNF). A context free grammar consists of 4 components:

- a set of terminal symbols. Terminal symbols are also called tokens.
- a set of non terminal symbols.
- a set of productions. Each production is separated in a left and in a right part. The left part is a non terminal symbol described by the right part. The right part consists of a sequence of terminal and non terminal symbols.
- a start symbol, which is sometimes also called the root symbol of the grammar. The start symbol must be a non terminal symbol.

For the generation of a parser the Unix-Tool yacc can be used. This tool takes the grammar in a kind of BNF. With each symbol of the right hand side of a production an action can be associated, which shall be executed when the symbol appears.

This tool will be used in the project to generate a parser checking the input source code with respect to the grammar of the programming language C. Furthermore the actions are used to find positions where the code has to be instrumented.

The parser generated by yacc is intended to be used together with a scanner generated by lex. yacc is described in chapter 3.2.2

3.1.3 Semantic Analysis

In the semantic analysis the source code is analysed in order to find semantic errors and type information is collected for the following phases. This analysis is called static checking, too.

During this phase type checking, control path checks, verification of unambiguousness and name verification is carried out.

This phase is interesting for the envisaged tool, as at this point a more intensified checking can be carried out than a normal compiler does. Not only the type of a variable is important but also its usage, e.g. whether a variable is set before reading it or whether a variable is set and afterwards it is never used.

3.1.4 Intermediate Code Generation

Some compilers generate an explicit intermediate version of the source code after the analysis. This intermediate version is seen as a program of an abstract machine and is machine independent.

The advantage of an intermediate language is that a compiler can be developed easier for different machines and a machine independent optimiser can be implemented based on the intermediate language.

This phase is not important for the project as no code transformation takes place.

3.1.5 Code Optimisation

In the phase of code optimisation the intermediate code shall be improved according to several performance criteria, e.g. usage of processor and memory. As a basis for this the occupancy of registers and the usage of the variables is analysed more intensively.

Some of the techniques used usually during code optimisation, e.g. those concerning the analysis of the read and write accesses of variables, will be employed in this project during an advanced static analysis phase, see above.

3.1.6 Code Generation

In the final phase a compiler takes the optimised intermediate code and builds up a program for the target machine.

The envisaged tool itself does not generate executable code. Rather the read and analysed source code is instrumented and emitted in the same language and saved in a temporary file. This file is used as input for the proper compiler.

3.2 Compiler Generation Tools

In the following two chapters the Unix-Tools lex and yacc are described. As [HEROLD, 1999] described these tools the chapter of lex and yacc refer to this book. These tools are used to generate scanner and parser for lexical and syntactical analysis.

3.2.1 lex

lex was developed in the seventies as a tool for the lexical analysis. A scanner generated by lex can be connected with a parser generated by yacc, which is described later. The corresponding GNU tool is called flex.

lex is a kind of compiler, which transfers a lex program into a C program. For this the lex program has to be written in a specific language. lex programs are tables of regular expressions and accompanying parts of C programs.

The table is translated by lex into a C program which reads an input text from standard input. The text is decomposed into strings matching the regular expressions of the lex program. Each time a string is found the corresponding C program part is executed which belongs to the matching regular expression.

A lex program can be divided into three parts. The first part is the definition part in which C variables, C constants and regular definitions are contained. The C code has to be written between the characters `%{` and `%}`. Then a part follows consisting of lex rules which are stated in a kind of table. Thereby every rule consists of a regular expression and an action, provided as statements C. The third part consists of definitions which are needed by the actions. These are implemented in the language C. In a lex program the first and the third part are optional. As all strings, which are covered by no rules, are printed unchanged to `stdout`, the rules part is optional, too.

In the following the structure of a lex program is shown.

```
/* definitions */

%%
/* lex rules */

%%

/* user specific definitions */
```

lex is shown by the characters %% that the following instructions refer to the lex rules. The second %% indicates the end of the rules part respectively the beginning of the user specific definitions. If these are omitted, the characters %% can be left out, too.

In the following a few lex rules are printed and explained.

In the definition part names can be given to regular expressions. This is done by stating a name followed by the regular expression. The name can then be used in the rules part, as a shorthand for the regular expression.

```
DIGIT        [0-9]
LETTER       [a-zA-Z_]
```

In the first line the name DIGIT is associated with the set [0-9]. So the digits 0 to 9 are matched by the name DIGIT. In the second line all letters and the underscore are matched by the set [a-zA-Z_] named LETTER. If a name is used in a regular expression, the name has to be written between braces, e.g. {LETTER}.

Every lex rule which is stated in the rules part is composed of a pattern (regular expression) and an accompanying action. This action is executed as soon as the input string matches to this pattern. A pattern is described by a regular expression. It has to start always in the first column of a line and has to be separated from the action by one or several blank characters or by tabulators.

A regular expression is an expression, which describes which strings are matched. In the simplest form the pattern can be a string, more complicated patterns are described by rules. In rules meta characters have a special meaning. They are used to build the rules. Meta characters are `\ ^ $ . [ ] | ( ) * + ? { } " / % < >`.

To use meta characters as a normal character in a pattern, a backslash has to be put before the meta character, for example `\*`. A backslash is put before the escape characters of C, too. Escape sequences are shown in Table 3-1.

escape sequence	description
\b	backspace character
\n	new line character
\r	return character
\t	tabulator character
\\	backslash character
\’	prime character
\”	quotation marks character
\.	dot character

Table 3-1 Escape sequences and their description

The meta character dot ´`.`´ plays a special role. It stands for any single character except of the new line character.

As shown before in an example sets can be built, writing the characters in square brackets. One of the character `A`, `B` or `C` can be matched by the regular expression `[ABC]`. Whole scopes can be indicated, too. So the regular expression `[a-z]` matches all small letters. If the meta character ´`^`´ is put before the elements of a set, the elements are matched which are not contained in the set, e.g. `[^0-9]` matches all characters besides digits.

If the meta character ´`^`´ is indicated before a character outside of a set, so the line beginning is meant. The end of a line is marked by the meta character ´`$`´.

By the meta character ´|´ an alternation is expressed, e.g. A|B matches one of the characters A or B. By the pattern AB the string A followed subsequently by B is matched. If after a regular expression the meta character ´*´ is put, this means that the pattern given by the regular expression is repeated zero, one or several times. E.g. AB*C matches the strings AC, ABC, or ABBC and so on. Replacing the meta character ´*´ by the meta character ´+´ means that the regular expression has to be repeated at least once. E.g. AB+C matches ABC, ABBC and so on. If the pattern is optional the meta character ´?´ has to be used, e.g. AB?C matches AC or ABC.

As within arithmetic expression, parentheses can be used to group regular expressions and to change priority among the operators.

If different sub strings of an input string could be matched by several patterns, the pattern is used which matches the longest sub string. If exactly the same input string is matched by different patterns, the pattern which appears first in the table of rules is chosen.

If several patterns should cause the same action, the action need not be written many times, but the patterns can be combined using the meta character '|', see above. If a pattern should not cause an action an empty action has to be indicated that means a semicolon or an empty block ´{}´.

lex disposes variables and functions which can be used by the programmer within the actions. The variables or functions disposed by lex have the prefix yy. With the variable yytext the string can be accessed which is matched by the pattern. The variable is a pointer to character (char *). The length of the string, which was matched by the pattern, is saved in the integer variable yyleng. The string pointed at by yytext is given out to the stream yyout when calling the macro ECHO. This stream is by default stdout. If the option %option yylineno is set, the number of the currently read line can be queried by help of the variable yylineno.

If in an action should read single characters out of the input stream the function `input` can be called. The `input` function returns the read character. This is necessary for example to read a comment in a C source file. The regular expression “`/*`“ indicates the beginning of a comment. After that all characters are read with the function `input` until the end of the comment (‘`*/`’) is reached. If characters have to be written back to the input stream the function `unput` can be used. As parameter the character to be written has to be indicated.

To start the scan procedure `yylex` has to be called. In this function the character of the input stream are read by `input`. If the end of file character `EOF` is read by the function `input`, the function `yywrap` is called. This function has to be implemented or has to be switched off by the option `%option noyywrap`. In the function `yywrap` the input stream can be switched to another file so that this one can be scanned. In this case the function `yywrap` has to return a value unequal `0`. If the value `0` is returned the scanning will be finished and `yylex` returns with `0`.

In the following a short lex program is presented which determines the quantities of the characters and lines in an input stream.

```
%{
int numLines = 0;
int numCharacters = 0;
%}
%option noyywrap
%%
\n   { numLines++; numCharacters++;
.    { numCharacters++; }
%%
int main(){
    yylex();
    printf( "# of lines = %d, # of chars = %d\n",
             numLines, numCharacters );
}
```

In the definition part the integer variables numLines and numCharacters are defined and initialised by 0. After that the option noyywrap is set. In the rules part new line characters are recognised by the first rule and as actions both variables are incremented by one in the corresponding action. All "remaining" characters are matched by the second rule, using the meta character '.', and as action the variable numCharacters is incremented by one. In the main function the scanner is started by calling yylex. After the yylex function returned, the number of lines and characters are printed to stdout.

In the following example the scanner should recognise names and numbers of an input stream and output them to stdout.

```
%option noyywrap
LETTER  [a-zA-Z]
DIGIT   [0-9]
%%
{LETTER}+           { printf( "name = %s\n", yytext); }
{DIGIT}+            { printf( "integer = %d\n", atoi( yytext)); }
{DIGIT}+\.{DIGIT}*  { printf( "float = %f\n", atof( yytext)); }
[ \n\t]             {}
.                   { printf( "bad characters %s\n", yytext); }
%%
int main( int argc, char **argv){
    yyin = fopen( argv[1], "r");
    yylex();
}
```

In the definition part the option noyywrap is set. Afterwards two regular expressions are defined. The regular expression LETTER matches the capital and small letters from A to Z, the regular expression DIGIT matches the digits from 0 to 9. The first rule in the rules part matches one or several characters of the set LETTER and outputs them as a name. The second rule matches one or several characters of the set DIGIT and outputs the corresponding integer number.

The third rule recognises simple float numbers. The float number has to consist of minimum one or several predecimal digits and a dot followed by none, one or several postdecimal digits. The recognised float number is printed out as float. In the fourth rule the blank character, the new line character and the tabulator character are matched, but no action is to be executed. At the end all remaining characters not covered by the rules before are printed out to `stdout` as bad characters. In the `main` function the file is opened for which the file name was handed over as argument to the program. After that `yylex` is called to scan the file.

The usage of lex in combination with yacc, is described in the following chapter on yacc.

3.2.2 yacc

yacc was developed as a tool for the syntax analysis in the seventies. The name yacc means yet another compiler compiler and indicates ironically the popularity of parser generators of the seventies. The corresponding GNU tool is called bison.

yacc is a tool which takes as input a description of the syntax of the text to be analysed. The description is a derivation of the Backus-Naur-Form (BNF). yacc takes this description and produces a C program which analyses and processes the input data according to the given structure. Additionally by writing C code it can be specified what has to be done when structure elements in the input text are recognised.

The structure of a yacc program is divided into three parts, in a similar way as a lex program. The first part deals with the definition part in which global valid C definitions or yacc specific statements are indicated. The C definitions have to be written between the characters `%{` and `%}`. The yacc specific statements include `%token` with which the names of tokens are defined and `%start`, with which the start symbol is indicated.

The second part is the rules part. In this part all rules are written which the generated parser should cover. The rules are indicated in a way which is similar to the well known Backus-Naur-Form. On the left side of such rule the non terminal symbol is written, which is described by the right hand side. Both sides are separated by a colon. The rule is completed by a semicolon. The right hand side can be empty or can consist of a sequence of terminal, non terminal symbols and actions. An action can consist of several C statements and has to be written between braces (‘{}’). Each non terminal symbol has to appear at least once on the left side. In order to inform the parser with which non terminal symbol he should start, the start symbol can be indicated with %start. If this is not done the first rule is used as start point. If a non terminal symbol is described by different right hand sides, the rules can be summarised. The right hand sides have to be separated by the character ‘|’.

On the right hand side an action can appear before or after each symbol. In the action code values can be accessed which are delivered by the symbols. The value of the first symbol can be accessed by the variable $1, and the value of the second one can be accessed by the variable $2 and so on. The value that has to be delivered by a symbol has to be assigned to the specific yacc variable $$.

The variables are of type int. However, the types of the variables can be defined by the user. For this the instruction %union {...} has to be used. Within the braces the data types and the accompanying variable names have to be indicated, with which the type respectively the valid union field is selected later. In the following definition of the union a field of the type float and the accompanying name real are defined.

```
%union {
    float real;
}
```

The type of the return value of a terminal symbol can be defined by the %token instruction. The type has to be declared after the %token instruction in angle brackets (‘<>’) and before the terminal symbols. The type has to be one of the fields' name in the union. If a terminal symbol should deliver a floating point value, the `%token` instruction has to be written like

```
%token <real> FLOAT_CONST.
```

For a non terminal symbol the type of the return value can be defined, too. This happens with help of the `%type` instruction. The type has to be declared after the `%type` instruction in angle brackets ('<>') and before the non terminal symbols. The type has to be one of the fields' name in the union. If the non terminal symbol `result` should deliver a floating point value, the `%type` instruction has to be written like

```
%type <real> result.
```

If there is no type declaration for a non terminal symbol, the type of the return value is `int`.

To start the parser the yacc function `yyparse` has to be called. This function returns the value 0, when parsing was successful. If an error occurs the value 1 is returned. The function returns when the end of the file is reached which have to be parsed or when an error occurred. In an action a return can be forced by calling one of the macros `YYACCEPT` or `YYABORT`. With the macro `YYACCEPT` `yyparse` returns the value 0, respectively with `YYABORT` the value 1.

In the function `yyparse` the function `yylex` is called. This function is not defined by yacc but has to be disposed by the programmer. The `yylex` function recognises tokens from the input stream and returns them to `yyparse`. This function can be implemented in C and can analyse the input stream. Another possibility is that a lex program can be written from which the `yylex` function is called. At this, as described in the previous chapter, the lex program recognises the tokens of the input streams and returns them. The following Figure 3-1 shows the communication between `yylex` and `yyparse`.

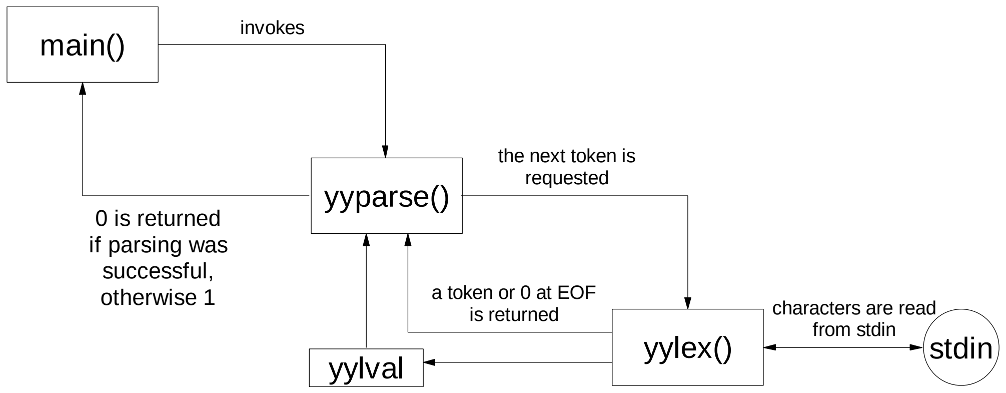
main()
invokes
yyparse()
the next token is requested
0 is returned if parsing was successful, otherwise 1
a token or 0 at EOF is returned
yylex()
characters are read from stdin
stdin
yylval

Figure 3-1 Communication between yylex and yyparse

A further interface between lex and yacc is the global communication variable `yylval`. The default type of the variable is `int`. A value can be assigned to the variable `yylval`, as a further description of a token. E.g. if an integer constant is recognised in the input stream, the constant value can be assigned to `yylval` and `yylex` returns the corresponding token for integer constant. If different types of such values have to be returned to yacc, the types have to be defined in the yacc union statement as described above. In this case the variable `yylval` gets the type of this union. E.g. if the yacc union is defined as above, the float value can be assigned to the field `real` of the variable `yylval` when a float constant is recognised. The values which are assigned to `yylval` can be accessed by the `$` variables of corresponding terminal symbols.

The third part consists of the user defined definitions which are needed by the actions. These are implemented in the language C. Additionally the function `yyerror` has to be implemented which is called by `yyparse` if a syntax error occurs. Here information of the kind of error and the place of the error can be reported.

yacc generates a parser, which consists of a finite state machine with a stack, which is called parser stack. The state machine can always read a token, the lookahead token, in advance. Depending on the current state, which stands on the upmost place of the stack, and on the lookahead token the new state of the state machine is determined and a transition to it is made. At the beginning the parser starts always in the state 0.

Four operations are at the disposal of the state machine. Depending on the current state the state machine decides if a lookahead token is necessary, in order to decide which operation can be carried out. If the lookahead token is necessary `yylex` is called to get the new token and it is filed in a new state on the stack. This operation is called shift. If the right hand side of a rule can be matched totally by the states on the stack, these states are removed from the stack as these have been investigated and are not relevant any more. This operation is called reduce. If the total input stream has been investigated in this way using the given rules and the read lookahead token is the end of file (`EOF`), the operation accept is executed and the parse procedure finishes. The fourth operation is the error operation. This one is executed if one of the rules is broken. At this the parser reports an error and finishes the syntax analysis.

When compiling a yacc program conflicts in the grammar can be recognised by the yacc compiler. There are shift/reduce and reduce/reduce conflicts. With shift/reduce conflicts there is the possibility to shift in the current state the next token to the stack or to reduce a rule. With reduce/reduce conflicts several rules can be reduced at the same time. If these conflicts are not eliminated from the grammar, the generated parser behaves in a shift/reduce conflict the shift operation is used and in a reduce/reduce conflict the first rule of the grammar, which is suitable, is reduced.

In the following a yacc program is described for a calculator with the four fundamental arithmetic operations and the co-operating lex program shall be described, too.

First of all the lex program is shown which recognise integer numbers and floating point numbers and the fundamental arithmetic operators.

```
D    [0-9]
E    [Ee][+-]?{D}+

%%
{D}+                 { yylval.integer = atoi( yytext);
                       return INT_CONSTANT;
                     }
{D}+{E}              |
{D}*"."{D}+({E})?    |
{D}+"."{D}*({E})?    { yylval.real = atof( yytext);
                       return FLOAT_CONSTANT;
                     }
"+"                  { return ADD; }
"-"                  { return SUB; }
"*"                  { return MUL; }
"/"                  { return DIV; }
\n                   { return RESULT; }
"q"                  { return QUIT; }
[ \t]                {}
.                    { printf( "bad character\n"); return 0; }
%%
```

In the definition part the sets `D` for digits and `E` for exponents are built. The first rule in the rules part matches all integer numbers and assigns the corresponding value to the yacc variable `yylval.integer`. As token this rule returns `INT_CONSTANT`. The following three rules were summarised and cover the possibilities of floating point numbers allowed in the programming language C. The float value is assigned to the yacc variable `yylval.real` and the token `FLOAT_CONSTANT` is returned. The following four rules match the arithmetic operations and the respective token for the operator is returned. As soon as the new line character is recognised the token `RESULT` is returned. When recognising the character ‘q’ the token `QUIT` is returned. Blanks and tabulators are thrown away, that means they are read from the input stream but nothing is done and no token is returned. By the dot meta character all other characters are matched and `0` is returned as token. In this case, the output of the lex compiler is written into the file `calculation.h`.

The following yacc program defines the grammar of the calculator.

```
%{
#include <stdio.h>

int yylex();
%}

%union {
    int integer;
    float real;
}

%token ADD SUB MUL DIV RESULT QUIT
%token <integer> INT_CONSTANT
%token <real> FLOAT_CONSTANT

%type <real> term
%type <real> factor
%type <real> constant

%start lines

%%
lines    :  QUIT               { printf( "bye\n"); YYACCEPT; }
         |  line lines
         ;
line     :  term RESULT        { printf( "result: %f\n", $1); }
         ;
term     :                     { $$ = 0; }
         |  factor             { $$ = $1; }
         |  term ADD factor    { $$ = $1 + $3; }
         |  term SUB factor    { $$ = $1 - $3; }
         ;
factor   :  constant           { $$ = $1; }
         |  factor MUL constant { $$ = $1 * $3; }
         |  factor DIV constant { $$ = $1 / $3; }
         ;
constant :  INT_CONSTANT       { $$ = (float)$1; }
         |  FLOAT_CONSTANT     { $$ = $1; }
         ;
%%

#include "calculator.h"

void yyerror( char *pString){
    fprintf(stderr, "%s\n", pString);
}

int main( int argc, char **argv){

    return yyparse();
}
```

In the definition part `stdio.h` is included and the function prototype of `yylex` is given. After that the yacc union type with fields `integer` of type `int` and `real` of type `float` is defined. Furthermore all tokens which are used are defined by `%token`. Additionally the token `INT_CONSTANT` delivers a value in the field integer of the variable yylval. The token FLOAT_CONSTANT delivers the value of the field real of the variable yylval. Then the return values of the non terminal symbols are defined by %type. At this all return values are of type real. The non terminal symbol lines is defined by %start as the beginning of the analysis.

In the rules part the grammar of the calculator is defined. At this the non terminal symbol lines can consist of several line or of the token QUIT. The application is finished with the latter token. The non terminal symbol line is composed of the non terminal symbol term and the token RESULT. As action the result is printed to stdout. A term can consist of an empty instruction, a factor or of a further term which is followed by an operator ('+' or '-') and a factor. The return value of the empty instruction is always 0. If a term is simply a factor, the value of the factor is returned. With the other two alternatives the return value of the term is computed by taking the return value of the right hand side term added respectively subtracted by the return value of factor. A factor can consist of a constant or a further factor followed by an operator ('*' or '/') and a constant. If a factor is simply a constant, the value of constant is returned. With the other two alternatives the return value of factor is computed by taking the return value of the right hand side factor multiplied respectively divided by the return value of constant. A constant is either an INT_CONSTANT or a FLOAT_CONSTANT. At this the value which was saved in the variable yylval.integer or yylval.real is read with the variable $1 and is assigned to the variable $$. Therefore the value serves as return value of the respective symbol.

At the end the output file of lex, calculator.h, is included and the functions yyerror and main are defined. In the yyerror function a report is printed if an error occurs. In the main function yyparse is called.

3.3 Principles of Testing

During each phase of the software lifecycle the respective documents should be evaluated [SCHACH,1999]. There are different methods of validation depending on the kind of the documents. This chapter deals with the validation of the source code by testing it and the different methods doing this.

Dijkstra said that testing could not prove the absence of errors but only their presence.

3.3.1 Test Methods

For testing software there are two methods:

- One is black box testing, by which the functionality of the software is tested. The implementation of the software is not considered. Depending on the inputs the outputs are compared with the expected results. [BEIZER, 1990] wrote that this testing takes the user's point of view.
- The other method is white box testing which is also called glass box testing. This means a structural testing of the software by which in contrary to black box testing the implementation details are taken into account. As a side effect the source code is analysed more exactly.

The test cases which are used by one method, can also be used by the other method. Both methods complement each other. In principle it could be tried to find all mistakes in the software by functional testing, but this would last forever. Structural testing can be performed in a finite period of time but not all errors might be found, even if the software is completely executed.

For the dissertation only structural testing is of interest and considered in the following.

3.3.2 Structural Testing

With structural testing static and dynamic testing can be distinguished. Static testing means that only the source code is investigated. With this respect, each compiler is so to speak a test tool, which investigates the source code for certain language defined rules. Additionally, further investigations about the source code can be done. Metrics can be measured, e.g. the relation of comments and statements or by the relation of reading and writing a variable. Furthermore, it can be checked whether a defined variable is used or whether a variable was set before reading.

For dynamic testing test cases have to be implemented, as dynamic testing only takes place at runtime. When dynamic testing is performed the code coverage achieved by the test cases can be investigated. Several levels of coverage are distinguished and explained in the following chapter.

3.3.3 Code Coverage Levels

In this chapter the different levels of code coverage are presented. The levels Function, Statement, Decision, Condition and Path Coverage are described and it is shown how they can be calculated [BEIZER, 1990], [WOODWARD, 1980].

Function Coverage:

Function Coverage means that all called functions and all implemented functions are set into a relation.

$$\text{Function Coverage} = \frac{\text{executed functions}}{\text{existing functions}}$$

However, the conclusion that can be drawn from this figure is very weak and can scarcely be used for the judgement of test cases.

Statement or Line Coverage:

Statement or Line Coverage means that it is checked which statements are executed in one function. A similar coverage is the Block Coverage. Here, statements are summarised to blocks, which have only one entry-point and one exit-point.

The Statement Coverage is defined as follows:

$$\text{Statement Coverage} = \frac{\text{executed statements}}{\text{existing statements}}$$

This figure has more expressive power than the pure Function Coverage. The following function can be taken as an example.

```
void function( int i){
    if (i > 5) {
        statement1;
    } else {
        statement2;
    }
}
```

For any value `i` greater than 5, a Statement Coverage of only 66% is achieved. To achieve a Statement Coverage of 100%, at least one more test case is necessary, which executes the else branch of the `if`-statement, i.e. a value for `i` less or equal than 5.

Decision Coverage:

A 100 % Line Coverage does not ensure that all possible paths were executed. A stronger figure is delivered by Decision Coverage, with which the executed paths are evaluated. The Decision Coverage is calculated as follows:

$$\text{Decision Coverage} = \frac{\text{executed branches}}{\text{existing branches}}$$

The following example shows the weakness of statement coverage.

```
void function( int i){
    int *p = NULL;
    if (i > 5) {
        p = &v;
    }
    *p = 0;
}
```

If in a test case this function is called with a value for `i` greater than 5, then the pointer p is initialised with the address of the variable v. After that the value 0 is written into the variable on which the pointer p points (i.e. the variable v). By this test case a 100% Statement Coverage is achieved. The conclusion may be drawn that for a 100% coverage no more test cases are necessary. However, the error that the pointer p is not initialised, respectively has the value `NULL`, in all cases where the value for `i` is less or equal than 5 is not detected.

If a Decision Coverage is carried out on that function, only a result of 50% is achieved. When a 100% Decision Coverage is achieved using a value for `i` less or equal than 5 then the error at the assignment `*p = 0` is recognised.

Condition Coverage:

With the Condition Coverage all conditions, which are necessary to execute a branch, are considered.

The Condition Coverage is calculated as follows:

$$\text{Condition Coverage} = \frac{\text{executed conditions}}{\text{existing conditions}}$$

Considering the expression `a > 0 && b > 0` in an `if` statement. There are four possibilities to get a result:

A > 0	B > 0	result
false	false	false
false	true	false
true	false	false
true	true	true

Table 3-1 Conditions of the expression

To achieve a 100 % Condition Coverage for this expression, at least four test cases are necessary to meet the conditions.

Path Coverage:

The strongest evaluation is the Path Coverage. A path is defined to be a sequence of statements beginning from the entry point of a function to one of its exit points. With this evaluation it is checked how much paths are executed. To achieve a 100% Path Coverage, a nearly endless number of test cases is necessary, e.g. as every number of possible executions of a loop has to be tested separately. This is the reason why a 100% Path Coverage is not often achieved in practise, especially in the case of nested loops.

The Path Coverage is calculated as follows:

$$\text{Path Coverage} = \frac{\text{executed paths}}{\text{existing paths}}$$

4 Analysis

The analysis and later the design and implementation deals only with the realisation of a prototype and not a complete test tool. In the chapter Summary and Conclusion the additional functionalities for such a test tool are described.

First the requirements of the prototype are considered and after that the use cases are set up and some scenarios are described.

4.1 Requirements

In the following chapters the requirements of the Instrumentation-Tool and of the Evaluation-Tool are shown.

4.1.1 Requirements of the Instrumenta tion-Tool

In the following the requirements of the Instrumentation-Tool are shown.

Requirement 100:

The Instrumentation-Tool should be for source code written in the programming language C using the ANSI-C standard. The source code should be faultless, which means that the source code is executable.

Requirement 200:

If an incorrect code occurs only a simple error handling should be used to indicate the error position by the Instrumentation-Tool.

Requirement 300:

A static analysis of the source code should be executed by the Instrumentation-Tool.

Requirement 400:

The source code should be instrumented by the Instrumentation-Tool for Function Coverage, Block Coverage and Decision Coverage analysis.

Requirement 500:

The Instrumentation-Tool should create a database, which is filled by the instrumented code when executing test sets and which should be evaluated by the Evaluation-Tool.

Requirement 600:

The instrumented code should be stored in a temporary file by the Instrumentation-Tool.

Requirement 700:

To compile the instrumented code the standard compiler should be called by the Instrumentation-Tool.

4.1.2 Requirements of the Evaluation-Tool

In the following the requirements of the Evaluation-Tool are shown.

Requirement 100:

The Evaluation-Tool should provide a list of all instrumented functions.

Requirement 200:

The Function Coverage of the test sets should be requested by the Evaluation-Tool.

Requirement 300:

The Block Coverage of each function should be requested by the Evaluation-Tool.

Requirement 400:

The Decision Coverage of each function should be requested by the Evaluation-Tool.

4.2 Use Cases

Based on the requirements the following use cases can be set up.

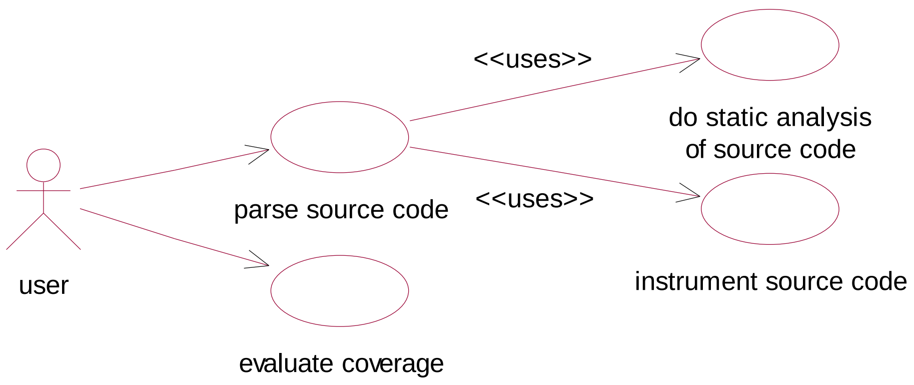

Figure 4-1 Use cases of the Code Coverage Tool

The identified use cases are described in the following.

Use Case "parse source code"

The source code file is opened and is processed by the parser. At the same time the parser builds a tree. Parsing is executed with help of a lex and yacc program. While parsing the use case "do static analysis of source code" is used to do the static analysis. After parsing the use case "instrument source code" is used to instrument the source code.

If errors occur while parsing the application is finished and the position of the error is printed.

Use Case "do static analysis of source code"

If a variable declaration of a primitive data type is recognised while parsing, information on this variable is pushed on the variable stack. When a write access to a variable occurs it is checked whether the variable has been declared and if it is in the scope. In this case in the corresponding variable information the write flag is set.

When a read access to a variable occurs, it is checked whether the variable has been declared and if it is in the scope. In this case in the corresponding variable information the read flag is set. Additionally it is checked, if the write flag in the variable information is set. If it is not set an error is reported that there was a read access to an uninitialised variable.

If the scope of a variable is left the variable information is removed again from the stack. At this time it is checked if the variable was used for reading as well as for writing. If the variable was not used a warning is reported, that the variable was declared but not further used.

Use Case "instrument source code"

In general the tree which was built while parsing is run through and the parsed statements are written unchanged in a temporary file. However:

If a function definition is recognised the code is instrumented for Function Coverage after the declaration part and is written in the temporary file.

If a block begin is recognised, the source code is instrumented for the Block Coverage after the declaration part and is written in the temporary file.

If a select statement, an iteration statement or a labelled statement is recognised the code is instrumented for the Decision Coverage and written in the temporary file.

For each instrumented unit the file name, the function name, the kind of coverage and a running number is written in an Information Database to locate and arrange the coverage point. Additionally in a second database, the Coverage Database, an execute flag is written for each coverage point.

Use Case "evaluate coverage"

With help of the Information Database and the Coverage Database the respective kind of coverage for all functions or for the indicated functions only are determined. This depends on the parameters which the user indicates when starting.

4.3 Scenarios

To make the working of the tool clearer some scenarios are shown and described in the next chapters.

4.3.1 Scenario Instrument

The scenario in Figure 4-1 shows the procedure of the Instrumentation-Tool.

The user starts the application and hands over all necessary compiler flags and source code file names to the application. After that each file name is forwarded to the `Preprocessor` object. For each file the preprocessor of the GNU compiler (gcc) with the option '-E', only for preprocessing, is called by the `Preprocessor` object. The preprocessor is shown as actor in the scenario. It executes the preprocessing and saves the source code in a temporary file. When preprocessing all preprocessor statements are released. Afterwards the parser with the name of the temporary file is called to execute the static analysis and the instrumentation. The instrumented source code itself again is saved in a temporary file. This file is handed over to the `Compiler` object. The `Compiler` object calls the GNU compiler (gcc) with the indicated compiler flags given by the user. The GNU compiler is shown as actor in the scenario. The compiler compiles the instrumented source file and saves the result as object file.

If the Instrumentation-Tool is called with the compiler flag '-c', exactly this scenario is run through. It includes the compiler call, but finishes afterwards. For linking a new call of the Instrumentation-Tool is necessary indicating the object files as arguments.

When linking the object files are handed over to the `Linker` object. This object calls the GNU linker (gcc) then, to link the files to an executable one. The linker is shown as actor in the scenario.

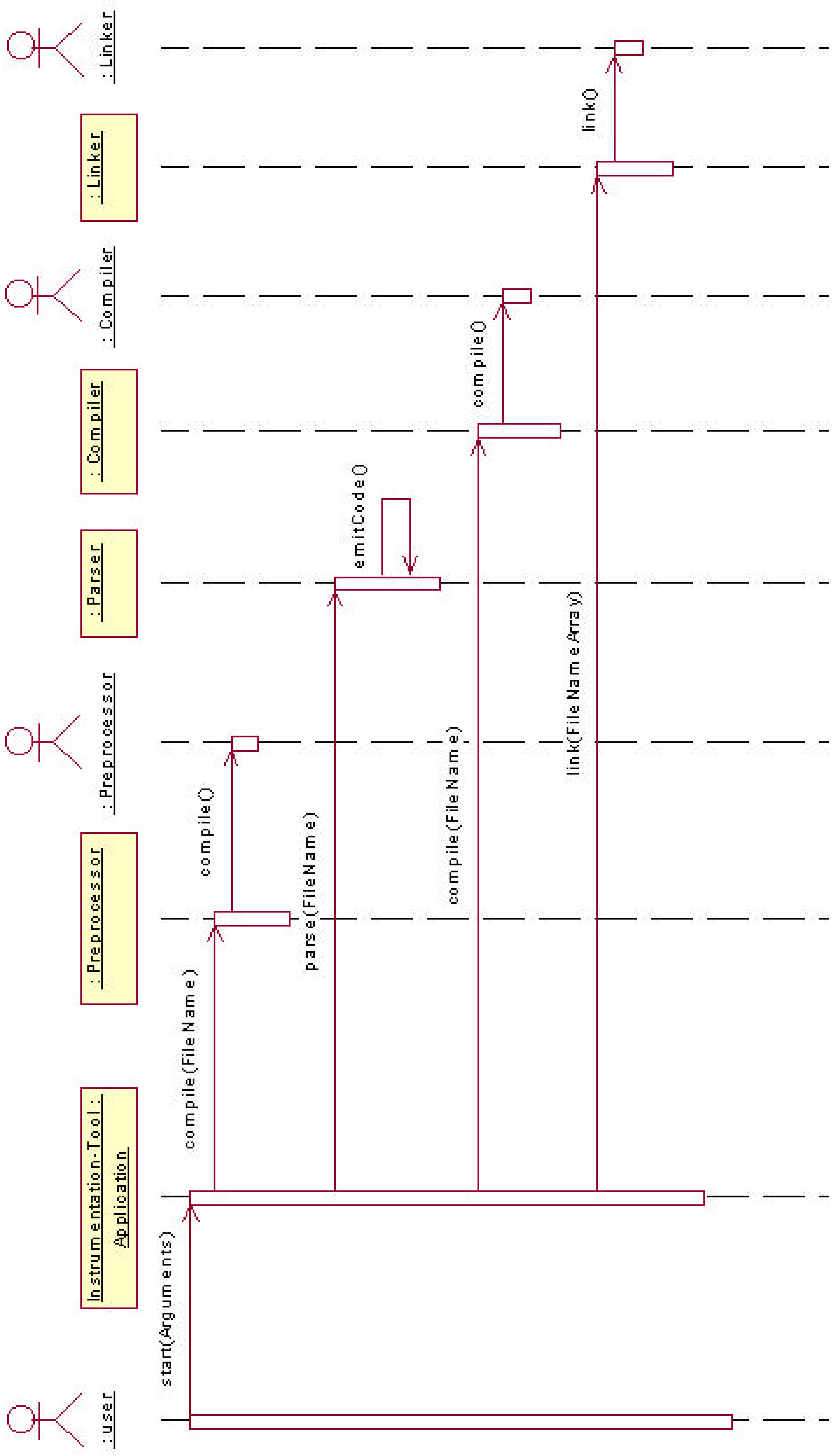

Figure 4-1 Scenario of the Instrumentation-Tool

4.3.2 Scenario Variable Access

The scenario in Figure 4-1 shows in detail the static analysis of the variable access. In the diagram no alternative procedures were shown as these would overload the diagram. The alternative procedures are only described.

The scenario shows the parser during parsing. The parser recognises a beginning of a block and pushes a scope to the `VariableStack`. If the parser recognises a variable declaration it creates for the variable declaration a new `VariableInformation` object with `VariableType` and `VariableName` and pushes this object to the `VariableStack`.

If the parser recognises a write access to a variable, it requests the `VariableInformation` from the `VariableStack` with help of the `VariableName`. The `VariableStack` looks for the `VariableInformation` object in the stack and returns it. The parser sets the write flag in the `VariableInformation` object after that.

If the `VariableInformation` for the used variable can not be found this means that the variable is not declared and therefore there is an error. This error is shown to the user.

If the parser recognises a read access to a variable, it requests the `VariableInformation` from the `VariableStack` with help of the `VariableName`. As well as with the write access the `VariableStack` looks for the `VariableInformation` object and returns it. The parser sets the read flag in the `VariableInformation` object after that. Additionally it is checked, if the write flag is set. If the flag is not set a variable is accessed, which has not been initialised. This leads in most cases to an error. The user is informed of the uninitialised variable.

If no `VariableInformation` object was found on the stack, like in the write access, an error is shown, too.

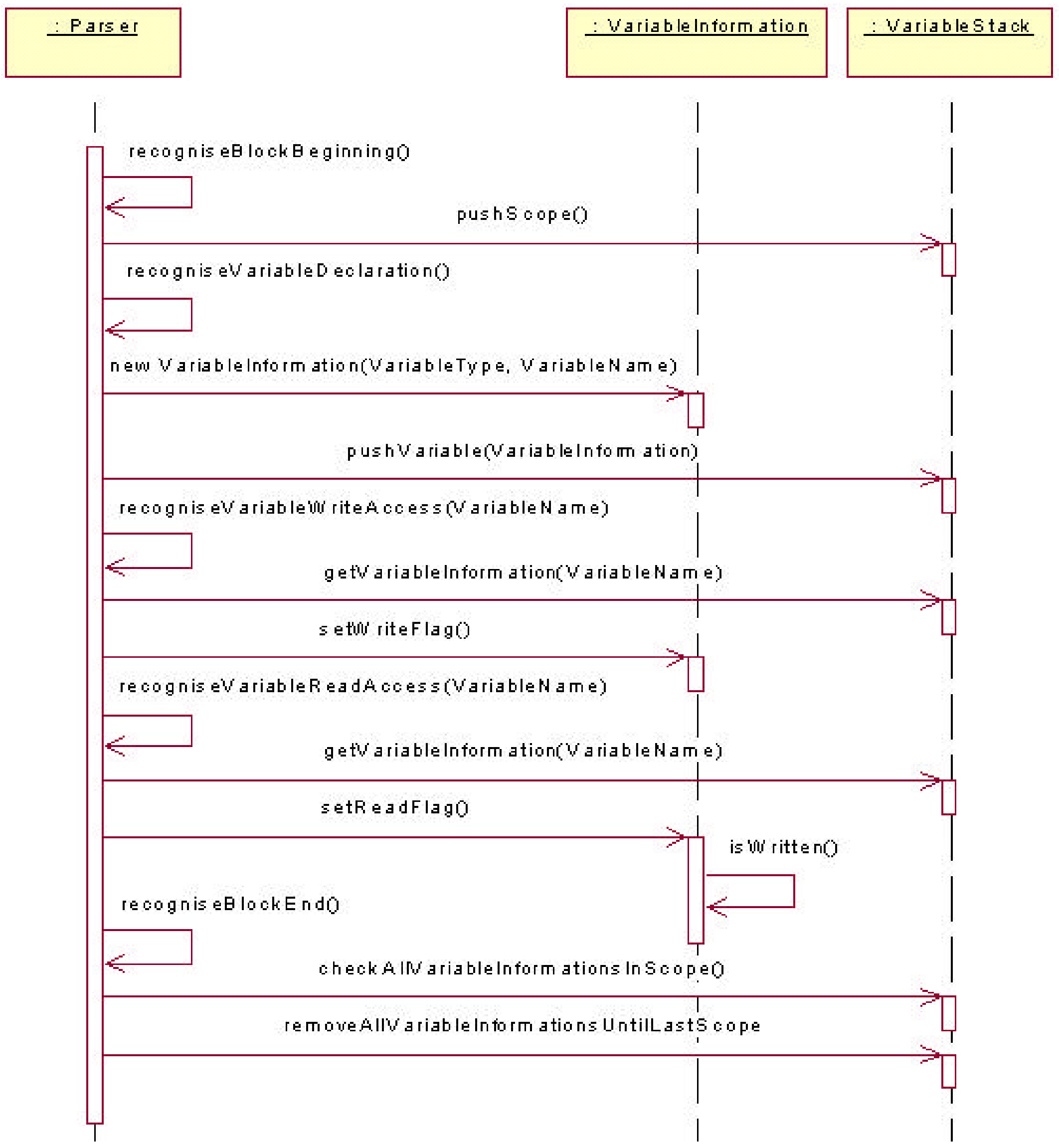
: Parser
: VariableInformation
: VariableStack
recogniseBlockBeginning()
pushScope()
recogniseVariableDeclaration()
new VariableInformation(VariableType, VariableName)
pushVariable(VariableInformation)
recogniseVariableWriteAccess(VariableName)
getVariableInformation(VariableName)
setWriteFlag()
recogniseVariableReadAccess(VariableName)
getVariableInformation(VariableName)
setReadFlag()
isWritten()
recogniseBlockEnd()
checkAllVariableInformationsInScope()
removeAllVariableInformationsUntilLastScope

Figure 4-1 Scenario of a variable access

If the parser recognises the end of a block, all `VariableInformation` objects which lay in the current scope on the stack are checked, if write and read flags are set. If both flags are not set, the user's attention is called that a variable was declared, but that there is no use for it after that. If the write flag is set, the user is shown that a variable was declared and described but the contents is never read.

After that all `VariableInformation` objects, which lay in the current scope are removed from the stack.

4.3.3 Scenario Emit

The scenario in Figure 4-1 shows emitting of instrumented source code.

While parsing the source code, the parser creates an internal parse tree. The tree nodes represent non terminal symbol. For each symbol an individual class has to be created which are used to instantiate the tree nodes. The parse tree is run through recursively from the root and each node writes its equivalent source code to the `OutputFile` handed over as arguments. At the places where an instrumentation is necessary, calls to coverage functions are built in and this code also written to the `OutputFile`. Additionally information is entered in the Information Database are entered and an `ExecuteFlag` is created in the Coverage Database. The information comprise `InputFileName`, `FunctionName`, `Coverage` and `CoveragePointNumber`, which are necessary for a unique identification. The parameter `Coverage` indicates the coverage level of the `CoveragePoint`.

In the scenario a simplified emitting is shown. The parser calls the root object with the parameters `OutputFile` and `InputFileName`, in order to emit the code. The root is an object of `ExternalDeclaration`. The sub node of an `ExternalDeclaration` object is an object of `FunctionDefinition`. This one calls the object of `CompoundStatement` and hands over the `OutputFile`, the `InputFileName`, the `FunctionName`, which was determined while parsing, and the `Coverage`. The level of coverage is Function Coverage. In the `CompoundStatement` object the new coverage point with the parameters `InputFileName`, `FunctionName` and `Coverage` is added to the Information Database and the `CoveragePointNumber` is returned to the `CompoundStatement` object. The `CoveragePointNumber` is a running index, with which an access to a `CoveragePoint` in the Coverage Database is possible. After that an `ExecuteFlag` in the Coverage Database is added for the `CoveragePoint`. At the end the source code is instrumented with a coverage function and it is emitted to the `OutputFile`.

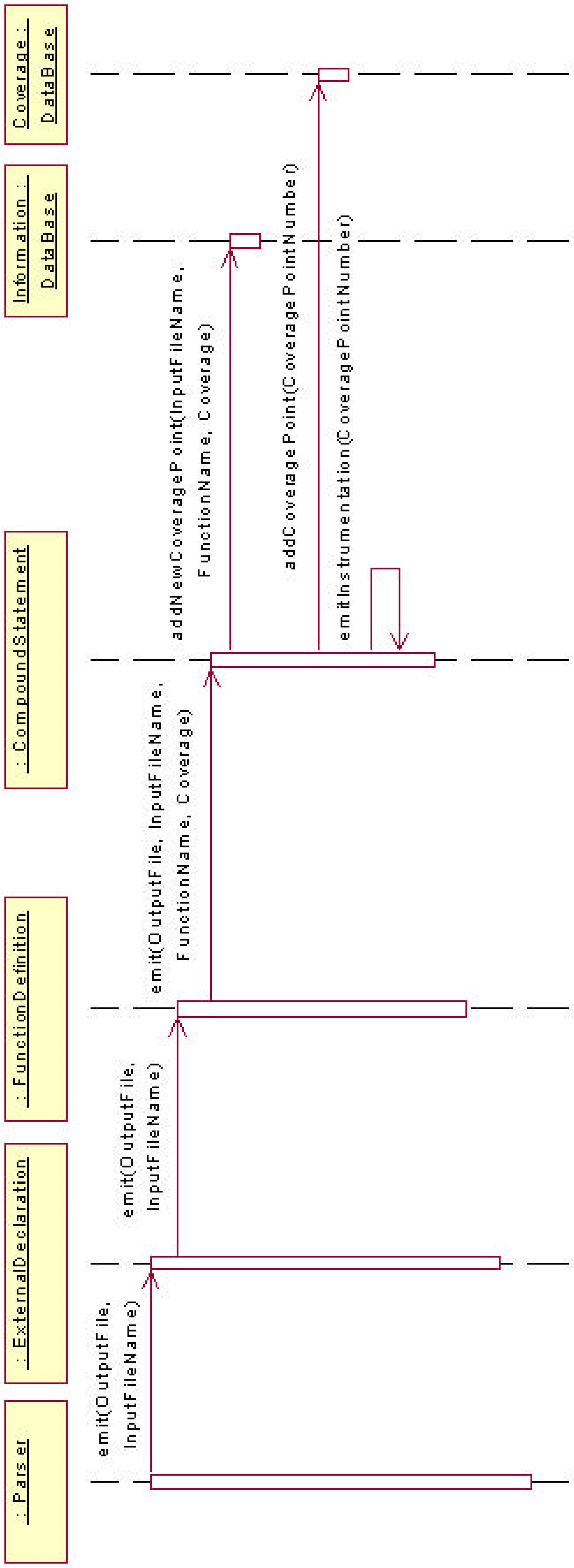
: Parser
: ExternalDeclaration
: FunctionDefinition
: CompoundStatement
Information : DataBase
Coverage : DataBase
emit(OutputFile, InputFileName)
emit(OutputFile, InputFileName)
emit(OutputFile, InputFileName, FunctionName, Coverage)
addNewCoveragePoint(InputFileName, FunctionName, Coverage)
addCoveragePoint(CoveragePointNumber)
emitInstrumentation(CoveragePointNumber)

Figure 4-1 Scenario of emitting the instrumented code

4.3.4 Scenario Coverage Evaluation

The scenario in Figure 4-1 shows the procedure of the Evaluation-Tool.

The user starts the application and indicates the arguments, which coverage should be evaluated. After that the Information Database and the Coverage Database are read. Depending on the arguments which were indicated at the call, the respective object, which determines with help of the Information Database and the Coverage Database the respective coverage, is called for the Function Coverage, Block Coverage or Decision Coverage. At the end the results are reported to the user.

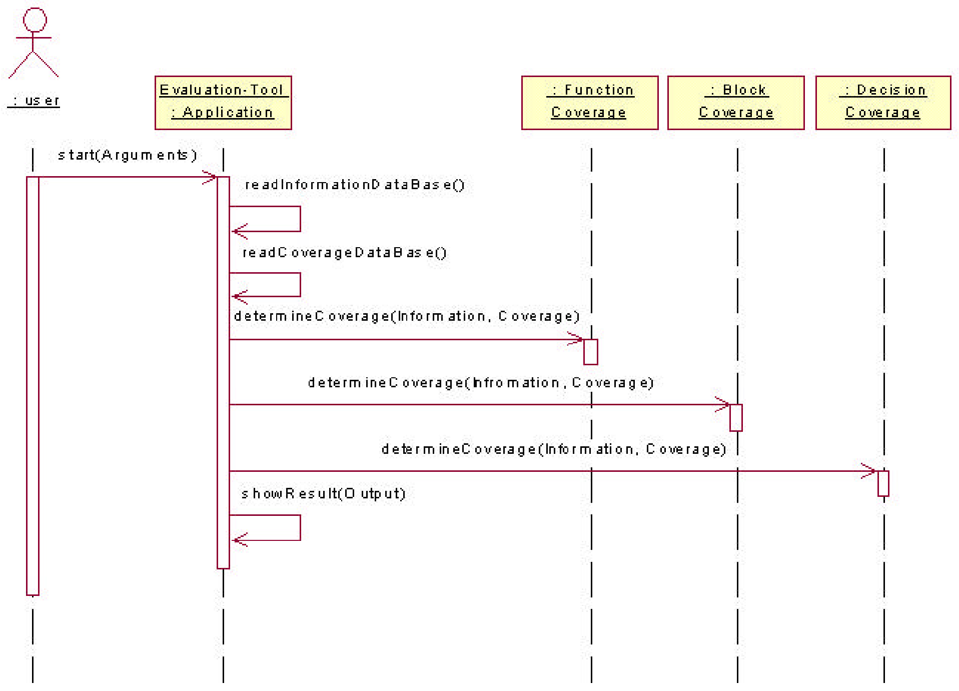
: user
Evaluation-Tool : Application
: Function Coverage
: Block Coverage
: Decision Coverage
start(Arguments)
readInformationDataBase()
readCoverageDataBase()
determineCoverage(Information, Coverage)
determineCoverage(Infromation, Coverage)
determineCoverage(Information, Coverage)
showResult(Output)

Figure 4-1 Scenario of the Evaluation-Tool

5 Design

The following subchapters deal with the design of the single components. Primarily it is about finding and using classes and the relation of them. The use cases and scenarios described above are basis for that.

5.1 Instrumentation-Tool

In the scenario in Figure 4-1 the domain classes and their functionality can be recognised. The resulting classes are:

- `InstrumentationTool`
- `Preprocessor`
- `Parser`
- `Compiler`
- `Linker`

The class `InstrumentationTool` realises the interface to the application and is the central control object. The object is activated with the method `start` in the `main` function of the application, with which all command line parameters are transferred to the `InstrumentationTool` object. After that the method `parseCommandLine` is called, which receives the command line parameters and all compiler arguments in `m_pCompilerArgumentArray` and increments for each argument `m_numberOfCompilerArguments`. If the compiler argument '`-c`' is recognised while parsing the command line, the flag `m_isOnlyCompiled` is set to `true`. If the compiler argument '`-o`' is recognised, the succeeding argument is interpreted as output file name and is assigned to `m_pOutputFileName`. Arguments with no preceding '`-`', are interpreted as source file names and are stored in `m_pSourceFileArray`. All other compiler arguments are not interpreted by the Instrumentation-Tool and are only passed through.

After that with the method `createDataBases` it is checked if the databases are available. If this is not the case the Information Database and die Coverage Database are created.

After that the `Preprocessor` is triggered by calling the method `precompile`, to precompile the indicated source file. Then the `Parser` is triggered by calling the method `parse` to parse the indicated source file. Finally the `Compiler` is triggered to compile the indicated source file. If the attribute `m_isOnlyCompiled` the `Linker` is triggered by calling the method `link` to link the indicated object files.

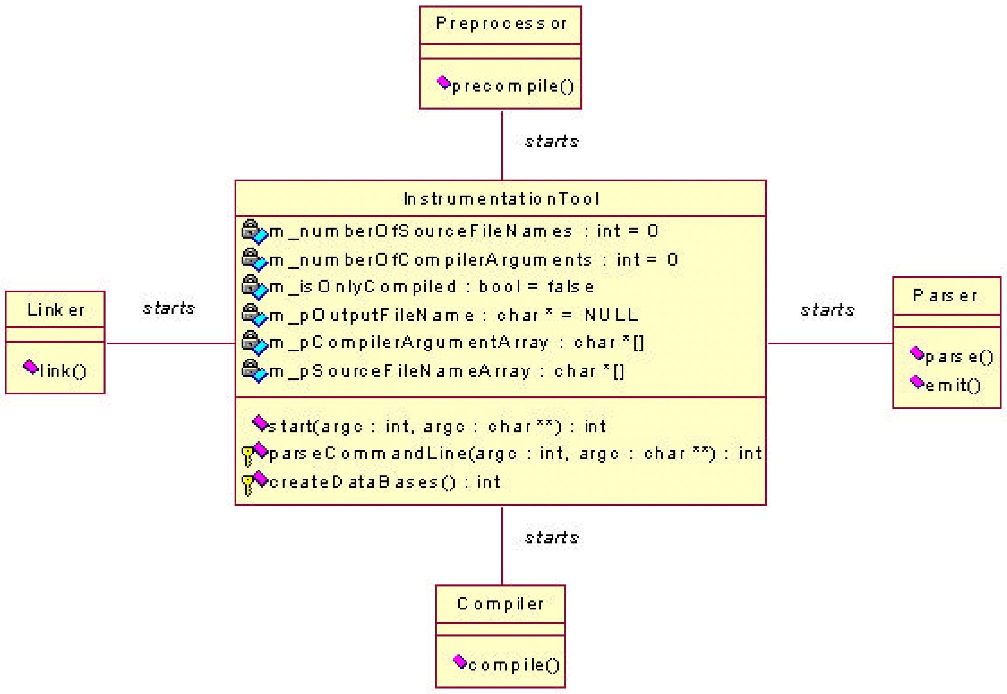
Preprocessor
precompile()
starts
InstrumentationTool
m_numberOfSourceFileNames : int = 0
m_numberOfCompilerArguments : int = 0
m_isOnlyCompiled : bool = false
m_pOutputFileName : char * = NULL
m_pCompilerArgumentArray : char *[]
m_pSourceFileNameArray : char *[]
start(argc : int, argc : char **) : int
parseCommandLine(argc : int, argc : char **) : int
createDataBases() : int
Linker
link()
starts
starts
Parser
parse()
emit()
starts
Compiler
compile()

Figure 5-1 Simple class diagram of the Instrumentation-Tool

The attributes `m_pCompilerArgumentArray` and `m_pSourceFileNameArray` are arrays with constant array size. In order to remove the limit of the number of compiler arguments, respective source file names, both arrays have to be replaced by linked lists.

5.1.1 Preprocessor

The preprocessor instructions in the source code are processed by the preprocessor of a standard compiler. The single task of the class `Preprocessor` is to call the preprocessor from a standard compiler. To trigger the preprocessor of the GNU compiler gcc, the options '`-E`', '`-P`', '`-C`' and '`-ansi`' are necessary. The file name for the output file can be indicated by the option '`-o`'. To avoid conflicts with file names, the `Preprocessor` inserts an underscore '_' before the output file name. In the method `precompile` the class `Preprocessor` builds a command string with the options mentioned above and the options, which are transferred as parameters to the application, and then the class calls the standard compiler. The return value of the `precompile` method indicates, if the precompile process was successful.

5.1.2 Parser

The class `Parser` should serve as an interface to the yacc parser. In the method `parse` the file name for the file which has to be parsed, is built with help of the parameter `pSourceFileName`. This is done by putting an underscore '_' before the source file name. Additionally the output file name is built by putting a double underscore '__' before the source file name. After that the `parse` function of the yacc parser is called with the parameters output file name and input file name.

In the `parse` method of the class `Parser` the function `yyparse` is called. Due to that the real parser generated by yacc is started. While parsing a parse tree is built, whose root is an object of `ExternalDeclaration`.

For each non terminal symbol a class with the same name is provided [HOLMES, 1995]. If a rule for a non terminal symbol is reduced, so a new object of the corresponding class of the non terminal symbol is created and it is initialised with the return values of the rule's symbols and is saved. The new object serves each as return value of the non terminal symbol. Each class owns different constructors in order to cover the different rules of the corresponding non terminal symbol. Additionally such a class owns a method `emit` which has as parameter at least one file pointer, to emit the code into a file again.

An exception are those non terminal symbols, which are part of a list of non terminal symbols. The non terminal symbols which are elements of a list, are displayed by the class of the non terminal symbol responding the list. The non terminal symbol `declaration_list` is an example of such a symbol. It is defined to be either a `declaration` or a `declaration_list` followed by a `declaration`. The classes which should realise such lists, own a method `append`, which owns as a parameter a pointer to an object of the own class. By calling `append` the parameter object is added to the end of the list.

In a similar way the non terminal symbol `translation_unit` was treated as a list of `external_declaration`. The class `TranslationUnit` therefore does not exist.

Figure 5-1 shows the connection between classes representing some important parse tree nodes. The assignment of classes and non terminal symbols is described in detail in chapter 9.1.

If the input file could be parsed successfully, the method `emit` is called with the parameters `pOutputFile` and `pFunctionName` of the root object. At this the parse tree is run through recursively and all elements of the parsed source code and the ones of the instrumented source code are written in the `pOutputFile`.

The return value of the `parse` method shows, if parsing was successful.

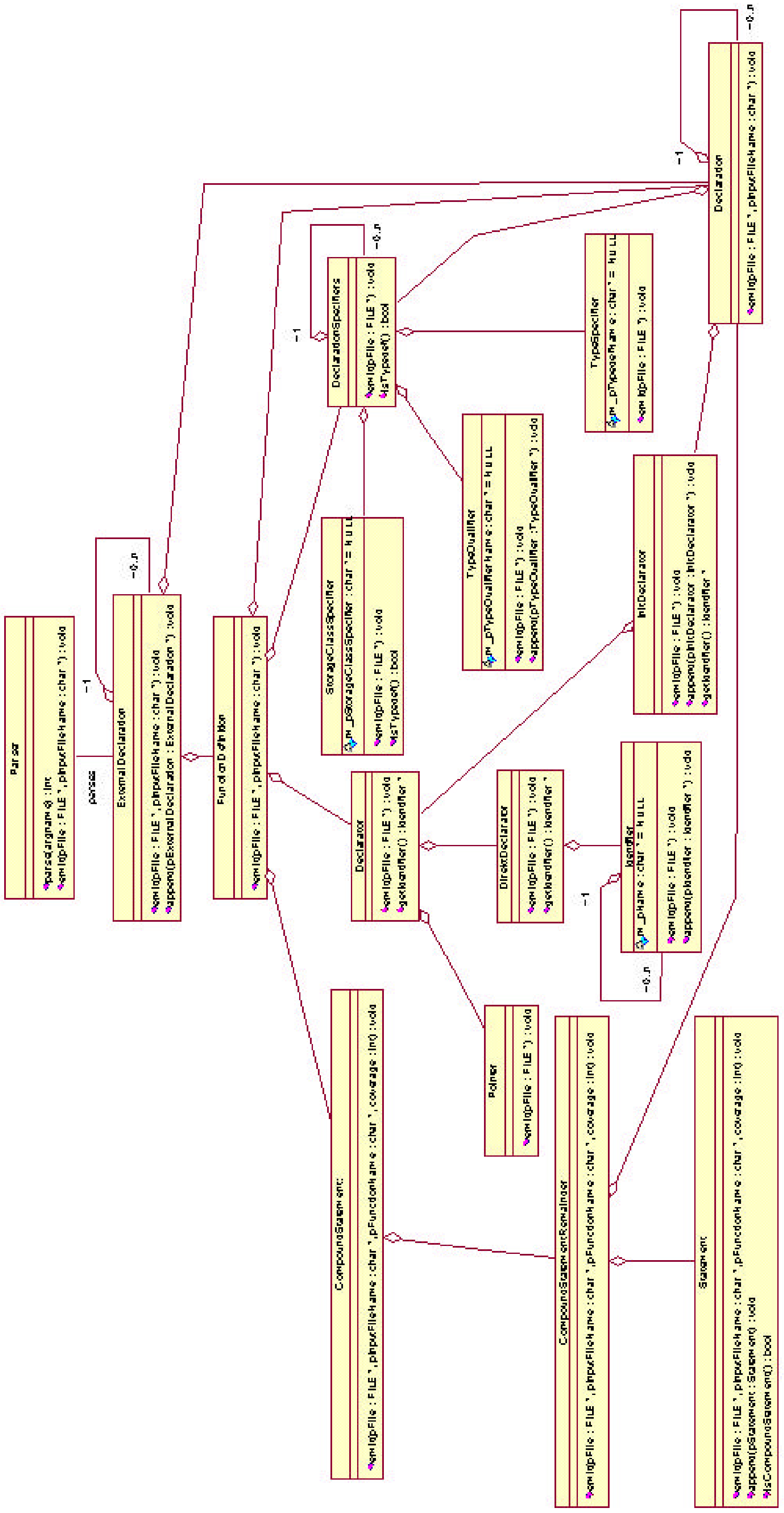

Figure 5-1 Part of the class diagram of the parser

5.1.3 Compiler

A standard compiler is used for compiling the instrumented source code, too. The class `Compiler` has the function to arrange the command line string and to call the standard compiler. For this the method `compile` is placed at disposal. The command line string, which calls the compiler, is built of the compiler options, the output file name and the source file name, which are transferred as parameters. The return value of the `compile` method indicates, if the compile process was successful.

As standard compiler GNU gcc is used.

5.1.4 Linker

The class `Linker` is the interface to the linker. As actual linker a standard linker is used, too. The function of the class Linker is, to arrange the command line string for the linker and to call the standard linker. This happens in the method `link`. As parameters the compiler arguments, the output file name and the source file names which have to be linked, are transferred. When arranging the command line string the library option '`-ltct`' is added at the end. By this option the library `libtct.a`, in which the coverage functions are implemented, is linked. The library is described in chapter 5.3 in detail. The return value of the `link` method indicates, if the link process was successful.

As standard linker GNU gcc is used.

5.2 Databases

For the coverage tool two databases are necessary. These are:

- Information Database
- Coverage Database

Generally speaking a database management system could be used for them together with SQL for inserting and querying the necessary information. However, this would lead to an additional overhead. For that reason both databases are realised by files. Access to the databases is provided by interface functions, which are implemented in the library `libtct.a` (see below).

In the following chapters the information, which is stored in the respective database, is described.

5.2.1 Information Database

The Information Database records binary Information of the coverage points, which are inserted when instrumenting, and records the quantity of the available coverage points.

The coverage point information consists of the file name and the function name, in which the coverage point is inserted, the level of coverage and an exact index with which the coverage point is addressed. To simplify the reading of the stored strings, the length of each string is recorded as an additional information, too.

The quantity of the available coverage points is stored in the first four bytes (the size of an integer value) of the database file.

In the prototype the name of the Information Database file is called `tct.inf`. In a later version the name could be set with a system variable from outside.

5.2.2 Coverage Database

The Coverage Database records for each coverage point the quantity how often the coverage point was executed. For each coverage point a field of 4 bytes (the size of an integer value) is reserved in the database which can be addressed by the index.

In the prototype the name of the Coverage Database file is called `tct.cov`. As well as in the Information Database the name could be changed with a system variable in a later version.

5.3 Coverage Library

In the Coverage Library all interface functions are implemented for the access to the databases. This library is written in the language C as it has to be linked to the applications which are written in C. In the following all functionalities of the interface functions are described.

The functions `__tct_createInfoDataBase` and `__tct_createCoverageDataBase` are used to create the respective database and to initialise them. If the databases exist already when calling this functions the databases remain unchanged.

If a database should be opened this can be done by calling `__tct_openInfoDataBase` or `__tct_openDataBase` for the respective database.

For reading and writing the quantity of coverage points there are the functions `__tct_readNumberOfCoveragePoints` and `__tct_writeNumberOfCoveragePoints`.

In order to be able to add new coverage points when instrumenting, there is the function `__tct_addNewCoveragePoint`, which returns the index, with which it is possible to access the coverage point. At this the coverage point information is written in the Information Database and the number of coverage points is increased and is written back to the Information Database. Additionally the field for the coverage point is written in the Coverage Database and is initialised with 0. After that the coverage function `__tct_coverage` insert into the source code with the index as a parameter. With a negative index an error occurred and the coverage point information and the coverage field could not be entered in the databases.

With the coverage function `__tct_coverage` the index has to be indicated in order to increase the coverage field for the right coverage point. If the prototype should be used in a multi-process or multi-threaded environment, calls to this function have to by synchronised, so that no write and read conflicts occur when accessing the Coverage Database in parallel.

To simplify the evaluation of the coverage there are the two functions `__tct_readInfoDataBase` and `__tct_readCoverageDataBase`, with which the respective database is read.

5.4 Evaluation-Tool

In the scenario in Figure 4-1 the domain classes and their functionalities can be recognised. The classes and their relations are shown in Figure 5-1 .The classes are:

- `EvaluationTool`
- `FunctionCoverage`
- `BlockCoverage`
- `DecisionCoverage`
- `CoverageInfo`
- `CoveragePoint`

The class `EvaluationTool` realises the interface to the application and is the central control object. The object is activated in the `main` function of the application with the method `start`, with which all command line parameters are transferred to the `EvaluationTool` object. As a result the method `parseCommandLine` is called, which parses the command line parameters and which sets `true` the corresponding coverage flags `m_doFunctionCoverage`, `m_doBlockCoverage` or `m_doDecisionCoverage`. After that the Information Database and the Coverage Database are read using the method `readDataBases`. For each coverage point, which is stored in the Information Database, a `CoveragePoint` object is created and saved in an array in the

`EvaluationTool` object. The entries of the Coverage Database are stored in the `m_pIsExecutedArray` array.

After reading the databases a `CoveragePoint` object is created for each function, which is inserted in the Information Database, and is stored in an array. The size of the array is fixed. As consequence only a finite quantity of `CoverageInfo` objects can be inserted. If this number should be variable, a linked list has to be used.

After that, according to the command line parameters, the method `determineCoverage` is called for an object of the corresponding coverage classes.

At the end the coverage results are reported to the user using.

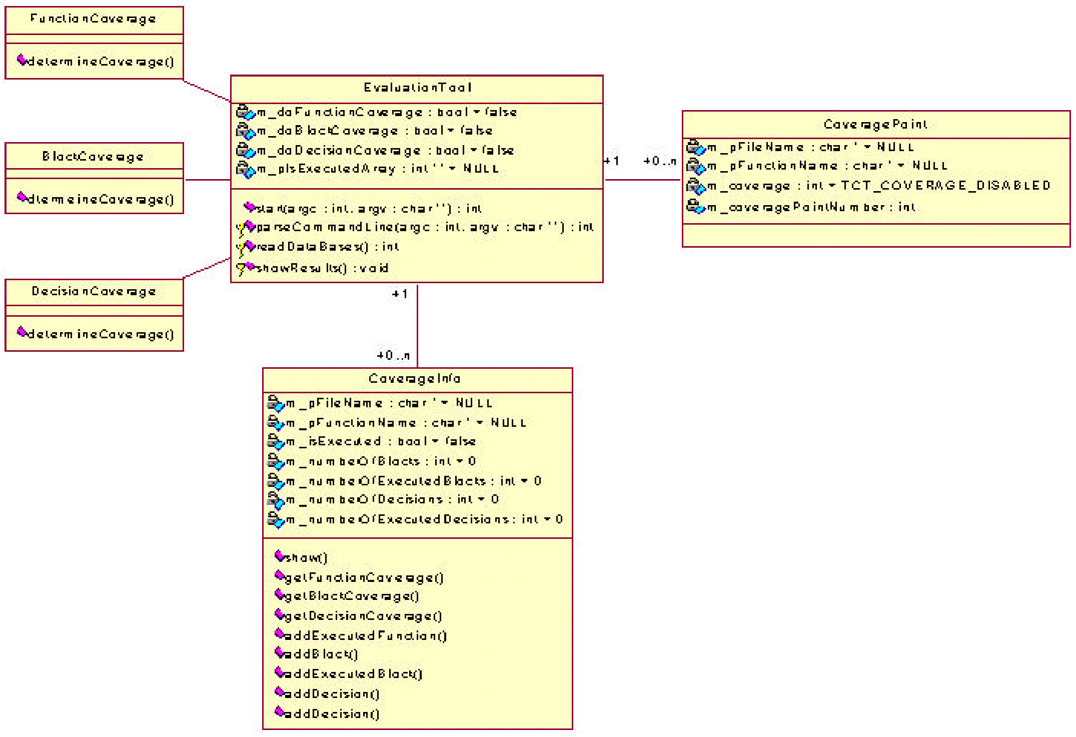
FunctionCoverage
determineCoverage()
BlockCoverage
dtermeineCoverage()
DecisionCoverage
determineCoverage()
EvaluationTool
m_doFunctionCoverage : bool = false
m_doBlockCoverage : bool = false
m_doDecisionCoverage : bool = false
m_pIsExecutedArray : int ** = NULL
start(argc : int, argv : char **) : int
parseCommandLine(argc : int, argv : char **) : int
readDataBases() : int
showResults() : void
+1
+0..n
CoveragePoint
m_pFileName : char * = NULL
m_pFunctionName : char * = NULL
m_coverage : int = TCT_COVERAGE_DISABLED
m_coveragePointNumber : int
+1
+0..n
CoverageInfo
m_pFileName : char * = NULL
m_pFunctionName : char * = NULL
m_isExecuted : bool = false
m_numbeOfBlocks : int = 0
m_numbeOfExecutedBlocks : int = 0
m_numbeOfDecisions : int = 0
m_numbeOfExecutedDecisions : int = 0
show()
getFunctionCoverage()
getBlockCoverage()
getDecisionCoverage()
addExecutedFunction()
addBlock()
addExecutedBlock()
addDecision()
addDecision()

Figure 5-1 Class diagram of the Evaluation-Tool

5.4.1 Coverage Point

A `CoveragePoint` object is created for each coverage point. The class `CoveragePoint` implements a structure, which is used to store the necessary information for each coverage point in the Information Database. The structure comprises fields for the file name and the function name, in which the coverage point is located as well as for the level of coverage and the index.

5.4.2 Coverage Information

A `CoverageInfo` object is created for each function. It has attributes for a file name and a function name. Additionally it owns a flag, which shows if the function was called and was executed. For Block Coverage as well as for Decision Coverage there are two attributes to store the number of coverage points of the respective coverage and the number of executed coverage points.

To set the number of coverage points, there are methods to add points for the respective level of coverage.

The respective coverage for a function can be queried by the methods `getFunctionCoverage`, `getBlockCoverage` and `getDecisionCoverage`.

Additionally there is the method `show`, which creates a formatted output.

5.4.3 Coverage Classes

The classes `FunctionCoverage`, `BlockCoverage` and `DecisionCoverage` represent the different coverage levels. All three classes own a method `determineCoverage`. In this method the `CoverageInfo` array is run through and for each function and for each level of coverage the respective number of coverage points and the number of executed coverage points is determined. This happens with help of the `CoveragePoint` array.

6 Testing

The testing of the coverage Tool is divided into three phases. In the first phase, the module test, each class is tested separately. For each class drivers and stubs are necessary. In the second phase, the component test, classes which have a close relation, are connected together to a component. Afterwards all interactions of the connected classes and their interfaces to the external environment are tested. At the end, in the third phase, called the integration test, all components are connected and their functionality is tested.

6.1 Module Test

For testing the classes `Precompiler`, `Compiler` and `Linker` a test application is necessary which is named gcc. This test application is called in the same way as the real GNU gcc compiler and has the function to check the transferred command line parameters.

For testing the symbol classes, for each class a test bed is created. In this it is checked if all available constructors initialise the attributes correctly and if they emit correct code when called. Additionally, it is checked if all variables were removed from the heap by the destructors.

The library is tested separately, too. It is checked, if the library functions create the Information Database and the Coverage Database correctly and if they insert the information correctly in the databases. Additionally the database read functions are checked.

6.2 Component Test

In the second phase the classes `InstrumentationTool`, `Preprocessor`, `Compiler` and `Linker` are connected as a component in order to test the external interfaces to call the programs and the internal interfaces between the class `InstrumentationTool` and the classes `Preprocessor`, `Compiler` and `Linker`. A stub class is inserted for the representing `Parser` class.

To test parsing and emitting, all symbol classes and the yacc parser are connected to a test parser. As input files correct C source code files, covering a wide area of the complete C grammar, are used. Additionally faulty C source code files are given to the test parser in order to see whether the `Parser` recognises errors and stops after an error message.

The component test of the Evaluation-Tool was dropped, as the single component is the system itself.

6.3 Integration Test

During the integration test the components are connected and the interaction of them is tested. For the test data from the parser component test can be used.

For the integration test all classes of the Evaluation-Tool are connected and are checked in the context of both databases. A synthetically created Information Database and Coverage Database is used for the tests in order to jet computable results.

At the end the databases, which were created for the test C source code files by the Instrumentation-Tool, are used for testing the Evaluation-Tool.

7 Summary and Conclusion

In this project a prototype was developed, with which it was shown how a static analysis for read and write accesses of variables with primitive data types can be carried out. As well it was shown by the prototype, how the source code can be instrumented for a later Function, Block and Decision Coverage. The functions which are called from the instrumented source code are included in a runtime library, which has to be linked to the executable.

Furthermore an Evaluate-Tool was developed, with which it is possible to determine the respective coverage for the whole instrumented source code, or for chosen functions.

By the prototype a framework was built, in which the required additional functionalities of a complete test tool can be integrated.

For a complete test tool the analysis must be enhanced. Read and write accesses have to be analysed for variables of complex data types. Additionally a boundary check of static arrays has to be carried out. During static analysis, metrics could be set up, e.g. to compute the relation between source code documentation and program statements or to check of how many statements a function consists. These metrics are interesting, as it was shown that with a small documentation to code relation or with functions with many program statements the error probability is higher than with detailed documented functions and functions with little program statements.

For the dynamic analysis the tool could be extended for the Condition and Path Coverage. The use of dynamic memory often causes errors. Therefore during the dynamic analysis a rigid check of the allocation and release of memory has to be done. Additionally the boundaries of dynamic arrays have to be checked. By these checks segmentation faults and memory leaks can be detected.

For an effective evaluation of the coverage results the statement of the coverage degree is not enough, as the information which sections were executed gets lost. The coverage results should rather be visualised in a kind of editor, who opens the source code and highlights the executed sections. So the user has a fast overview, at which places he has to improve or extend his test cases.

The use of such a test tool can save time during the test phase, otherwise more time has to be spent during the design phase to consider the necessary test cases. The quality of a system is not improved automatically by using a test tool. There is only the possibility to show the quality of the test case and therefore a higher quality of the system can be reached. To this end the coverage results have to be considered carefully and the test cases have to be improved and extended accordingly.

8 References and Bibliography

[BEIZER, 1990] Beizer, Boris (1990), Software Testing Techniques, 2nd Edition, New York, Van Nostrand Reinhold, ISBN 0-442-20672-0

[MYERS, 1991] Myers, Glenford J. (1991), Methodisches Testen von Programmen (Engl. The Art of Software Testing), 4th Edition, Munich, R. Oldenbourg, 3-486-21877-8

[SCHACH,1999] Schach, Stephen R. (1999), Classical And Object-Oriented Software Engineering, 4th Edition, Singapore, McGraw Hill, ISBN 0-07-116761-7

[KERNIGHAN, 1990] Kernighan, Brian W., Ritchie, Dennis M. (1990), Programmieren in C (Engl. The C Programming Language), 2nd Edition, Munich, Hanser, ISBN 3-446-15497-3

[AHO, 1988] Aho,Alfred V., Sethi, Ravi, Ullman, Jeffrey D. (1988), Compilerbau Part 1 and 2 (Engl. Principles of Compiler Design), 1st Edition, Munich, Addison-Wesley, ISBN 3-89319-150-X and 3-89319-151-8

[SCHMITT, 1992] Schmitt, Franz Josef (1992), Praxis des Compilerbaus, 1st Edition, Munich, Hanser, ISBN 3-446-16517-7

[HOLMES, 1995] Holmes, Jim (1995), Object-Oriented Compiler Construction,1st Edition, Englewood Cliffs, Prentice Hall, ISBN 0-13-630740-X

[HEROLD, 1999] Herold, Helmut (1999), Linux-Unix Profitools, 3rd Edition, Germany, Addison-Wesley, ISBN 3-8273-1448-8

[WOODWARD, 1980] Woodward, Martin R., Hedley, David, Hennell, Michael A. (1980), “Experience with Path Analysis and Testing of

Programs", IEEE Transaction on Software Engineering, Vol. SE-6, No. 3, pp 278-286

[NTAFOS, 1988] Ntafos, Simeon C. (1988), "A Comparison of Some Structural Testing Strategies", IEEE Transaction on Software Engineering, Vol. 14, No. 6, pp 868-874

[CORNETT] Cornett, Steve, "Code Coverage Analysis", http://www.bullseye.com/coverage.html

IPL, "An Introduction to Software Testing", http://www.iplbath.com/p820.pdf

IPL, "Structural Coverage Metrics", http://www.iplbath.com/p823.pdf

9 Appendixes

9.1 C Grammar

The following deals with the C grammar, which refers to [KERNIGHAN, 1990] and [HEROLD, 1999]. At this in the first line there is the non terminal symbol, which is described by the following lines. All non terminal symbols are written in small letters, the tokens are written in capital letters or are shown as character directly. The names, which are stated in brackets with the non terminal symbol, are the used class names respective the return type. The brackets which are stated at the end, are only necessary, that yacc gets no shift/reduce and reduce/reduce conflicts. The brackets are not covered by own classes. The respective non terminal symbol returns the ascii value of the bracket.

```
translation_unit(ExternalDeclaration):
      external_declaration
      translation_unit external_declaration

external_declaration(ExternelDeclaration):
      function_definition
      declaration

function_definition(FunctionDefinition):
      declarator compound_statement
      declaration_specifiers declarator compound_statement
      declarator declaration_list compound_statement
      declaration_specifiers declarator declaration_list
            compound_statement

declaration(Declaration):
      declaration_specifiers ';'
      declaration_specifiers init_declarator_list ';'

declaration_list(Declaration):
      declaration
      declaration_list declaration

declaration_specifiers(DeclarationSpecifiers):
      storage_class_specifier
      declaration_specifiers storage_class_specifier
      type_specifier
      declaration_specifiers type_specifier
      type_qualifier
      declaration_specifiers type_qualifier

storage_class_specifier(StorageClassSpecifier):
      AUTO
      REGISTER
      STATIC
      EXTERN
      TYPEDEF

type_specifier(TypeSpecifier):
      VOID
      CHAR
      SHORT
      INT
      LONG
      FLOAT
      DOUBLE
      SIGNED
      UNSIGNED
      struct_or_union_specifier
      enum_specifier
      TYPEDEF_NAME

type_qualifier(TypeQualifier):
      CONST
      VOLATILE

struct_or_union_specifier(StructOrUnionSpecifier):
      struct_or_union left_brace struct_declaration_list right_brace
      struct_or_union IDENTIFIER left_brace struct_declaration_list
            right_brace
      struct_or_union IDENTIFIER

struct_or_union(StructOrUnion):
      STRUCT
      UNION

struct_declaration_list(StructDeclaration):
      struct_declaration
      struct_declaration_list struct_declaration

init_declarator_list(InitDeclaration):
      init_declarator
      init_declarator_list ',' init_declarator

init_declarator(InitDeclaration):
      declarator
      declarator '=' initializer

struct_declaration(StructDeclaration):
      specifier_qualifier_list struct_declarator_list ';'

specifier_qualifier_list(SpecifierQualifier):
      type_specifier
      type_specifier specifier_qualifier_list
      type_qualifier
      type_qualifier specifier_qualifier_list

struct_declarator_list(StructDeclarator):
      struct_declarator
      struct_declarator_list ',' struct_declarator

struct_declarator(StructDeclarator):
      declarator
      ':' constant_expression
      declarator ':' constant_expression

enum_specifier(EnumSpecifier):
      ENUM enum_remainder
```

```
enum_remainder(EnumRemainder):
      left_brace enumerator_list right_brace
      IDENTIFIER left_brace enumerator_list right_brace
      IDENTIFIER

enumerator_list(Enumerator):
      enumerator
      enumerator_list ',' enumerator

enumerator(Enumerator):
      IDENTIFIER
      IDENTIFIER '=' constant_expression

declarator(Declarator):
      direct_declarator
      pointer direct_declarator

direct_declarator(DirectDeclarator):
      IDENTIFIER
      left_parenthese declarator right_parenthese
      direct_declarator '[' ']'
      direct_declarator '[' constant_expression ']'
      direct_declarator left_parenthese parameter_type_list
            right_parenthese
      direct_declarator '(' ')'
      direct_declarator left_parenthese identifier_list
            right_parenthese

pointer(Pointer):
      '*' pointer_remainder

pointer_remainder(PointerRemainder):
      /* empty */
      type_qualifier_list
      pointer
      type_qualifier_list  pointer

type_qualifier_list(TypeQualifier):
      type_qualifier
      type_qualifier_list type_qualifier

parameter_type_list(ParameterType):
      parameter_list
      parameter_list ',' ELLIPSIS

parameter_list(ParameterDeclaration):
      parameter_declaration
      parameter_list ',' parameter_declaration

parameter_declaration(ParameterDeclaration):
      declaration_specifiers declarator
      declaration_specifiers
      declaration_specifiers abstract_declarator

identifier_list(Identifier):
      IDENTIFIER
      identifier_list ',' IDENTIFIER

initializer(Initializer):
      assignment_expression
      left_brace initializer_remainder

initializer_remainder(InitializerRemainder):
      initializer_list right_brace
      initializer_list ',' '}'

initializer_list(Initializer):
      initializer
      initializer_list ',' initializer

type_name(TypeName):
      specifier_qualifier_list
      specifier_qualifier_list abstract_declarator

abstract_declarator(AbstractDeclarator):
      pointer
      direct_abstract_declarator
      pointer direct_abstract_declarator

direct_abstract_declarator(DirectAbstractDeclarator):
      left_parenthese abstract_declarator right_parenthese
      '[' ']'
      '[' constant_expression ']'
      direct_abstract_declarator '[' ']'
      direct_abstract_declarator '[' constant_expression ']'
      '(' ')'
      left_parenthese parameter_type_list right_parenthese
      direct_abstract_declarator '(' ')'
      direct_abstract_declarator left_parenthese parameter_type_list
            right_parenthese

statement(Statement):
      labeled_statement
      expression_statement
      compound_statement
      selection_statement
      iteration_statement
      jump_statement

labeled_statement(LabeledStatement):
      IDENTIFIER ':' statement
      CASE constant_expression ':' statement
      DEFAULT ':' statement

expression_statement(ExpressionStatement):
      ';'
      expression ';'

compound_statement(CompoundStatement):
      '{' compound_remainder

compound_remainder(CompoundRemainder):
      '}'
      declaration_list '}'
      statement_list '}'
      declaration_list statement_list '}'

statement_list(Statement):
      statement
      statement_list statement

selection_statement(SelectioStatement):
      if_head statement
      if_head statement ELSE statement
      SWITCH '(' expression ')' statement

if_head(IfHead):
      IF '(' expression ')'

iteration_statement(IterationStatement):
      WHILE '(' expression ')' statement
      DO statement WHILE '(' expression ')' ';'
      FOR '(' expr ';' expr ';' expr ')' statement

expr(Expression):
      /* empty */
      expression

jump_statement(JumpStatement):
      GOTO IDENTIFIER ';'
      CONTINUE ';'
      BREAK ';'
      RETURN ';'
      RETURN expression ';'

expression(Expression):
      assignment_expression
      expression ',' assignment_expression

assignment_expression(AssignmentExpression):
      conditional_expression
      unary_expression assignment_operator assignment_expression

assignment_operator(AssignementOperator):
      '='
      MUL_ASSIGN
      DIV_ASSIGN
      MOD_ASSIGN
      ADD_ASSIGN
      SUB_ASSIGN
      LEFT_SHIFT_ASSIGN
      RIGHT_SHIFT_ASSIGN
      AND_ASSIGN
      XOR_ASSIGN
      OR_ASSIGN

conditional_expression(ConditionalExpression):
      logical_or_expression
      logical_or_expression '?' expression ':'
            conditional_expression

constant_expression(ConstantExpression):
      conditional_expression

logical_or_expression(LogicalOrExpression):
      logical_and_expression
      logical_or_expression OR_OP logical_and_expression

logical_and_expression(LogicalAndExpression):
      inclusive_or_expression
      logical_and_expression AND_OP inclusive_or_expression

inclusive_or_expression(InclusiveOrExpression):
      exclusive_or_expression
      inclusive_or_expression '|' exclusive_or_expression

exclusive_or_expression(ExclusiveOrExpression):
      and_expression
      exclusive_or_expression '^' and_expression

and_expression(AndExpression):
      equality_expression
      and_expression '&' equality_expression

equality_expression(EqualityExpression):
      relational_expression
      equality_expression EQUAL_OP relational_expression
      equality_expression NOT_EQUAL_OP relational_expression

relational_expression(RelationalExpression):
      shift_expression
      relational_expression '<' shift_expression
      relational_expression '>' shift_expression
      relational_expression LESS_EQUAL_OP shift_expression
      relational_expression GREATER_EQUAL_OP shift_expression

shift_expression(ShiftExpression):
      additive_expression
      shift_expression LEFT_SHIFT_OP additive_expression
      shift_expression RIGHT_SHIFT_OP additive_expression

additive_expression(AdditiveExpression):
      multiplicative_expression
      additive_expression '+' multiplicative_expression
      additive_expression '-' multiplicative_expression

multiplicative_expression(MultiplicativeExpression):
      cast_expression
      multiplicative_expression '*' cast_expression
      multiplicative_expression '/' cast_expression
      multiplicative_expression '%' cast_expression

cast_expression(CastExpression):
      unary_expression
      left_parenthese type_name right_parenthese cast_expression

unary_expression(UnaryExpression):
      postfix_expression
      INC_OP unary_expression
      DEC_OP unary_expression
      unary_operator cast_expression
      size_of unary_expression
      size_of left_parenthese type_name right_parenthese

unary_operator(UnaryOperator):
      '&'
      '*'
      '+'
      '-'
      '~'
      '!'

postfix_expression(PostfixExpression):
      primary_expression
      postfix_expression '[' expression ']'
      postfix_expression '(' ')'
      postfix_expression left_parenthese arg_expression_list
            right_parenthese
      postfix_expression '.' IDENTIFIER
      postfix_expression PTR_OP IDENTIFIER
      postfix_expression INC_OP
      postfix_expression DEC_OP

primary_expression(PrimaryExpression):
      IDENTIFIER
      constant
      STRING_LITERAL
      left_parenthese expression right_parenthese

arg_expression_list(AssignmentExpression):
      assignment_expression
      arg_expression_list ',' assignment_expression

constant(Constant):
      INTEGER_CONSTANT
      CHARACTER_CONSTANT
      FLOATING_CONSTANT
      ENUMERATION_CONSTANT

size_of(SizeOf):
      SIZEOF

left_parenthese:
      '('

right_parenthese:
      ')'

left_brace:
      '{'

right_brace:
      '}'
```

9.2 Management of the Project

The milestone plan of the interim report, which was attached to the report, was a helpful tool managing the project at the beginning. As the implementation was a greater time expenditure than expected the time plan of the interim report did not correspond to reality at the end. However the background, aims and objectives stated in the interim report could be realised in the dissertation project.