Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

Magisterarbeit

Automatic Loop Bound Analysis of Programs written in C

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines

Diplomingenieurs

unter der Leitung von

Ao.Univ.Prof. Dr. Peter Puschner
Institut für Technische Informatik 182/1

und als verantwortlich mitwirkenden Universitätsassistenten

Dipl.-Ing. Dr. Raimund Kirner
Institut für Technische Informatik 182/1

eingereicht an der
Technischen Universität Wien,
Technisch-Naturwissenschaftliche Fakultät

durch
Martin Kirner Bakk. tech.
Matr. - Nr. 0027593
Würtzlerstraße 23/1/10, A-1030 Wien

Wien, am 31. August 2006 ....

Automatic Loop Bound Analysis of Programs written in C

Abstract

The knowledge about the worst-case execution time is important for the design of real-time systems. Without a save upper bound for the execution time it cannot be guaranteed that the system will meet all its deadlines. As part of the worst-case execution-time calculation, it is important to know how many times the body of a loop will be executed after entering the loop header for the first time. Traditionally, loop bounds had to be provided explicitly, in the form of source-code annotations to support timing analysis of real-time programs.

This thesis presents a method that is able to calculate a lower and an upper bound for the number of iterations of different loop types by analyzing the semantics of a source code, written in the high-level language C. Only if the number of iterations of a loop depends on unknown variable values, annotations about the value bounds have to be given in the source code. The analysis of loops is done at the source code level. For every supported loop, the result of the loop-bound calculation is written back into the source file to support the further steps of the WCET analysis.

Keywords: Hard Real Time Systems, Worst Case Execution Time (WCET), Static Analysis, Loop Bound

Automatic Loop Bound Analysis of Programs written in C

Kurzfassung

Das Wissen um die maximale Ausführungszeit ist wichtig für das Design von Echtzeitsystemen. Ohne einer sicheren oberen Grenze für die Ausführungszeit kann nicht garantiert werden, dass das System alle seine zeitlichen Grenzen einhält. Als Teil der Analyse der maximalen Ausführungszeit ist es wichtig zu wissen, wie oft der Schleifenrumpf bei erstmaligen Treffen auf den Schleifenkopf ausgeführt wird. Traditionell wurde diese Begrenzung zur Unterstützung der zeitlichen Analyse explizit als Kommentar in den Quellcode eingefügt.

Diese Diplomarbeit präsentiert eine Methode, welche eine untere und eine obere Grenze für die Anzahl der Durchläufe von verschieden Schleifentypen durch Analyse der Semantic eines in der Programmiersprache C geschriebenen Quellcodes berechnet. Nur wenn die Anzahl der Schleifendurchläufe von unbekannten Variablenwerten abhängt, sind zusätzliche Kommentare mit Wertebegrenzungen dieser Variablen im Quellcode erforderlich. Die Analyse der Schleifen wird auf dem Level des Quellcodes durchgeführt. Für jede unterstützte Schleife wird das Ergebnis der Berechnung in den Quellcode, zur Unterstützung weiterer WCET Analyseschritte, als Kommentar zurückgeschrieben.

Schlüsselwörter: Harte Echtzeitsysteme, Maximale Ausführungszeit (WCET), Statische Analyse, Begrenzung von Schleifendurchläufen

Acknowledgements

I dedicate this work to my family for all their love and support, which had made it possible for me to complete my study successfully and also to write this thesis.

With special thanks to my brother Raimund for his advices and his helping hands during the time of study and also during finishing this work. And further also for his friendly support within private fields during this time.

Thanks also to Dr. Helen Treharne for her support during writing part of this thesis during the time of stay at the University of Surrey in Guildford, UK. Also thanks to my colleagues from the department Technische Informatik, Technical University of Vienna, for their technical support.

At least, I will not forget to say thanks to my girlfriend Gerlinde Freiler for her love and support outside the world of study and computer science.

Danksagung

Ich widme diese Arbeit meiner Familie, die es mir durch ihre Liebe und Unterstützung ermöglicht hat, mein Studium erfolgreich zu absolivieren und diese Diplomarbeit zu schreiben.

Wobei ich speziell meinem Bruder Raimund für seine Ratschläge und Hilfsbereitschaft während der Studienzeit und auch während dem Abfassen dieser Arbeit danken möchte. Und ebenso für seine freundschaftliche Unterstützung im privaten Bereich während dieser Zeit.

Dank gebührt ebenso Dr. Helen Treharе für ihre Hilfestellungen an der Fertigstellung dieser Arbeit während meines Aufenthaltes an der Universität von Surrey in Guildford, UK. Auch den Kollegen des Instituts für Technische Informatik an der Universität Wien möchte ich für ihre fachliche Unterstützung meinen Dank aussprechen.

Und nicht vergessen möchte ich zuletzt, meiner Freundin Gerlinde Freiler, für ihre Liebe und Unterstützung, auch außerhalb der Welt des Studiums und der Informatik, zu danken.

Contents

I Abstract **i**

II Kurzfassung **ii**

III Acknowledgements **iii**

IV Danksagung **iv**

1 Introduction **1**
- 1.1 Motivation 2
- 1.2 Contribution 3
- 1.3 Outlook 4

2 WCET Analysis **5**
- 2.1 Introduction into WCET Analysis 5
- 2.2 *WCET* analysis by measurement 8
  - 2.2.1 Evolutionary testing 9
- 2.3 Static WCET Analysis 12
  - 2.3.1 Calculation of the *WCET* 13
  - 2.3.2 Integer Liner Programming 13
- 2.4 Hybrid Analysis 15

3 Related work **16**
- 3.1 Flow Facts calculation 16
  - 3.1.1 Flow Analysis by using Abstract Interpretation 16
  - 3.1.2 Detection and Exploitation of Value-Dependant Constraints . 19
- 3.2 Using Flow Facts for WCET Analysis 20
  - 3.2.1 Tree-Based WCET Analysis on Instrumentation Point Graphs 20

4 Loop Bound Analysis **23**
- 4.1 General definitions for the Control flow graph 23
- 4.2 Loops with multiple exits 24
  - 4.2.1 Construction of the control flow graph 25
  - 4.2.2 Finding all branches that can affect the number of loop iterations 26

4.2.3 Expansion of the branch tree . . . 29
4.2.4 Calculation of required information . . . 31
4.2.5 Determine the range of iterations when each of these identified branches could be reached . . . 34
4.2.6 Determining the minimum and maximum loop iterations . . . 35
4.2.7 Calculate loops with iteration branches using the equality operator . . . 37
4.3 Loops with a non-constant loop-invariant number of iterations . . . 40

5 Realization of the framework **42**
5.1 Syntactical analysis of the input file . . . 42
5.1.1 Preprocessing . . . 42
5.1.2 Lexical Analysis . . . 42
5.1.3 Syntactical analysis . . . 44
5.2 Syntax tree . . . 46
5.3 Type table . . . 50
5.4 Symbol table . . . 51
5.5 Control Flow Graph . . . 53
5.6 Loop bound analysis . . . 56
5.6.1 Finding all branches that can influence the number of loop iterations . . . 56
5.6.2 Construction of the branch tree . . . 58
5.6.3 Calculation of information about each branch condition . . . 59
5.6.4 Determination of the minimum and maximum number of loop iterations . . . 61

6 Evaluation **65**
6.1 Types of loops that can be successfully bounded . . . 65
6.1.1 Grammatical description of valid expressions . . . 67
6.2 Types of loops that cannot be successfully bounded . . . 71
6.3 Loops that maybe are bounded incorrectly . . . 73
6.4 Examples . . . 74
6.4.1 Example loop 1 . . . 75
6.4.2 Example loop 2 . . . 77

7 Conclusion **79**
7.1 Future work . . . 80

List of Figures

1.1 Structural representation of the working method of the analyser . . . 3

2.1 Structural representation of a real time system (from [Kir03]) 6
2.2 Plot of the execution time probability of a task within a *WCET* system 7
2.3 Test set-up for *WCET* analysis by measurement (from [PN98]) . . . 9
2.4 Graphical representation of evolutionary testing 10
2.5 Recombination of parent individuals to get a new generation of individuals 12
2.6 Annotated control flow graph for flow facts derivation [Kai05] 14

4.1 Sequence of valid statements in C 25
4.2 Example loop with multiple exits 26
4.3 Algorithm to find the set of Iteration Branches for a loop 27
4.4 Control flow graph of the example in figure 4.2 28
4.5 Branch tree with all identified branches 29
4.6 YACC-grammar for logical '||' and '&&' expressions 29
4.7 Condition tree for example in figure 4.2 30
4.8 Branch tree after expanding 30
4.9 Transformation of *SWITCH-CASE*-statements 31
4.10 Branch tree with calculated ranges of iterations 35
4.11 Graph after calculation of minimum and maximum iteration values . 38
4.12 Some example loops using equality operators 39
4.13 Loop with a non-constant loop-invariant number of iterations 41

5.1 Structure of the syntax tree of C language programs 47
5.2 Example code to show the structure of the symbol table 52
5.3 Graphical representation of the symbol table from the example code 52
5.4 Some example code to show the construction of the control flow graph 54
5.5 Control flow graph of the example code within figure 5.4 55
5.6 Graphical representation of an expression tree 60

6.1 Example loop 1 that *can* be bounded by the algorithm 65
6.2 Example loop 2 that *can* be bounded by the algorithm 66
6.3 Example loop 3 that *can* be bounded by the algorithm 66
6.4 Example loop 4 that *can* be bounded by the algorithm 67

6.5 Example loop 5 that *can* be bounded by the algorithm 67
6.6 YACC replacement-rule for an assignment-expression 68
6.7 Examples of valid and invalid expressions 68
6.8 Modified YACC replacement-rule for an conditional-expression . . . 68
6.9 Examples of valid and invalid expressions 69
6.10 Modified YACC replacement-rule for a selection-statement 69
6.11 Examples of valid and invalid selection statements 69
6.12 YACC replacement-rule for a logical-or-expression 70
6.13 YACC replacement-rule for a logical-and-expression 70
6.14 Modified YACC replacement-rule for a primary-expression 70
6.15 Example loop 1 that *can not* be bounded by the algorithm 71
6.16 Example loop 2 that *can not* be bounded by the algorithm 71
6.17 Example loop 3 that *can not* be bounded by the algorithm 72
6.18 Example loops 4 a) and 4 b) that *can not* be bounded by the algorithm 72
6.19 Example loop 5 that *can not* be bounded by the algorithm 73
6.20 Example loop 1 that *maybe* is bounded false 74
6.21 Different calculation results, depending on the variable type 75
6.22 Calculation example loop 1 75
6.23 Branch tree for the example loop of figure 6.22 76
6.24 Calculation example loop 2 77
6.25 Branch tree for the example loop of figure 6.24 77

7.1 Types of loops that could be successful bound 80
7.2 Alias occurrence that could influence the number of iterations 81
7.3 Example of a natural *GOTO* loop 81
7.4 Example of independent nested loop 81
7.5 Example of nested loops that depends on outer loops 82

List of Tables

2.1 Derived *Flow Equations* from figure 2.6 14
2.2 Explanation of used symbols within section Integer Linear Programming 15

4.1 Required information for each *known* iteration branch 32
4.2 How to determine the *adjust* value and the direction of change of the iteration branch 33
4.3 Calculated information for each branch of the tree in picture 4.8 . . 34
4.4 Notation that is used within the assignment rules 36
4.5 Adjust value and direction of change for branches with equality operator 39

5.1 Some often used examples of *YACC* declarations 45
5.2 *YACC* rule to recognise the different types of function definitions . . 46
5.3 *YACC* rule to recognise the different types of loop definitions 46
5.4 Data structure to store all details of a function 48
5.5 Data structure of a statement 49
5.6 Data structure of an expression 49
5.7 Structure of the type table 51
5.8 Structure of the symbol table 53
5.9 Structure of a node within the control flow graph 55
5.10 Return values of the function `is_postdominated_by(...)` 57
5.11 Structure of the elements within the branch tree 58
5.12 First half of the structure *var_info* 60
5.13 Second half of the structure *var_info* 62

Chapter 1

Introduction

Due to technical improvement of computer hardware, IT[1] penetrates more and more into new application areas. New development methods makes it possible to let the size of computer hardware shrink continually and also reduces the production costs. Therefore, different kinds of computer systems will be found also within formerly traditional mechanical equipment and becomes a important part of the human life.

Today, the main share is hold by personal computers that are mainly used within industry but also to fill the private sectors. Beside this domain, computer systems are also used to interact with the environment. Such kind of computer systems are often installed within other objects and called *embedded systems*. The current trend within the computer science is to hide the presence of computer systems from the user. Because computer systems are always getting more complicated and it should be avoided to overtax users without special education. This trend has been summarised as *ubiquitous computing*. For these relatively young areas, *sensors* and *actuators* are additionally needed to interact with the environment.

Every time when an interaction with the environment is needed, this interaction has to fulfil some timing requirements. For example, within the holiday season it is useless to get informed about traffic jam when someone is right in the middle of it and there is no way out. This may be sound funny but there are also situations where it may be dangerous for living beings to miss some timing requirements. A modern *ABS*[2] car brake system has to measure and regulate the braking pressure every few milli seconds. Otherwise the braking distance could increasing and bring the passengers into a dangerous situation. Such systems with fixed timing requirements are also called *real-time systems*.

In addition to the functional correctness of the service, there must be a way to show that a computer system can offer its service guaranteed within a fixed time-

[1] **I**nformation **T**echnology
[2] **A**nti **B**locking **S**system

line. Or, in case of an erroneous system, to show that the service could not delivered within this fixed timeline. The keywords for the timing behaviour are *BCET*[3] and *WCET*[4], which represent a lower and upper bound for the execution time. Both execution-time representations could be calculated *dynamically* (e.g. by measurement), *statically* (e.g. with analytical methods) or as a combination of both. Within this thesis the calculation of loop bounds, which is part of the *WCET* analysis, is done by static methods.

1.1 Motivation

For the calculation of the *BCET* and the *WCET* of a program it is important to know the execution time for each instruction on the target hardware. But a sequence of instructions could also be executed reiterated. Therefore, also the number of reiterations must be known. Structures within a programming language, which are used to express such reiterations are called *loops* and the number of reiterations are called *loop bounds*. Traditionally, these loop bounds have been inserted within the source code by hand before or prompted during analysis. Writing these loop bounds by hand could be annoying and for complicated loops also error prone. Another disadvantage of hand-written user annotations is that they may become invalid if the source code has been changed. Because these changes could also influence the number of iterations and therefore these previous assertions has to be recalculated and updated within the code. Meanwhile, several work focuses on methods to calculate the loop bounds automatically. The approach followed within this thesis is to calculate the loop bound directly at the presentation level of the source code, while most other analysers performs their analysis at low level representations like assembler. The implementation of this analyser is based on a publication of Healy, Sjödin, Rustagi, Whalley and v. Engelen in [HSR+00]. The main difference lies within the level of representation. While this work directly uses a C language file as input, the original work was targeted to analyse an *assembly* source program. An advantage of this approach is that the result can directly inserted into the original source file. Loops that cannot calculated by the algorithm can be bound by hand afterwards. Figure 1.1 on page 3 shows a structural representation of the working method of the actual implementation. As input file will be taken a C source file. This file could also contain user insertions for variable value ranges that cannot be calculated automatically. As described within the following sections, these annotation are not needed for every unknown variable, but could help to calculate the result more accurately. The given input file is handed over to the *preprocessor* to replace all constants and defines. After the preprocessing, all possible flow facts will be calculated and stored within the internal representation structure. Following, the loop bound for all supported loops is calculated. The calculated result are inserted in a compatible style to the language *WCETC* (described in [Kir02]) within the

[3]**B**est-**C**ase **E**xecution **T**ime
[4]**W**orst-**C**ase **E**xecution **T**ime

source file. This modified source file can be handed over to any *WCETC* compatible *WCET* analyser for further calculation.

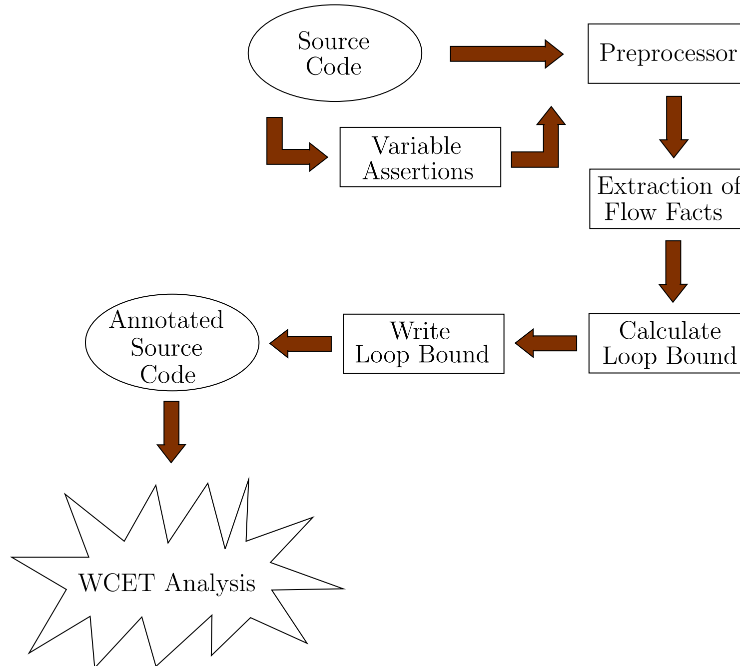


Figure 1.1: Structural representation of the working method of the analyser

1.2 Contribution

Within this work, the theoretical method behind and some concrete implementation details of a loop analyser for further *WCET* analysis are described. In contrast to several other related works the described analysis is performed directly at C source files. Additionally to the basic algorithm, which has been described within [HSR+00], this thesis outlines:

- How to parse and represent the information of the input source file for further analysis. And also how to reduce the gained information without influence of the calculation result.
- A method to create a control flow graph out of the previous parsed source file. Extraction of *flow facts* by investigation of the 'C' language semantic.
- Some additional expansions that must be performed to analyse a program at C source level are introduced. Also some transformations of possible 'C' language constructs that not fulfil the required structure are explained.

- An algorithm to extract the needed information for the following loop bound calculation is explained in detail. Last, the preparation of the calculation results using the language *WCETC* is shown.

1.3 Outlook

The content of this thesis is structured as follows:

Chapter 2 gives a detailed overview over the main topics of the current research area of *WCET* analysis. It outlines some vocabulary of the *WCET* field and describes different actual approaches to overcome and solve existing problems within this area.

Chapter 3 summarises some related work within the area of *WCET* analysis. It contains not only work descriptions about loop iteration bounding directly but also descriptions about the entire *WCET* analyses.

Chapter 4 describes in detail the theoretical background of the method used within this work. It concentrates on the calculation of loop bounds. Only when necessary for better understanding, additional implementation details of the framework has been included.

Chapter 5 shows some of important implementation details of the actual framework realisation. It outlines some principal details of the lexical and syntactical analysis using the tools *FLEX* and *YACC*. Also how the control flow graph is calculated and of course the implementation of the theoretical background, described within chapter 4.

Chapter 6 gives some examples of loops that could be successfully bounded by the algorithm. Also some types of loops that could not be successfully bounded are listed. Additionally there is a list of some existing restrictions for expressions within the input code that cannot handled by the current implementation and can lead to incorrect results. At least some calculation results of loops are given to show how this tool implementation works.

Chapter 7 gives a short summary of the possibilities of the algorithm and also the pros and cons of the approach. Further, a list of some future work that has not realised within the current implementation, is given.

Chapter 2

WCET Analysis

Within this chapter, the *WCET* analyses in general is described. This general description also outlines the most important terms that are used within this research area. Afterwards, the course of *WCET* measurement and of a static *WCET* analysis is described. The loop bound analysis described in chapter 4 is also based on statical analysis.

2.1 Introduction into WCET Analysis

Computer systems often interact with their environment. This interaction may be processing measurement information from an external sensor and control an external actuator depending on this processed information. If these course of events will be periodically repeated, there is only limited time from getting the sensor value to change the actuator position. The time, when these course of events have to be finished, is called deadline. Such computer systems that must fulfil timing requirements are called *real-time systems*. Figure 2.1 on page 6 depicts a rough possible structure of such a system. The sensor on the left side will periodically transmit measurements of physical values. This physical value could be the temperature of the cooling water of a car. The actuator on the right hand side could be a valve that controls the water circulation depending on the temperature of the water. Then, the information system between these two parts gets the actual temperature and calculates from this value the next position of the valve to stay within a fixed temperature range. Every mark within the continuous timeline where the sensor takes a new measurement is called observation time ($\hat{t}_{observation}$). The timeline mark when the valve reaches its new position is called reaction time ($\hat{t}_{reaction}$). The passed time between the measurement and the reaching of the new position ($\hat{t}_{reaction}$-$\hat{t}_{observation}$) is called response time ($t_{response}$). Within the field of *real-time systems* this *response time* has to be bounded to a certain limit. Because the transmission of a car may be damaged if it is not running within a proper temperature range. In case that the *information processing system* has not only to control the position of the valve

but also to control some other actuators, the system must decide which task[1] will be processed first. The decision of which task will be processed first, is taken by a *scheduler*. There exist several realisations of such a scheduler. One of the simplest would be, to process that task first, that has the nearest deadline in the future. To guarantee that all tasks within a *real-time system* will be meet their deadline, the execution time of each task must be known. The maximum of the execution time is called ***w**orst **c**ase **e**xecution **t**ime* (WCET). Similary, the minimum of the execution time is called ***b**est **c**ase **e**xecution **t**ime* (BCET). Using the *WCET*, it can be verified whether all tasks of a real time system meet its deadline. In contrast, the *BCET* can be used to prove that not every deadline can be met. But also to guarantee that the system does not respond faster than expected. Depending on the consequence of missed deadlines the domain of *real time systems* is split up into two subdomains.

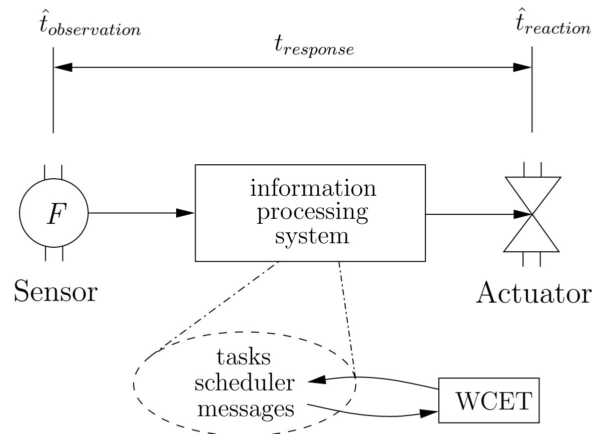


Figure 2.1: Structural representation of a real time system (from [Kir03])

If a missed deadline leads to a critical failure of the whole system it is called a *hard* real time system. Otherwise if a missed deadline only leads to a reduced service quality of the system, it is called a *soft* real time system. Therefore, *soft* real time systems can be said to follow the *best effort strategy*. This means that the task scheduler will order the tasks such that the number of deadline misses is low. In case of a *hard* real time system, the reaching of all deadlines must be guaranteed.

This explains why the knowledge of the temporal behaviour of the tasks is fundamental within the design process of a *real time system*. The temporal behaviour of a task is given by its deadline and by its *execution time*. Figure 2.2 on page 7 shows the execution time distribution of a task. The probability of the *WCET* is in general small compared to the average execution time. But it is most important for the behaviour of a *hard real time system*. Because within a *hard* real time system it

[1]Within this thesis always the synonym *task* will be used. This synonym will be also include the execution of a whole software. Because the thesis itself deals not with internal software architectures.

is important that the tasks of a system meet their deadline under all circumstances. If the consequences of a missed deadline are catastrophic it is not sufficient that in general all deadlines are reached. The execution time of a task is normally near the average time. But the real execution time 'ET_{Actual}' depends on several facts.

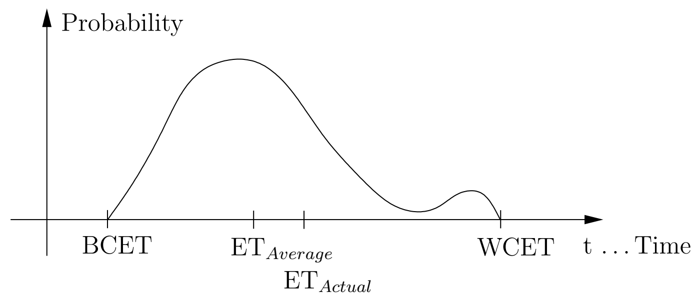


Figure 2.2: Plot of the execution time probability of a task within a *WCET* system

In many cases, the input data of a task do influence its execution time. If not all input values are within the defined ranges of the specification, the system must include some mechanism of *error handling*. But *error handling* will not be within the scope of this work. Therefore, interested reader are referred to additional literature. A nice summary of *real-time systems* could be found in [Kop97].

Another reason for different execution times comes from internal processor states. Processors of early generations are of much more simpler architectures than complex processors today. Therefore, it was possible to determine the execution time of single commands more precisely. Within modern processors many additional performance-increasing technologies are used. Such technologies lead to a larger amount of possible internal processor states. It is obviously that additional internal states in the same way increase the complexity of analysis. This complexity could be reduced by several approximations. But the main drawback is that that they often lead to imprecise *WCET* analysis results. Thus, a system may be classified to not fulfil the requirements but in reality it does. The following list summarises some technologies that are used within modern processors and possibly influence the *execution time*.

- **Pipelining:** The instruction processing is split up into several stages like *FETCH*, *DECODE*, *EXECUTE* and *WRITE BACK*. This makes parallelism possible by overlapping instruction execution. If this overlapping execution could not be included within the analysis, the calculated *WCET* result could be significant higher as it would be within reality.

- **Caching:** Memory is often structured by hierarchy and builds a compromise between hardware costs and access times. Mass storage hardware like hard disks can be produced cheap but needs long access times. Therefore, processors

often use a so-called *cache* memory with faster access times. Used data will be intermediately stored within this cache memory. Before new data will be loaded from external memory it will be looked if this data has been stored within the cache. If they have been previously stored, they will be loaded from the cache; otherwise they would be loaded from the external memory. The difficulties of *WCET* analysis of processors that use cache memory are:

1. Caches can reduce the execution time significant by intermediate storage of previously used data.
2. But caches can also reduce the execution time if every cache access is a *cache miss*.
3. Ignorance of cache memory leads often to pessimistic analysis results by not including the possible speed gain.

Another problem within the analysis of the temporal behaviour of hardware can lie within its documentation. Processors are not always designed for the use within *real time systems*. Therefore, companies do not or only roughly analyse the temporal behaviour of their products and the available documentation is lacking of such detailed information. If some information must be estimated for the execution time analysis, it has to be save. This possible overestimation will additionally lead to pessimistic analysis results. On the other hand, for a special group of processors it is impossible to predetermine the timing behaviour by the manufacturer. For *configurable processors* an extra application-specific firmware could be written by application developers. The final instruction timing depends also on this firmware and could therefore not be evaluated by the hardware manufacturer.

2.2 *WCET* analysis by measurement

Compared to statical *WCET* analysis, the approach of determining the *WCET* by measurement has a long tradition. But for the real *WCET* value, all possible execution scenarios evaluated must be evaluated. For the complex structures of modern computer systems this will become infeasible. Increasing hardware improvements have been introduced to reduce the average execution time. The idea behind is to create a system that follows the *best effort* strategy. This strategy let the *WCET* analysis to become more and more complex. Because the possible internal states of such modern computer system will be increase enormously. By extracting representative input values, it could be tried to find execution traces of input dependant statements within the source code that may lead to *WCET* scenarios. The arduous work of extracting proper input values could be done manually. But this could also still be solved by heuristic algorithms. Within section 2.2.1 on page 9 the possibility of evolutionary testing is described. This method iteratively produces input values while the execution time in the optimal case converges to the real *WCET*. Another problem within the measurement based approach is the so-called *probe effect*. Additional *break points*, *input-* or *output values* influence the execution time. Figure

2.3 shows a possible realisation of a test-environment for measurement-based timing analysis.

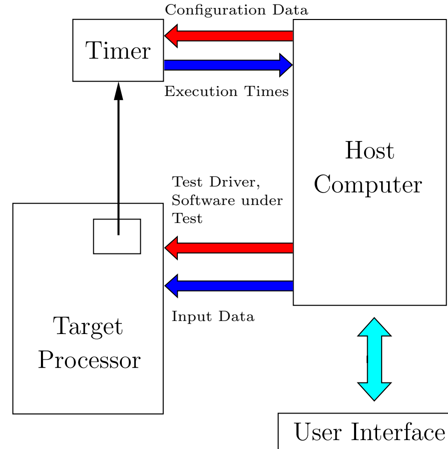


Figure 2.3: Test set-up for *WCET* analysis by measurement (from [PN98])

Within this test-environment, the *WCET* of the *target processor* should be evaluated. To minimize the possibility of *probe effect* occurrences, all calculations of input values and time measurements will be done outside of the target processor. The initialisation and the disposal of input values will be done by the *host computer*. After the execution of the *real-time* program has been started on the target computer, the external timer is triggered to start the measurement. The measurement stops when the calculated output values are visible at the output ports of the target computer. Although there is no guarantee that the longest measured execution time is near the real *WCET*, execution time measurement is still an option. Both, static *WCET* analyses and execution time measurement can be used to control the calculated result of each other. Comparing the results of each methodology we got the following equation:

$$\text{Measured_}WCET \leq \text{Real_}WCET \leq \text{Statically_analysed_}WCET \tag{2.1}$$

In common, static analysis is used during the development of real time systems. After the development process, measurement based analysis can be used to control if there was no fundamental error within static analysis and the system will fulfil its requirements.

2.2.1 Evolutionary testing

As described previously, execution time measurements covering all possible execution scenarios will be nearly infeasible. Hence, it must be searched for input data that lead to a execution time near to the real *WCET*. Within the area of *evolutionary testing*, it will be tried to find input data that lead to a maximal execution time,

using heuristic methods. Figure 2.4 on page 10 shows a diagram of the working course of evolutionary testing. It can be realised with *genetic algorithms*. This algorithm and its terms have been overtaken from the biological research. *Generic algorithm* consists of:

- **Genes:** Represents the smallest factor of the input data. For example, this will be some integer constants.
- **Chromosomes:** Unions from several genes to units. A *chromosome* could be a matrix that is part from a matrix multiplication.
- **Individuals:** Consists of a number of chromosomes and represents a whole input data set. For each individual, a *fitness value* will be calculated. This value will be used to judge each individual and will be calculated by a problem specific value function.
- **Population:** Several individuals are grouped into a population. The number of individuals depends on the complexity of the problem.

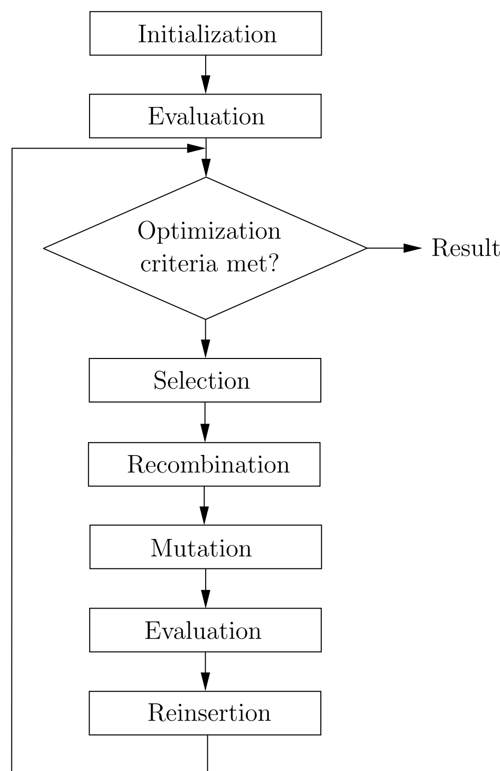


Figure 2.4: Graphical representation of evolutionary testing

The terms described above are be used to outline the method behind evolutionary testing. The steps of evolutionary testing within the scheme of figure 2.4 can be described as follows:

1. **Initialisation:** The first population will be initialised manually or with random values.
2. **Evaluation:** Within this step, the fitness value for each individual will be calculated. Depending on *WCET* analysis, this *fitness value* is the execution time of a run with the actual individual as input value. Within a test setup, as described in figure 2.3, the *fitness value* is generated by the external timer.
3. **Optimization criteria:** Determines if the exit condition that depends on the calculated fitness values is fulfilled. The *exit condition* depends the individual implementation. A possible approach is to stop the evaluation after a previous fixed number of runs without getting a new *WCET* value.
4. **Selection:** Depending on their fitness values, some individuals will be selected and recombined. Selecting only individuals that have lead to high *WCET* values will possibly resulting into local optima. To overcome this problem various selection algorithms have been developed. One possible approach is *simulated annealing*[2] where not always individuals with high fitness values are selected.
5. **Recombination:** To increase the measured *WCET* values, new individuals are generated. This generation will be done as shown on figure 2.5. The parent individuals are selected as previously described.

   Some genes of the parents will be mixed up by chance. This will be lead into new individuals with possible the same or a higher *WCET*. The recombination could also be repeated with the previous generated individuals. It could also included to the selection algorithm that the number of a possible generation deepness is fixed.
6. **Mutation:** Each bit within a *gene* are reversed by a, typically fixed, probability. New individuals not only depend on older generations, but also will be generated independently. This measures could also help to escape local optima.
7. **Reinsertion:** The number of individuals within a population is restricted. Therefore, the fitness value (*WCET* value) of all new individuals is ascertained. Individuals with low fitness values are removed from the population to get the

[2]this algorithm is adopted from metal processing. Glowing metal can be formed easier. Continually annealing metal will become hard to form. This means that at the beginning the probability of selecting an individual with lower fitness value is higher. At the end, individuals with high fitness values will be selected more likely. This could possibly avoid that the search will end at local optima.

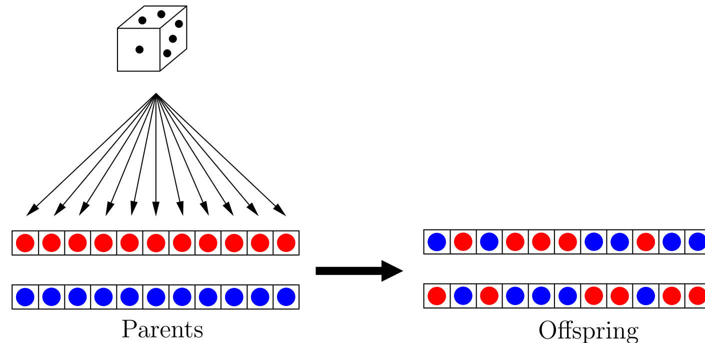


Figure 2.5: Recombination of parent individuals to get a new generation of individuals

same number of individuals within the population as at the beginning. This procedure is conform to *Darwins*[3] *laws* on evolution.

Additional information about *genetic algorithm* can also be found in [PN98].

2.3 Static WCET Analysis

As previously described, static *WCET* analysis is used to calculate save upper bounds of the *WCET*. It will be tried to extract the needed information for further calculation out from the source code. But unfortunately, not all information can be directly extracted from the semantic analysis of a program. To get the information needed for *WCET* analysis that cannot extracted from the language semantic, a programmer may insert some additional information (flow facts) by annotations. The following list describes three different methods to extract and collect *flow facts* of a program code:

- **Abstract interpretation:** The complexity of the *WCET* analysis could be decreased by performing some kind of abstraction during the extraction of flow facts. One kind of abstraction could be to perform all calculations using value ranges instead of perform the calculations for each concrete value separately.
- **Simplified methods:** This approach uses annotations within the source code to simplify the calculation process. Providing an upper and a lower bound for value ranges or for the number of loop iterations leads to less complex and maybe more exact analyses.
- **Symbolic computation:** The *flow facts* are calculated by solving algebraic equations in symbolic form. The computation complexity of *symbolic execution*

[3] Charles Darwin was born on 12.02.1809 in Shrewsbury, England and † on 9.04.1882. Within the book *On the Origin of Species by Means of Natural Selection* he developed his famous theory about evolution.

lies between the complexity of *abstract interpretation* and that of *simplified methods*.

2.3.1 Calculation of the *WCET*

The *WCET* will be calculated by following the longest path within the control flow graph of the analysed program and summing up the execution time of each single instruction. As described by existing research, this path will be calculated by following one of the following approaches:

- **Tree-based calculation:** The calculation of the *WCET* will be done in a hierarchical manner. A tree, corresponding to the *syntactical parse tree* of the target program, will be evaluated in a *bottom up* manner. For each type of compound statement, like *loops* or *IF* statements, rules will be used to determine the *WCET* at each level of the tree. Such tree-based calculation techniques in general are simple and fast, but are also limited with regard to path specifications.

- **Path-based calculation:** The *WCET* is calculated by evaluation of all possible paths through a program. To evaluate all paths, the program will be split up into several scopes. A possible scope would be build by a loop. Each scope will be investigated hierarchically to find the longest path out of all possible paths.

- **Implicit path enumeration technique (IPET):** This method is also based on the *control flow graph*. It translates the *control flow graph* systematically into a set of constraints. One possible realisation of *IPET* could be *I*nteger *L*inear *P*rogramming (ILP), which is described in section 2.3.2.

2.3.2 Integer Liner Programming

Instead of tracing the execution time of all possible paths and determine the maximum out of them, several methods of *Integer Linear Programming* could be used. Figure 2.6 depicts a control flow graph for a small C example code. This example graph will be used in the following lines to explain the main idea behind this analysis method. The basic blocks have been named from B_1 until B_7.

A variable x_i for each basic block B_i gives the maximal number of executions. Using the control flow graph in figure 2.6, the *flow equations* given in table 2.1 on page 14 can be derived. These equations builds structural constraints for the control flow graph. On the one hand, the number of all incoming edges e_m of a basic block must be the same as the number as all outgoing edges e_n. On the other hand, this number must be the same as the maximal number of executions for the basic block itself. All symbols that have been used in equation 2.1 are summarized in table 2.2.

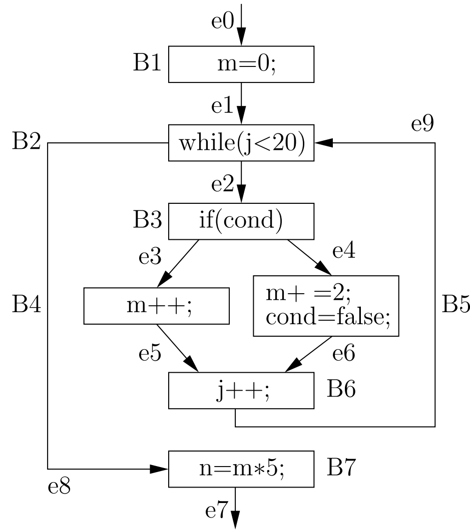


Figure 2.6: Annotated control flow graph for flow facts derivation [Kai05]

$e_0 = e_1 = x_1$	$e_1 + e_9 = e_2 + e_8 = x_2$
$e_2 = e_3 + e_4 = x_3$	$e_3 = e_5 = x_4$
$e_4 = e_6 = x_5$	$e_5 + e_6 = e_9 = x_6$
$e_8 = e_7 = x_7$	

Table 2.1: Derived *Flow Equations* from figure 2.6

Additional to the previous described *structural constraints* we can use *functional constraints*. Such functional constraints can result from the context of a program. For example, x_5 (the maximal number of executions of basic block $B5$) can be bounded with '≤ 1', as the condition of basic block $B3$ would be true maximal once. Such functional constraints can be also inserted as comment within the source code by the programmer. As a result of a value restriction for the variable j to values '≥ 0' by the programmer, the number of executions of basic block $B3$ can be limited to '$x_3 \leq 20 * x_1$' by the analyser.

All these defined or derived constraints lead to the *I*nteger *L*inear *P*rogramming - equation 2.2 for the *WCET*. The *WCET* is the maximum sum of all statement execution times, dependent on the number of executions for each statement and all functional constraints. A piece of code that should be analysed must be natural, which is expressed for the example in figure 2.6 by $e_1 = 1$ and means that there is only one entry point. At least, the number of all incoming edges must be the same

as the number of all outgoing edges.

$$\begin{aligned} \text{WCET} = max\Big(& \sum c_i * x_i \mid \\ & \mid e_1 = 1 \wedge \Big(\forall\, i \in \{1, \ldots, N\} \textstyle\sum_{e_m \in E_{m,i}} e_m = \sum_{e_n \in E_{n,i}} e_n = x_i\Big) \wedge \\ & \wedge \text{ functional constraints } \Big) \end{aligned} \tag{2.2}$$

B_i ... Basic Block number i, $i \in [1 \ldots N]$
N ... Number of Basic Blocks
c_i ... Execution time of Basic Block B_i
x_i ... Maximal number of executions of Basic Block B_i
$E_{m,i}$ ... Set of incoming edges of Basic Block B_i
$E_{n,i}$ ... Set of outgoing edges of Basic Block B_i

Table 2.2: Explanation of used symbols within section Integer Linear Programming

2.4 Hybrid Analysis

Alternative to the previous described calculation techniques, research time is also spent on possible combinations, of both, measurement and static techniques. To obtain a reliable *WCET* result, the code is split into blocks. The execution time of these blocks is estimated by measurements. To avoid overestimation within the analysis, this blocks should be coarser than basic blocks. The entire *WCET* will be calculated by putting the measurement results for each separate block together. Compared to other pure analytical methods, it needs less effort to reuse this analysis method for different hardware technologies.

Chapter 3

Related work

This chapter introduces different approaches to calculate or process *flow facts* within the area of *WCET* analysis. The main focus is pointed at automatic *flow facts* derivation without user support. Each of the introduced algorithms is summarised to show the idea behind. More interested readers are referred to the corresponding bibliography entries at the end of this thesis.

3.1 Flow Facts calculation

Flow Facts are used to describe all possible paths through a computer program. The paths itself could be made visible within a *control flow graph*. But *flow facts* are more powerful. They can also describe the behaviour, like when each path and how often a path will be taken. This information will be used for several purposes; for example, to interpret a source file or to calculate the execution time of a computer program. The aim of this thesis is to support the calculation of the worst case execution time (*WCET*). Some of this *flow facts* can be directly extracted from the source code. Others can not be directly extracted or it is too complicated to extract them from the semantics of the source code. Instead, this information could be also inserted by manual annotations. An often used annotation within *WCET* analysis is the number of iterations of a loop. Apart from the method used within this thesis, there exists several different approaches to calculate this loop bound automatically.

3.1.1 Flow Analysis by using Abstract Interpretation

3.1.1.1 Flow Analysis of Object-Oriented Real-Time Programs

Jan Gustafsson and Andreas Ermedahl presents in [GE98] a method for automatic derivations of path and loop annotations in object-oriented real-time programs. Within this paper, the idea is shown on the programming language *Smalltalk*. The analysis is done by using *abstract interpretation*. Every time in the program where the value of a variable is of interest, a control point is set. To every control point,

one ore more environments "σ_i", which hold the possible values of the variables at this control point, are associated. These environments are created by *semantic functions*, which are subdivided into functions for *statements* and functions for *condition rules.*

$$\sigma_y = S[expression]\sigma_x \quad \ldots \quad \text{for statements}$$
$$\sigma_y = C[condition]\sigma_x \quad \ldots \quad \text{for conditions}$$

Where σ_x is the input environment and σ_y is the new created output environment. The analysis is performed within two environments for two alternative paths in an *ifTrue*-construct. The idea behind the analysis of loops is to transform loops recursively into *ifTrue*-constructs.

$$\text{S[[C] whileTrue: S]}\sigma \quad = \quad \text{S[[C] ifTrue: [S [C] whileTrue: S]]}\sigma$$

A loop is *rolled out* until it terminates. The *ifTrue*-environment creates two new environments until the loop terminates. This leads to a heavy increase of the number of environments. Therefore, environments could be merged at different control points. Which means that the ranges of each variable within the environments are merged. The loop iteration bounds are determined using the value ranges within the environments of the loop headers. Merging of environments avoids explosion of the number of environments but could also lead to overestimation of the loop bound. To avoid non-terminating of the analysis in case of unbounded loops, *timing budgets* have been introduced. This timing budget must be a realistic upper bound of the execution time and is the only required manually annotation within the source code. The actual execution time is calculated during the analysis and continuously compared to the timing budget. If this budget is exceeded the analysis is stopped with an error message.

Characterization:

+ Termination of the program is guaranteed by using *timing budgets.*
+ Stopping criterion in the case of unbounded execution time
+ Wrong timing budgets leads in the worst case only to previous ending of the analysis.

– Exponential increase of environments without environment merging.

– Merging of environments leads to overestimation.

– Without environment merging, all possible paths must be evaluated.

3.1.1.2 Flow Analysis of C-programs

A method to calculate *flow facts* automatically without the need for the programmer to insert manual annotations for the loop bounds is introduced in [GLSB03] by Jan Gustafsson, Björn Lisper, Christer Sandberg and Nerina Bermudo. The authors describe the algorithm behind by explaining their developed analysis tool. The *input* of their tool will be an intermediate code, originated from a *C* program. It produces as *output* a set of *flow facts* that describe and constrain the possible flow within the program. After optimising and production of an *internal representation* of the code, the analysis itself can be summarised by the following steps:

1. *Static Single Assignment Construction (SSA):* SSA is a form of *data flow description* and is used to store and also to simplify the program for further analysis.

2. *Removal of Non-Conditionals:* Within this step, all statements that do not influence the program flow are removed. This is used to reduce the data for the next analysis step.

3. *Scope Graph Construction:* Transforms the *control flow graph* (CFG) into a *scope graph.* This *scope* format is used to support unstructured code and recursion.

4. *Syntactical Analysis:* Tries to express the number of iterations of simple loops as recurrence equations. If these equation can be expressed within a closed form then the number of iterations could be directly get without further analysis.

5. *Abstract interpretation:* All loops that could not successfully bounded within the syntactical analysis will be bounded using *abstract interpretation.* This means that abstract values (value intervals) instead of real values are used during the calculation.

Characterization:

+ Works theoretically for all loop constructs of the supported language.

+ Needs no further annotation within the source code.

+ Removes unnecessary information before analysing

+ Efficiency gain by means of combination of *syntactical analysis* and *abstract interpretation.*

− High implementation effort used for abstract interpretation.

− Requires simulation of all paths within every iteration.

3.1.2 Detection and Exploitation of Value-Dependant Constraints

In [HW99], Christopher Healy and David Whalley describe a method to bind the number of loop iterations by using *effect-based* and *iteration-based* constraints. These constraints are used to predict the control flow within the execution of a program.

Effect-Based Constraints: First, all variables and registers that determine the direction of a conditional branches within the control flow graph are identified. It tries to predict the successors of a node within the control flow graph at a certain point. To indicate the behaviour of a conditional branch, three different states are distinguished:

1. *Unknown:* If a value of a condition is unknown, no statement about the behaviour can be made.
2. *Falltrough:* If the behaviour of the conditional branch can be determined and the target of the branch is the sequential successor.
3. *Jump:* If the behaviour of the conditional branch can be determined and the target of the branch is *not* the sequential successor.

All assignments to variables, that are used within branches, are marked with a set of states. For example, the assignment '`a=0;`' results in the condition '`if(a>0)`' within a *basic block* with the index number *4* to *jump* to its successor and therefore marked with **4J**. Additionally, the behaviour of a branch could be related to the behaviour of other branches. If a condition '`if(a>0)`' leads to a *fall through*, an other condition '`if(a==0)`' always leads to *jump*. The behaviour of a branch within a loop will remain unchanged until the value of variable will be changed.

Iteration-Based Constraints: A *basic induction variable* is a variable that is incremented or decremented by a constant value within every loop iteration. If this variable is compared to a constant, it is possible for the algorithm to calculate the range when a branch will jump or fall through. Otherwise, if such a variable is compared to a loop invariant non-constant with a relational operator '`!=`' or '`==`', it could be determined that this condition will be true once, even if the value of the non-constant will be unknown. This calculated range information will be stored for each edge within the control flow graph.

This derived information will be used within the analyser to determine the minimum and maximum number of iterations for each loop. A list with all calculated constraints for each path trough a loop is stored. As a path through a loop we understand a path from the first *basic block*[1] within the loop body back to the loop header. A value constraint is valid within a path until new constraints are encountered. A new found effect-based constraint nullifies the previous constraint. Finally, if such a constraint is still valid at the end of a path, it will be propagated

[1] A sequence of sequential statements with only one entry point and only one exit point

to all other paths within the loop. After this propagation is completed, a *can follow* matrix will be created to see which path can follow another path. This matrix, in combination with the previous calculated edge ranges, will be used to determine the longest possible path through a loop. The longest found path will be the foundation of the following *WCET* analysis.

Characterization:

+ Also applicable for nested loops
+ Analysis possible without value assertions
− Tighter estimations will still require value assertions
− Increasing memory consumption for storage of intermediate results
− Hard to verify whether the longest path was found

3.2 Using Flow Facts for WCET Analysis

Section 3.1 had outlined different approaches to derive *flow facts* from a given source code. The aim of this section is to show how these derived *flow facts* can be used to calculate the **W**orst **C**ase **E**exution **Time** (WCET).

3.2.1 Tree-Based WCET Analysis on Instrumentation Point Graphs

As described in section 2.3.1 on page 13, three typical methods are available to calculate the *WCET* of a program. Adam Betts and Guillem Bernat introduces in [BB06] a tree-based hybrid[2] analysis method for *WCET* calculation. It combines low-level measurement-based and high-level static analysis techniques to reduce possible underestimation and overestimation, as it likely happens when these techniques are used each separately.

The basis for this analysis method is the *control flow graph* (see section 4.1 on page 23 for further details) of the input program. This *control flow graph* will be expanded with *instrumentation points* (ipoints $\in \hat{\mathcal{I}}$), which represent individual time stamp instructions. This placement of ipoints is a fundamental issue when measurement based and static techniques are combined. Because it determines the precision and resolution of further analysis. The *ipoints* can be placed at arbitrary locations, therefore each basic blocks can be split up into a set $\hat{\mathcal{B}}$ of *sub basic blocks*. The resulting flow graph is called **I**nstrumentation **P**oint **G**raph (IPG) and is defined as:

Definition 1. *An **IPG** is a connected directed graph* $\Gamma(\hat{\mathcal{I}}, E', s, e)$ *such that* $(i, j) \in E'$ *if and only if there exists a path* $i \rightarrow b_1 \rightarrow \ldots \rightarrow b_n \rightarrow j$ *such that each* $b_i \in \hat{\mathcal{B}}$

[2] a combination of statical and measurement-based *WCET* analysis

and $n \geq 0$.

with $\hat{\mathcal{B}}$ ... set of *ipoints*, E' ... set of directed edges, s ... start node, e ... end node;

To minimize the *probe effect*[3], the number of *ipoints* will be reduced and several *sub basic blocks* combined to *instruction blocks* (IB). Afterwards, the IPG is decomposed hierarchical into an *Itree*, which is used to calculate the *WCET* within the following analysis. The structure of the *Itree* consists of four different nodes:

- An *alternative* node is a rooted n-ary tree that models the selection of paths.
- A *sequence* node is a rooted n-ary tree that models a non-empty path to a post-dominator node.
- A *loop* node is a rooted binary tree that models all paths between header and tail.
- A *meta-loop* is a rooted n-ary tree that models several sub loops. The subloops have different tails but the same loop header and shares a common path p.

The previous described *Itree* will be used to compute *WCET* estimates by applying several rules. The following two equations show how the *WCET* for *loop* structures and *meta-loop* structures is calculated:

$$WCET(loop) = ((WCET(body) + WCET(iteration\ \ edge)) * k \tag{3.1}$$

$$WCET(meta) = \sum_{i=1}^{n} WCET(loop_i) + WCET(p) * \sum_{i=1}^{n} k_i \tag{3.2}$$

In equation (3.1), k denotes the upper bound of the analysed loop. These loop bound will be calculated with methods, described within section 3.1. Methods to bound the number of iterations are not within the scope of the actual described related work. In equation (3.2), the *WCET* of a *meta-loop* is the sum of all sub loops plus the *WCET* of any path p common to all sub loops. Path p is always executed before any sub loop is encountered and therefore the *WCET* of path p is factored by the upper bound of every sub loop. The *WCET* of a single instruction block is determined by measurement.

Characterization:

+ Compared to non-hybrid methods, reduced underestimation and overestimation of the *WCET*.

[3] the collection of timing data at the *ipoints* affect the execution time of program

+ This work presents an algorithm in a pseudo language for each step within the tree modifications.

− Additional effort for the creation of the instrumentation tree is needed. This special three has to be created out of the standard control flow graph.

− This work does not discuss the calculation of all flow facts in detail.

Chapter 4

Loop Bound Analysis

This chapter describes the theoretical background used to analyse a source file of the procedural programming language 'C' and to calculate the upper and lower bound of loop iterations. The concept of how to find the bounding iterations for loops in SPARC assembler files was introduced in [HSR+00] and [HSW98]. Within this thesis, the methods are adapted to analyse files of the high level language C directly without translation into assembler files. Because translation to assembler files differs sometimes depending on the used compiler and requires additional transformations of annotations to the assembler file and vica versa. This work is also compatible to the programming language WCETC introduced in [Kir02].

4.1 General definitions for the Control flow graph

Expression consists of constants (e.g. "12"), identifiers (e.g. "var_1") and operators (e.g. "+" or "-"). A valid expression would be "var_1 = 1", which assigns to the identifier "var_1" the constant value "1".

Statement consists in the simplest case of only one statement, followed by a semicolon. Several statements, surrounded by a beginning "{" and a completing "}"-bracket, are combined to a *block statement*. A "{"-bracket increases the depth level of a statement and a "}"-bracket decreases it. But there also exists more complex statements like loops and branches. The analysis of such statements is the main focus of this thesis.

Node = Basic Block consists of a list of consecutive statements and has a single entry point at the beginning and a single exit point at the end. All statements inside a node have the same depth level. If a statement consists of a branch (e.g. an *IF*-statement) or the next statement has a different depth level, a node ends with the actual statement.

Predecessor Every node has zero or more predecessor-nodes. Several nodes build a *tree* (=called *Control Flow Graph*), where the nodes are connected with

directed *edges*. A way from one node to another is called a *path*. A node i is called a predecessor of a node j, if they are connected with a directed edge and node j follows node i within the tree.

Successor In the same way like the predecessor, a node has zero or more successor nodes.

Dominator A *node* or *basic block* i dominates another node j if there is *no* path from the tree head to the node j which contains *not* node i. For instance, the header of a *natural* loop[1] predominates all nodes inside the loop.

Postdominator A *node* or *basic block* i postdominates another node j if there is *no* path from node i to the exit of the Control Flow Graph which contains *not* node j.

(A node always dominates and postdominates itself)

Functions A C-program consists of a number of *functions*. A function consists of a function header and a function body. The header consists of the function-*name*, the arguments (=the input values of a function) and the type of the return value. The body of a function consists of variable declaration list and a list of statements, which forms nodes and a whole tree within the *control flow graph*.

4.2 Loops with multiple exits

To calculate the number of loop iterations of a natural loop the steps of the following list must be performed. Within this section each step is explained using example C-codes like in figure 4.2 on page 26.

1. Parse input C source files (see chapter 5.1.3 for further details)
2. Construction of the control flow graph (see also chapter 5.5 for further details)
3. Find all conditional branches inside the loop that can affect the number of loop iterations
4. Construct the branch tree with the loop header and all identified branches
5. Expand the branch tree if there are expressions with alternative internal control flow or also if there are *SWITCH-CASE*-statements
6. Determine the range of iterations when each of this identified branches could be reached
7. Calculate the minimum and maximum number of iterations for the loop

[1] *natural* means that the loop has only a single entry point

4.2.1 Construction of the control flow graph

After the source code is parsed in the usual way there are some specialities required for the analysis. A detailed description of the implementation of the construction of the control flow graph will be given in section 5.5. This section only outlines some highlights that are necessary for the following analysis. Figure 4.1 shows a short selection of possible expression statements of the language C.

```
void main( void ){
    int arr[20], i=1;      /* line 01 */
    arr[i++]=5;            /* line 02 */
    arr[i>0?--i:i]=6;      /* line 03 */
}
```

Figure 4.1: Sequence of valid statements in C

The corresponding *YACC*[2]-grammar of line 1 and 2 in figure 4.1 is:

```
postfix_expression '[' expression ']'
```

This means that the index of an array could be an arbitrary expression. If this expression contains an assignment or a conditional assignment (line 3 in figure 4.1) it must be extracted as separate statement.
The statement

```
arr[i>0?--i:i]=6;
```

will be changed to an *IF*-statement like

```
if(i>0){
    i=i-1;
}
arr[i]=6;
```

This is an additional branch which must be evaluated within the calculation (see section 4.2.2). Additionally, such expressions could influence the calculation of information (see also section 4.2.4). The main drawback of this approach is that it is not working for all possible expressions. Therefore, a number of restrictions for the input files have been formulated. These restrictions are listed and outlined in detail in chapter 6 on page 65.

[2]**Y**et **A**nother **C**ompiler **C**ompiler

```
void main( void ){
   extern int x;

   for(i=0, j=0; i < 200; i++, j+=2 ){
      if(i > 150){
         /* do something */
      }

      if((j > 160 && x) || (j > 450)){
         break;
      }
   }
}
```

Figure 4.2: Example loop with multiple exits

4.2.2 Finding all branches that can affect the number of loop iterations

Loop iteration Within this work, every time when it is spoken about *loop iterations*, the number of executions of the loop *header* is meant. Because if the condition of the loop header is always false, the loop header is executed once and the body of the loop is executed never.

Iteration Branch Is a branch inside a loop where the choice of which path in the Control Flow Graph will be taken, could directly of indirectly infect the number of loop iterations. The choice of which path will be taken depends also on a conditional expression. The structure of such an expression must be "variable relop[3] limit"; otherwise the branch is treated as *unknown.*

Unknown branches does not mean that the iteration bound of a loop could not be calculated. They only lead to maybe less accurate results with bigger ranges.

Back Edges An edge from a node inside a loop to the node, containing the loop header, is called *back edge.* Such a node is the last node within the loop or a node containing a *CONTINUE*-statement at the end.

A branch could directly affect the number of loop iterations if it has a successor S_x outside the loop, the successor S_x is the loop header itself or it is postdominated by the loop header. It could indirectly affect the number of iteration branches if

[3] relop . . . relational operator (e.g. '<' or '<=' or '>' or . . .)

each successor is postdominated by a different iteration branch. In figure 4.3 on page 27 an algorithm for finding such branches is presented (from [HSR+00]).

```
//Find the iteration branches that can directly affect the number of iterations
I = {}
FOR each block B in the loop L DO
    IF (B has two successors S1 and S2) THEN
        IF (S1 ∉ L) OR (S2 ∉ L) OR
           (S1 ∈ PostDom(Header(L))) OR (S2 ∈ PostDom(Header(L))) THEN
           I = I ∪ B
        END IF
    END IF
END FOR

//Find the iteration branches that can indirectly affect the number of iterations
DO
    FOR each block in B in the loop L DO
        IF (B has two successors S1 and S2) AND (B ∉ I) THEN
            IF (there exists J, K ∈ I AND J ≠ K AND
               S1 ∈ PostDom(J) AND S2 ∈ PostDom(K)) THEN
                I = I ∪ B
            END IF
        END IF
    END FOR
WHILE (any change to I)
```

Figure 4.3: Algorithm to find the set of Iteration Branches for a loop

$\mathbf{S_x} \in$ PostDom(Header(L)) Successor S_x is postdominated by the header of the loop L.

$\mathbf{S_x} \notin \mathbf{L}$ Successor S_x is located outside of the loop body. Such a successor could be a *BREAK*- or a *GOTO*-statement.

Now, the small C-function in figure 4.2 is used to show step by step how the loop bound is be calculated. First, it must be searched for all branches that had successors outside the loop and or that are postdominated by the loop header. Within the small example function, there is only the second *IF*-statement. If the condition is true, it has a *BREAK*-statement as successor, which is a jump to the first statement outside the loop. And if the condition is false, the closing brace of the loop is found as successor, which has a back edge to the loop header within the control flow

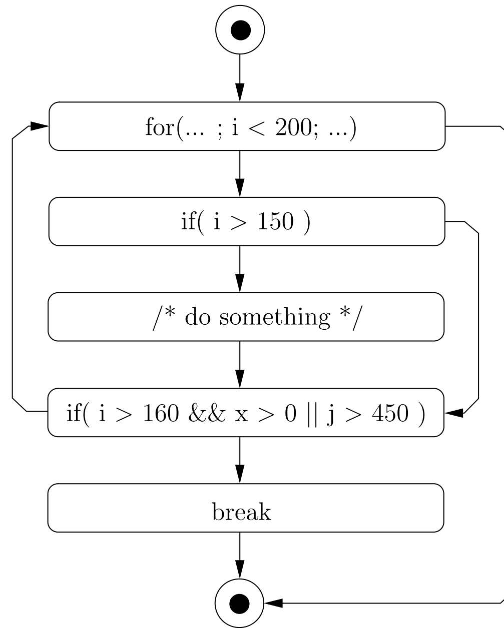


Figure 4.4: Control flow graph of the example in figure 4.2

graph (figure 4.4). The first *IF*-statement was not found because there is no direct successor outside of the loop and no successor, which is postdominated by the loop header. To store all found branches, a bitvector can be used and each bit within the bitvector represents a branch. But a branch is always the last statement of a node. Therefore the branch vector could be understand as node vector containing one bit for each node of the control flow graph (figure 4.4).

After identification of all branches the construction of the flow graph works in a similar way. For all branches that have a successor outside of the loop, a *BREAK*-node is set as successor. All other branches that are directly postdominated by the loop header (this means that there is no other branch between this and the loop header) get a *CONTINUE*-node as successor. Afterwards, the branch tree could be connected from top down. The loop header is also set as the root node of the branch tree. As successor will connected the first node, which is set in the branch vector and which is also dominated by the actual branch. For the example code of figure 4.2, the constructed branch tree is shown in figure 4.5. Note that the *IF*-branch contains a multiple condition. How to dissolve such branches into multiple branches with only a single condition of the form "`variable operator limit`" is the subject of section 4.2.3.

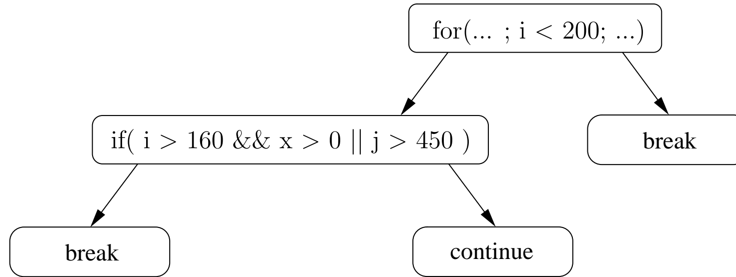
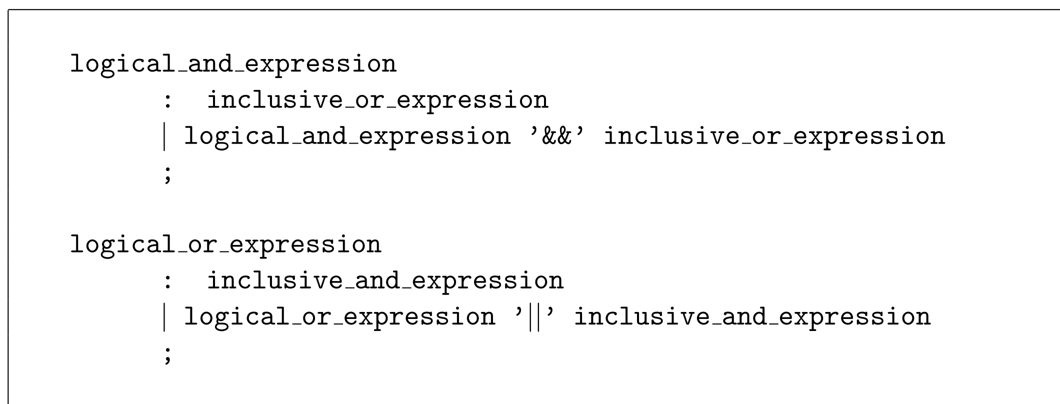


Figure 4.5: Branch tree with all identified branches



```
logical_and_expression
      :  inclusive_or_expression
      | logical_and_expression '&&' inclusive_or_expression
      ;

logical_or_expression
      :  inclusive_and_expression
      | logical_or_expression '||' inclusive_and_expression
      ;
```

Figure 4.6: YACC-grammar for logical '||' and '&&' expressions

4.2.3 Expansion of the branch tree

As mentioned before, if there are branches with multiple conditions they must be resolved into multiple branches with a single condition. Using the YACC[4] expression in figure 4.6 it could be constructed a tree out of the multiple condition.

For the testexample 4.2, the resulting tree is shown in figure 4.7. Now it is possible to expand the tree from top to down. A branch "`if( cond1 || cond2 )`" with the successor S_1 if the condition is true and successor S_2 otherwise could be expand to "`if( cond1 )`" with successor S_1 if the condition is true. If the condition is false the successor is a new inserted branch "`if( cond2 )`" with the original successors S_1 and S_2. For a branch "`if( cond1 && cond2 )`" with the successor S_1 if the condition is true and successor S_2 otherwise it is done in a similar way. The new branches are "`if( cond1 )`" with the successor S_2 if the condition is *false* and "`if( cond2 )`" with the original successors if the condition is *true*. The conditions *cond1* and *cond2* could be itself contain multiple conditions, so this tree could be

[4]**Y**et **A**nother **C**ompiler **C**ompiler

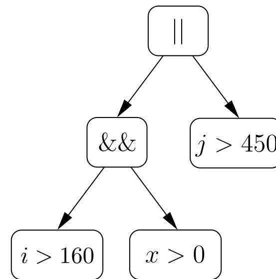


Figure 4.7: Condition tree for example in figure 4.2

expanded recursively without knowledge of the actual structure of cond1 and cond2. Using this method for the example in figure 4.2 with the corresponding branch tree in figure 4.5 will lead to the branch tree depicted in figure 4.10.

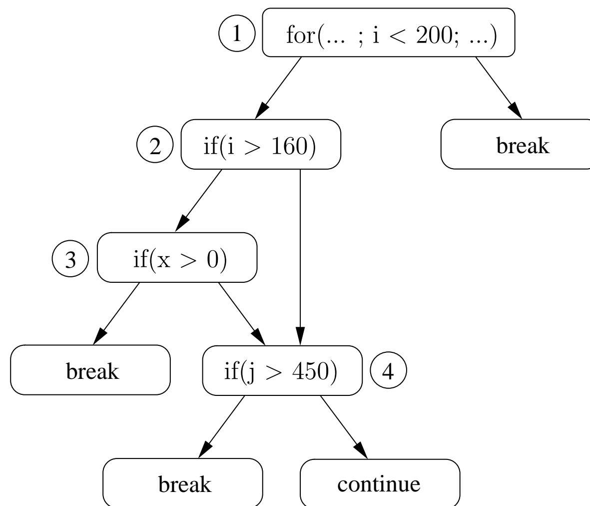


Figure 4.8: Branch tree after expanding

The next pieces of code within the high level language C that contain hidden *IF-ELSE*-branches inside are *SWITCH-CASE*-statement. As described within section 4.2.2, a *CASE*-statement could be found as branch that could affect the number of iterations, because it has a successor for each *CASE*-statement and another one for the *DEFAULT*-statement. This *CASE* statements are replaced by *IF* statements, where every following *CASE* statement is inside of the *ELSE* scope of the previous *CASE/IF* statements. Figure 4.9 shows for a small piece of code how this transformation works. This could also be done in the same way for a higher number of *CASE*-statements. After the transformation, the *SWITCH*-statement within the

branch tree is replaced with this sub-tree of *IF-ELSE*-statements. Additionally, the resulting *control flow graph* must be evaluated and the last basic block within the scope of a *CASE* statement, if it is not terminated with a *BREAK*, connected with the body of the following *CASE* statement (= fall-through of *CASE* statements).

`switch( x ){`
case 1:
...
break;
default:
...
break;
}
	if(x==1){
	...
	} else {
	...
	}

Figure 4.9: Transformation of *SWITCH-CASE*-statements

4.2.4 Calculation of required information

Within this section, it will be shown how the number of loop iterations when a branch from the branch tree will change its direction (or determined that a condition will be always *false/true*) is calculated. This information is required within the next step where the number of how often the loop header is examined during runtime will be estimated. Within table 4.1, all information, including the requirements that are used to classify a branch as *known*, are listed. If one of these table entries could not be found or calculated, the branch is classified as *unknown*. Being classified as *unknown* does not mean that a loop could not be calculated. Only the range of the calculated loop iterations will be bigger and therefore less accurate. In the worst case, the calculated range will be "<0...∞>" which does mean that no information about the loop iteration bound could have been calculated.

Referring to the testexample in figure 4.2, it does mean that branch 3 '`if(x > 0)`' is classified as unknown because the initial value of the external variable x could not be determined. The value *adjust*, which compensates the difference between the relational operator '<' and '<=' or '>' and '>=', can be read from table 4.2 for each branch. If all required information for a branch have been collected, equation 4.1 is used to calculate the loop iteration when each branch changes its direction. From table 4.2, it can also be read if a branch will never change its direction and will therefore always be true or false. Within this case, no further calculation using equation 4.1 is necessary. It includes also the case where $before_i + after_i = 0$ and therefore a divide by zero exception equation 4.1 is avoided.

Term	Explanation	Requirement
variable	The control variable which is used within the condition and compared to the limit value	Only assignments of the form 'var := var + c' are allowed. Where 'c' must be a constant. Additionally, the amount of change must be constant every loop iteration. To guarantee this requirement, every node containing an assignment to this variable must dominate every node with a back edge to the loop header.
limit	Value to which the variable is compared to.	Must be a constant or a variable which must not change its value inside the loop (see also section 4.3) each iteration.
relop	Relational operator used inside condition	Must be one of the following operator types: '<', '<=', '>' or '>='. If some additional requirements are fulfilled, the operator could also be an equality '==' or an non equality '!=' operator (see also section 4.2.7 fur further information.
initial	Initial or start value of the variable before entering the loop	The assignment to this value must be defined within the last basic block before the loop body begins. This value must also be a constant (see also section 4.3 how this requirement could be handled alternative)
before	Amount of change of the variable before reaching the actual branch at each iteration.	This amount of change must be constant at every loop iteration, otherwise the branch is *unknown*.
after	Amount of change of the variable after reaching the actual branch at each iteration.	This amount of change must be constant at every loop iteration, otherwise the branch is *unknown*.
adjust	Is used to compensate the difference of the relational operators '<', '<=', '>' or '>='.	Must be an integral value between the range of -1 and 1 (see table 4.2 how this value will be calculated)

Table 4.1: Required information for each *known* iteration branch

$$N_i = \left\lfloor \frac{limit_i - (initial_i + before_i) + adjust_i}{before_i + after_i} \right\rfloor + 2 \tag{4.1}$$

Operator	Condition	Test Result	*adjust*
<=	$first \leq limit$ & $incr > 0$	is false on the Nth iteration	0
<=	$first \leq limit$ & $incr \leq 0$	always true	
<=	$first > limit$ & $incr \geq 0$	always false	
<=	$first > limit$ & $incr < 0$	is true on the Nth iteration	1
<	$first < limit$ & $incr > 0$	is false on the Nth iteration	-1
<	$first < limit$ & $incr \leq 0$	always true	
<	$first \geq limit$ & $incr \geq 0$	always false	
<	$first \geq limit$ & $incr < 0$	is true on the Nth iteration	0
>	$first \leq limit$ & $incr > 0$	is true on the Nth iteration	0
>	$first \leq limit$ & $incr \leq 0$	always false	
>	$first > limit$ & $incr \geq 0$	always true	
>	$first > limit$ & $incr < 0$	is false on the Nth iteration	1
>=	$first < limit$ & $incr > 0$	is true on the Nth iteration	-1
>=	$first < limit$ & $incr \leq 0$	always false	
>=	$first \geq limit$ & $incr \geq 0$	always true	
>=	$first \geq limit$ & $incr < 0$	is false on the Nth iteration	0
Where $first = initial + before$, $incr = before + after$, N is defined in equation 4.1 and *adjust* is used in equation 4.1			

Table 4.2: How to determine the *adjust* value and the direction of change of the iteration branch

Table 4.3 shows the calculated information for each branch of our example code. The last column contains the result of the equation 4.1. Branch 3 is *unknown* and therefore no number of iteration when the branch changes direction could be calculated. Branch 4 is *known*, but as it could be seen in the next section, the number of iteration will never be reached and therefore replaced with ∞. Another example where equation 4.1 is not useful will be the following loop:

```
for( i=0; i>10; i++ ){ some_statement; }
```

The value of *initial*+*before* (=0) is less than the limit of 10 and therefore the condition is false at the beginning and exits immediately. This means that the number of loop iterations is 0 because the body of the loop is never executed. Another example for a loop that may never exit is the following:

```
for( j=0; j<10; j-- ){ some_statement; }
```

The value of *initial*+*before* (=0) is less than the limit of 10 and the update value (*before*+*after*) is -1 and therefore less than 0. This fulfils case 2 within the second operator class of table 4.2 and the condition will be always true and the loop may never exit.

branch	*variable*	*limit*	*relop*	*initial*	*before*	*after*	*adjust*	*known*	*iter*
1	i	200	$<$	0	0	1	-1	$\surd$	201
2	i	160	$>$	0	0	1	0	$\surd$	162
3	x	0	$>$	–	0	0	–	–	–
4	j	450	$>$	0	0	2	0	$\surd$	227

Table 4.3: Calculated information for each branch of the tree in picture 4.8

4.2.5 Determine the range of iterations when each of these identified branches could be reached

The next step within the calculation of the loop iteration bound is to determine when each of the branches could be reached. This information is calculated in top-down order. First, to the loop header is always assigned the range from 0 (=node_range_min) to ∞ (=node_range_max), which is written as [0...∞]. All other nodes are assigned the result of the union of all incoming edge ranges. The ranges of all outgoing edges may be calculated using one of the following rules:

1. If the node is dedicated as *known* and the test result of the third column in table 4.2 is "*is false on the Nth iteration*":

   (a) if the number of iteration when a branch changes its direction (last column within table 4.3) is less than node_range_min and greater or equal as node_range_max: the range of the first edge (when the condition is true) is set to [*node_range_min...iteration of change*-1] and the range of the second edge is set to [*iteration of change...node_range_max*]

   (b) Otherwise the range of the first edge is assigned to [*node_range_min...node_range_max*] and the range of the second edge is assigned to [∞...∞] (because this edge will never be reached)

2. If the node is dedicated as *known* and the test result of the third column in table 4.2 is "*is true on the Nth iteration*":

   (a) if the number of iteration when a branch changes its direction (last column within table 4.3) is less than node_range_min and greater or equal as node_range_max: the range of the first edge (when the condition is true) is assigned to [*iteration of change...node_range_max*] and the range of the second edge is set to [*node_range_min...iteration of change*-1]

   (b) Otherwise the range of the first edge is assigned to [∞...∞] (because this edge will never be reached) and the range of the second edge is assigned to [*node_range_min...node_range_max*]

3. If the node is dedicated as *known* and the test result of the third column in table 4.2 is "*always true*" then to the first edge is assigned the same range as to the node itself and to the second edge is assigned the range [∞...∞].

4. If the node is dedicated as *known* and the test result of the third column in table 4.2 is "*always false*" then to the first edge is assigned the same range as to the node itself and to the second edge is asigned the range [∞...∞].

5. If the node is dedicated as *unknown* to all outgoing edges are assigned the same range as the node itself. This is done, because there is no information when each direction is followed.

Figure 4.10 shows the branch tree with all node ranges and edge ranges for the example in figure 4.2. Within this directed graph, every node, where the branch is classified as *known*, is marked with a K otherwise with a U. Note that the condition `if(j > 450)` of node 3 will never be true and therefore the edge has the assigned range [∞...∞].

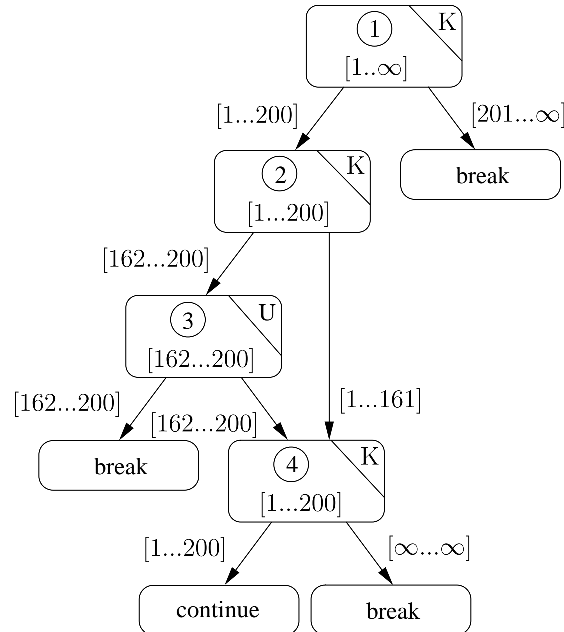


Figure 4.10: Branch tree with calculated ranges of iterations

4.2.6 Determining the minimum and maximum loop iterations

The calculated ranges for each branch in the previous section and its edges now are used to calculate the minimum and maximum number of the iterations of the entire loop. For each branch the minimum and the maximum iteration number are calculated in bottom-up order. At the end, the minimum and maximum iteration number of the root node (=loop header) represents the range for the loop itself. Table 4.4 announces all terms that are used within the following rules. These rules lay down how the values to each term are assigned.

[*edge_range_min...edge_range_max*]
edge_range_min: lowest iteration value when this edge can be reached
edge_range_max: highest iteration value when this edge can be reached

<*edge_exit_min...edge_exit_max*>
edge_exit_min: first iteration when this edge may lead to a break
edge_exit_max: first iteration when this edge must lead to a break

[*node_range_min...node_range_max*]
node_range_min: lowest iteration value when this node can be reached
node_range_max: highest iteration value when this node can be reached

<*node_exit_min...node_exit_max*>
node_exit_min: first iteration when this node may lead to a break
node_exit_max: first iteration when this node must lead to a break

Table 4.4: Notation that is used within the assignment rules

1. If an edge is leading to a *BREAK*-node than both the *edge_exit_min* and the *edge_exit_max* values are assigned to the *edge_range_min* value. Because this edge can only taken once and the loop will be leaved. A *BREAK* is the only way to leave a loop and therefore the only place where bounded values can be introduced.

2. If an edge is leading to a *CONTINUE*-node than both the *edge_exit_min* and the *edge_exit_max* values are assigned to ∞. Because a *CONTINUE* will only point to the loop header and therefore not hold any information about the loop exit.

3. If an edge is not pointing to a *BREAK* or to a *CONTINUE* node it is pointing to a node, representing a conditional branch. Within this case, the *edge_exit_min* value is simply assigned to the *node_exit_min* value of the target node and the *edge_exit_max* value is assigned to the *node_exit_max* value of the target node. This could be done because the ranges of each edge are calculated exactly as described in the previous section. This is slightly different as it was described in [HSR$^+$00] where the edge ranges are assigned to [1...N-1] and [N...∞]. N stands for the number of iteration when the branch changes its direction.

4. If a node of the branch tree is marked as *known* then the *edge_exit_max* value for this node is set to the smallest number of the *edge_exit_max* values of both

outgoing edges. Because the loop has to be leaved if a *BREAK* occurs and the higher iteration bound for a *BREAK* could never be reached.

5. If a node of the branch tree is marked as *known* then the *edge_exit_max* value for this node is set to the largest number of the *edge_exit_max* values of both outgoing edges. Because the algorithm has no information about the branch behaviour and therefore must take the upper exit bound.

6. The *node_exit_min* value of a branch tree node is set to smallest number of the *edge_exit_min* values of both outgoing edges. For the lower exit value it must take the first possibility when a *BREAK* could be reached. And therefore there is no need to distinguish between *known* and *unknown* branches.

Figure 4.11 shows the same branch tree as in figure 4.10 but now with calculated exit ranges using the rules described before. Branch node 4 has received the node exit ranges $< \infty ... \infty >$ because the edge pointing to the *BREAK* node will never be reached. Branch node 3 has been assigned the node exit ranges $< 162 ... \infty >$ because the iteration number when the first outgoing edge leads to a break is 162. But the branch is classified as *unknown* and there is no guarantee that the edge will ever be taken and therefore the greatest possible range must be taken. And the loop header and therefore the entire loop has received the iteration bound $< 26 ... 101 >$ because the lower iteration number when a path leads to a *BREAK* is 26. But there is no guarantee that this path will be taken. And at the iteration number of 101, the second edge absolutely leads to a *BREAK*.

4.2.7 Calculate loops with iteration branches using the equality operator

In section 4.2.4, branch nodes that contains an equality operator ($==$ or $!=$) were classified as unknown. This will be result in a safe but also bigger range of loop iterations. But there are types of loops using equality operators, which can be successfully bounded by the implementation of the previous sections if they are fulfil some additional requirements. Figure 4.12 contains some simple example loops to show the problematic nature of loops containing equality operators. Loop 4.12 a) (*some_statement* is any statement that not influence the number ob loop iterations) can be bounded because it is guaranteed that the induction variable i reaches a point where the condition becomes false. A short look at example 4.12 b) exhibits that this loop must be unbounded because the variable i there does never reach a value that let the condition become false. Another problem, which can occur while dealing with such types of loops is shown in example 4.12 c). Because there is an *unknown* branch, which can prevent the loop from reaching its *BREAK* statement. This loop may have a bounded number of loop iterations but there is no guarantee because the value of x could not be determined (section 4.3 will show how this problem could be mastered). The following table lists a number of rules when also

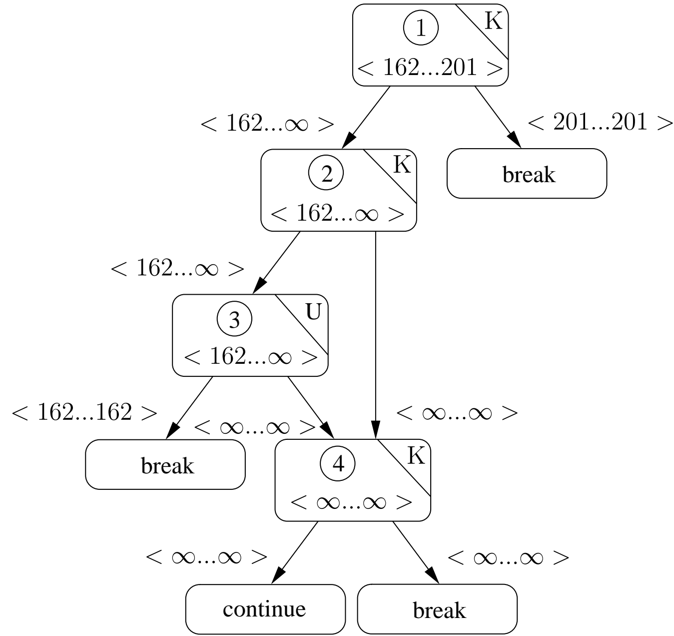


Figure 4.11: Graph after calculation of minimum and maximum iteration values

for loops, containing equality operator, a bounded number of loop iterations could be calculated, if all of them are fulfilled.

1. Every node containing a back edge to the loop header must be dominated by the branch containing an equality operator. Because if this condition is not fulfilled it is not guaranteed that this branch will meet its exit condition although the branch variable is assigned to the value, which makes the branch change its direction. Example 4.12 c) does not fulfil this rule and could therefore not be calculated.

2. One of the outgoing edges of an iteration branch that contains an equality operator must lead to a *BREAK*. Because it must be guaranteed if a branch changes its direction that one of the directions leads to an exit point.

3. The following expression, which is part of equation 4.1 must result in an integral value:
$$\frac{limit_i - (initial_i + before_i)}{before_i + after_i}$$
This means that a variable must reach its limit at a certain iteration branch. Example 4.12 b) does not fulfil this requirement and therefore the variable does never meet its limit of 10 (only 0, 4, 8, 12, . . . but not 10).

If these requirements are all fulfilled, the branches are calculated in the same way as described in the previous chapter. Only table 4.2 will be replaced with table

4.5 to calculate the ranges and adjust values for every node and also the ranges for each edge. Note that if the update value of a variable is 0 than the condition is always false or always true and we do not need to use equation 4.1 and therefore avoid a divide by zero exception.

a)

```
for( i=0; i!=10; i++ ){
    some_statement;
}
```

b)

```
for( i=0; i!=10; i+=4 ){
    some_statement;
}
```

c)

```
for( i=0; ; i++ ){
    if( x>5 );
        continue;
    if( i==20 );
        break;
}
```

Figure 4.12: Some example loops using equality operators

Operator	Condition	Test Result	*adjust*
$==$	$first < limit$ & $incr > 0$	is true on the Nth iteration	-1
$==$	$first > limit$ & $incr < 0$	is true on the Nth iteration	1
$==$	$first = limit$ & $incr = 0$	always true	–
$==$	$first = limit$ & $incr \neq 0$	is false on the 2nd iteration	–
$==$	otherwise	always false	–
$!=$	$first < limit$ & $incr > 0$	is false on the Nth iteration	-1
$!=$	$first > limit$ & $incr < 0$	is false on the Nth iteration	1
$!=$	$first = limit$ & $incr = 0$	always false	–
$!=$	$first = limit$ & $incr \neq 0$	is true on the 2nd iteration	–
$!=$	otherwise	always true	–
Where $first = initial + before$, $incr = before + after$, N is defined in equation 4.1 and *adjust* is used in equation 4.1			

Table 4.5: Adjust value and direction of change for branches with equality operator

4.3 Loops with a non-constant loop-invariant number of iterations

As it has been shown in the previous section, it is not always possible to bound the number of loop iterations; because the iteration number where some branches changes its direction could be unknown. There are different ways to handle loops that could not be calculated. The program may be prompting the user to specify the number of loop iteration interactively or it may specified within the source code as annotation. But for long and complicated loops it may be difficult to determine the number of loop iterations by hand and therefore there is no guarantee that the user my specify the right loop bound. Otherwise variables within branch conditions are often loop invariant. Using this circumstance it is possible to relax the requirement within table 4.3 that the values of *limit* and *initial* had to be constants. For the initial value of a branch condition variable or if the limit is also a variable the last basic block before the loop body starts is evaluated. But if there is no assignment with a constant found it is possible to let the user specify an upper and a lower bound for the variable. Because this is less error prone as letting the user specify an upper and a lower bound for the entire loop. This could also be done interactively. To stop the calculation for each branch that could not be classified as known and ask the user for upper and lower bound could be annoying. Therefore this work is only based on user annotations within the source code. Similar as in the programming language WCETC of [Kir02] the grammar of the programming language C will be expanded for value bound annotations before a branch that could not be classified as *known* occurs. Within the header file "wcet.h" in [Kir02] a new macro

WCET_VALUE_BOUNDS(x,y,z)

has been inserted that is replaced by the preprocessor with

value_bound (x) minimum (y) maximum (z)

if *LANG_WCET* is defined within the source code. Otherwise it will be replaced by an empty sequence when it will be translated with a usual C compiler. The variable x of the assertion stands for the variable name and y for lower variable bound and in the same way z for the upper variable bound. If such an assertion for the *limit* or the *initial* value of a branch condition is found and the defined minimum and the maximum values are identical the loop bound could be calculated automatically in the usual way. When there is a difference between the minimum and maximum values, the value in the calculation will be replaced first with the minimum and afterwards with the maximum value. The upper bound for the loop iterations is the biggest result of both upper bound calculations and the lower bound is smallest result of both lower bounds. Figure 4.13 shows a simple example function with an annotation for the value limit of variable a. The number 20 is the minimum and the number 30 is the maximum value for a. Using this provided values, the upper and lower loop iteration bound could be easily calculated as $< 20...30 >$.

```
int calculate( int a ){
   int res=1;

   WCET_VALUE_BOUNDS( a, 20, 30 )
   for( i=0; i<a; i++ ){
      res *= i;
   }
   return a;
}
```

Figure 4.13: Loop with a non-constant loop-invariant number of iterations

If there is more than one annotation for different values within the code it must be calculated the upper and the lower bound for each permutation of this values. For example three different upper and lower bounds for three different variable values have been parsed from the code. This would result in the following combinations:

val1	val2	val3
min	min	min
max	min	min

val1	val2	val3
min	min	max
max	min	max

val1	val2	val3
min	max	min
max	max	min

val1	val2	val3
min	max	max
max	max	max

where *min* represents the minimum value of the annotation and *max* the maximum value. As it can seen, this results in 2^n (n is the number of assertions) different combinations, which have to be calculated. This means that a loop with 3 unknown values with different minimum and maximum value assertions needs approximately an 8 times greater calculation time. Therefore, a programmer of code for WCET[5] analysis could decrease the calculation time if he would follow some coding conventions to make loop bounds predictable (see [HSR+00] for further details).

[5] **W**orst **C**ase **E**excution **T**ime

Chapter 5

Realization of the framework

Within this chapter an overview is given about the method of how to calculate the minimum and maximum iteration of a natural loop[1], described in the previous section, had been implemented within this master thesis. To ease the understanding of the implementation, we partitioned it into several steps and each step is described within an own subsection.

5.1 Syntactical analysis of the input file

5.1.1 Preprocessing

Before starting the analysis, the GCC preprocessor is called to replace all macros and include the code of all used header files. To divert the output stream of the preprocessor into the input stream of the analyser, the unix system call `popen()` is used. The function `popen()` creates a pipe for the return stream, forks the actual process and invokes a shell to call the preprocessor. If the system call was successful, the preprocessed input file can be processed within the analyser.

5.1.2 Lexical Analysis

Finite Automates are used to describe the syntax (structure) of input data. It consists of *states*, *transitions*, one distinguished *start state* and also one distinguished *end state*. All transitions are marked edges that connect the states together. If an input is read, it starts from the start state and depending on the sequential input symbol it follows the marked transitions and changes the state or stays where it is. If the end state is reached the input sequence is accepted.

Regular Expressions are an effective description of a finite automate. A regular expression consists of a sequence of recognisable symbols and some abstract symbols. Abstract symbols for example are '[' and ']a', which represents a

[1]a loop with only a single entry

symbol class. Another abstract symbol could be an asterisk '*', which can follow a symbol class and indicates zero or an endless number of symbols within the symbol class. E.g. [1-9][1-9]* can be used to recognise natural numbers. [1-9] indicates a digit between 0 and 9 and [1-9]* indicates zero or an endless number of digits like '1234'.

Token accepted individual sequence described by a regular expression

The aim of a lexical analyser is to translate a textfile into a stream of tokens. Lexical analysis of different programming languages is very similar. Therefore writing the code for such a scanner can be done automatically. Within this work, the scanner generator *FLEX*[2] was used. This generator needs a specification file as input and creates a output file with the scanner function `yylex()`, which can be called from another source file. Every call of the function `yylex()` returns a new recognised token. The specification file consits of three with '%%' separated sections. The first section is the definition part. There can be placed all variable definitions, reusable parts of a regular expression or `#include` instructions. Every line in this section is copied directly into the output file. The next section is the most important part. There must be placed all regular expressions where each of it describes an individual token. For each regular expression there must be placed an action that will be executed if a token is recognised. The last section can be used to define some functions, which are called as action for any regular expression. Within this work, all rules in the configuration file have the form

```
''tokenname'' or regular expression
          { count(); if( !attribute_found ) return(token); }
```

All elements between the opening '{' and the closing '}' braces are executed as action. The function `count()` calculates the line and the column of the beginning of the token within the input file. This information is required to write the result of the analysis on the right place within the output file. To distinguish between the source of the input and the source of an included file, the line and column are set to -1 for tokens of the include code. The flag *attribute_found* is set if the *GCC* compiler extension `__attribute__` was found. If this flag has been set, the following tokens are ignored. Otherwise the token identifier (e.g. "GOTO") is returned. Using the specification file, the call

```
flex -flags 'specification file'
```

produces a file called `lex.yy.c`, which can be compiled and linked to the whole program.

[2]**F**ast **Lex**ical analyser

5.1.3 Syntactical analysis

Context free grammar A context free grammar produces a context free language. Beginning from a non-terminal start symbol, a context free grammar produces a sequence of terminal symbols using grammar rules iteratively. This sequence could also be empty ($\varnothing$). A grammar is context free if on the left side of the grammar rules are only single non-terminal symbols. Such non-terminals will be replaced independent from the surrounding *context*.

Grammar rule All grammar rules of a context free grammar are of the form $A \to \gamma$. Where A is a non-terminal symbol and γ a sequence of terminal symbol and non-terminal symbols. The empty sequence will be produced with the rule $A \to \epsilon$. A formal description of these rules is:

$$\forall (w_1 \to w_2) \in \textit{Grammar rules}: (w_1 \in \textit{Nonterminals}) \wedge (w_2 \in (\textit{Terminals} \cup \textit{Nonterminals})^*)$$

Terminal is a symbol that cannot be replaced anymore, e.g. a *token*, returned by the scanner.

Non-terminal is a symbol which can be replaced with a sequence of terminal and non-terminal symbols using the grammar rules described before.

Startsymbol is a single non-terminal start symbol, that will be replaced first.

Similar to the lexer as described in the previous section, the parser for the syntactical analysis is generated using the parser generator *YACC*[3]. This compiler generator needs as input a configuration file, which has the form

```
%{
    C-declarations
%}
    YACC-declarations
%%
    Rules
%%
    Helping functions
```

Within the section *C-declarations*, all C functions used within the parser will be declared (also the external scanner function `int yylex()`). Additionally all `#include` directives and global variables definitions will be placed at this section.

The *YACC-declarations* of the next section will be replaced by declarations describing the characteristic of the C language grammar. Table 5.1 shows a list of some this declarations, which have been used within this master thesis.

[3]**y**et **a**nother **c**ompiler **c**ompiler

`%union{ struct func *fn, ...}`	Type of $$, the non-terminal on the left side of a *YACC* grammar rule
`%token GOTO`	Indicates that the symbol *GOTO* is a token (= terminal symbol)
`%type <fn> function_definition`	Indicates that the non-terminal symbol *function* has the type $< fn >$ which must be part of the union defined in the first line
`%start program`	Indicates that the nonterminal *program* is the start symbol of the whole grammar

Table 5.1: Some often used examples of *YACC* declarations

For every terminal or non-terminal used within the grammar rules it must be indicated as token (with *%token*) or have a declared type (with *%type <type>*).

The longest and most important part of the configuration file will be the section containing the *YACC* grammar rules. These rules are context free (as described within the beginning of this section) and have the form

```
non_terminal :  /* empty */
             |  (terminals ∪ non_terminals) { action }
             ;
```

The ':' separates the left side of the grammar rule from the right side. On the right side, the '|' separates different possible replacement rules and the semicolon ';' at the end terminates the whole rule. Between the opening '{' and the closing '}' braces after each replacement rule there could be some C functions inserted which will be executed if this replacement rule would be used. Figure 5.2 shows how a function could be described using a context free grammar. All symbols are written with lower case signs, which means that they are non-terminals. There are four possibilities of how a function could be defined. The first rule means that there is a declared return value (*declaration_specifier*), followed by the function name (*declarator*) and the arguments (*declaration_list*). And at least is the function body (*compound_statement*) placed, which contains in the most cases a list of statements. As second example in figure 5.3 on page 46, the possible definitions of C language loops, also described by a *YACC* grammar, are shown. The key words *for*, *do* and *while* are recognised as tokens and therefore written in upper cases. Also the opening '(' and closing ')' round braces between the inverted commas means that they are terminal symbols. The *expression* or *expression_statement* contains the loop condition which has be analysed in chapter 4. The symbol *statement* is also a non-terminal and would be replaced by a list of statements and forms the loop

body, which could also have multiple exits (see section 4.2 for further details).

```
function_definition
    : declaration_specifiers declarator declaration_list
      compound_statement
    | declaration_specifiers declarator compound_statement
    | declarator declaration_list compound_statement
    | declarator compound_statement
    ;
```

Table 5.2: *YACC* rule to recognise the different types of function definitions

How to use this and all the other rules that describe a valid C program and how to create a syntax tree of the input source file is shown in the next section (section 5.2). At the end of the configuration file there is the section *Helping functions* left. This section can be used to define some C language helping functions that may be used between the '{' and '}' braces where the action for each of the grammar roles could be defined.

```
iteration_statement
    : WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | FOR '('expression_statement expression_statement
             maybe_expression ')'
      statement
    ;
```

Table 5.3: *YACC* rule to recognise the different types of loop definitions

5.2 Syntax tree

This section describes how the syntax tree is created using the *YACC*-rules of the previous section. The tree is created from top down. Figure 5.1 on page 47 gives a graphical representation of the stored syntax tree at the end of the lexical analysis. All functions are stored in a single linked list. The loop analysis takes place local at each function so there is no gain of performance if a double linked list will be used. For each function, a pointer to a linked list of all statements inside of of it is stored. During the analysis there is of often the need to get the successor of a statement.

But there are also a number of accesses to predecessor of a statement. Therefore a double linked list was chosen as the most efficient way to store the data. And for each statement, as can be also seen in the same figure, the expressions are stored in a hierarchical order as a directed linked tree. Within the next lines of text, there is also a short description of the used code segments within the implementation of this work.

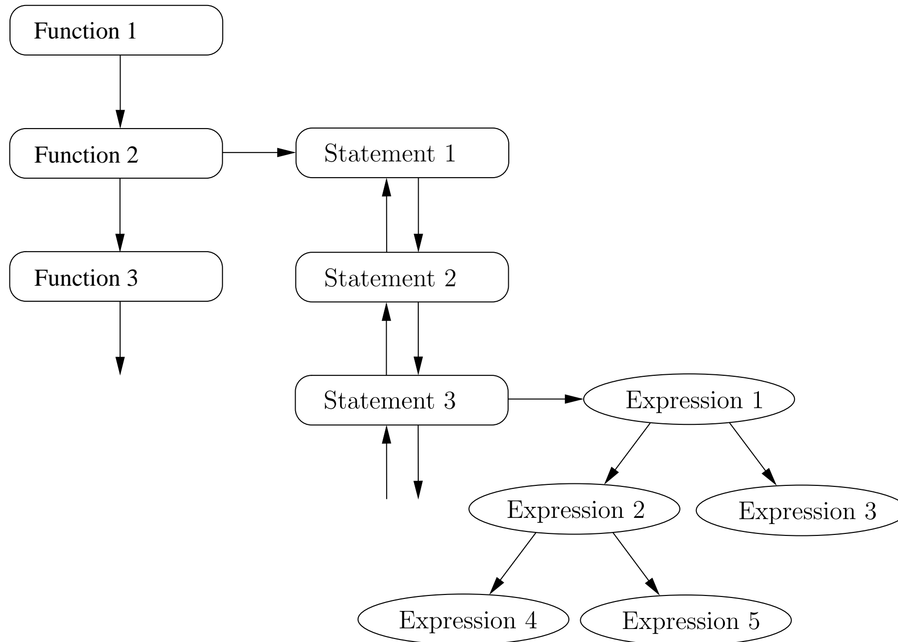


Figure 5.1: Structure of the syntax tree of C language programs

Table 5.4 shows the piece of code from the data structure which is used to store the parsed information for each C function definition within the source code. "`num_nodes`" and "`**nodes`" are used to store the nodes of each function. How to group the statements of a function into nodes and how they are used within the control flow graph are described in section 5.5. The vector "`*labels`" is used to store all labels that are used as target of the jump statement *GOTO*. Because a label can be declared at every position inside a function. And therefore, within the worst case the algorithm must search within all predecessor statements until the function header is reached and again to search all successors down to the end of the function until the target of a *GOTO* statement is found. Using this vector, only the list of all labels had to be evaluated until the target is found. This can be save a lot of search time compared to the worst case where every statement has to be evaluated. The pointer at the last line "`*symb`" refers to a list of all symbols that can be accessed from this function.

```
struct func *next;              /*!< Pointer to the next function */
struct ident *name;             /*!< Name of function */
struct stmt *stmts;             /*!< First statement */
int num_nodes;                  /*!< Number of cfg nodes */
struct cfg_node **nodes;        /*!< Nodes of cfg */
vector<struct stmt *> *labels;  /*!< Stores all labels of a function */
registered_symbols_t *symb;     /*!< Pointer to the last declared
                                     symbol in this function */
```

Table 5.4: Data structure to store all details of a function

One step deeper into the hierarchy of the syntax tree goes table 5.5 on page 49. It shows the implementation of one member of the double-linked statement list of each function. One important member of this is the integer `depth_level`, which indicates the hierarchy level where a statement is located. For global statements this value is 0, for arguments of the function 1 and within the function ≥ 2. Whenever a new scope (beginning with an opening curly brace '{') begins this value is incremented. And at the end of a scope (ends with a closing curly brace '}') it will be decremented. This in important e.g. for finding the right successor within nested *IF-ELSE*-statements as described in section 5.5. The two members, `line` and `column`, indicate the position of the beginning of the statement within the source code. This position is used at the end of the analysis where the result of the calculation must be written between the loop header and the loop body. The pointers `*symboltab` and `*typetab` are referring to the list of symbols and to the list of types that can be seen from the position of the actual statement (see section also section 5.4 fur further details). Within the case that the actual statement is of type *FOR*, *WHILE* or *DO* the pointer `*details` will point to a structure containing the calculation result (see section 4 for further details). Now, the expression level of figure 5.1 is described. As mentioned in previous sections, a statement could contain an expression (e.g. a condition or a expression statement). For the analysis within this work it has been found that this information could be best stored as directed binary tree. Because, if it must be evaluated if a variable has only constant assignments, then all expressions must be considered where the root node is an assignment. If such an expression is found and the left child is the variable then the right child has to be a constant. Table 5.6 shows the corresponding data structure, which is used to store such an expression tree. The enum "`type`" of type `expr_type` stores, which kind of expression it is. The enum "`expr_type`" contains a large number of members so not all of them could be described here. In assistance for all others, the members *E_VARIABLE* (indicates that the expression is a variable), *E_ASSIGN* (representative of '=') and *E_NUM_CONSTANT* (means that the expression is a number constant like '1.0') are introduced.

```
enum stmt_type type;             /*!< Statement type */
struct stmt *next;               /*!< Next statement */
struct stmt *prev;               /*!< Previous statement */
struct expr *expr;               /*!< Expression */
int depth_level;                 /*!< Depth-level of the statement */
symbol_t *symboltab;             /*!< Pointer to the symbol-table */
type_t *typetab;                 /*!< Pointer to the type-table */
int line;                        /*!< Line number in sourcecode of
                                      this statement */
int column;                      /*!< Column in the code where
                                      this statement starts */
struct loop_details *details;    /*!< Contains all analysis details */
```

Table 5.5: Data structure of a statement

Within the case that an expression is a variable, the pointer "`*ident`" refers to a structure, which stores the name of the variable and the pointer to the corresponding symbol within the symbol list (see also section 5.4).

```
enum expr_type type;             /*!< Operator/expression */
struct expr *kids[2];            /*!< Operands of operator */
struct ident *ident;             /*!< Identifier for variables/
                                      function calls */
double val;                      /*!< Value of number constants */
```

Table 5.6: Data structure of an expression

The *double* value "`val`" stores the number if an expression is a number constant. Now, to show it with an example, the expression of the following expression statement

```
var1 = 3;
```

should be stored within an expression tree. Within the *YACC* specification file, the following grammar rule is used to describe such an assignment expression. Like this piece of code, in most *YACC* grammar implementations they are in hierarchical order. If in this case the expression is not an assignment expression, then it could be a conditional expression. This hierarchy ends with an `primary_expression`, which could be a number constant or an identifier.

```
assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;
```

Using the example of few lines before, `var1` goes the hierarchy from *unary_expression* down to an identifier. There is a new expression with type *E_VARIABLE* created. The pointer "`kids[0/1]`" are set to *NULL* and the pointer "`*ident`" are set to a structure where the name "*var1*" and corresponding the element of the symbol tree (section 5.4) will be stored. The constant number "3" is stored in a similar way. The only difference is that the `*ident` pointer is set to *NULL* and the number "3" is stored in `val`. And now, the previous described *YACC* rule finds the assignment operator "=" within the non-terminal symbol *assignment_operator*. Within this rule, also a new expression, but now with the type *E_ASSIGN*, is created. The previous created expressions are set as *kids*, with the variable as `kids[0]` and the constant as `kids[1]`. This is the idea behind the storage of expressions as binary tree. Sometimes, such stored tree must be changed to make the analysis easier afterwards. Because an expression of an expression-statement could also be a composed one like "`var1 = var2 = 3;`". Then we have to change it into two separate expressions "`var1 = 3;`" and "`var2 = 3;`" to get the usual structure *variable = constant*. Afterwards, this separate binary tree will be joined together with a concatenation expression *E_CONCAT*. There are also many other necessary changes, but the idea behind is always nearly the same and therefore not further described within this work.

5.3 Type table

The *type table* is used to store all non-elementary types that occurs during parsing of the source code. Such non-elementary types are *enum*, *struct*, *union* or new types defined with *typedef*. Every time when a new variable declaration is found the corresponding type is stored together with all other variable details. And therefore this type table can be used to get the real structure behind a variable name. The type table is implemented as linked list of data structures, which is shown within table 5.7 on page 51. Composed types are stored as sub-lists within linked elements of the same structure. The `name` of the type is a string, which is used to identify the type within the source code. The next element *typespec* stores which kind of composed type (e.g. *STRUCT*) it is, *T_POINTER* is used for pointer types and *T_TYPEDEF* for a new declared type name. If it is only an elementary type (e.g. *int*) the name of the elementary type will be stored. For composed ones like *struct* the number of subtypes and a reference to the list of subtypes is stored. If a type is a subtype of a composed one, then the corresponding `member_name` must also be stored. Because only the type name is not enough to indicate each type member. Each part of the program should only see the types that are declared for it and therefore, like for statements, the `depth_level` for each type is stored.

```
char* name;                 /*!< name of the type */
typespec_t typespec;        /*!< kind of type */
int numsubtype;             /*!< number of subtypes for composed types */
struct type *subtype;       /*!< dyn. alloc. array for sub-types */
struct type *next;          /*!< next type element in list */
int depth_level;            /*!< statement-level of declaration-point */
char *member_name;          /*!< name, if struct- or union-member */
```

Table 5.7: Structure of the type table

A new type is always be inserted at the beginning of the list. If a scope end, indicated with the closing curly brace '}', is found the pointer for the current type list is set to the first element with a *depth_level* less than the level of the current type. The corresponding type table for a source file has a similar structure as the corresponding symbol table which is depicted in figure 5.3 on page 52. The following short example shows how a new type is stored within the list.

```
typedef int counter_t;
counter_t counter1;
```

First, a new element is inserted at the beginning of the current type list. This new element has the `typespec` *T_TYPEDEF* and the `name` *counter_t*. The subtype counter `numsubtype` is set to '1'. As subtype, a reference to the structure and *T_NUMBER* as type, representative for *int*, is stored. All other elements for the subtype are set to the initial values. The pointer `*next` refers to the old current type list, which is the next element within the new list. The `depth_level` is set to the same value as the level of the surrounding statement where the type is declared. In the following line, the declared variable has the new defined type *counter_t*. So the reference to the current type table can be followed until the first type with name *counter_t* is found. The result of the search has the type *T_TYPEDEF*, so it can be dereferenced and the reference to the subtype *t_number* is stored as type for the variable *counter1*.

5.4 Symbol table

The *symbol table* is structured similarly as the *type table*, described in the previous section. The example code within figure 5.2 is used to demonstrate the construction of the symbol table using a valid C code. The pointer *current_symbol_list* always holds a reference to current position within the symbol table. Every time a new symbol is declared it will appended at the beginning of the *current_symbol_list*. After the new symbol is appended, the pointer *current_symbol_list* will be set to this

new symbol. At the end of a scope this pointer is set to the first symbol which has a depth level less then the depth level of the current target of this pointer. This leads to a symbol table, which may contain several branches inside.

```
int symbol1;
if( condition1 ){
   int symbol2;
} else {
   int symbol3;
   if( codition2 ){
      int symbol4;
   } else {
      int symbol5, symbol6, symbol7;
   }
}
```

Figure 5.2: Example code to show the structure of the symbol table

For the example code on page 52, figure 5.3 shows the resulting symbol table. With every opening curly brace '{' a new scope begins. As it can be seen on this figure, after *symbol1* and *symbol2*, the symbol table splits up because there are different scopes and within each of this scope new symbols had been declared. Now, we describe the data structure used to store new symbols within the symbol table. The code of the data structure is shown within figure 5.8.

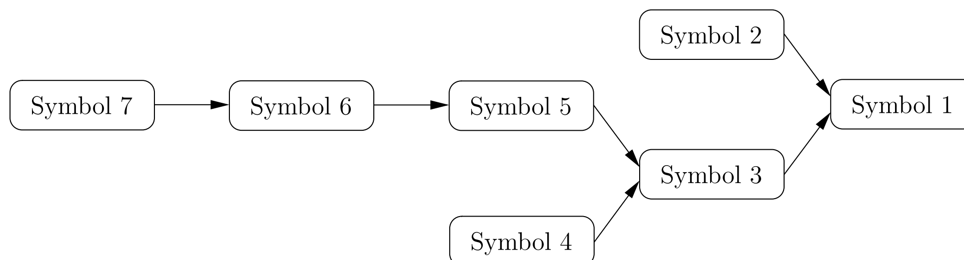


Figure 5.3: Graphical representation of the symbol table from the example code

One of the most interesting details of this structure may be the reason of the introduction of the elements `numsubsymb`, `subsymb` and `subsymbof`. They are used if the type of the symbol is a composed one. Because in this case, for each subtype of the main symbol type also a subsymbol must be created. This subsymbols are used for accesses like the following

```
symbol->member1 = element1;
symbol.member1 = element2;
```

where for each assignment the reference to the subsymbol is stored. To get the reference of the outer symbol itself the pointer `*subsymbof` of the stored subsymbol must be followed. If the outer symbol is reached this pointer is *NULL*. Every time a variable is used again within the source code after the declaration, the pointer *current_symbol_list* has to be followed. The first found symbol with the same name is the right one; because the last declarations always has been inserted at the beginning. And each statement has also only access to that path of the symbol list with variables that are valid inside of the actual scope.

`char *name;`	*/*!< Name of the symbol */*
`type_t *ptype;`	*/*!< Pointer to the type structure */*
`int numsubsymb;`	*/*!< Number of subsymb. */*
`struct symbol *subsymb;`	*/*!< Dyn. alloc. subsymb. array */*
`struct symbol *subsymbof;`	*/*!< Pointer to the parent-symbol */*
`struct symbol *next;`	*/*!< Next element in list */*
`int depth_level;`	*/*!< Depth level of the symbol (0=global, 1=paramter, >1=local) */*
`int id;`	*/*!< Unique id for a symbol */*
`struct registered_symbols` *rsymbol;	*/*!< Pointer to the registered symbol */*

Table 5.8: Structure of the symbol table

5.5 Control Flow Graph

After the source file has been parsed successfully and the creation of the syntax tree is finished, there must be created a *control flow graph* for each function. This *control flow graph* is necessary for the further analysis of the code. Every node within this flow graph represents a basic block[4]. During the construction, the following steps must be performed:

1. Finding all basic blocks of each function.
2. Storage of the first and the last statement of each block.
3. Finding all successors of each basic block.
4. Finding all predecessors of each basic block.

[4]see section 4 for further definitions around the control flow graph

To find all basic blocks, the syntax tree at the statement level must be evaluated. At the beginning, the algorithm starts with a new basic block. If the next statement has a depth level greater or less then the depth level of the actual statement, the current basic block ends with the actual statement. The second reason for ending the current basic block is when the actual statement is a branch statement like *FOR*, *IF*, or *SWITCH*. If the current basic block ends and the last statement of this block was not the last statement of the function, a new basic block starts. When the beginning and the end of each basic block have been determined, the first and the last statement is stored for each block. The statements are double linked with its successor and predecessor and therefore only the first and the last statement is necessary for each basic block. After determining all basic blocks (=nodes of the control flow graph), the first basic block is set as root node within the control flow graph. To find and store all successors of each node is a little bit more tricky; because also the semantic of the program must be taken into account.

```
if( condition1 ){
   if( condition2 ){
      some_statement1;
   }
} else {
   some_statement2;
   some_statement3;
}
```

Figure 5.4: Some example code to show the construction of the control flow graph

Figure 5.4 shows a short code example, which could be a source of errors if the grammar of the C language would not followed exactly. During parsing, an *IF* or corresponding *ELSE* statement is terminated by an pseudo *END_IF* statement to let the algorithm know when corresponding *IF-ELSE*-statements are finished. Normally, the successor of an *END_IF* node is the sequential successor node. The only exception is an *IF-ELSE*-statement that itself is inside of an *IF*-block of an *IF-ELSE*-statement. Because the sequential successor would be the *ELSE*-statement of the surrounding *IF-ELSE*-statements. But the right successor is the next node containing an *END_IF* statement with the same depth level as the found *ELSE* statement. Otherwise, if the condition of the surrounding *IF*-statement is true, the path within the control flow graph goes through the body of the *IF* statement *and* also through the body of the *ELSE*-statement. And this path is erroneous. Figure 5.5 on page 55 shows the well constructed control flow graph to the corresponding code within figure 5.4.

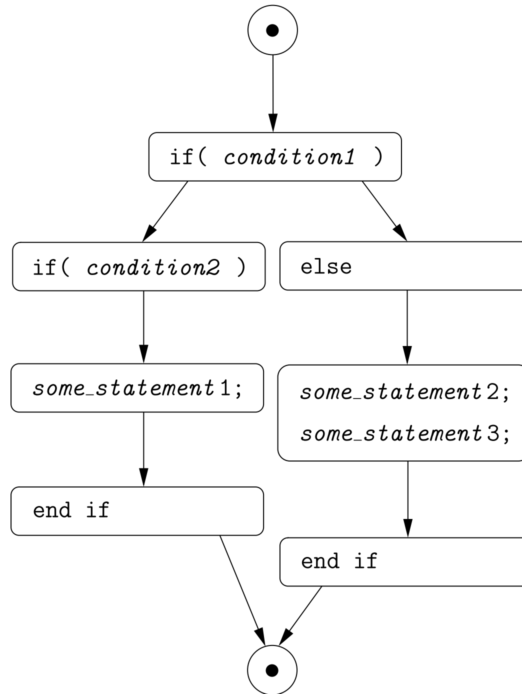


Figure 5.5: Control flow graph of the example code within figure 5.4

```
int id;                           /*!< Node id */
struct stmt *first,               /*! <First and */
            *last;                /*! <Last statement of sequence */
int num_preds;                    /*! <Number of predecessors */
int **preds;                      /*!< Predecessors */
int num_succs;                    /*!< Number of successors */
int **succs;                      /*!< Successors */
struct branch *branch_tree;       /*!< Used during analyses */
```

Table 5.9: Structure of a node within the control flow graph

Setting of the predecessor nodes for each node of the control flow graph can be done straight forward. Because every node that has received the actual node as successor can be set as predecessor. Last, a short introduction to the corresponding data structure that is used to store a control flow graph node, is given. Table 5.9 shows all elements of the data structure that are used within the source code. Before all predecessor and successor nodes will be set, every node gets a unique id. This id number is stored within the first line of the data structure and is used to store the

predecessors and successors afterwards. To make the access of these elements much more clearly and easier to read, some macros like

```
#define CFG_GET_SUCC_ID(NODE_REF,INDEX) ( *( NODE_REF->succs[INDEX] ) )
```

are used. Now, all necessary data are ready for the analysis, which is described within the following section.

5.6 Loop bound analysis

This section shows some of the implementation details of the methodology described in chapter 4. Within the first subsection a short overview of some implementation details about the construction of the branch tree is given. The following subsection explains how and which values for each branch must be calculated. Last, the calculation of the loop iteration bound is described.

5.6.1 Finding all branches that can influence the number of loop iterations

As described within section 4.2.2, all branches that can influence the number of loop iterations are found within two steps. First, all branches that could directly influence the number of iterations are identified and afterwards, all branches that could indirectly influence the number of iterations are identified. A bitvector is used to mark all identified branch nodes[5]. This bitvector is an array with the same size as the number of nodes of that part of the control flow graph that is representing the loop. At the beginning, every element of this array are initialised with '0'. When a branch should be marked the corresponding element of the bitvector must be set to '1'. The main piece of code of the algoritm within the whole implementation, which has been introduced within figure 4.3 on page 27 at the previous chapter, is the function

```
bool is_postdominated_by( struct func *pFn, int nNode, int nPostDom )
```

and is used to find out if a node with *id*[6] `nNode` is postdominated by the node with *id* `nPostDom`. Within the first step of algorithm 4.3, a node is marked if a call of the previous introduced function returns *TRUE*. As arguments, the reference to the actual function that contains the loop, the *id* of one of the successors of the actual branch node and as postdominator the *id* of the node that contains the loop header, are used. The second reason to mark a branch node within the first step is, if one the successors is located outside the loop. A node can be identified as located outside of the loop when his *id* is higher as the *id* of the last loop node. Because the node *id*'s have been assigned arising during the construction of the control flow

[5] *branch node*: a basic block or control flow graph node where the last statement of this node has more than one successor

[6] *id* is an element of the structure shown on table 5.9

graph. When all branches that can indirectly affect the number of iterations should be found, the function `is_postdominated_by(...)` has to be called for every node where the corresponding flag is not set within the previous described bitvector. As argument for the postdominator node, one the nodes where the corresponding flag has been set is taken. If the function returns *TRUE*, the flag of the evaluated branch node is set to '1'. This procedure is repeated until no new flag has been set within the bitvector. Table 5.10 shows, how the return value of the function `is_postdominated_by(...)` is calculated. The first case within this table means that a node cannot be postdominated by a previous defined node. The loop header has an *id* less (*FOR* of *WHILE*) or equal (*DO-WHILE*) as the loop end (=last loop node). Therefore, the return value is *FALSE* if an *id* greater as the *id* of the last loop is reached. Otherwise, if the loop header is reached without looking (e.g. *CONTINUE*) for, this path is not postdominated from a node with *id* `nPostDom` (third case of table 5.10). The next case means that a node cannot be reached from the end of the loop body and therefore the return value is false again.

Condition	*Return value*
nPostDom != *id* of the loop header && nNode > nPostDom	*FALSE*
nPostDom == *id* of the loop header && nNode > nEnd	*FALSE*
nPostDom != *id* of the loop header && nNode == *id* of the loop header	*FALSE*
nPostDom != nEnd && nPostDom != nHeader && nNode == nEnd	*FALSE*
nNode == nPostDom	*TRUE*
All previous conditions are *FALSE* and one of the successor nodes of `nNode` is not postdominated by `nPostDom`	*FALSE*
Everything else	*TRUE*

Table 5.10: Return values of the function `is_postdominated_by(...)`

The only exception is the loop header, which will be reached from the last loop node. But every node postdominates itself and therefore case five returns *TRUE*. If all of the previous cases are not true, then this function is called recursive for each successor. If one of these paths is not postdominated by `nPostDom`, then the

node itself is not postdominated `nPostDom` and the whole function returns *FALSE*. If a direct successor of a node is identified by the previous described function as be postdominated by the loop header or be postdominated by a node outside the loop, then the marker for this node is set within the bitvector. Within the second step of algorithm 4.3, all nodes that can indirectly influence the number of loop iterations, are found with use of this bitvector. If a node is not marked and all its successors are postdominated by different marked nodes than the marker for the node itself is set. The algorithm stops if no new marker of the bitvector has been set.

5.6.2 Construction of the branch tree

After identifying all nodes that can directly or indirectly infect the number of iterations, a directed acyclic graph will be created with all identified nodes. The graph must be acyclic to avoid endless loops during path analysis. Before describing details of the construction of the branch tree, the elements of the data structure for a tree node will be listed within table 5.11 and explained for short at the following.

```
struct cfg_node *node;           /*!< Pointer to a node containing
                                      the branch */
enum branch_node_type type;      /*!< Type of the node */
struct var_info *var_info;       /*!< Information about induction
                                      variable */
int num_preds;                   /*!< Numer of predecessors */
struct branch **preds;           /*!< Pointer to the predecessors */
int num_succs;                   /*!< Number of successors */
struct branch **succs;           /*!< Pointer to the successors */
```

Table 5.11: Structure of the elements within the branch tree

The pointer "`*node`" is a reference to the corresponding node within the control flow graph which contains the actual branch. The "`type`" is used to distinguished between real branch nodes and pseudo nodes which represents *BREAK* and *CONTINUE*. The pointer "`*var_info`" refers to a structure that stores all information about the condition of the branch. The next elements of the table are used to store all predecessor and also all successor nodes within the branch tree.

As preparatory work for the following construction, a branch tree node must be created for every branch of the previous set bitvector with type "*B_NODE*". The construction of the tree starts with all the branches where one of its successors is outside of the loop. In this case, a pseudo branch tree node with type "*B_BREAK*" and no successors will be created. Every node that has an successor outside the loop is pointint to this created *BREAK* node as successor afterwards. It is important

for the bottom up analysis within the following sections that all nodes refers to the same terminal node. Now, all nodes with set mark within the bitvector are connected together. For the root node (=loop header), the first successor (successor, if the condition is true) is the first branch node where the mark is set within the bitvector. And the second successor is the *BREAK* node. For all other nodes, the first successor (if not already set to the *BREAK* node) is the first branch node where the mark is set and which postdominates the first successor[7] of the branch. If there is no such successor branch node found (where the mark is set) a pseudo node with type "*B_CONTINUE*" is set as successor. The second successor is set in the same way. Figure 4.5 on page 29 is showing a graphical representation of such a branch tree after construction. If some branch contains a non single branch condition (e.g. ``if(*cond1* && *cond2*)") or if the branch statement has the type "*S_SWITCH*" the branch tree is expanded as described within section 4.2.3 on page 29. The pointer "`*node`" of the splited up nodes refers to a copy of the original pointer target. But the condition of the copy is replaced with the corresponding single condition. Therefore, all successors within the control flow graph can be evaluated in the same way during the following analysis.

5.6.3 Calculation of information about each branch condition

In section 4.2.4 on page 31, a branch is classified as *known* if all information, which is necessary to determine the number of iteration when the branch changes its direction could be calculated. Table 5.12 lists the first half of the elements of the structure *var_info*, which are stored for each branch within instances of the structure from table 5.11 as target for the pointer "`*var_info`". The meaning of each of this elements are described within table 4.1 at chapter 4 on page 32. If the value of one of these members could not be calculated, the branch is classified as unknown for the following analysis. For the elements *limit* and *initial*, there exits additional minimum and maximum values. These additional elements are used if the value of the element itself could not be extracted from the parsed source file and user defined annotations for the minimum and maximum value had been placed within the source file (see section 5.6.4 for differences within determining the minimum and maximum loop iterations). The value for "`*variable`" and the relational operator "`relop`" could be directly extracted from the branch condition if it is of the type "*variable operator limit*". The value of "`limit`" could be a constant, a variable or an expression like "*expression operator expression*". Where the *operator* could be '+', '−', '∗' or '/'. If the "`limit`" is a constant it can be stored straight forward. Within the case that is a variable (e.g. a condition like "*variable relop variable*"), it must be guaranteed that this variable is always constant for this loop. So the limitation within this work is, that there must be an assignment with a constant to this variable in the last basic block before the loop body starts. It is also important not only to evaluate *expression statements* but also pay attention to side effects

[7] This is the first node which follows the branch within the control flow graph

within sub-expressions (e.g. *array[variable++]=x;*).

`struct symbol *variable;`	*/*!< Pointer to the variable */*
`bool known;`	*/*!< TRUE if expression is known, else FALSE */*
`double limit;`	*/*!< The value being compared to the variable */*
`double limit_min;`	*/*!< Value bounds can be inserted within the source code */*
`double limit_max;`	*/*!< Value bounds can be inserted within the source code */*
`enum expr_type relop;`	*/*!< Operator used to compare the variable and the limit */*
`double initial;`	*/*!< Value of the variable when loop is entered */*
`double initial_min;`	*/*!< Value bounds can be inserted within the source code */*
`double initial_max;`	*/*!< Value bounds can be inserted within the source code */*
`double before;`	*/*!< Amount of change of the variable before reaching the branch */*
`double after;`	*/*!< Amount of change of the variable after reaching the branch */*
`enum adjust adjust;`	*/*!< Adjustment value for difference of relational operators */*

Table 5.12: First half of the structure *var_info*

And additional, every path through the loop must be evaluated to guarantee that the value of this variable does not change within the loop.

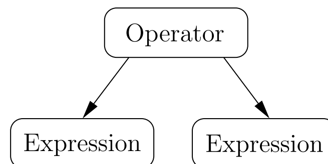


Figure 5.6: Graphical representation of an expression tree

If the "`limit`" is an expression it has been stored as expression tree during parsing as shown within figure 5.6. Therefore, the value can be calculated recursive for the whole expression. If a variable is inside of the expression, it must be handled in the

same way as described before. If there are multiple variables inside the expression and one of them cannot be bounded, the "`limit`" itself cannot be bounded and the branch becomes *unknown*. For the control variable (stored as "`*variable`"), the initial value, the amount of change before the branch is reached and the amount of change after reaching the branch must be calculated. The initial value is determined in the same way as a single *variable limit* by searching constant assignments within the basic block before the loop starts. Before the amount of change can be calculated, it must be proofed that it is constant within every path. This is done by two nested loops. The outer loop iterates over all statements of the loop within the linked statement list. If such a statement contains an assignment to a control variable then the inner loop iterates over all nodes of the control flow graph. If a node has the loop header as successor it must be dominated by the node which contains the assignment. Otherwise, the amount of change is not constant on every path. To determine if a node is dominated by another node, every successor must evaluated recursive. If a successor node is the dominator node, then this path is dominated by this node. Otherwise, every successor of the successor node must be evaluated. If the loop header is reached before reaching the dominator node then this path is not dominated by the dominator node. A branch must be classified as *unknown* if the amount of change is not constant on every path. The amount of change before reaching the actual branch can be calculated by following a path until the loop header is reached. It must be constant for every path and therefore always the first predecessor can be followed. For every node all corresponding statements and their expression trees must be evaluated. Only assignments of the form "$variable_x = variable_x + constant$" are allowed. Therefore, the amount of change within each loop iteration is a summation of all constants within the assignments. For the calculation of the amount of change after reaching the actual branch the path of the first successor is followed in the same way as described before, until the *loop end* is reached. And the last value "`adjust`" can be calculated straight forward using equation 4.1 on page 32.

5.6.4 Determination of the minimum and maximum number of loop iterations

When the information collection for each branch is finished, then the last two remaining steps of the analysis are

1. determining the ranges of when each of the branches of the branch tree can be reached and
2. determining the minimum and maximum number of loop iterations.

All elements that must be calculated during these two steps are stored within the second half of the structure "`var_info`", which is part of the structure listed in table 5.12. The second half of the previous mentioned structure "`var_info`" is listed within table 5.13 and will described for short within the following lines. The first

of the remaining two steps is calculated in top-down and the second in bottom-up order. Within bottom-up order, all values of the current branch node could not calculated until all successors of the node within the branch tree have been calculated. To let the algorithm know when the values of all successor nodes have been calculated, the "`set_counter`" is incremented for each calculated successor. The elements "`edge_range_*`" and "`edge_exit_*`" are all available as pairs for the first and the second successor. The method behind of how this values are calculated has been introduced within section 4.2.5 and 4.2.6 and will described further within this section. The implementation has been simply realised with *IF-ELSE* structures based on this method.

```
int set_counter;              /*!< Used to know if all predecessor values
                                   are set */
double node_range_min;        /*!< First iteration on which it is possible to
                                   execute node */
double node_range_max;        /*!< Last iteration on which it is possible to
                                   execute node */
double node_exit_min;         /*!< First iteration when this node may lead
                                   to a break */
double node_exit_max;         /*!< First iteration when this node must lead
                                   to a break */
double edge_range_min1;       /*!< Lowest loop iteration when the first edge
                                   can be reached */
double edge_range_max1;       /*!< Highest loop iteration when the first
                                   edge can be reached */
double edge_range_min2;       /*!< Lowest loop iteration when the second
                                   edge can be reached */
double edge_range_max2;       /*!< Highest loop iteration when the second
                                   edge can be reached */
double edge_exit_min1;        /*!< First iteration when the first edge can
                                   be reached */
double edge_exit_max1;        /*!< First iteration when the first edge must
                                   lead to a break */
double edge_exit_min2;        /*!< First iteration when the second edge
                                   can be reached */
double edge_exit_max2;        /*!< First iteration when the second edge
                                   must lead to a break */
```

Table 5.13: Second half of the structure *var_info*

The next interesting detail of the implementation is the realisation of the determination of the minimum and maximum loop iterations using annotations for upper and lower bound of variables. Because if there are an upper and also a lower bound with different value, then it is not known if the upper bound or the lower bound, or maybe both, could influence the number of iterations. Therefore, the iteration bounds must be calculated for both values. The result is an union of both ranges, where the smallest value of the lower bound and the biggest value of the upper bound must be taken. If there are more than one unknown variables for which value bound annotations have been provided, the iteration bound must be calculated for each minimum and maximum value combination. This fact leads to the problem of how to calculate each possible combination within the efficientest way. The minimum and maximum values could be represented as '0' for the minimum and '1' for the maximum value. This leads to 2^n possible combinations with n as the number of annotations with different upper and lower bound. To represent these combinations, a binary bitvector with n elements can be used. The simplest implementation method would be to implement a binary counter that counts from “`0 0 0 ...0`” up to “`1 1 1 ...1`”. But the disadvantage of this simple method is the relatively high calculation overhead. Because before setting the next higher priority bit to '1' all lower priority bits down to the least significant bit must be reset to '0'. For a large number of n this is very inefficient. To avoid this unnecessary calculation overhead this bitvector is calculated recursively until all bits are set. The calculation starts with an uninitialised bitvector of the *size* 'n':

x	x	x	...	x	x	x
n	n-1	n-2	...	3	2	1

Afterwards, a *set* function, with this vector and the index of the highest priority bit as arguments, is called. This function sets the bit with the given index first to '1' and calls itself with this changed vector and the by one incremented index as arguments afterwards. After this first function call, the bitvector has been changed to

1	x	x	...	x	x	x
n	n-1	n-2	...	3	2	1

and

0	x	x	...	x	x	x
n	n-1	n-2	...	3	2	1

and the new index is now '$n-1$'. The same procedure is done once more but now the bitvector is set to '0' at the actual index position. This recursive function calls is repeated until the index position reaches 0. Now the algorithm knows that all positions have been set. And the next step is to set the input data for the following analysis according the previous calculated bitvector. If the value for the corresponding variable is set to '1', the higher annotation value is taken for initialisation and vica versa. For big loops with many unknown branches and corresponding annotations within the code, it must taken into account that the calculation time also increases by the previous announced factor 2^n.

After the ranges for all loops within a source file have been calculated, the results must be inserted within the source file. To avoid changes of the original source file, it is copied and stored with the same name; but now with additional leading "__". The annotation starts after the closing round ')' brace of *FOR* and *WHILE* loops or after the *DO* statement, followed by a blank space, of *DO-WHILE* loops. Therefore, the overall style of the source is still remaining unchanged after the insertion. As mentioned within the previous chapter, this work has been implemented compatible with the programming language WCETC, introduced in [Kir02]. Within this language, the syntax for the annotation of the loop bound is

WCET_LOOP_BOUNDS(*variable name*, *lower bound*, *upper bound*)

and has been followed within this work.

Chapter 6

Evaluation

This chapter presents the possibilities and the limits of the analysis method and the actual implementation. Every characteristic property is illustrated with a small example loop and also supplemented with a short explanation. More complex loop bound calculations are shown within section 6.4. The style of used annotations for the value bound of some variables are compatible with the language WCETC from [Kir02].

6.1 Types of loops that can be successfully bounded

Within this section, it is described which types of loops this actual implementation is able to calculate. There are small examples listed to made the explanations more clearer and understandable. Figure 6.1 shows a simple loop that can be calculated straight forward. But is shows some of the important limitations of the method.

```
for( i=0; i<100; i++ ){
   some_statement;
}
```

Figure 6.1: Example loop 1 that *can* be bounded by the algorithm

To calculate when a branch changes its direction, the initial value for the induction variable must be given as constant. As relational operator only “<”, “<=”, “>” and “>=” allowed. With some additional limitations also “==” and “! =” are allowed. The limit of the condition must be constant for every loop execution. And the next important limitation is that the amount of change for each induction variable must be constant within every loop iteration. If this requirements are fulfilled there could also be multiple exits within the loop. Within figure 6.2, an example

loop is depicted, which has two possible exits. There is no limitation regarding the number of the exits. The complexity for loops without user assertions is $O(n)$.

```
for( i=0, j=2; i<1000; i++, j+=2 ){
   some_statement;
   if( j > 4000 ){
      break;
   }
}
```

Figure 6.2: Example loop 2 that *can* be bounded by the algorithm

But also if some branches of the loop are unknown, an iteration bound could be calculated. The range of this iteration bound could be larger compared to loops with known branches. Within the worst case the range could be between 0 and ∞ and therefore has no expressiveness. The iteration range of the example loop within figure 6.3 has been determined by the algorithm between 1 and 301. The inner branch has been marked as unknown because there could no initialisation value determined. And therefore, the lower iteration value was set to 1 for the case that this branch condition is fulfilled within the first iteration.

```
extern int j;
for( i=0; i<300; i++, j++ ){
   if( j>200 ){
      break;
   }
}
```

Figure 6.3: Example loop 3 that *can* be bounded by the algorithm

Such less accurate calculations could be minimised with value bound annotations for each unknown variable. Figure 6.4 shows the same example code as figure 6.3 but now with an annotation for the variable j. This annotation minimizes the loop iteration bound from [1...301] to [1...102]. But is must also be mentioned that annotations results in performance lost within big loops with many annotations. Because the calculation must be done with all possible combinations of these values. The calculation result is an union of all calculated ranges. Therefore, 2^m calculations must be performed, if m is the number of annotations with different upper and lower bound.

```
extern int j;
for( i=0; i++, j++ ){
   WCET_VALUE_BOUNDS( j, 100, 300 )
   if( j>200 ){
      break;
   }
}
```

Figure 6.4: Example loop 4 that *can* be bounded by the algorithm

The next code example, depicted in figure 6.5, shows a loop with an equality operator. Within the case of equality operators, three additional requirements must be fulfilled. First, such a branch must be included within every path that ends in a back edge. Second, one of the outgoing transitions must lead to a *BREAK*. Last, equation 4.1 on page 32 must be result in an integral value. Otherwise, this equation will never changes its direction and the loop could potentially be unbounded.

```
for( i=0; i!=100; i++, j+=2 ){
   if( j==200 ){
      break;
   }
}
```

Figure 6.5: Example loop 5 that *can* be bounded by the algorithm

6.1.1 Grammatical description of valid expressions

As described in previous sections, expressions with alternative internal flow are translated into into *IF-ELSE* constructs. These translations are necessary to create a valid *control flow graph*. But unfortunately, not all allowed expressions of the language C can be translated into such *IF-ELSE* constructs. Based on the *YACC* grammar of the language C (all found *YACC* grammar descriptions were nearly equivalent) an expression can be translated into a single '`assignment_expression`', or into comma separated '`assignment_expression`'s. The *YACC* grammar rule for an assignment expression is shown in figure 6.6. This '`assignment_expression`' can

be translated into a list of '='-separated 'unary_expression's with a conditional expression at the end.

```
assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;
```

Figure 6.6: YACC replacement-rule for an assignment-expression

If one of these 'unary_expression's is set between round braces, it again can be a new expression. This circumstance leads to a possible expression as shown in figure 6.7 b) that cannot translated hierarchically into *IF-ELSE* statements. Therefore, the grammar must be restricted in that way that at the bottom of these unary expressions must be an identifier (variable). A valid example of such an expression is shown in figure 6.7 a).

$a = b = (c > 1)?2 : 3;$	$a = (d > 1)?b : c = (d > 1)?2 : 3;$
a) Vailid expression	**b) Invalid expression**

Figure 6.7: Examples of valid and invalid expressions

The next step down within the grammar hierarchy leads to conditional expressions. The original replacement rule says that the first alternative after the '?' also can be an expression.

```
conditional_expression
    : logical_or_expression
    | logical_or_expression '?' assignment_expression ':'
                                        conditional_expression
    ;
```

Figure 6.8: Modified YACC replacement-rule for an conditional-expression

In that case, comma separated assignments are possible. But the algorithm is only able to handle single assignment expressions as shown in figure 6.9 a). Therefore, the replacement rule has been modified as shown in picture 6.8 to avoid expressions that cannot be handled like 6.9 b).

$j = (c > 1)?a = b = 2 : a = b = 3;$	$j = (c >= 1)?d* = 3, e = 4 : 4;$
a) Valid expression	**b) Invalid expression**

Figure 6.9: Examples of valid and invalid expressions

The next problem of the original grammar rules lies within the selection statements. A typical expression for the condition that can be handled is shown in 6.11 a). It is possible to translate this *IF* statement iterative into an *IF-ELSE* construct with only single conditions. Such valid conditions can be produced by the grammar rules shown in figure 6.12 and 6.13.

```
selection_statement
    : IF '(' logical_or_expression ')' statement
    | IF '(' logical_or_expression ')' statement ELSE statement
    | SWITCH '(' identifier ')' statement
    ;
```

Figure 6.10: Modified YACC replacement-rule for a selection-statement

An expression that cannot be translated by the algorithm into corresponding *IF-ELSE* statements is shown in 6.11 b). Therefore, the grammar replacement rule for selection statements in figure 6.10 has been modified so that only 'logical_or_expression's are allowed.

`if(( a<=5 )||( b>6 ))`	`if( j<=( c>=1 )?d++:4 )`
a) Valid expression	**b) Invalid expression**

Figure 6.11: Examples of valid and invalid selection statements

Most all quite common expressions can be expressed in that way. Additionally, the expression of a *SWITCH* statement has been changed to an identifier. A 'switch(a){ case 5: ... }' statement construct can be changed to *IF-ELSE* statements like 'if(a==5){ ... ' by following this replacement rule.

```
logical_or_expression
   : logical_and_expression
   | logical_or_expression '||' logical_and_expression
   ;
```

Figure 6.12: YACC replacement-rule for a logical-or-expression

```
logical_and_expression
   : inclusive_or_expression
   | logical_and_expression '&&' inclusive_or_expression
   ;
```

Figure 6.13: YACC replacement-rule for a logical-and-expression

At last, to allow grouped expressions like '(a<=5 || a>8) && b<10' for the condition of a selection statement, the replacement rule of a primary expression at the bottom of the hierarchy has been modified as shown in figure 6.14. The replacement of '(expression)' by '(logical_or_expression)' makes it possible to define also such, with round braces, grouped expressions. But avoids expressions that cannot be translated.

```
primary_expression
   : IDENTIFIER
   | CONSTANT
   | ( logical_or_expression )
   ;
```

Figure 6.14: Modified YACC replacement-rule for a primary-expression

6.2 Types of loops that cannot be successfully bounded

As it had been shown within the previous section, the described methodology has also some limitations. During these section, some examples are used to show the main limitations that could made the iteration bounding impossible.

- **Non-constant amount of variable change:** The example within figure 6.15 violates the paradigm that the amount of change must be constant within every loop iteration. Theoretically, for a human user it is possible to calculate the number of iterations for this loop. But for the algorithm it is not possible, because it could not calculate when the loop header changes its direction. Because this is the only branch that could lead to a *BREAK*.

```
for( i=0, j=0; i<20; i++ ){
   if( j>10 ){
      i++;
   }
}
```

Figure 6.15: Example loop 1 that *can not* be bounded by the algorithm

- **Function calls with variable references:** Within the following example, the amount of change for the variable i seems to be constant for each loop iteration. But the function call gets a reference to this variable and maybe could change its value. Therefore, it is no guarantee that this value is left unchanged and the algorithm must handle the condition of the loop header as unknown.

```
int i=0;
do{
   i++;
   some_function( &i );
} while( i<100 );
```

Figure 6.16: Example loop 2 that *can not* be bounded by the algorithm

- **Invalid induction variable updates:** The example within figure 6.17 shows why the assignments within a loop are limited to "*variable = variable + constant*". Because the induction variable i changes its sign within every iteration

of the loop and therefore has no constant growth. But also when the variable value itself would continually grow, the amount of change also depends on the variable itself and the amount of change is not constant too.

```
int i=1;
while( i<20 ){
   i*=-3;
}
```

Figure 6.17: Example loop 3 that *can not* be bounded by the algorithm

- **Nested loops:** Another group of loops that must be excluded from the calculation are nested loops. There must also be distinguished between different types of nested loops. Nested loops that are dependent on the enclosing or inner loop and such that are independent. The first group is complicated because the number of iterations must be calculated as a whole. Within the second group, the loops could in principle be calculated independent, but the sub loops inside leads to endless loops during the path analysis. Therefore, nested loops are not supported in general within the current version. At last, within figure 6.18 on the right side (b)), an example is shown that could be a source of an endless loop during path analysis.

```
for( i=0; i<20; ){
   if( i<30 ){
      some_statement;
      goto LABEL1;
   }
   LABEL1:
      i++;
}
```

a)

```
for( i=0; i<20; i++ ){
   if( i<=10 ){
      LABEL2:
         some_statement;
   } else {
      goto LABEL2;
   }
}
```

b)

Figure 6.18: Example loops 4 a) and 4 b) that *can not* be bounded by the algorithm

Within the example on the left side (a)) it is obvious that there is no sub-loop; because the target of the *GOTO* statement is a node with a higher index. But if the successor of a node that contains a *GOTO* statement has an index less than the node itself, it is for the algorithm not possible to detect,

if the control flow graph contains a sub-loop or not. This is the case in the example on the right side. During the construction of the control flow graph, the node that represents the body of the *IF* statement gets a lower index as the node representing the body of the *ELSE* statement, because it is defined before. So the target of the *GOTO* statement is a node with a lower index and does not form a loop. But within the current implementation is not possible to distinguish *GOTO* statements that form a loop and such that do not. Therefore, if a loop contains a *GOTO* statement, the number of iterations could not be calculated if the target node is not outside the loop or is also not a node with a higher index.

- **Multiple conditions with negation operator:** The example loop in figure 6.19 shows another type of loops that cannot be handled by the current implementation. Because conditions with negation operators cannot be translated into multiple *IF-ELSE* statements with single conditions in the same way as described within the previous chapters for conditions without negation operator. A negation operator requires some kind of algebraic transformations, which are not implemented within the current version.

```
while( !((i<=5) && ((j>20) || (j<5)) ){
   some_condition;
   i++, j++;
}
```

Figure 6.19: Example loop 5 that *can not* be bounded by the algorithm

6.3 Loops that maybe are bounded incorrectly

Within this section, some examples are shown that maybe will be bounded false by the algorithm. Because there a some circumstances that can not detected by the algorithm. These circumstances must be excluded by definition.

- **Aliasing:** One of these problems is called *aliasing* and is shown within figure 6.20. If there is a pointer definition within a node of a control flow graph that had a lower index as the last node before the loop body starts, than it could refer to a basic induction variable and the analyser does not take notice of it. Because the actual implementation evaluates the parse tree only local at function level. But these reference assignments could also be placed outside of the function, global or within another file. For example, the expression statement “`*pt=5;`” within the figure 6.20 not only changes the value of the variable “`*pt`” itself.

```
int *pt = i;
   ⋮
i=0;
*pt=5;
while( i<100 ){
   i++;
}
```

Figure 6.20: Example loop 1 that *maybe* is bounded false

Because the pointer "*pt" and variable "i" refers to the same memory location. Therefore, the value of the variable "i" changes its value too. But during local analysis, this behaviour could not detected and therefore must be excluded by definition.

- **Variable overflow:** A further problem of the algorithm could be variable overflow. All numbers are stored during the lexical analysis with type *signed double*. For example, an assignment of "0xff'' to a signed character variable with 8 bits memory will result in "-1". But within the token analysis the function

  ```
  double strtod( const char *nptr, char **endptr);
  ```

  is used to convert a read string into a number. This function returns "255" as result for the string "0xff" that deviates from the overflow result during an execution of the source program. Another example is shown in figure 6.21 a) on the left side. The calculation result of the character variable "c3" will be "4". Because the range for 8 bit *unsigned chars* lies between "-127" and "127". But within the analyser the variable is stored with type *double* as shown on the right side. There is no overflow and therefore the calculation result of the analyser is *incorrect*. But this overflow problem will be fixed in the next version release of this analyser.

6.4 Examples

Within this section, the calculation results of some example loops are shown. To make the results more readable, a branch tree with numbered edges and nodes for each example has been inserted. The calculated results are explained separately using this numbers.

```
char c1 = 126;
char c2 = 126;
char c3 = c1 * c2;
```

a)

```
double f1 = 126;
double f2 = 126;
double f3 = f1 * f2;
```

b)

Figure 6.21: Different calculation results, depending on the variable type

6.4.1 Example loop 1

The first example demonstrates the calculation of a *DO-WHILE* loop with multiple exits. The C language code of this example is depicted within table 6.22. It contains one *IF* statement with multiple conditions. This statement must be split up into two separate *IF* statements with only single conditions (see also section 4.2.3 on page 29 for further details).

```
int i=0, j=1;
do{
   if( j>1000 ){
      break;
   } else {
      if( j<500 ){
         some_statement;
      }
      if( j>500 && i<150 ){
         break;
      }
   }
   j += 10;
   i++;
}while( i<5000 );
```

Figure 6.22: Calculation example loop 1

The corresponding branch tree after branch expansion is depicted within figure 6.23. The required information of table 4.1 on page 32 could have been calculated for every node of the branch tree. Therefore all nodes are marked as *known*. Within the following lines the calculated results of the ranges when a node can be reached (within '[' and ']' brackets) and the number of iterations (within '<' and '>' brackets) when a nodes leads to a loop *exit* are listed.

	Range	Exit
Node 1	$[1 \ldots \infty >$	$< 51 \ldots 51 >$
Outgoing edge 1	$[1 \ldots 4999]$	$< 51 \ldots 51 >$
Outgoing edge 2	$[5000 \ldots \infty]$	$< 5000 \ldots 5000 >$
Node 2	$[1 \ldots 4999]$	$< 51 \ldots 51 >$
Outgoing edge 3	$[1 \ldots 100]$	$< 51 \ldots 51 >$
Outgoing edge 4	$[101 \ldots 4999]$	$< 101 \ldots 101 >$
Node 3	$[1 \ldots 100]$	$< 51 \ldots 51 >$
Outgoing edge 5	$[1 \ldots 50]$	$< \infty \ldots \infty >$
Outgoing edge 6	$[51 \ldots 100]$	$< 51 \ldots 51 >$
Node 4	$[51 \ldots 100]$	$< 51 \ldots 51 >$
Outgoing edge 7	$[\infty \ldots \infty]$	$< \infty \ldots \infty >$
Outgoing edge 8	$[51 \ldots 100]$	$< 51 \ldots 51 >$

The outgoing edges of *node 4* are leading to a *CONTINUE* (edge 7) and to a *BREAK* (edge 8) node. Therefore, the result when *node 4* leads to a break is 51 for the upper and also for the lower bound. Because iteration 51 is the only one when an exit can be reached. Next, as we can see at figure 6.23, both outgoing edges of *node 2* are may lead to a *BREAK*.

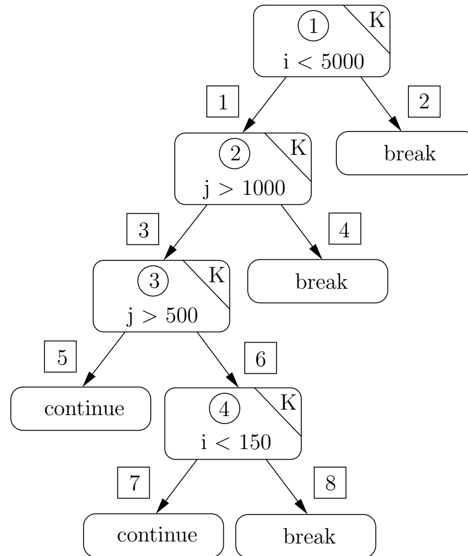


Figure 6.23: Branch tree for the example loop of figure 6.22

A look at the calculation results shows that both exits can be reached. But if a *BREAK* has been reached once, the loop execution stops and not further exit can

be reached. Therefore, the calculated result when *node 2* leads to a break is also 51 (the smallest value of both outgoing edges) for the upper and lower bound. The same calculation model leads to the result that *node 1*, which represents the loop header, will be executed exactly 51 times.

6.4.2 Example loop 2

The following example loop contains a branch where the number of iterations when it changes its direction could not be determined. Unknown branches often leads to less accurate calculation results. Therefore an assertion (see also section 4.3 for further details) for variable *j* with upper and lower bound has been placed before the branch definition within figure 6.24.

```
extern int j;
for( i=0; i!=100; i++ ){
   WCET_VALUE_BOUNDS( j, 20, 60 )
   if( j > 50 ){
      goto LABEL1;
   }
}
LABEL1:
   ⋮
```

Figure 6.24: Calculation example loop 2

Figure 6.25 shows the corresponding branch tree of the example loop within figure 6.24. The target of the *GOTO* statement is outside of the loop and therefore replaced by a *BREAK* node. With use of the annotations, every node could be calculated and therefore also *node 2* is marked as *known*.

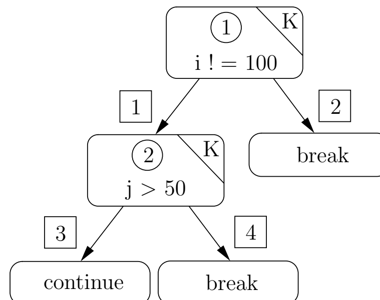


Figure 6.25: Branch tree for the example loop of figure 6.24

As described within the sections 4.3 and 5.6.3, the results must be calculated for all possible combinations of assertions. Within the previous described example there is only one annotation that leads to two independent calculations. The first calculation uses the value *20* for the initial value of the variable j. This calculation leads to the results of the following table:

	Range	Exit
Node 1	$[1 \dots \infty]$	$< 100 \dots 100 >$
Outgoing edge 1	$[1 \dots 100]$	$< \infty \dots \infty >$
Outgoing edge 2	$[101 \dots \infty]$	$< 101 \dots 101 >$
Node 2	$[1 \dots 100]$	$< \infty \dots \infty >$
Outgoing edge 3	$[1 \dots 100]$	$< \infty \dots \infty >$
Outgoing edge 4	$[\infty \dots \infty]$	$< \infty \dots \infty >$

Within this calculation the condition of the branch of *node 2* is always *FALSE* and the number of iterations when this branch leads to a *BREAK* is ∞. Therefore, the calculation result is *100*, which is the number of iteration when the loop header itself changes its direction, decremented by 1; because the body is not executed anymore when the loop header has changed its direction. The second calculation uses the upper bound *30* of the annotation for variable j which leads to the result of the following table:

	Range	Exit
Node 1	$[1 \dots \infty]$	$< 1 \dots 1 >$
Outgoing edge 1	$[1 \dots 100]$	$< 1 \dots 1 >$
Outgoing edge 2	$[101 \dots \infty]$	$< 101 \dots 101 >$
Node 2	$[1 \dots 100]$	$< 1 \dots 1 >$
Outgoing edge 3	$[1 \dots 100]$	$< 1 \dots 1 >$
Outgoing edge 4	$[\infty \dots \infty]$	$< \infty \dots \infty >$

For the value *30* of the variable j the loop exits after the first iteration. The end result after all variations of the annotation values is the union of all separate calculation results. As lower bound is taken '1', which is the lower number when the second calculation leads to the *BREAK*. The upper bound is taken from the first calculation, because 100 is the higher number of iteration when the loop may exit. After the calculation

```
WCET_LOOP_BOUNDS( 1, 100 )
```

is inserted as result between the loop header and the loop body within the source file for further *WCET* analysis.

Chapter 7

Conclusion

Within this thesis an implementation of a loop analyser for further *WCET* analysis has been described. It is used to analyse source code, written in C, without transformation into intermediate or assembler code. It based on a publication by Healy at al. in [HSR+00] for bounding the number of loop iterations at assembly code level. To have a tight upper and lower bound of the number of iterations of loops is necessary for statical *WCET* analysis. Traditional approaches for *WCET* analysis have mainly used annotations for the loop bound within the source code. Calculating the loop bound manually could be arduous and for complex loops also error prone.

Therefore this loop analyser tries to extract the required information out of the source code. If the loop bound could be determined, the result is written back into the source code as annotation. These annotations are compatible to the language *WCETC*, introduced in [Kir02]. Afterwards, the revised source file can be handed over to a timing analyser.

The main difference of the actual work is the level of representation. Whereas the basis publication are used for analysis at assembly code level, this thesis had been implemented to analyse code, written within the high level language *C*. This level of representation requires several transformation to bring the code in a form that can by analysed by the algorithm. As described within chapter 4 and 5, only branches with a single condition of the form '$variable_x$ *operator limit*' are allowed. This requires transformations of branches with multiple conditions, but also transformations from *SWITCH-CASE* statements into a representation with *IF-ELSE* statements. Additionally, the condition of a *FOR* statement could be empty and must be replaced by an condition that is always *true*.

Unfortunately, the types of loops that can be successfully bounded by the algorithm are restricted. These types of loops that will be bounded successfully are shown within figure 7.1 on page 80. The current implementation is only able to bound

natural[1] *C* loops constructed by *FOR*, *WHILE* and *DO-WHILE*. These natural loops can be bounded also within the case of multiple exits - figure 7.1 (a) - and in combination with annotations, also when the number of iterations is non-constant - figure 7.1 (b). A more detailed description of loops that can be bounded and possible restrictions within the calculation are given within the previous chapter.

```
for(i=0; i<10; i++){
  ⋮
  if( any_condition ){
     break;
  ⋮
  }
}
```

a) Loop with multiple exits

```
for( i=0; i<x; i++){
  ⋮
}
```

b) Loop, where the number of iterations is non constant

Figure 7.1: Types of loops that could be successful bound

7.1 Future work

Apart from loops where a tight loop bound can be calculated, there are several limitations that could prevent some loops of being successfully bounded. Although such loops are maybe of one of the types as described within figure 7.1, in some cases a calculation of a tight loop bound is not possible. The limitations of the implementation have already been described within the previous chapter. But some of these loops in principle can be calculated by the algorithm after some structural changes. One of this actual limitations are shown within picture 7.2. The input file is split up into its functions and one function is evaluated after the other. But within future versions of the actual work the code will be considered globally. Additionally, all basic blocks between the function and the loop header will be evaluated. After these changes it would also be possible to recognise most of all alias dependencies (in combination with an aliax matrix). With the exception of some special cases, alias dependencies of induction variables can be successfully handled and the corresponding loop can be bounded always correctly.

Another point of weakness of the actual implementation is the limitation to standard loops. The insertion of a search algorithm for non-standard loops as shown within figure 7.3 will be one of the topics of future versions. The difficulties within such loops are that *GOTO* statements have the power to form loops with complicated

[1] *natural* means that the loop has only a single entry point

```
int i;
int *p = i;
  ⋮
i=0;
  ⋮
*p=10;
while( i!=10 ){
   some_statement;
}
```

Figure 7.2: Alias occurrence that could influence the number of iterations

control flow graphs (e.g. two loop headers). The chosen implementation of the control flow graph is also a reason why nested loops within the current version are not supported. Therefore, a redesign of the current implementation of the control flow graph is also planed for future versions.

```
LOOP_HEAD:
   some_statement;
   if( condition ){
      goto LOOP_HEAD;
   }
```

Figure 7.3: Example of a natural *GOTO* loop

Figure 7.4 shows an example of two nested loop where the number of iterations of the inner loop is independent from the enclosing loop and vica versa.

```
for( i=0; i<10; i++ ){
   for( j=0; j<10; j++ ){
      some_statement;
   }
}
```

Figure 7.4: Example of independent nested loop

Independent in that case that no one of the depicted loops changes the induction variables of each other. The inner loop could be evaluated in nearly the same way as described within chapter 4. Additionally, it only must be guaranteed that the outer loop not changes the induction variable 'i' of the inner loop. But using the current implementation, the evaluation of the outer *FOR*-loop leads within an endless loop after reaching the header of the inner loop. Because all possible paths within a loop are followed and the last node of the inner loop points to its header and again the same paths are followed. A possible approach for further improvement will be to copy the control flow graph and cut out the inner loop. Additionally, the inner loop must be evaluated to guarantee that there will be no influence of the induction variables of the outer loop. There seems to be no limitation regarding to the deepness of the nested loops. Another type of nested loops that should be supported within later versions are depicted within figure 7.5. The number of iterations of the inner loop from these two nested loops depends on the induction variables of the outer loop.

```
for( i=0; i<10; i++ ){
   for( j=i; j<10; j++ ){
      some_statement;
   }
}
```

Figure 7.5: Example of nested loops that depends on outer loops

A possible algorithm to determine such, so called *non-rectangular loops*, had been published within [HvEW99], which is based on a previous publication by Sakellariou in [Sak97]. Additionally to this previous presented improvements, further research time will be spent into new algorithms to support supplementary types of loops.

Bibliography

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, Schweiz, first edition, 1986.

[BB06] Adam Betts and Guillem Bernat. Tree-based WCET analysis on instrumentation point graphs. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 8–15. IEEE, April 2006.

[BR05] Claire Burguiére and Christine Rochange. History-based schemes and implicit path enumeration, 2005.

[CM05] Eli D. Collins and Barton P. Miller. A loop-aware search strategy for automated performance analysis. September 2005.

[EES02] Andreas Ermedahl, Jakob Engblom, and Friedhelm Stappert. A unified flow information language for WCET analysis, 2002.

[Elm02] Wilfried Elmenreich. *Systemnahes Programmieren*. Institut für Technische Informatik der Technischen Universität Wien, Vienna, first edition, 2002.

[GE98] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs, 1998.

[GLSB03] J. Gustafsson, B. Lisper, C. Sandberg, and N. Bermudo. A tool for automatic flow analysis of c-programs for wcet calculation, 2003.

[HSR+00] Christoper A. Healy, Mikael Sjödin, Viresh Rustagi, David Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, pages 121–148, May 2000.

[HSW98] Christopher A. Healy, Mikael Sjödin, and David B. Whalley. Bounding loop iterations for timing analysis. In *Proc. IEEE Real-Time Technology and Aplications Symposium*, pages 12–21, June 1998.

[HvEW99] Christopher A. Healy, Robert van Engelen, and David B. Whalley. A general approach for tight timing predictions of non-rectangular loops.

In *Proc. Real-Time Technology and Applications Symposium, Work in Progress Session*, pages 11–14, June 1999.

[HW99] Christopher A. Healy and David B. Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proc. Real-Time Technology and Applications Symposium*, pages 79–88. IEEE, June 1999.

[Kai05] Andreas Kaiser. Verfahren zur Bestimmung der Worst Case Execution Time (WCET). Lecture notes, June 2005.

[Kir02] Raimund Kirner. The Programming Language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[Kir03] Raimund Kirner. *Dissertation - Extending Optimising Compilation to Support Worst-Case Execution Time Analysis.* Institut für Technische Informatik der Technischen Universität Wien, Vienna, first edition, 2003.

[Kop97] Herman Kopetz. *Real-Time systems. Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, New York, first edition, 1997.

[PN98] Peter Puschner and Roman Nossal. Testing the results of static worst-case execution-time analysis. *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 134–143, December 1998.

[Sak97] Rizos Sakellariou. Symbolic evaluation of sums for parallelising compilers. In *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, pages 685–690, 1997.

[WM01] Joachim Wegener and Frank Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, pages 241–268, November 2001.

[ZK04] Andreas Zeller and Jens Krinke. *Open-Source-Programmierwerkzeuge.* dpunkt.verlag GmbH, Heidelberg, second edition, 2004.