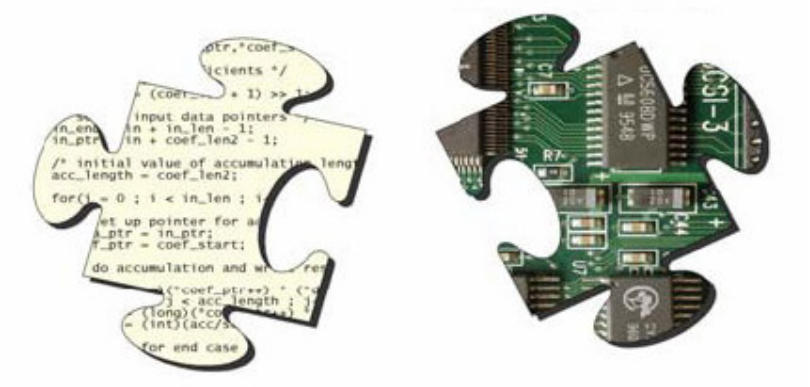
# Evaluating program flow analysis for WCET calculations at Volvo CE



A Master's Thesis by:
Dani Barkah

Department of Electronic, Computer, and Software Systems (ECS)
School of Information and Communication Technology (ICT)
Royal Institute of Technology, Sweden

# Abstract

A prerequisite for creating a safe and predictable real time computer system is to have knowledge about its timing behavior during operation. The Worst Case Execution Time (WCET) is one of the key values to consider when predicting this timing behavior, since it ensures that every task meets its deadline in the system. A missed deadline in systems with critical tasks can lead to severe consequences.

Dynamic WCET analysis is a widely used method to derive WCET value in the industry today. The method is based on using simulation and measurement techniques. The WCET value is derived by performing a set of automatic tests on the system and measure its execution time. To ensure that a safe WCET value is derived the automatic tests must cover the worst case scenario, which is tough to find when the systems increase in size and complexity.

Static WCET analysis is an alternative method, where it is guaranteed that an overestimated WCET value is obtained. The method uses mathematical models of the system hard and software to calculate WCET. Therefore, this method has been the main research subject for many research groups. One of the key components in making a safe and tight WCET value, using static WCET analysis technique, is program flow information.

The information describes the possible program flows through the program; this information can be either provided to the WCET analyzer manually, by user, or automatically, by a flow analyzer. Deriving flow information manually is done by:

- Studying and analyzing the program source code, and manually calculating loop bounds, detecting infeasible paths and making recursion bounds.
- Providing the information to a WCET analyzer, usually by writing them in form of annotations in a program language supported by the analyzer.

The steps above are time consuming and unsafe when the program complexity increases, that makes it preferable to use flow analysis to provide flow information. This has been shown in a previous study at Volvo Construction Equipment, where the WCET analysis tool aiT has been used. This work will complete the study by evaluating an automatic method, using flow analyzers to derive flow information for the analyzed code.

Flow analyzers use mathematical methods to model and derive flow information of program source code. This flow information makes deriving WCET estimates safe, simple and requires no or minimum user interaction. Therefore, Volvo CE is interested in the development of static WCET analysis tools where the flow information is obtained automatically.

This work evaluated SWEdish Execution time Tool (SWEET), a prototype WCET analysis tool built by a research group from Mälardalens University (MDH). The results are compared to the ones obtained in previous work using aiT, where the flow information is derived manually.

We have found that SWEET derived the same flow information that was derived manually, but since it is a prototype tool a lot of manual work was needed. The manual work consisted mainly of adapting the Volvo CE application to the SWEET analysis environment. This work has also contributed in development of SWEET, adding new functionality and making it more users friendly.

# Contents

**Abstract** ..... **2**

**Introduction** ..... **6**

1.1 Background ..... *6*

1.2 Thesis Focus ..... *7*

1.3 Thesis Outline ..... *7*

1.4 Abbreviations ..... *8*

1.5 Related Work ..... *8*

**Concepts** ..... **9**

2.1 Real Time Systems ..... *9*
2.2.1 Time-Triggered Tasks ..... 9
2.2.2 Event-Triggered Tasks ..... 10

2.2 Execution Time Analysis ..... *10*

2.3 Static WCET Analysis ..... *11*
2.3.1 Flow analysis ..... 11
2.3.2 Low-level analysis ..... 12
2.3.3 WCET Calculations ..... 12

2.4 Dynamic WCET analysis ..... *13*

2.5 Hybrid WCET Analysis ..... *13*

**Volvo CE** ..... **14**

3.1 Hardware ..... *15*

3.2 Software ..... *15*

3.3 Code structure ..... *15*

**SWEET** ..... **16**

4.1 NIC ..... *16*
4.1.1 Syntax ..... 17
4.1.2 Memory model ..... 18

4.2 Annotations ..... *18*

4.3 Flow Analysis ..... *19*
4.3.1 Scope Graph and Flow Facts ..... 19
4.3.2 Abstract Execution ..... 20
4.3.3 Calculation of Flow Information ..... 20
4.3.4 Flow Analysis Results ..... 21

4.4 Low-level Analysis & WCET Calculation ..... *22*

**aiT** ..... **24**

5.1 User Annotations ..... *24*

5.2 aiSee ..... *25*

**Project Tools** ..... **26**

6.1 Hardware ..... *26*

6.2 Software ..... *26*

6.3 Scripts ..... *26*
6.3.1 c2nic ..... 27
6.3.2 build ..... 27
6.3.3 cilly_to_c ..... 27
6.3.4 runsweet ..... 27
6.3.5 sweet-nic ..... 28
6.3.6 dot2pdf ..... 29

6.4 SWEET options ..... *29*
6.4.1 Type Check Annotations, -tca ..... 29
6.4.2 Slicing: -ss and -nops ..... 29
6.4.3 Value Analysis: -nova, pge and -pga ..... 29
6.4.4 Abstract Execution: -noae, -lb, -ip and –sm ..... 29
6.4.5 State Variable Value Analysis (SVVA), -svva ..... 30
6.4.6 Draw & Print, -d, -v ..... 31

**Method** ..... **32**

7.1 Source code preparation & NIC generation ..... *32*

7.1 Source code preparation & NIC generation ..... *33*
7.1.1 Function Wrapper ..... 33
7.1.2 Phase 1 Result ..... 33

7.2 Deriving SWEET Annotations ..... *33*
7.2.1 Function Wrapper - Revisited ..... 34
7.2.2 State Variable Annotations ..... 34
7.2.3 Phase 2 result ..... 34

7.3 Flow Analysis using SWEET ..... *35*
7.3.1 The General Case ..... 35
7.3.2 The Special Case ..... 36
7.3.3 Phase 3 Results ..... 36

7.4 Source Binary Generation & aiT Flow Annotations ..... *36*
7.4.1 Converting Dot to PDF ..... 37
7.4.2 Interpreting SWEET Flow Facts ..... 37
7.4.3 Writing aiT Flow Annotations ..... 38
7.4.4 Phase 4 Results ..... 39

7.5 WCET Analysis using aiT ..... *39*

7.6 Method Summary ..... *39*

**Results** ..... **40**

8.1 SWEET Annotations ..... *40*

8.2 SWEET Flow Facts ..... *41*
8.2.1 Infeasible Edges, Nodes and Paths ..... 41
8.2.2 Loop Bounds ..... 41

8.3 Analysis Time ..... *42*

8.4 aiT Annotations ..... *42*

8.5 WCET Comparison ..... *43*

8.6 aiT v1.5 vs. v2.0 ..... *43*

8.7 Additional Information ..... *43*

8.8 Problems and Solutions ..... *44*

**Conclusions** ..... **45**
9.1 Preparation Work ..... *45*
9.2 WCET Effects ..... *45*
9.3 SWEET for Volvo ..... *45*
9.4 State Variable Value Analysis ..... *45*
9.5 Future Work ..... *46*
**Bibliography** ..... **47**

# Chapter 1

# Introduction

Embedded computers has become very common in many commercial products; vehicles, airplanes, telephones and microwave ovens. Some of these computers have time-critical tasks such as controlling airbag control systems, control of anti-lock braking system (ABS) and engine control. Imagine what will happen in a car crash if the airbags inflates too late or if an airplane engine control system responds too late on pilot commands. These examples show the need of critical systems to be predictable and have a reliable behavior, since their failure has catastrophic outcomes.

Time critical tasks usually have deadlines which are very important to fulfill; if these are not fulfilled it could have disastrous effects on the system. These are a part of predictable and reliable embedded computer systems; also called hard real-time systems.

## 1.1 Background

There is several ways to make sure that the deadlines are met in hard real-time systems. One way to do it is to find WCET (Worst Case Execution Time) estimates for the tasks in the system. It is one of the most time consuming tasks for system designers while designing hard real-time systems. In practice to estimate WCET is done by measuring its execution time with different combinations of input data.

The measurement technique, also called dynamic WCET analysis, is widely used in development of real time systems today. The main drawback with using this technique is that there is no guarantee that the WCET estimates obtained are safe, i.e., the WCET is not underestimated. Furthermore, to find the test case where the WCET value is obtained becomes more difficult, when the tasks become larger and more complex.

Another method to determine these estimates is to use static WCET analysis, which is based on mathematical models of the real time systems hardware and software. This method guarantees that the WCET estimates obtained are not underestimated, thus making analysis result safer and more trustworthy. Furthermore, this method requires information about program flow, e.g., loop bounds and recursion depths to bind WCET estimates.

All the WCET estimates in this thesis are determined by the static analysis method. Several kinds of static WCET analysis tools exist in the market today; most of these require detailed flow information on possible execution paths in order to determine the longest possible path in a program. This includes upper loop bounds and execution dependencies.

Flow information is provided by the user who manually studies the code and finds it. This could be a time consuming task when the code increases in size and complexity. Not to mention that this has the possibility of being the source of error. That created the need of making flow information calculations more automatic by making the tools less user dependent.

One commercial static WCET analysis tool is aiT [18]. This tool determines the longest possible path through a program by analysing its binary file and displaying the results in a combined flow and call graph. The graph also contains the WCET estimates for the program. This tool was used to derive WCET

estimates for Volvo CE code by Daniel Sehlberg as a part of his master thesis [3], where he provided the necessary flow information manually, to aiT.

This work will evaluate another static WCET analysis tool called SWEET, SWEdish Execution time Tool. The main interest is to evaluate the flow analysis part of this tool. The flow analysis part is developed to obtain the flow information needed automatically, i.e., no code studies are needed to derive them. This makes WCET analysis work; less user dependent, able to obtain safer WCET estimates and less system development time consuming. This thesis is a step forward in the investigation of using static WCET analysis tools in Volvo CE's embedded system development.

Volvo CE uses the Rubus Operating System as a platform for their software applications. Rubus is created by Arcticus Systems [5]. Rubus is designed to maintain different kinds of real time tasks; time-triggered tasks (which runs periodically), event triggered tasks and interrupts. The applications are written in c-code and consist of many if-statements, case-statements and some loops. For the reliability and predictability purposes there is no dynamic memory handling.

## 1.2 Thesis Focus

This thesis is a project in cooperation between Volvo Construction Equipment and Mälardalen University. The purpose is to analyze the tasks in Daniel Sehlberg's thesis with a prototype static WCET tool called SWEET (SWEdish Execution time Tool), which will automatically find the flow information.

The evaluation will consist of performing flow analysis with SWEET on the task and to make a comparison with the flow information that Daniel found manually. One of the interesting aspects is to find out how much preparation work is needed in order to make the code analyzable by SWEET and to make sure that the provided input values are correct.

Daniel analyzed thirteen tasks which varied in size and complexity; most of these tasks are small and has simpler functions, but there are some programs which are complex and large. Task size and complexity measurements are presented in his thesis [3].

To summarize, the focus of this thesis is to:

- Find the number of input value annotations needed by SWEET to calculate correct flow information.
- Study the type of flow information calculated by SWEET for each task.
- Investigate the effects of this information on WCET results.
- Translate SWEET results to aiT annotation and compare it with flow information found by hand.
- Investigate the upper limits for SWEET flow analysis.
- Compare the WCET results derived by aiT 2.0 with the one derived by aiT 1.5 [3].

## 1.3 Thesis Outline

Chapter one will continue by introducing some abbreviations used in this paper. Chapter two presents the real time systems and WCET analysis concepts. Chapter three will introduce Volvo CE and the electronic systems used in their products. Chapter four will introduce SWEET and the WCET calculation method used, and chapter five will briefly introduce aiT, its annotation style and visualizing tool.

Chapter six will introduce the hardware and software used in this thesis; this includes the scripts used and the SWEET options used to analyze the tasks. In chapter seven the method used to analyze the tasks in this thesis will be described. The answers for the questions, i.e. results, from previous sections are answered in chapter eight. The conclusions drawn in this project about SWEET, aiT are presented in chapter eight. Chapter nine presents some future work and suggestion for improvements that can be done in the WCET analysis field.

## 1.4 Abbreviations

The list below presents the abbreviations used in this paper.

ABS – Anti lock Brake System
BCET – Best Case Execution Time
ACET – Average Case Execution Time
WCET – Worst Case Execution Time
Volvo CE – Volvo Construction Equipment
SWEET – SWEdish Execution time Tool
RM – Rate Monotonic
EDF – Earliest Deadline First
IPET – Implicit Path Enumeration Technique
ECU – Electronic Control Unit
CAN – Controller Area Network
TECU – Transmission Electronic Control Unit
Rubus OS – Rubus Operating System
Rubus VS – Rubus Visual Studio
NIC – New Intermediate Code
BB – Basic Block
GDL – Graphical Description Language
SML – Standard ML
CIL – C Intermediate Language
OCAML – Objective Caml
SG or sg – Scope Graph
SGH or sgh – Scope Graph Hierarchy
CG or cg – Call graph
PDG – Program Dependency Graph
SVVA – State Variable Value Analysis

## 1.5 Related Work

Beside Daniel Sehlberg's aiT evaluation at Volvo CE, Daniel Sandell's has evaluated the tool using Enea's real-time operating system OSE in his Master Thesis in 2004. A comparison between static WCET and dynamic WCET analysis has been preformed by Ola Eriksson and Yina Zhang at CC Systems [30].

Stefan Bygde's Master Thesis [13] about Abstract Interpretation explains in detail the theory behind the abstract execution concept used in SWEET. The theory behind SWEET is presented for the first time in 1998, since then a number of articles and dissertations have been written describing SWEET in more detail [9].

Two PhD theses are written by Jan Gusstafsson [32] in 2000, and Andreas Ermedahl [1] in 2003, about the method and algorithms used in SWEET. More information about SWEET can be found on the WCET research group webpage [9].

# Chapter 2

# Concepts

## 2.1 Real Time Systems

According to TechWeb [6] a Real Time System is:

"As fast as required. A real-time system must respond to a signal, event or request fast enough to satisfy some requirement. Real-time often refers to process control and embedded systems."

As mentioned above; apart from the requirement of delivering results a real time system has a time constraint which must be fulfilled. There are two types of real time systems; hard real-time system where it is crucial to system operation that the system timing requirements are *always* fulfilled. The other type is called soft real-time system, where it is *preferred* that the timing requirements are fulfilled.

An example of hard real-time is the flight control system in airplane. It is very crucial that the airplane's control system never fails to meet a certain deadline. A soft real-time system example is internet video (video streaming). If a frame is dropped (system fails to meet deadline) it would not have disastrous impact on system operation, i.e., the user would survive with a missed frame but not a failure in a flight control system.

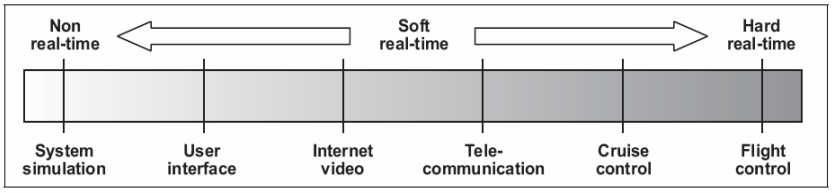



Figure 2.1: Electronic systems sorted by their real-time requirement, [1].

A real time system consist of program entities called tasks. Each task usually performs a specific operation. A task can control the fuel consumption; another one may control the engine oil level, etc. Tasks are sorted in two categories, time-triggered and event-triggered.

### 2.2.1 Time-Triggered Tasks

This kind of task has often a release time, a period, a deadline and a WCET. A release time is often set in the beginning of the task period time and the deadline is set in the end of it. These tasks are executed periodically according to a schedule. The schedule could be created statically, before run time, or dynamically, during run time. A scheduler in real-time system decides the execution order of the tasks using some scheduling algorithms.

One popular algorithm in real-time systems is Rate Monotonic (RM) scheduling, where tasks with the shortest period are executed first. Another example is Earliest Deadline First (EDF), where the task with the shortest time to its deadline is executed first.

### 2.2.2 Event-Triggered Tasks

Tasks in these systems respond to an event, this event could be the press of a button or a lever movement. If the task is critical it could preempt the task which is currently running, in other cases it waits for some spare time in the schedule.

Event-triggered tasks often have a priority and a stack size. The priority is often used to decide the criticality of the task. Priority is often the attribute which decides if the task is critical or not, i.e. can it preempt the currently running task.

Event-triggered tasks are often executed according to their priority; a higher priority leads to a earlier position in the execution queue. The priority is often decided off-line, i.e. each task is given a priority value before it is executed and it never changes during execution. This schedule is called event-triggered paradigm with fixed priorities.

## 2.2 Execution Time Analysis

Every real-time embedded system, especially hard real-time system, has a need of predictable execution times in order to make the system predictable and safe. Three different kinds of parameters are estimated by execution time analysis; WCET, BCET and ACET.

Worst-Case Execution Time (WCET) is defined as the longest possible execution time a program can ever achieve when running on its target hardware. The opposite of WCET is the BCET, Best-Case Execution Time, where the shortest execution time of the program is given. Average-Case Execution Time (ACET) is a time value between WCET and BCET; it depends on the distribution of the execution time of the program [1].

The goal of timing analysis is to estimate WCET and BCET for a program. To make those estimates fulfill the hard real-time system requirements they have to be safe. A safe estimation in WCET analysis is greater than or equal to the actual program WCET. Useful estimates need to be tight, that the provided WCET value has an acceptable level of overestimation; see Figure 2.2.

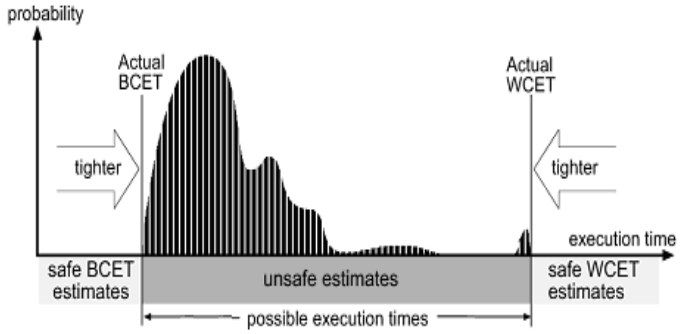



Figure 2.2: The distribution of execution times in a real-time application, [1].

Timing analysis is divided into two fields; dynamic timing analysis and static timing analysis. Dynamic estimates are obtained by measurement, which involves running the system in a lab environment simulating the input values and taking the highest execution time value for WCET and lowest for BCET. Static timing analysis uses mathematical models of the system to derive the estimates.

The main difference between using analysis and measurement is that the value provided by measurement might not be safe, i.e., it may underestimate the WCET value. This occurs when the test cases do not cover the real WCET, i.e. the subset of the possible execution paths (test cases) do not cover the path which takes the longest time to execute [2].

## 2.3 Static WCET Analysis

WCET values, in a real time system, are important to ensure that system tasks meet their deadline. WCET values are used to specify the time triggered tasks periods in an execution schedule. This makes WCET analysis the most important field in the static timing analysis field.

The mathematical models used in static WCET analysis must correctly represent both the hardware and software in the system. These models properties and all possible input combinations must be considered to derive correct WCET estimates, due to their impact on the execution time of the program.

Deriving WCET estimates for a program involves deriving hardware timing characteristics and the possible program execution flow. Hardware timing characteristics entails execution time of individual instruction, memory access time, etc. The possible program execution flows derives execution frequency of each program instruction [2].

Accordingly, static WCET analysis is usually divided into three steps:

- Flow analysis (possible program execution flows).
- Low-level analysis (hardware timing characteristics).
- Calculation stage (where the results from the first stages are combined).

Two commercial WCET analysis tools using static WCET analysis are; Bound-T [31] from Tidorum Ltd and aiT [18] from AbsInt. Figure 2.3 visualizes the three steps in static WCET analysis.

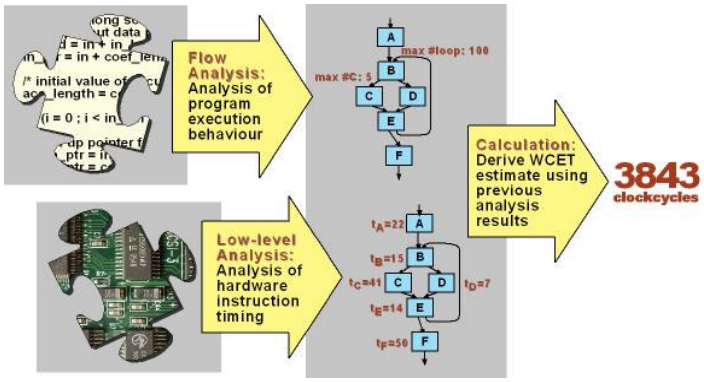



Figure 2.3: The steps in static WCET analysis [9].

### 2.3.1 Flow analysis

Flow analysis is the method used to derive execution flow information automatically. Deriving flow information includes calculating the maximum number of iterations for every loop in the program (loop bounds), identifying infeasible execution paths, deriving execution frequency of code parts etc. To ensure safe WCET estimates, the flow analysis must include all possible execution paths in the analyzed program.

***Value Analysis***

Value analysis is a method to determine the value range for processor registers and local variables at every program point. The results of this analysis method are used to determine infeasible paths and loop bounds. The resulting value ranges are used to determine the indirect memory accesses, which are important to infeasible path and loop bound calculations [16].

***Loop Bounds***

Loop bounds information must be provided to derive WCET estimates; otherwise execution frequency of the instructions inside the loop body will never be known, which makes it impossible to derive WCET estimates. Correspondingly, the recursion depth of a program must also be derived in order to derive the estimates.

Several WCET analyzers provide ways to give these bounds manually, but this method has the tendency of being the source of erroneous information. Thus, automatic loop bound analysis is a very important area in static timing analysis research projects. For further reading in the subject, please read [2] where some automatic methods are mentioned.

***Infeasible Paths***
Compared to loop bound analysis; identifying infeasible paths is not required to derive WCET estimates, but it might tighten them. According to [2], infeasible paths are defined as: "Paths which are executable according to the control flow graph structure, but not feasible when considering the semantics of the program and possible input data values". There are two types of infeasible paths:

- Infeasible paths caused by semantic dependencies
- Input-data limitation depended infeasible paths

For more detailed explanation please read [2] (section 2.2) and [16] (section 3.2).

### 2.3.2 Low-level analysis

The main goal of this step is to create a timing model of the system hardware. The model is used to derive the processor time for each individual instruction, which is highly affected by target hardware features. This includes the effects of various hardware enhancing features, such as branch prediction, pipelines, memory system, out of order execution etc [10].

The timing model contains timing properties of the processor and some other system hardware which affects the instruction timing. There is no need of detailed information as long as timing of the instructions can be derived. Usually safe approximations of hardware information are provided, since it is very difficult to find the precise timing. A typical example is assuming that all instructions are fetched from the slowest memory in the system [10].

The main difference between a cycle accurate processor simulator and the timing model is that in the timing model, a timing bound is derived with respect to all possible executions of the instruction. In contrary the cycle-accurate simulator uses just one single concrete execution of the instruction. It is worth to mention that in this step the binary code of the program must be used, since it has to analyze the timing for each program instruction.

### 2.3.3 WCET Calculations

This stage combines the results of the previous two stages to calculate WCET and to derive the worst-execution path for the program. The calculation method used in this stage can be divided into three main categories; Implicit Path Enumeration Technique (IPET), structure-based and path-based calculation.

***IPET***
As the category name indicates, the method uses an implicitly defined longest execution path of the program to calculate the WCET. Program flow information and basic block[*] execution time bounds are represented using logical and/or algebraic constraints. Usually in an IPET Control Flow Graph (CFG); the execution time for each individual basic-block is denoted $T_{entity}$ and execution frequency of basic blocks is denoted $X_{entity}$. The WCET is then derived by maximizing the sum:

$$\sum_{i \in entities} X_i * T_i$$

***Path-based***
The execution times are calculated for each individual part of the code, which could be a basic block or another well defined code part. This means that each part of the code has its own worst case execution path and thereby its own WCET. The overall WCET is then derived by sum the timing values of each individual

[*] **Basic Block**: code that has one entry point, exit point and no jump instructions contained within it [12]

code part. The main difference between this method and IPET is that possible execution paths are explicitly defined.

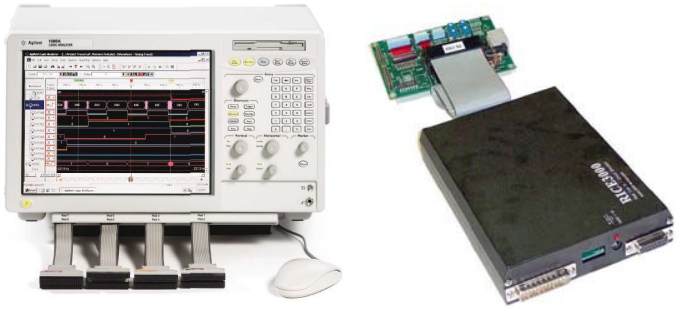
***Structure-based***
This method uses a syntax tree representation of the program; the syntax tree nodes illustrate the structure of the program (if-statement, loops, etc). The tree leaves represents the basic blocks of the program. Traversing the tree follows some given rules, where each node represents an equation which expresses its timing based on its children node timings.

For further reading in the subject of WCET calculations, detailed description of methods and illustrative examples can be found in [11] and [12].

## 2.4 Dynamic WCET analysis

Dynamic WCET analysis is used today by several real-system developers; this method is based on executing the program with many different inputs and measuring the execution time for each run. To obtain the WCET the input must be found, in general obtaining such an input combination is very hard. Therefore it is impossible (in general) to guarantee that the program WCET has been found, without using some form of mathematical proof, i.e. analyzing the program by static timing analysis.

The advantage of using this method is that the actual hardware can be used to execute the program, which gives more accurate worst case timing information. There are different kinds of measurement techniques used in practice, the most important techniques and tools used today are: oscilloscopes, logic analyzers, hardware traces, high-resolution timers, performance counters, emulators, etc. These techniques are described in more detail in [10].



a) Logic Analyzer b) Emulator

Figure 2.4: Dynamic WCET analysis tools, [27][28]

## 2.5 Hybrid WCET Analysis

This method is a combination of the methods described above, i.e. static and dynamic. Basically, the idea is to create a model of the program with static analysis of the source code and prepare to run on the real hardware by partitioning it using measurement points. These partitions are then executed on the real hardware and their timing measurement results are brought back to the static analysis and used to derive a WCET estimate for the program.

The main disadvantage of this method is that there is no guarantee to derive safe WCET estimates, since the timing of the partitions are obtained using measurements. On the other hand, the main advantage is that no timing model for the hardware is needed; instead it is replaced by structured measurement of the program partition using the real hardware. RapiTime, from Rapita Systems Ltd [29], is a WCET analysis tool which calculates WCET estimates using hybrid WCET analysis.

# Chapter 3

# Volvo CE

Volvo Construction Equipment (CE) is the one of the leading manufacturers of construction equipment in the world. Their products range includes motor graders (VMG), articulated haulers (ART), excavators, wheel loaders (WLO) and backhoe loaders. The main office and the development of engines, electronics, axels and transmissions for VMG, ART and WLO are located in Eskilstuna, Sweden. This thesis work was performed at the electronic system department, 14700, in Eskilstuna.

The electronic system architecture in Volvo CE vehicles is a distributed computer system. The ECUs (Electronic Control Unit) are nodes in the system, which communicate with each other via a CAN bus and a J1587 bus. The CAN bus is a high speed bus therefore used for data exchange, the slower J1587 bus is primarily used for diagnostic and service of the system.

The number of ECU used varies depending on the functionality required in the vehicle. The most used ECUs are listed below:

- Vehicle ECU
- Engine ECU
- Transmission ECU
- Instrument ECU
- Cabin ECU

In this thesis thirteen tasks from the articulated hauler transmission ECU (TECU) are analyzed with SWEET, identical to the tasks analyzed in Daniel Sehlberg's thesis [3].


Figure 3.1: Volvo's articulated hauler (ART) [3]

## 3.1 Hardware

The ECU hardware is built around the Infineon C167CS microcontroller, which is a single chip 16-bit microcontroller with a four-stage pipeline. The memory space in C167CS is 16 Megabyte in size and it is linearly addressed. The memory is shared by instructions, data, registers and I/O ports in accordance with the Von Neumann memory architecture. C167CS has a clock frequency of 33 MHz.

## 3.2 Software

The operating system used in the system is called Rubus OS. Rubus is a real time operating system created by the Swedish company Arcticus Systems [5]. Rubus is designed to provide services managing data flow and related computation in the real-time distributed computer system. Rubus supports execution of three types of real-time system threads (hereafter called tasks):

- Green tasks: The interrupt tasks in the system have the highest priority in Rubus, i.e. they can preempt all other kinds of tasks in the system. When a green task interrupt arrives it is handled immediately, these are served by an interrupt service routine which takes care of the most critical tasks in the system. An example is the CAN communication controller.

- Red tasks: The time-triggered tasks in the system are called red tasks; they are executed accordingly to a schedule which is generated off-line. The red tasks are preemptive, but they are seldom pre-empted since they often control some vital operation of the machine. The green threads are the only kind of tasks which can preempt a red task, i.e., red threads have the second highest priority level in Rubus.

- Blue tasks: The lowest-priority tasks in the system. These tasks are event-triggered and preemptive. Blue tasks are executed when no other higher priority task needs the CPU, i.e., blue tasks execute when the CPU is idle.

Volvo CE develops the system using Rubus Visual Studio (VS), which is a graphical development tool used to create system components with ports and create communication paths between them. A system component is a thread visualized as a box with in-ports, out-ports and internal state variables. A component can be one of the thread types mentioned above.

## 3.3 Code structure

The tasks are written in ANSI C and use the Tasking Tool Chain [26] to compile the target CPU assembler instructions. Volvo mainly uses a combination of red and blue threads (tasks) in their system; a task is a function with a structure pointer as an argument, each task has its own private structure. The structure has three main categories of members; input-, output- and state members.

The input members are pointers to data areas which contain the input data for the task; the main reason of choosing pointers (instead of declaring private data areas for each task) is performance. Copying the input data to the tasks own data area every time a task is executed is time consuming and decreases system performance. The input pointers are often structure pointers, which often has two members; an input value and a port status.

On the other hand, output members are data areas which are declared privately for each function. Similar to an input member, the output member could be a structure with several members. The state members are similar to output members, but they are declared as private data areas for the task.

# Chapter 4

# SWEET

The SWEdish Execution time Tool, SWEET, is a prototype tool developed by the WCET research group at Mälardalen University. SWEET handles full ANSI-C programs including recursion, unstructured code and pointers. Unlike most of the other WCET analysis tools SWEET has an integrated intermediate code compiler. SWEET performs its flow analysis using the intermediate representation of the program. It also has a low-level analysis part needed to calculate WCET estimates and find the longest execution path in the program.

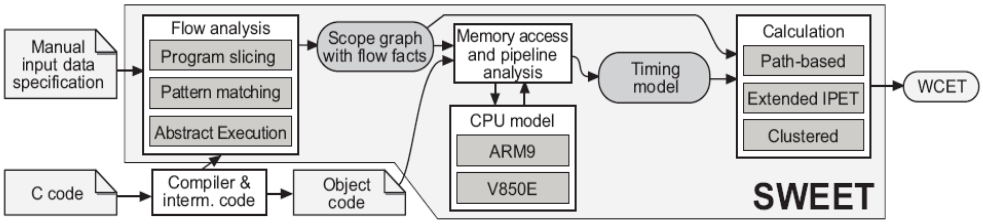



Figure 4.1, SWEET modules [11]

The estimates are derived with SWEET in four steps:

- ***NIC:*** the C-code is fetched to the NIC compiler, in this step the program source code is compiled into intermediate code format. Two types of files are produced; ".nic" and ".tcd".

- ***Annotations:*** to be able to derive correct flow information the input data has to be specified in form of an annotation. The purpose of this stage is to provide value ranges for the input variables and add information which complements the NIC code.

- ***Flow Analysis:*** Flow-analysis calculates the necessary flow information to bound tight and safe WCET estimates. The flow analysis result is stored in set of files where it is presented both textually and graphically.

- ***Low-level analysis & WCET calculations:*** the tcd-files and some other files form the input to this stage where the low level analysis is done together with WCET calculations.

SWEET is built from several different modules, see Figure 4.1, these allows This makes SWEET more flexible to adding new functionality and new algorithms for optimizing WCET analysis.

## 4.1 NIC

The New Intermediate Code (NIC) format is an intermediate code format designed specially for embedded systems by a research project at Uppsala University targeting whole program analysis. The main purpose of this code is to represent the embedded system code to facilitate code analysis and compilation. NIC instructions are generated from the C-code. One benefit of analyzing the NIC format of the program is the possibility of integrating the compiler into a static WCET analysis framework [4].

### 4.1.1 Syntax

The implicit expressions written in C are expressed in a simpler way in NIC, which makes it harder to map the NIC code to its equivalent C code. One C-code line could be expressed in more then 10 lines NIC of instructions. This appears when the C-line is expressing an operation between two different data types where many type conversions are involved.

However, NIC representation of the C code is close enough that all data and control structures have a direct equivalent in NIC code. This makes it easier to understand the flow analysis results, even if they are addressed to NIC code snippet instead of C-code snippets. Table 1 lists the operators supported in NIC; the number in parenthesis represents the number of arguments that the operator use.

| Arithmetic operators | |
|---|---|
| neg (1) | negation |
| add (2) | addition |
| sub (2) | subtraction |
| u_mul (2) | unsigned multiplication |
| s_mul (2) | signed multiplication |
| u_div (2) | unsigned division |
| s_div (2) | signed division |
| u_mod (2) | unsigned remainder |
| s_mod (2) | signed remainder |
| **Bit operators** | |
| l_shift (2) | left shift (second argument is number of steps) |
| r_shift (2) | right shift (second argument is number of steps) |
| r_shift_a (2) | arithmetical right shift (second argument is number of steps) |
| not (2) | bitwise not |
| and (2) | bitwise and |
| or (2) | bitwise or |
| xor (2) | bitwise xor |
| trunc (2) | truncate bits (second argument is number of bits) |
| z_ext(2) | zero-extend bits (second argument is number of bits) |
| s_ext(2) | signed-extend bits (second argument is number of bits) |
| **Relational operators** | (returns true/false) |
| cmp_eq(2) | compare integers for equality |
| cmp_ne(2) | compare integers for non-equality |
| u_cmp_lt(2) | unsigned less than comparison |
| s_cmp_lt(2) | signed less than comparison |
| u_cmp_ge(2) | unsigned greater than or equal comparison |
| s_cmp_ge(2) | signed greater than or equal comparison |
| u_cmp_gt(2) | unsigned greater than comparison |
| s_cmp_gt(2) | signed greater than comparison |
| u_cmp_le(2) | unsigned less than equal comparison |
| s_cmp_le(2) | signed less than or equal comparison |

Table 4.1: NIC operators [13]

The NIC syntax is similar to the LISP language syntax; the language can be seen as a high-level assembler language. Data values are presented as bit strings where the operation applied on the bit string decides how the string should be interpreted. For example, s_div 32 (32, 32) is a division where the s indicate that the arguments for the operation are signed 32-bit integers and division results are also stored in a 32-bit integer.

Data type sizes, Endianness[†] and alignment are some target architecture depended properties which are not presented in NIC format. These program properties have to be specified by the user [4].

### 4.1.2 Memory model

Three different memory spaces are supported by NIC for data storage:

1. Registers: used mainly for temporary data. For example, function "A" calls function "B" with some arguments (input data); the registers are then used as a temporary storage for argument data.

2. Global memory: used for data needed from the program startup (hereafter called "global data"). Example: Global variables in C, common for all functions in the program.

3. Local memory: used for data needed in function entries (hereafter called "local data"). All the variables declared inside a function are stored in this memory space.

Memory frames are used for allocating global and local data. NIC specifies frame id, size and when it should be allocated. There is no actual memory address assigned for each data frame, instead the frame id is used to make a reference to the frame. In order to map a frame id to a specific memory address a memory environment is used, a more detailed explanation is presented in [4].

Global frames are needed from the program startup, which means they are allocated and given values at program startup. The local frames are allocated at functions entries and de-allocated when the functions return. When a new frame is allocated the current memory environment has to be updated with the new mapping, where the new frame id is mapped to the actual memory address where the frame is allocated. This means that every time a function is called a new frame is allocated, with the guarantee that a newly allocated frame is referred by the given frame id.

## 4.2 Annotations

The main goal of this step is to define the input values or values at certain points in the program. Annotations are stored in a text file (.ann), where each line in the file corresponds to one annotation. The annotation is divided into two sections; the location in the NIC code where the annotation will be inserted (called the "where" section), and the other section what its functionality is (called the "what" section).

The addressing of the position in the NIC code, and the "where" section, can be written in three different ways, either by: explicit address, entry block of the function or the parameter number of the function. The second section, the "what" section, could contain either: a value range (integer or float), an annotation for a pointer or an annotation for a pointer pointing to an interval of offsets.

*Examples:*

```
1. main par 1 int 0 1 ;
2. func BB0 5 reg 1 int -1 5 ;
3. func entry reg 2 float -1.4711 2.4711 ;
4. main entry store input  0 32 int 1 5 ;
5. main entry store input  4 32 int 1 10 ;
6. main entry store structure 0 32 pointer variable 0;
```

The first annotation in the list is used to set an interval of integers [0...1] for the first parameter (par 1) of the program (main). The second example is used to annotate register nr 1 in the program func () with an interval of signed integers [-1…5], the BB0 5 indicates that the annotation is placed in basic block nr 0 as the fifth instruction. The third example has a similar function as the second example; the main difference is that the annotation is placed at the function entry which is set automatically by SWEET.

Example nr 4 sets a 32-bit integer interval (1-5) in the global variable input with offset 0, the offset facilitates writing intervals to variables inside a data structure which is the case in example 4. The variable

[†] **Endianness** is the attribute of a system that indicates whether integers are represented from left to right or right to left.

input is a data structure containing several members; the offset is set according to the size of the members. Example 5 a range is set for the member nr 2 in the "input" structure; since the first member has allocated byte 0 to 3 (32-bit integer) member nr 2 has an offset of 4 as written in example 5.

Example nr 6 is used when the structure contain pointers which are not assigned to any variables. This annotation is useful when the software is built from several standalone functions, which are analyzed separately. These systems often have one main function which contains the initialization for all the functions in the system, making the main function very large and unmanageable when generating a NIC representation for one single function. Therefore, SWEET offers the user the possibility to write such assignments after NIC code generation.

## 4.3 Flow Analysis

This step in SWEET is based on a multi phase approach:

- First an optimization phase where a technique called program slicing is used to reduce code amount to be analyzed, i.e., it restricts the analysis to the part of code that affects the program flow [17].
- Second phase is analyzing the code using the value analysis and the pattern-matching techniques, in this phase simpler case loop bounds are found.
- Third, a unique method is used, called abstract execution. The goal of this step is derive flow facts, i.e. loop bounds, infeasible paths and node execution bounds. It is not necessary to run all three phases in flow analysis.

### 4.3.1 Scope Graph and Flow Facts

In order to describe how abstract execution works there is a need for defining a scope graph and the flow fact language syntax. Execution behavior of the program is represented using a format called scope graph [1]. A scope (node) in the graph represents a code snippet or program part which can be repeated, e.g., functions and loops. A containment relationship is represented in a scope graph, e.g., if a loop is called inside a function then the loop scope will be below the function. This is possible because the scope graph is acyclic [2].

The scope graph used in SWEET describes how scopes are invoking other scopes. If a function or a loop is called several times in the program, each call would be represented with it own scope, i.e., the graph is context sensitive. The context sensitivity in the scope graph makes the analysis more accurate and costlier (in time and computer power) since each context will be analyzed separately [2].

Flow facts are the flow information derived for each basic block in a program scope. A flow fact has the following syntax in SWEET:

***scope: context: linear constraint***

***Scope***
This field defines the scope name where the constraint is effective. The scope name in SWEET contains the name of all the scopes which are invoked to get to the target scope, i.e. the call string in the scope graph from the top scope to the targeted scope.

***Context***
In this field the context range where this constraint is effective is set. Here it is possible to specify if the constraint holds; for each individual scope iteration (specified as <range>) or for all the possible scope iterations (specified as [range]). If the range is left out, i.e. the field contain either "<>" or "[]", then the specified flow fact holds for all iterations [2].

***Linear constraint***
The flow fact in SWEET represents either loop bounds or infeasible paths. If BB3 is a loop header, and the loop iterates maximum four times, then SWEET expresses this as: BB3 <5; in other words BB3 is executed maximum four times. If BB3 is an infeasible node, the expression would be: BB3 = 0. More detailed information about scope graphs and flow facts can be found in [1] and [2].

### 4.3.2 Abstract Execution

Abstract execution, in [4] is defined as:

"A form of symbolic execution, based on abstract interpretation"

Abstract execution uses abstract operators and abstract values for program variables when executing the program in a so called abstract domain. The main difference between this form and the traditional abstract interpretation is that in abstract execution all possible executions at a certain program point are analyzed separately.

The abstract domain in SWEET is a domain of intervals. It means that each variable in the program will hold an abstract value, which is an interval instead of a single value. Abstract execution inherits that, at each program point new intervals are calculated from current ones for each variable in the program, from abstract interpretation. The new intervals then represent the concrete values the variables could hold at that point.

The Abstract interpretation framework guarantees that the set of possible values are always represented by the abstract value. Thus, no execution paths will be missed by the analysis. But since the abstract values may overestimate the value range, the analysis may produce flow constrains which are not tight. In case of infeasible path analysis this means that some infeasible execution paths are considered feasible, consequently the result of WCET will be less tight [4].

In some condition nodes the abstract execution will consider both true and false branches as possible nodes in the execution path. In order to hold the result of both possible paths, two abstract states must be created. This forces the abstract execution to handle many abstract states in parallel. Furthermore, the states have a tendency to grow exponentially with the length of the execution paths. This demands SWEET, and any other tools using abstract execution, to use a merging mechanism to save space and analysis time.

Merging the states is usually done at program points where different paths join, but there are several kinds of merging strategies where the user has the freedom of choosing the actual merge points. That makes the user able to choose between a fast analysis and a more precise one. The algorithms in SWEET merge the abstract states which belong to the same scope iteration and program point.

To read more about the abstract execution algorithm used by SWEET please read section 4 in [4].

### 4.3.3 Calculation of Flow Information

The methods used by SWEET to calculate flow information, i.e. loop bounds and infeasible paths, are all using the abstract execution technique. This section contains a brief presentation of these methods.

***Loop Bound Calculation***

A safe upper and lower bound of the total number of iterations of a scope is found in this analysis. This method is most useful to find loop bounds in single loops. Each scope has a counter which is set to 1 at the scope entry and incremented each time the scope is executed. The counter is then incremented at the scope exit.

For nested loops a simple variation of the single loop analysis is used. The inner loop has its own counter which is incremented each time the inner loop is visited. Thus, this will calculate the maximum and minimum number of iterations, for the inner loop, for any single iteration of the outer loop.

***Calculation of Infeasible Nodes, Edges and Paths***

In SWEET an infeasible node is defined as the node which is never visited in any execution of a certain scope. In this method a bit array is used to detect infeasible nodes; for each scope execution there is bit array, each bit corresponds to a node in the scope. The array is initiated to zero; once the node is visited the corresponding bit is set to one.

When the scope is terminated, these bit arrays are collected and merged together in bitwise manner. The bits which are still zero indicate that the corresponding node is never executed during the scope execution. It is worth mentioning that an infeasible node is not the equivalent of dead code, since there could be other scopes of the same code where the node is executed. (Recall that the analysis is context sensitive.)

A generalization of the method mentioned above is used to calculate upper bounds of node executions. The bit array is then replaced by an integer array. The array elements are then execution counters initiated to zero and incremented each time a correspondent node is visited. The number in the counter is the maximum number the node is executed during the scope iterations.

These algorithms are also applied to detect infeasible edges; an example is an if-statement which is always true according to its conditions. The false edge of the if-statement is then infeasible. Furthermore, infeasible node pair detection gives a node pair, which can not be executed together in the same execution state of the program. These together with the infeasible node detection facilitate finding infeasible execution paths in the analyzed program.

### 4.3.4 Flow Analysis Results

The flow analysis results are presented in a set of files:

- Scope graph file (file.sg), containing the scope graph and the results of algorithms which were chosen to analyze the program.
- Scope graph dot file containing the scope graph (file.sg.dot). When the file is read in an appropriate visualizing tool, the scope graph is drawn.
- Scope graph hierarchy dot file (file.sgh.dot), which draws the relationship between scopes.
- Call Graph dot file (file.cg.dot) draw the call map of program functions, i.e., the relationship between functions in the program. In addition it gives an idea of how many contexts each function has, i.e., how many times it is executed during program execution.

***Scope graph file***

Figure 4.2 shows an example of a scope graph file, here follows a description of every section in the file:

- **scope:** The scope name contains the name of all functions which were invoked in this scope, beginning with the root function.
- **Type:** a scope could be either a function or a loop. Here it is also specified if the function is the root or if it is another ordinary function.
- **Backedges:** In case of having a recursive function or a loop, the basic block where the function jumps back into is specified here.
- **maxiter:** the number of scope iteration is set here, in case if it is a function this value is often 1 or 0 (if the scope is never invoked).
- **header:** the basic block which contains the header code of the function is specified here.
- **facts:** the most important field of the scope graph file; all the results from the different algorithms are presented here.
- **subordinates:** the scopes which are invoked by this scope are presented here.
- **basicblocks:** the basic blocks which belong to this scope are listed here.
- **internaledges:** the relationship between scopes basic blocks are expressed here.
- **exitedges:** the exit edge(s) of the scope is presented here, this contains information on which scopes this scope is invoking.

***Scope graph dot file and Scope hierarchy file***

These files contain a graphical description of program flow and hierarchy; SWEET uses the DOT program [14] to present the program flow and the program's scope hierarchy. Graph Visualization Software, Graphviz [15], can be used to view these graphs.

Figure 4.2-4 shows examples of a Scope hierarchy dot file, a Scope graph file and a Call Graph file. The latter represents the functions call hierarchy inside the program.

```
scopegraph

scope main:
%% ------------------------
%%    Type = Function (root)
%%    Backedges:
%% ------------------------
 maxiter 1;
 header BB1;
 facts
  main: [ ] : BB1 <= 1; %% -ip : min max node count
 subordinates
   main_prime1;
 basicblocks
   BB1, [ ];
   BB2, [ ];
   BB3, [ ];
   BB4, [ ];
 internaledges
            BB1 -> BB4;
            BB1 -> BB2;
            BB2 -> BB3;
            BB3 -> BB4;
 exitedges
         BB4 -> (main_prime1, BB9);
         BB5 -> exit;
end scope
```

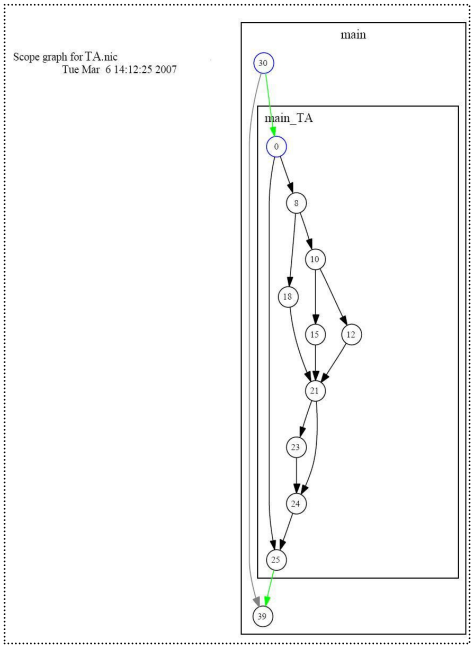



Figure 4.2: A snippet from a scope graph file and a Scope graph dot file.

## 4.4 Low-level Analysis & WCET Calculation

The low-level analysis part of SWEET supports NECV850E and ARM9 processors at the moment, therefore this part is not so interesting in this thesis. Worth mentioning is that the low level analysis in SWEET is divided into three phases; memory analysis, pipeline analysis and the calculation phase.

In memory analysis phase, the memory which is accessed during execution of the program is determined. That includes instruction cache analysis if the target hardware has such cache. From this phase execution facts are derived. Execution facts can be information about the memory areas an instruction will refer to during runtime and also information about if the instruction will hit or miss the cache [11].

The execution facts are then used to impose worst-case timing behavior for every instruction in pipeline analysis phase. In this phase a trace-driven cycle-accurate CPU model is used. The final phase in SWEET is the calculation phase, where three different kinds of calculation techniques are supported; fast path-based, global IPET and a hybrid clustered technique [11].

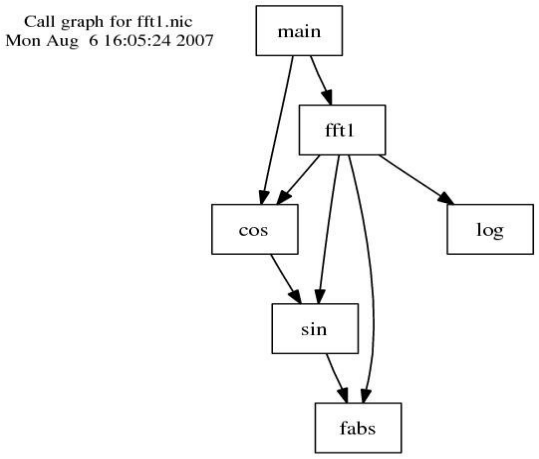



Figure 4.3: Call graph representation for Fast Fourier Transform function.

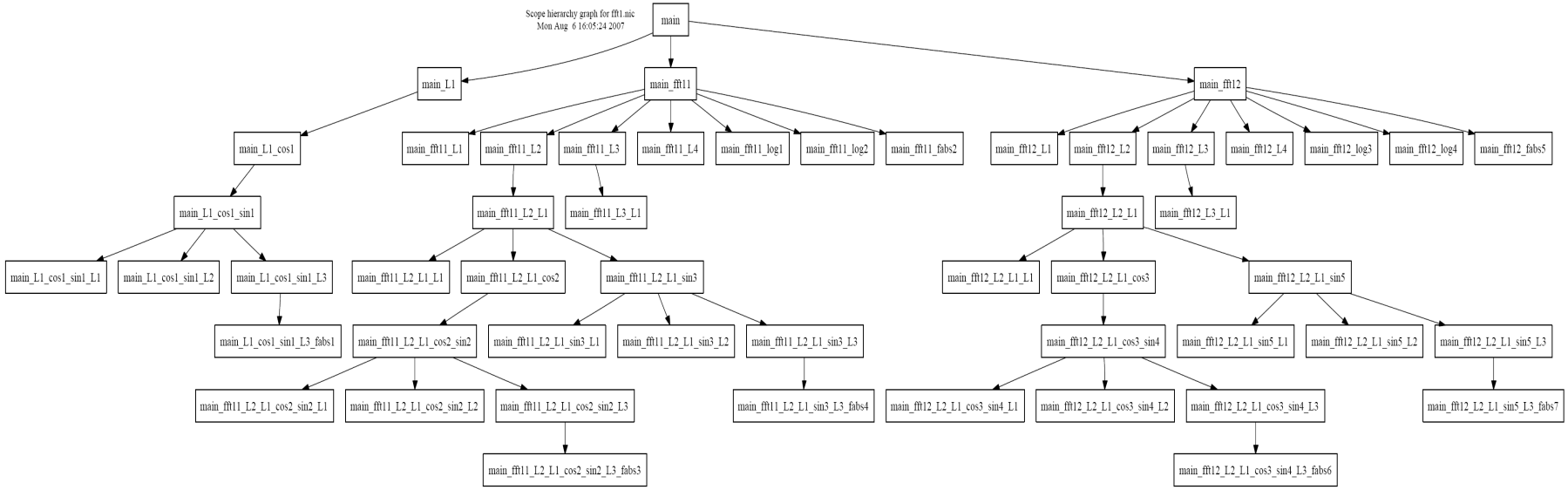



Figure 4.4: Scope graph hierarchy representation for Fast Fourier Transform function.

# Chapter 5

# aiT

aiT is a commercial WCET analysis tool used by vehicle and aerospace industries. aiT is developed and produced by the German company AbsInt [18]. The purpose is to obtain the execution time upper bound for code snippets in the binary of the program, these snippets can be subroutines or tasks in a task oriented system. The reason for using program binary and not the source code is: information about instruction, data addresses and register usage is only provided in the binary format. This information is important to consider when analyzing the cache and memory behavior of the program [11].

The tool needs user input in form of annotations in order to derive WCET for these subroutines. The annotation could be upper bounds for loops and some flow information (flow facts) which must be provided by the user. Some of these annotations also improve the precision of WCET results but they are not necessary to derive them. Apart from the usual annotations, aiT supports annotations specifying the value of the registers and variables. This feature is useful for the analysis of state dependent software, i.e., software which is executed in different modes depending on the value of the registers or variables [11].

The aiT results are presented in form of a flow graph, visualized by aiSee [19]. The node in the graph represents the basic block of the program. A basic block has the same memory address as the first instruction in it. Both aiT and SWEET has the same definition of the basic block which facilitates the mapping between their annotations considerably. For more information about the tool and its workspace, please read Daniel Sehlberg's thesis [3] where he used the tool to analyze the same tasks which are analyzed in this thesis.

Furthermore, the paper written by C. Ferdinand and R. Heckmann [16] contains a detailed description of the analysis techniques used in aiT.

## 5.1 User Annotations

The aiT annotations can be divided into two categories:

- Mandatory annotations, these annotations are needed by aiT to be able to analyze the code. A good example is annotations which set upper bounds for loops, recursion depths and unresolved calls. If these are not set aiT will warn the user to this unbounded problem and terminate the analysis.
- Optional annotations are used to tighten the WCET results, infeasible paths and excluding pairs are examples of annotations which tighten the WCET result.

There are two different ways of specifying annotations, either in a separate input file or with a special kind of comments in the source code of the analyzed task. Since in this project the same tasks which were analyzed by Daniel Sehlberg [3] are used, we chose to annotate in a separate file as he did in his project.

Beside flow fact annotations and loop bounds, the annotation file contains the following annotations: Compiler name, CPU clock rate, memory addresses, register values and context specification. Examples of these annotations are presented in [3], section 3.3.2.

There are two ways to write annotations: the first one is using the absolute address of the basic block (or instructions) which are needed when annotating, the second way is to use so called relative addressing. A

relative address consists of the routine name followed by some parameters, which specifies the basic block or instruction needing the annotation. Absolute addressed annotations are easier to write, since it is easy to find the addresses either in the flow graph file or in the warnings aiT print out in case of needing an annotation [3].

The down side is that when the source code is recompiled it could cause changes in the absolute addresses which makes the annotations useless. This makes the relative addressing method a more desirable way to write the annotations, since aiT will automatically map these to the correct absolute address each time the project is recompiled. Worth mentioning is that even the relative addresses needs to be rewritten, if the subroutine source code has been restructured and recompiled. Another pro is that the annotations are more readable when written in the relative addressing style.

**A:** `snippet 0x0: 0x231e is never executed;`
**B:** `snippet"_abs" is never executed;`

Figure 5.1: an annotation which specifies that the routine"_abs" is infeasible, i.e. dead code. **A** written with absolute address style and **B** in relative address style.

## 5.2 aiSee

aiSee is a visualization tool created by AbsInt. aiSee calculates and visualizes a custom layout for graphs written in Graph Description Language (GDL). aiSee views the analyzed routine call graph, Figure 5.2.A, as the last step of its analysis. It also displays the predicted WCET for the program above the graph. For each subroutine in the call graph there is a flow graph which describes the execution flow through it. The subroutines are visualized as boxes with the subroutine name in it and a bar showing how much WCET the subroutine has relative to the overall WCET.

When the subroutine box is unfolded it shows the flow graph in the subroutine. The flow graph nodes are basic blocks and edges representing the possible execution paths in the subroutine, see Figure 5.2. Under each basic block there is a white box where information about its WCET and the maximum number of visits to it is displayed. There is even another level of unfolding where each instruction in the basic block is displayed, see Figure 5.2.C. Furthermore, the basic block box in aiSee could contain either the start address of the basic block; see Figure 5.2.B, or the source code which corresponds to the instructions in the basic block.

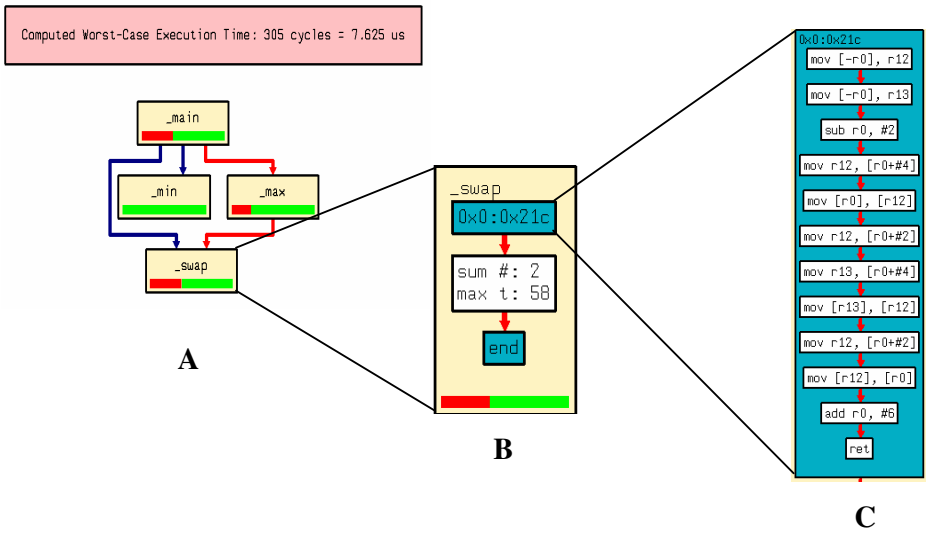



Figure 5.2: **A**: Call Graph displayed in aiSee, **B**: Flow graph for subroutine"_swap" and **C**: Basic block unfolded in"_swap".

# Chapter 6

# Project Tools

The hardware and software used in this project are described in this chapter. It begins with a description of the hardware used, followed by a description of the two main WCET analysis tools used in this project, SWEET and aiT, together with a description of the scripts used to facilitated the project tasks.

## 6.1 Hardware

The main hardware used in this project was two PCs, one with Windows XP professional and the other one with Linux Ubuntu 6.10 installed. No emulators or simulators were used for this project since the main focus was program flow analysis. The other results are taken from Daniel Sehlberg's thesis. The Windows computer was a Pentium 4 with 3.4 GHz clock rate and 1 GB RAM. The Ubuntu machine was also a Pentium 4 but with 2.0 GHz clock rate and 512 MB RAM.

The main reason for using Ubuntu is that the NIC compiler, also a prototype, works best in UNIX/LINUX environments. It is very time consuming task to make the NIC compiler to work in UNIX emulators in Windows environment. SWEET on the other hand runs in Windows (CygWin), Mac OS and Linux environments.

## 6.2 Software

SWEET is the main software used in this project, and since it is a prototype no version number could be set on it. The front end part of SWEET, the NIC compiler, and the low level analysis part of SWEET remained unchanged throughout the whole project. This because the main focus was on the flow analysis part of SWEET. SWEET was modified about ten times during the project, adding new functionality and adapting the tool to Volvo Code.

In order to generate the necessary intermediate code which is used as input for SWEET, we used the NIC compiler. The NIC compiler used in this project is constructed with the lcc [21] and some SML [22]. To cover the cases where the analysis used code from several different source files the CIL [23] and OCAML [24] are used, where the code needed is merged making one single file for the NIC compiler.

aiT C16x/ST10 version 2.0 was also used in project. Since the project results are compared to the result from Daniel Sehlberg's thesis, all 13 tasks had to be reanalyzed. Due to the fact that the WCET result from Daniel's analysis needs to be verified (he used version 1.5). Worth mentioning is with the aiT 2.0 a new version of aiSee is supported. The new aiSee 3.0 is has better user interaction than its predecessors.

## 6.3 Scripts

SWEET is commando based tool with no GUI, therefore several shell scripts were used to facilitate the work with SWEET. Some of these were specially written for this project and others are used to connect different parts (modules) in SWEET. In this section we will briefly describe the most important ones and how to use them.

### 6.3.1 c2nic

This script is the first step in WCET analysis with SWEET; it executes the necessary commands to run the lcc and SML compilers. The input for this script is the C-file of the analyzed application, the output is a text file (.nic) containing the intermediate representation of the application. The `c2nicarm` is a variation of `c2nic`, the only difference is that the latter also produces the input file (.tcd) for low level analysis low SWEET for the ARM9-based platforms.

### 6.3.2 build

build is an alternative script to `c2nic`, which has the same functionality as `c2nic` and `c2nicarm` together. The main difference is that build has the directory name, containing the application c-file as input parameter. The script build performs the NIC code generation in four steps:

1. Preprocessing, using lcc.
2. Patching, make the c-file more readable.
3. Compilation, running rcc.
4. Translation, translating the result from lcc to NIC.

The script has the option `–wcet` if the low level analysis will be used. If this option is selected a .tcd file will be produced. The output is a directory structure containing the files produced in every step above. This makes it easier to organize the project files and aids the user to find cause of errors if such occurred during this step of analysis. This script was frequently used during this project since it created a simple and ordered structure for the analyzed tasks, where each task had its own directory containing all the necessary files to analyze the task.

### 6.3.3 cilly_to_c

As mentioned in section 6.2, there are cases where the task or program needed to be analyzed had been constructed in such way that it depended on code from several different source files. This is the case with the analyzed code in this project. Therefore it was necessary to write a script to handle these cases. `cilly_to_c` performs NIC code generation in three steps:

1. Preprocessing; preprocesses all the “to-be-merged” c-files with the gcc compiler [25].
2. Merging; the necessary source files are merged into one file using cilly.
3. Build, calling the script build with a new result file as input and –wcet as an option.

Since the `build` script is a part of `cilly_to_c` the results of running these scripts are the same as the build script, i.e. a directory structure.

### 6.3.4 runsweet

The script runsweet is created by the SWEET developers with the intention of running SWEET with a combination of options creating four different analysis modes. The modes are listed below with the options to run SWEET in them:

1. *Single path, Basic mode*: The simplest analysis mode, only simple loop bounds are calculated. This because SWEET needs these facts in order to bind WCET for the analyzed code. By single path we mean that no value interval is assigned for the variables which affect the program flow, i.e. no annotations are used in this mode.

   **Linux command:** `runsweet –s –b directory-name;`

2. *Single path, Advanced mode*: The flow analysis is run in single path mode, but using more algorithms to calculate the maximum set of flow facts, i.e. all types of loop bounds and infeasible paths.

   **Linux command:** `runsweet –s –a directory-name;`

3. *Multi-path, Basic mode*: In this mode an annotations are used to assign the variables affecting the program flow, but using the calculation of the simple loop bounds

   **Linux command:** `runsweet –m –b directory-name;`

4. *Multi-path, Advanced mode*: The most advanced mode and the most time consuming one, all the paths in the program are taken by assigning intervals for variables. Beside that, the full set of flow facts is calculated in this mode.

   **Linux command:** `runsweet –m –a directory-name;`

The output from the flow analysis using runsweet is a set of files:

- A Scope Graph file, SG file, containing the scope graph and the flow facts (file.sg)
- A dot file containing a visualized SG (file.sg.dot)
- A dot file containing the Scope Graph Hierarchy, SGH (file.sgh.dot)
- A dot file containing the Call Graph (CG) (file.cg.dot)
- A log file (file.sweet.log) containing the information SWEET prints out during the analysis; these could be diverse error messages, a result list for the algorithms used in the analysis, SWEET run time in CPU seconds, a visited basic-blocks and context counter, etc. See Figure 6.1.

In addition to the above, `runsweet` also supports running the low level SWEET. But since there is no support for the target platform in this project the low level SWEET part was turned off.

```
SWEET - SWEdish Execution Time tool - Mon Jun  4 13:35:01 2007
Reading file example/example.nic
Reads abstract assignments from annotation file example/example.ann
Successfully added 3 abstract assignments
Removed 2 symbols out of 7
Removed 0 registers out of 2
Removed 0 initializations of globals out of 3
Removed 22 symbols from symbol table out of 30
Successfully parsed input
Creates scope graph
Runs PDG slicing
Removed 1 symbols out of 5
Removed 0 registers out of 2
Removed 0 initializations of globals out of 3
Removed 1 symbols from symbol table out of 8
PDG slicing finished, used 0 cpu seconds to remove 7 statements out of 28
Re-creates scope graph based on PDG slicing
Writing control flow graph graphically to example/example.cfg.dot
Writing call graph graphically to example/example.cg.dot
Skipping value analysis
Running Abstract Execution
WARNING: Type of stored value by annotation:
         stmt_abs_reg_pointer 12
           (register 0 has_man) ((symbol "$1_a" org_name "a" has_decl (gkey
"1")) 0)
         is not the same type as value previously stored by NIC compiler: s:[-
2147483648,2147483647] CEbbeAbs32bitIntervalInteger
Created 2 flowfacts of type in (infeasible nodes)
Created 1 flowfacts of type ep (excluding node pairs)
Created 2 flowfacts of type ina (infeasible nodes all iterations)
Created 0 flowfacts of type ine (infeasible nodes each iteration)
Created 9 flowfacts of type mmnc (min and max node count)
We visited 12 contexts and 9 basic blocks out of 11 (81%)
Used 0.01 cpu seconds
Writing scopes and flowfacts textually to example/example.sg
Writing the scope graph graphically to example/example.sg.dot
Writing scope hierachy graphically to example/example.sgh.dot
Flow analysis finished, used 0.01 cpu seconds in total
```

Figure 6.1: Example of a runsweet log file.

### 6.3.5 sweet-nic

The main purpose of this script is to print out the NIC code after it is fetched into SWEET. The NIC code produced by the c2nic compiler has a basic block for each assignment in the c-code. SWEET modifies the NIC code by putting all statements in a sequence into one basic block and adds the annotations written by

the user to it. In this script sweet is executed with the option "`-printprogram p`" with value analysis and abstract execution disabled.

The output of this script is a text file (file.sn) containing the NIC code produced by SWEET, after adding the annotations written by the user to the original NIC file and some merging of basic blocks. This file is necessary for NIC to C mapping, since all flow facts produced by SWEET are based on the modified NIC code.

### 6.3.6 dot2pdf

This script is used to convert the dot files produced by SWEET to postscript format and PDF format. These facilitate presenting the graphs independent of operating system used. The input could be one of the dot files and the output is a postscript version of the file and a PDF version. The scripts uses two open source converters; a dot converter which converts dot to postscript and `ps2pdf` converter where the postscripts file is converted to PDF format.

## 6.4 SWEET options

Besides using runsweet to analyze the tasks in the project, we analyzed some tasks using SWEET directly by calling it from the command prompt with the proper options. This was done because some of SWEET features were implemented by the SWEET developers during this thesis. This resulted in adding new options to the option list of SWEET. This section contains some of the most frequently used SWEET options, both new and old ones. Some of these are also used in the script runsweet, which will be mentioned below for each option.

### 6.4.1 Type Check Annotations, -tca

This option is used to check if the annotations given by the user specify the same data type for the variable as stored by the NIC compiler. There is the possibility of specifying the kind of the variables that needed checking, i.e., local or global variables. This option was implemented during the thesis to facilitate annotation writing and check if the NIC compiler converted the source code correctly. This option is used in the runsweet script; the warning in Figure 6.1 is a print out made by this option.

### 6.4.2 Slicing: -ss and -nops

Slicing is a method used to reduce the code that needed to be analyzed, i.e., to remove all the statement which does not affect the program flow [17]. This method is used to increase SWEET performance; decrease analysis time and memory usage. SWEET have several different kinds of slicing methods; the most frequently used one in this thesis is the Program Dependency Graph (PDG) slicing, which is by default active in SWEET and can be turned off by using `-nops` option. A simpler variant is the simple slicing; which is activated by the option `-ss` turning both slicing methods on.

### 6.4.3 Value Analysis: -nova, pge and -pga

The abstract execution is the main analysis method used in this thesis, therefore the `-nova` option is usually used to turn the value analysis off. There were also cases where the value analysis had to be used, using the options `-pge` or `-pga` are such cases. `-pge` is used to print the global variables values at the program end point; this could be used to check the output value of a program using a certain input value.

The `-pga` has also the same function as the `-pge`, the difference is that in `-pga` all the possible input values are considered. Therefore, the output is an interval of values. The `-pga` and `-pge` also inspired the implementation of SVVA, which basically is built on these two options.

### 6.4.4 Abstract Execution: -noae, -lb, -ip and –sm

This option group is frequently used in this thesis, beside the `-noae` option which turns off the abstract execution there are three main options in this section:

- Loop Bound Calculation, `-lb`
- Infeasible Path Calculation, `-ip`
- State Merge, `-sm`

***Loop Bound calculation, -lb***
This option is used to calculate the upper loop bounds using abstract execution, see section 4.3.3. Two calculation levels are available; Level `0` where the upper bounds for the program loops is set as max scope iteration (maxiter) and Level `1` where additional upper bounds are found for inner loops and are presented as flow facts in the scope graph file.

Worth mentioning is that there exists another variant of loop calculation called Loop Analysis (`-lba`), which is based on value analysis. This method is considered to be a faster analysis than the method mentioned above, but since it has been implemented during this thesis it is still not well tested enough to verify that its result are correct.

***Infeasible Path calculations, -ip***
This option activates the infeasible path, edge and node calculations (see section 4.3.3). `-ip` has several parameters which specify what kinds of infeasible flow facts are calculated. The most frequently used parameters are: `in(a)`, infeasible nodes calculation for all iterations; `ine`, the same as the previous one but for each iteration; `ep`, excluding pairs ; `ip`, infeasible paths; `mnc` and `mec`, max node count and max edge count correspondingly.

***State Merge, -sm***
This option is mainly used to increase the performance of SWEET. The main reason is when using abstract execution the number of program states has the tendency to be very large, which slows down the analysis. Furthermore several of these states could exist in parallel at a certain program point which makes SWEET consumes more memory. When state merging is used the flow analysis flow information results are limited, this is because when the states are merged together the detailed information about each individual state is lost.

State Merge merges several states together at specified program points (controlled by setting parameters for `-sm`). These parameters are: `ft`, merging at function termination, a very mild merge; `lt`, merges at loop termination; `lbt`, merges at loop body termination and finally the `ifj` merge which merges the states at each if-statement join program point. The `ifj` is the most extreme merge of them all; at this merge level the analysis cannot tell the difference between the true and the false edge of if-statements. This makes the analysis produce less flow facts, for example the excluding pair algorithm is useless when using the `ifj` parameter.

### 6.4.5 State Variable Value Analysis (SVVA), -svva
It is necessary that the SWEET annotations provided by the user, provides correct value intervals for variables affecting the program flow. These variables can be input parameters, state variables and in rare occasions the program output. In this work state variables are used to store the program state for the next program run. These could be declared in the global area or/and locally in the function. Usually it is relatively easy to find out the value interval of the input parameters.

A more complex task is to find the interval of state variables. Therefore, as a result of this thesis, the WCET research group added a new SWEET functionality to derive annotations for state variables automatically. The method is called State Variable Value Analysis (SVVA), which is activated with the option `-svva`. This method uses the input range to derive the value range for variables which are specified by the user and the results are presented as SWEET annotations.

State Variable Analysis iterates the program execution using all the possible input value combinations together with results derived from the previous iterations. When the value range of the specified variables is stabilized, i.e., the value range is not affected by the input combination nor by the program state, a result was file produced where all the value interval annotations are stored. This file can be directly used as a annotation file for flow analysis. This function can be used either by using value analysis, fixed point iteration, or abstract execution.

The option `-svva` has five parameters: `infile`, the input file specifies which state variables are in need of a value interval; `outfile`, the output file contains the annotations produced by `-svva`.; `sl`, use slicing which do not remove the code statements which affecting the variables in the input file. `<function>`, here the user specifies the function name where the state variables are calculated, this name is also used in

the output file to specify the function where the annotation should be set. The last parameter is where the user specifies which kind of method SVVA will be using, i.e., abstract execution or value analysis.

It is common in embedded software that the main function (or what corresponds to it) is used to assign the global variables an initial value, which is usually 0. This has a great effect on SVVA result, if no function name is specified; SVVA will assume that the starting point for each execution is the main function. That will cause the variables to be set to their initial value, which makes SVVA to assume that they will always be it and thereby produce annotation with the value [0…0]. For this reason the `<function>` parameter is very important is specified correctly.

### 6.4.6 Draw & Print, -d, -v

SWEET generates the set of files mentioned in section 4.3.4 and 6.3.4 when the option `-d` or `-draw` is set. This option has been developed during the thesis, making it more flexible and has more user interaction by adding two parameters which control it.

The first parameter for `-d`, `f`, is used to set the function which will be the root of the graph. The second option sets the depth level, i.e., the context depth level for the SG. These additions are implemented to be able to deal with larger flow graphs, and graphs with increased complexity.

The other option in this section is –v or –verbatim, it prints out the details of an error message; i.e. In other words, not just the warning/error message is printed out, it prints also out the NIC code where the error occurred.

# Chapter 7

# Method

In the beginning of this report the purpose and the main task of this project was described in a more general way. Since we have now described the tools used and theory behind WCET and flow analysis it is time to describe the methodology used to derive flow information and WCET estimates. The analysis method is divided into five phases:

1. Source Code Preparation & NIC Generation.
2. Deriving SWEET Annotations.
3. Flow Analysis using SWEET.
4. Source Binary Generation & aiT Annotations.
5. WCET Analysis using aiT.

Figure 7.1 shows a flow graph representation of the method, where the input/output files of each phase is viewed. The arrows show the path to a WCET estimate through the different phases. The chapter will contain a description of each phase, the tools, the input and the output files used by it.

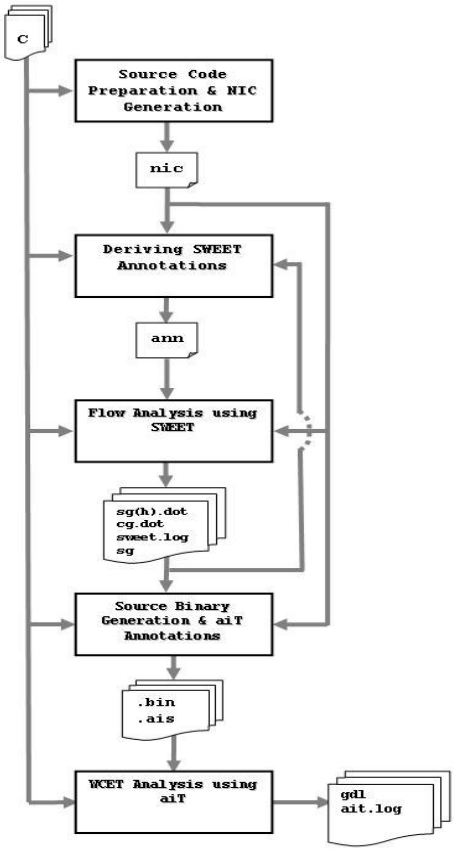



Figure 7.1: WCET analysis method flow chart

## 7.1 Source code preparation & NIC generation

The analyzed tasks are mostly written in C. However, there are some code sections written in assembler which belongs to Rubus OS. The system timer is an example of an assembler section in the system. The analyzed task had very little dependency on these assembler sections. Since the c2nic compiler supports only C code, the assembler sections which the tasks depended on had to be replaced by corresponding routines written in C.

A minor difficulty was that the system programming is done in a Windows environment which in theory should not be an obstacle, but there were some issues that had to be addressed prior to moving the code from the Windows environment to the Linux environment. An example is that all header file references were written with addresses using backslashes which is not supported in the preprocessing stage of the build script.

### 7.1.1 Function Wrapper

A more complex task in this section was isolating the task from the whole system. The isolation was necessary since it would be an infeasible task to compile the whole system source code to NIC format. The isolation was done by writing a wrapper for each one of the 13 tasks. The wrapper contains all the necessary declarations of data areas, structures and data/structure pointer used by the task. The wrapper also contains a root function (i.e., `main (void)`) containing a call to the task function with necessary parameters and the assignments of pointers and values to data areas.

The wrapper is in a sense a simpler version of the whole system. In some cases it also contains some math library routines, such as the modulus function `mod()`, which is still not supported by the NIC generator. These routines had to be written as functions in the wrapper and the math library paths were removed to force the use of them. There were also some arrays containing values needed by the task which had to be copied to the wrapper to reduce the size of the generated NIC code.

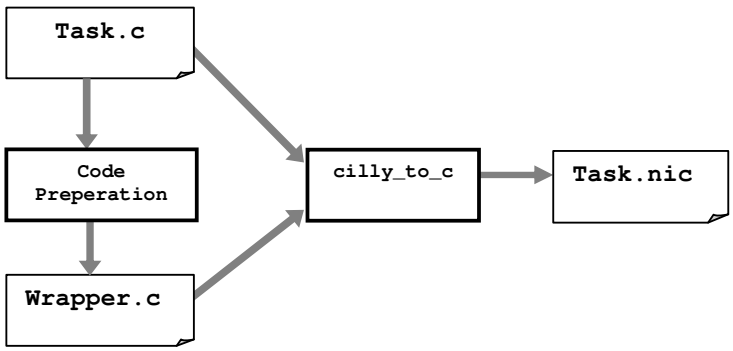



Figure 7.2: Phase 1, Code preparation & NIC generation

### 7.1.2 Phase 1 Result

The final step is NIC generation, using the `cilly_to_c` script. The input files are the function wrapper and the source code file of the task. The result of this phase is a NIC representation of the task. Worth mentioning is that each code statement is assigned a basic block number by the NIC generator.

This assignment makes the basic block definition a little bit blurred. For this reason SWEET first analysis step is merging these basic blocks together to form a basic block as defined in [1] section 2.3. Figure 7.2 visualizes the phase, the tools used, and the input and the output files of the phase.

## 7.2 Deriving SWEET Annotations

Most of the annotations written in this project assign value intervals, hereafter called value annotations, to variables which affect program flow, see annotation examples 4 and 5 in section 4.2. However, since the

Volvo code has a structure pointer as argument for all the analyzed tasks required using pointer annotations, see example 6 in section 4.2.
Basically we needed to find value intervals for the input and state members of the task's structure. The value intervals for the task's input members were extracted from source files located in subdirectories in the software project directory. Which was a relatively easy task comparing with finding the intervals for state members; see section 3.3 for details about Volvo code structures.

### 7.2.1 Function Wrapper - Revisited

As we mentioned in the previous section we had to isolate each task, in order to analyze it with SWEET. The wrapper used for task isolation is also used as a guide to find how the tasks associated memory area is organized. The need of knowing of how task variables and data structures are organized in the memory occurred when the structure members needed to be assigned value or pointer annotations.

The problem occurs when the NIC generator renames all the structure member/variables names to the structure name followed by an offset indicating at which offset in the structure data area contains its data. This makes it very hard for the user to map the code written in C to its corresponding NIC code. It also makes it hard to write an annotation to a variable, which is a structure member, without knowing where its data area is located, i.e., the offset value in bytes in the structure data area.

To facilitate this we assign a value to each input structure member in the main function. We search in the NIC file for the corresponding NIC code snippet for this assignment; the code snippet contains then the input structure data area offset where the assigned value is saved. This is repeated for every member in the input structure.

All the analyzed tasks had such assignments in this project, and to make it more safe all the assignment were repeated using pointer annotations for the pointer members of the structure and register annotation to point out the structure pointer (declared in the task wrapper) as its function parameter. Furthermore, the state variable value analysis (SVVA) requires the existence of these annotations. This is because of the value assignments written in wrapper. Which cause the resetting of all the state variables to their initial value.

### 7.2.2 State Variable Annotations

The state variables are special since they store their values after the task has finished executing and are used afterwards when the task is called again. Therefore, it was hard to know their value interval since they can represent a Boolean or be an accumulator which could be set to zero inside the code. That is when it reaches a certain value.

Furthermore, the state variables were used heavily in some tasks; some of the tasks had a larger number of state members then input members. This introduced the idea state variable value analysis (SVVA) in SWEET, which derives annotation for the state variables or any specified variable for that matter. Since the SVVA was implemented during this thesis it was not fully functional. But it successfully produced annotations for the task's state variables.

Resetting global variables is undesirable for SVVA, since it is built on saving the value of the state variables for next iteration. Each time the SVVA iterates with new input, see section 6.4.5 for more detailed functionality description, it begins the execution from the main function which resets all the variable values.

The solution was adding an argument, choosing where to begin the SVVA. By that we mean that the first run is from the main function, but after that the analysis runs from a user specified function. This in our case is the task function. This makes SWEET analyze a function with no value or pointer assigned from iteration nr 2. This is the reason why it needed the pointer and register annotations. The generated file in this phase is the annotation file (.ann), where some annotations are manually provided and other are SVVA generated.

### 7.2.3 Phase 2 result

Figure 7.3 shows the method process; note that the arrows show two ways through this phase, the dashed showing the use of SVVA analysis and the other one is without the SVVA. This is because there are some

tasks which didn't the need of SVVA, i.e. had no state variables to analyze which lead to that no input file for SVVA analysis is needed.

Note that there is an arrow back to the Task.ann from SVVA; this is because the annotations generated by SVVA are added to the manually written annotation file Task.ann.

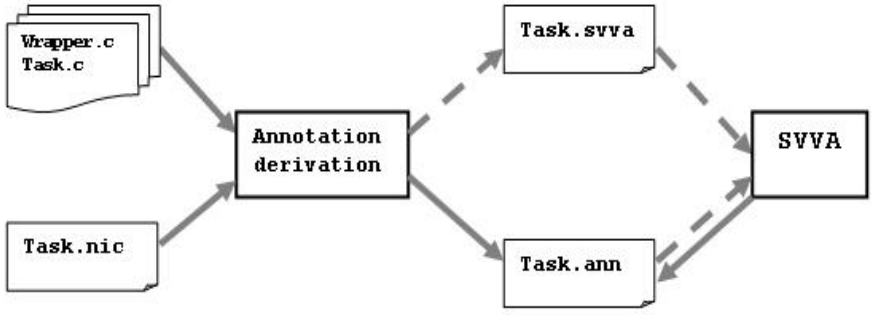



Figure 7.3: Phase 2, Deriving SWEET annotations.

## 7.3 Flow Analysis using SWEET

Flow analysis using SWEET can be done either by running a shell script or manually, using SWEET options, as mentioned in chapter 6. In this project the `runsweet` script was the most frequently used method, this because SWEET has so many options (with their arguments) that it resulted in very long option list when analyzing a task. The long option list made it difficult to guarantee that the same options were used during the analysis of all the 13 tasks. Furthermore, the runsweet script organized the project files, making them easier to find.

### 7.3.1 The General Case

As mentioned above the runsweet script is frequently used in this project, which is the general case. The phase is divided into four analysis steps using the analysis modes in runsweet, see section 6.3.4. These steps can be described as follows:

1. **Single path, basic mode -** The Task.nic generated in phase 1 is used in this step; the main reason using this analysis mode is to check if there is now unassigned pointer value or undeclared data area in the NIC file. SWEET asserts for all unsigned pointers and prints out an error message where the unsigned pointer is first located in the NIC code, its name, its size and the context where the error occurred is printed in the sweet.log file. During this step is we jump back and forth between phase 2, where the annotations for the pointers and registers are written, and phase 3. In this step we make sure that SWEET fulfills the analysis and generates the result files without any errors or warnings.

2. **Single path, advanced mode -** In this step we check if there are variables which need to be assigned value intervals. In this step SWEET usually produces flow facts of type infeasible nodes in the scope graph file, and it also gives information about how many basic blocks were visited during this run in sweet.log file. The basic block visits gives an idea of how much code SWEET has analyzed during this run. In this step it is often low since there is always an input value need to be assigned an interval to increase the number of visited basicblocks. The basic blocks which are found to be infeasible in this step often are if-statement branches which are controlled by an input or state variable. Having this information we jump back to phase 2 and write annotations to assign the input and state variables value intervals.

3. **Multi path, basic mode –** In this step we begin to use the first version of the annotation file, the main reason of performing this step is to check if there are any syntax errors in the annotation. Here we check if there is any difference in size and type of the annotated variables and the NIC generated size and type of the same variable. In this step more unsigned pointer could be discovered by SWEET. This is because SWEET now visits more basic blocks (it has more input combinations to analyze). Therefore the same process as in step 1 is followed to annotate these pointers.

4. **Multi path, advanced mode -** When the analysis reaches this step, a better version of the annotation file is available from step 3. However, there is no guarantee that all the pointers are assigned, since there could be still unvisited basic blocks because of some program flow affecting variables are still not annotated. The process described in step 2 is followed to maximize the number of visited basic blocks, hopefully visiting all the basic blocks. If that is not the case, there is some infeasible code (also called dead code) in the analyzed task.

These steps were used to derive flow information for twelve of the thirteen tasks. The remaining task was analyzed manually, i.e., we used some of SWEET options mentioned in section 6.4.

### 7.3.2 The Special Case

As mentioned above, we were unable to analyze one task using runsweet. This special case contained 33 condition statements, for each statement there is two possible paths which means that there are $2^{33}$ possible paths through the program. Each program path has its own abstract state; each state has its own copy of the whole program memory. This results in up to $2^{33}$ abstract states to analyze in this task.

The number of condition statements, and the fact that each abstract state has its own copy of the memory, made it impossible to analyze this task with `runsweet`. SWEET memory consumption increased rapidly when we analyzed this task with runsweet, this forced us to use the highest level of merging to decrease the number of the analysis states. This resulted in that we were unable to analyze this task with all the available infeasible path calculation options. See section 6.4 for more description of the options mentioned in this section.

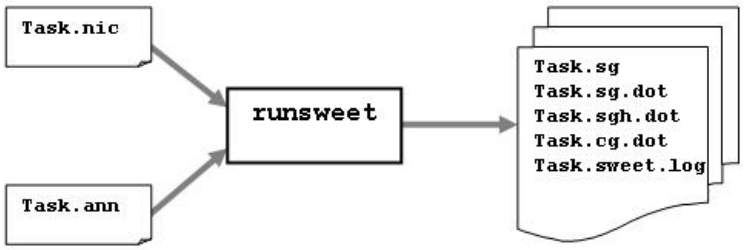



Figure 7.4: Phase 3, Flow Analysis using SWEET

### 7.3.3 Phase 3 Results

The result of the phase is the file set mentioned in section 6.3.4 with the exception of the special case task where the sweet.log file is not generated since we analyzed the task using SWEET manually. Figure 7.4 shows the result files from a general case analysis.

## 7.4 Source Binary Generation & aiT Flow Annotations

This phase prepares the tasks to be analyzed by aiT and translate the flow information generated by SWEET to aiT annotations. aiT uses the binary file which represents the whole ECU software to derive WCET estimates. The binary file used in this project was compiled using the c166-tasking compiler. It also has some options which had to be set to make an accurate analysis. These options configure the memory model used in the target hardware, register content and other hardware related option and values.

The binary and the analyzer settings are the same as Daniel Sehlberg used in his master thesis. We had to use these to make a fair comparison between the results of our analyses. The aiT annotations used in this project are also partially written by Sehlberg, since there are configuration annotations which are identical in both projects. Examples of these annotations are the clock rate annotation, known register value annotation and compiler annotation. For more information about the binary file, analyzer settings and annotations read chapter 3 in [3].

In this project we used a newer version of aiT to analyze the tasks, which made it necessary to reanalyze the tasks with the flow annotations Sehlberg found manually. The result, i.e. the flow graph, was used to derive flow annotations from SWEET flow facts. The flow annotations are derived in this step using the set of files generated in phase three, the process of mapping the flow facts is divided into the following steps:

1. Converting dot to PDF and generating the SWEET NIC file.
2. Interpreting SWEET flow facts.
3. Writing aiT flow annotations.

To prevent confusion it is worth to mention that this phase has a close relationship to fifth phase, with Sehlbergs project files we could regenerate the aiSee file (Project.gdl) which we use in the second step listed above.

### 7.4.1 Converting Dot to PDF

The purpose of this step is to aid the user to use and understand the flow graphs produced by SWEET in a more flexible way, there is some tasks in this project which have very large scope graphs therefore we need a tool with good zooming options(such as adobe acrobat reader). The dot files are converted using the dot2pdf script described in section 6.3.6.

The SWEET NIC file is the NIC representation of the code after entering SWEET. Before performing analysis SWEET merges together several basic blocks to one basic block. This merging is done to clarify which code statements are considered to be members of a basic block, which is defined in Section 2.3.3.

Therefore all the flow facts SWEET generates are addressed to the basic block, which the code statement the flow fact concerns is in it. To print out this NIC code we use the sweet-nic script, saving the results in Task.sn file.

### 7.4.2 Interpreting SWEET Flow Facts

Both SWEET and aiT visualize the flow graph using basic blocks as nodes, which facilitate the interpretation of the flow facts. The main difficulty in this step is to find the basic block(s) in aiT which corresponds to the basic block(s) which the flow fact is addressed to. The method we used is:

**First:** Visual pattern matching is made between the flow graph produced by SWEET and its correspondent in aiT; this to make sure that the same task has been analyzed.

**Second:** we take a look at the scope graph file where the flow facts are located, the basic block number extracted from the file is then found in the PDF flow graph file Task.sg.pdf.

**Third:** The NIC code corresponding to the basic block number is found in the Task.ann.sn file, the NIC code is then mapped to the corresponding C-code.

**Fourth:** The source code which corresponds to the SWEET basic block is then found in aiT flow graph.

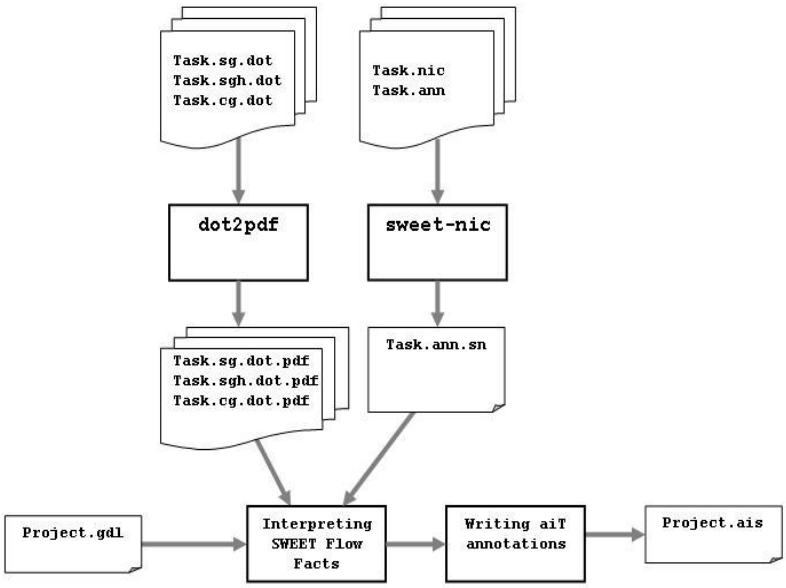



Figure 7.5: Phase 4, from SWEET Flow Facts to aiT Annotations

### 7.4.3 Writing aiT Flow Annotations

SWEET is a context sensitive analysis tool and presents its flow facts according to it, i.e., a basic block could have more than one flow fact if it is executed in more than one context. On contrary, aiT is not, which made it harder to interpret the calculated flow facts. The problem is when a flow fact does not yield for all program contexts according to SWEET flow analysis. This means that it is not guaranteed that the flow fact yield in the context where the longest execution path of the program is calculated.

It means that when we write the corresponding aiT annotation, the context which the annotation yields must be specified. But since aiT is not context sensitive, this kind of annotation does not exist. This occurred once, in task 12, where the task consists of several simpler functions. These call each other in different contexts functioning as a state machine, that meant that if a function is called more than one time each basic block in it has two flow facts of the same kind.

An example is when a basic block from the same function is called twice, in two different contexts. If SWEET calculates that it has been visited in one context and not in the other context (infeasible node calculation), it means that the basic block is not infeasible since it is visited sometimes during the analysis but in only one of the contexts. Therefore, the number of infeasible nodes in task 12, presented in table 8.2, is very large, but not all of them are infeasible in aiT.

As a matter of fact, the statistics over the visited nodes helps us do determine if any of these facts yield as an aiT annotation. If the number of visited nodes value is not 100%, it means one of the basic blocks has never been visited and therefore it is an infeasible node.

The interpretation of flow facts to aiT annotations is done in this step. As we mentioned in section 5.1 there are two annotation styles to choose between. We choose to write the annotations in absolute address style, because we would analyze a code which has been compiled and will not be changed. Furthermore, the style is easier to write due to the fact that basic block number could be found in the flow graph in aiSee tool. The number is the memory address where the first instruction in the basic block can be found. Figure 5.2 C shows the basic block number 0x0:0x21c is written on first line in the blue box.

The most frequently used aiT annotation is the flow constraint annotations. This is because these can be used to express the excluding pair flow fact easily. There were also some cases where it was necessary to use the infeasible path annotation due to the fact that code contained some infeasible code, i.e. dead code.

### 7.4.4 Phase 4 Results

The resulting files from this phase are the software binary file and aiT annotation file, Figure 7.5 shows the process where the SWEET flow facts are interpreted to aiT annotations. Note that the binary generation step is not visualized in Figure 7.5, nevertheless, this step exist in this phase of the analysis.

## 7.5 WCET Analysis using aiT

In the final phase of analysis process the annotation file (.ais), binary file (.bin) and machine setting file (.msf) is involved in this phase. The first two files are generated in phase 4 of this process. The machine setting file is generated by Sehlberg [3]. No changes in this file has been made since there is no changes in hardware used to run the analyzed code.

The results of this phase is two files; First is a log file (ait.log) where aiT print out information about the memory access for each routine (function), register content and WCET results for each function and the total estimate. The second file (.gdl) contain the graphical description of the program, i.e., the call- and the control flow graph. The .gdl file is used in aiSee to show these graphs together with the calculated WCET.

## 7.6 Method Summary

To summarize the above; a task is compiled with two compilers, one generating an intermediate format representation (NIC) and the other generating target hardware assembler instructions (Binary). The NIC file together with the C file is used to create an annotation file. The NIC file and the annotation file is then used as input to SWEET, where the task goes through program flow analysis. The flow information derived by SWEET are used to write aiT flow- and infeasible path annotations. These together with machine setting file and the binary are used to calculate WCET estimate for the task.

# Chapter 8

# Results

Using the analysis method described in Chapter 7, we will try to answer the questions in Section 1.2 in this chapter. The evaluation started with the simpler tasks and after SWEET successfully derived flow information for these we proceeded to analyze more complex tasks. The analysis results for each task are presented here.

## 8.1 SWEET Annotations

The number of input annotations needed by SWEET to derive correct flow facts varies between the analyzed tasks; this number depends on the size of the input structure. That is the number of input- and state- member that had to be annotated. Some the input members were also structures with at least two members, which had also to be annotated.

As mentioned in chapter seven, there were some tasks for which we were able to use SVVA to derive automatic state variable annotations. These tasks had additional annotations which were used to assign all the task pointers. Table 8.1 shows the number of annotations used for each task. “Interval annotations” are these which are necessary to derive flow facts without the use of SVVA and “Pointer annotations” are the number of additional annotations needed to analyze the task with SVVA. If zero then the SVVA is not used for the task. “SVVA annotations” are the number of interval annotations which are automatically derived by SWEET using SVVA.

| Task | Interval annotations | Pointer annotations | SVVA annotations |
|---|---|---|---|
| **1** | 4 | 0 | -- |
| **2** | 6 | 0 | -- |
| **3** | 7 | 0 | -- |
| **4** | 12 | 8 | 2 |
| **5** | 7 | 0 | -- |
| **6** | 7 | 0 | -- |
| **7** | 18 | 8 | 3 |
| **8** | 17 | 4 | 11 |
| **9** | 8 | 3 | 4 |
| **10** | 17 | 7 | 8 |
| **11** | 16 | 11 | 3 |
| **12** | 25 | 14 | 15 |
| **13** | 69 | 43 | 0 |

Table 8.1: Number of annotations needed for each task.

Task 13 is the only task which SVVA was unable to analyze; it would take infeasible amount of time to extract automatic annotations for this task. The reason for this is that it takes nearly 6 CPU hours for SWEET to execute the task for all input combination which corresponds to a single iteration for SVVA.

Since SVVA iterate this analysis several times, it would take months to extract the 85 state variables in task 13 with SVVA.

## 8.2 SWEET Flow Facts

The flow facts derived by SWEET are presented here. The section begins with presenting the flow facts derived by running flow analysis using the runsweet script. The results are presented according to the runsweet modes used to derive it. More information about `runsweet's` modes can be found in section 7.3. The loop bound analysis will be mentioned briefly, due to the limited number of the analyzed tasks which contain loops.

### 8.2.1 Infeasible Edges, Nodes and Paths

There are two analysis cases to consider; the general case, there we used the runsweet script, and the special case, there SWEET option were used. The subsection below will present the special case tasks result. In section 8.2.2 results for the remaining tasks are presented.

***Special Case Analysis***

Task 10 falls into the special case category; the highest level of merge had to be used in order to be able to analyze this task. The high merging level limits the use of all infeasible path `(-ip)` option in SWEET, only the max node- and edge count (mnc and mec) are available then. The analysis resulted in finding two infeasible if-statement edges in the task. These indicate that the code snippet in this edge is never visited during the analysis, which could be an indication of the existence of dead code in the task.

***General Case Analysis***

The remaining tasks were analyzed using runsweet. Table 8.2 presents the results when analyzing the tasks with the multi path, advanced mode in runsweet (see section 7.3.1). The table contains the number of flow facts derived for each task and its type, i.e., infeasible nodes, infeasible paths and excluding pairs.

Each one of the columns 2-5 corresponds to one of the infeasible path calculation option (-ip) arguments (see section 6.4.4 for detailed description). Furthermore, it is worth mentioning that there is more of the option –ip arguments used in runsweet, their result have not been presented since it would give the same flow information provided by the flow facts mentioned above.

| Task | Infeasible nodes (in) | Infeasible nodes all iterations(ina) | Infeasible paths (ip) | Excluding pairs (ep) |
|---|---|---|---|---|
| **1** | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 4 |
| **5** | 0 | 0 | 1 | 2 |
| **6** | 2 | 2 | 0 | 0 |
| **7** | 0 | 0 | 5 | 7 |
| **8** | 0 | 0 | 0 | 5 |
| **9** | 0 | 0 | 0 | 0 |
| **10** | -- | -- | -- | -- |
| **11** | 0 | 0 | 0 | 4 |
| **12** | 379 | 354 | 8 | 10 |
| **13** | 124561 | 121419 | 340 | 346 |

Table 8.2: Number of flow facts for each task sorted by flow fact type.

### 8.2.2 Loop Bounds

Only the tasks 6, 7 and 13 contained loops, see Table 8.3. The loop in task 7 was written in assembler which we had to remove to make the source compatible with NIC compiler. We rewrote the code section which depended on the loop.

SWEET loop bound analysis had no problem calculating bounds for these loops. We were able to perform the loop analysis on task 13, because loop bound analysis can be done separately and it's not depended on the flow analysis part where infeasible paths, edges and nodes are derived.

| **Task** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Loops** | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 18 |

Table 8.3: The number of loops in each task.

## 8.3 Analysis Time

A good measurement is the time SWEET takes to perform flow analysis for each task. Table 8.4 contains the time in CPU seconds for each task. Note that task 10 is analyzed differently from the other tasks therefore its time measurement is not directly comparable with the other times. The fact that less infeasible path calculations and higher merging level is used makes the analysis faster.

The last column presents a percentage value for the number of visited nodes during analysis; this value is indication if the task contains dead code. One exception is task 13; SWEET managed to analyze only a part of this task since the input annotations were never completed due to the time limitation of this project.

| **Task** | **Analysis Time** | **Visited nodes** |
|---|---|---|
| **1** | 2,75 | 100% |
| **2** | 0,02 | 100% |
| **3** | 2,72 | 100% |
| **4** | 7,47 | 100% |
| **5** | 0,12 | 100% |
| **6** | 0,07 | 92% |
| **7** | 5,65 | 100% |
| **8** | 0,41 | 100% |
| **9** | 0,04 | 100% |
| **10** | 15,08 | 97% |
| **11** | 4,35 | 100% |
| **12** | 173,34 | 99% |
| **13** | 20467 | 25% |

Table 8.4: Analysis times measured in CPU seconds and a percentage value of visited node during analysis.

Note that task 13 has taken approximately 6 hours to analyze, even though only 25% of the basic blocks were visited. A good prediction is that it would take more than 24 hours to perform a complete analysis of it; this makes task 13's size and complexity the upper limit of what SWEET can perform flow analysis on.

## 8.4 aiT Annotations

As mentioned in section 7.4.3, there are some cases where the context sensitivity gives us infeasible node flow facts which can't be interpreted to aiT annotations.

As we mentioned earlier, due to project time limitation and the limited flow analysis power of SWEET, we were not able to derive any aiT annotations for task 13. There is also the special case flow analysis, i.e., in task 10, where we could determine that there are two infeasible if-statement edges are infeasible.

Table 8.5, shows the number of SWEET produced flow facts that suitable aiT annotations. As comparison, the last column shows the number of annotations Sehlberg found by hand for the tasks. Note that the loop bound annotations are not counted with the flow fact numbers presented in it.

| Task | aiT Annotations by SWEET | aiT Annotations by Sehlberg |
|---|---|---|
| **1** | 0 | 2 |
| **2** | 0 | 1 |
| **3** | 0 | 1 |
| **4** | 4 | 3 |
| **5** | 3 | 6 |
| **6** | 2 | 2 |
| **7** | 12 | 6 |
| **8** | 5 | 5 |
| **9** | 0 | 4 |
| **10** | 2 | 8 |
| **11** | 4 | 4 |
| **12** | 19 | 18 |
| **13** | -- | 103 |

Table 8.5: aiT annotation automatically derived by SWEET and by hand by Sehlberg.

## 8.5 WCET Comparison

Most of the aiT annotations found by SWEET are the same as the ones found by hand, or they annotate basic blocks which the longest execution path doesn't go through. This was determined by analyzing the task with aiT twice, first with the manually found annotations and the second run with the new founded annotations by SWEET.

We draw the conclusion that SWEET does not necessarily derive flow information which tightens the WCET estimates. In task 12 we found an infeasible node, i.e. dead code, which couldn't be found by hand. There is also task 10 where more flow annotations could be found by, that because, as mentioned earlier, the limited performance of SWEET flow analysis.

## 8.6 aiT v1.5 vs. v2.0

To increase the fairness of our comparison all the analyzed tasks was reanalyzed with newer aiT version 2.0; this is because in Daniel Sehlberg's project aiT version 1.5 was used. The results differed but no further investigation was done to discover the reason, since it is out of thesis scope. Table 8.6 shows the WCET estimates for each task, sorted by the aiT version were used to estimate it.

| Task | aiT v1.5 | aiT v2.0 |
|---|---|---|
| **1** | 4,788 | 4,849 |
| **2** | 5,879 | 5,879 |
| **3** | 12,122 | 12,304 |
| **4** | 13,394 | 13,455 |
| **5** | 22,364 | 22,485 |
| **6** | 27,243 | 27,545 |
| **7** | 29,091 | 29,334 |
| **8** | 30,455 | 28,273 |
| **9** | 36 | 34 |
| **10** | 55,091 | 60,91 |
| **11** | 70,273 | 61,364 |
| **12** | 143,606 | 146 |
| **13** | 1447 | 1421 |

Table 8.6: aiT WCET estimates with version 1.5 and version 2.0 in microseconds.

## 8.7 Additional Information

Beside the flow facts presented in section 8.2, SWEET provided some additional information about the code style in the tasks. One problem in the code worth mentioning is the out of array addressing, which was revealed during the analysis of one task. After some investigation we found the cause of the problem, it was caused by one condition used in a While-loop.

## 8.8 Problems and Solutions

Since SWEET is still a prototype, several issue had to be addressed in order to begin the flow analysis with it. In this section we will present some of time consuming tasks which had to be done in order to perform successful flow analysis on Volvo code.

The most time consuming task during this project was to study the NIC compiler; i.e. how it organized memory, statements and functions. Under the first four weeks of this project we tried to analyze the simplest tasks using SWEET, unfortunately with failure since we could not reveal how the input structure was interpreted by NIC. SWEET always halted on unsigned pointers in the tasks. After some investigation, compiling some test cases and creating the wrapper, we could reveal how we address the input structures.

After this break through we were able to write the function wrapper for each task, and write the first input interval annotations for it. We began then to check each task to make sure that the input intervals are correct and all the possible pointers are correctly assigned. It took another four weeks to complete to perform this task.

During the whole project time SWEET was improved, adding new options and functionalities to improve its performance. Type Check Annotations (-tca), Generate “dot” files (-draw) and State Variable Value Analysis (-svva) presented in section 6.4 are some examples of new added and improved functionalities of SWEET.

# Chapter 9

# Conclusions

The main goal of this project was to evaluate the WCET analyzer SWEET using real life embedded system software, which has been successfully done. Since it was the first time that SWEET has been used on real life embedded code several modifications had to be done to it make more user friendly. SWEET managed to analyze almost all the thirteen tasks assigned to it, but due the time limitation of this project we were unable to analyze task 13 completely.

## 9.1 Preparation Work

The preparation work done on code making it analyzable by SWEET is still very time-consuming task, the main reason is the NIC compiler. There is not so much documentation available for this compiler since it is a university prototype. Some of the bigger delays during project time were caused by the NIC compiler, for example the memory organization and replacing the name of the structure members by the structure name and an offset.

This preparation work results as much effort as it was spend to find the derive flow annotations manually. A suggestion would be to change the NIC language with a more informative intermediate representation language.

## 9.2 WCET Effects

The flow facts calculated by SWEET were successfully translated to aiT annotations (except for task 13); these flow facts have been compared with the ones found by Daniel Sehlberg [3]. The following conclusion can be drawn; SWEET managed to find “at least” the same flow facts found previously by hand, but these didn’t affect the WCET estimates for these tasks.

By “at least” we mean that in some analysis cases there were infeasible nodes SWEET detected, which have never been detected before. In addition, some other information about the code was found when we analyzed the tasks with SWEET. This information could be a great help for writing safer embedded system code.

## 9.3 SWEET for Volvo

There is still a long way ahead for SWEET to become a commercial product. Even though the flow analysis part of it is nearly fully implemented it is too early to state if it is of any good use for Volvo. Furthermore, our analysis shows that the SWEET’s flow analysis capabilities are limited, due to the fact that task 13 is the biggest code which can be analyzed in reasonable amount of time.

We believe that task 13 is a example of code in Volvo software, which needs to be analyzed by a flow analyzer due to its complexity.

## 9.4 State Variable Value Analysis

This function is worth mentioning since it has been developed and implemented during the time this project went on. According to the shown results, this option works successfully. This makes us draw the conclusion that more research power should be put on developing this function further.

## 9.5 Future Work

The main focus of SWEET development should lay on making the analysis faster and user friendlier. Functionalities such as type check annotations, State Variable Value Analysis are worth further development, since they facilitate the use of SWEET considerably.

The flow analysis part should be more focused on in aiT. Our method shows that there is the possibility of making the flow analysis part of SWEET a plug-in for aiT, which could decrease the labor spent to find flow information for aiT.

Furthermore, a better intermediate code format is needed. In this thesis a lot of work was spent to adapt the code to the NIC compiler, which can be avoided if a more standardized intermediate code format is used in SWEET.

The algorithms in SWEET to derive flow information have tight relationship with each other. In the sense of that many of the complex algorithms are a further development of the simpler ones. This makes SWEET to perform a lot of redundant analysis and thereby many flow facts produced are redundant. A function that eliminates the redundant information would make the results more readable.

# Bibliography


[1] A. Ermedahl. A Modular Tool Architecture for Worst-Case Execution Time Analysis. PhD thesis, Faculty of Science and Technology, Uppsala University, June 2003.

[2] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis using Abstract Execution , The 27th IEEE Real-Time Systems Symposium (RTSS 2006), Rio de Janeiro, Brazil, December. 2006.

[3] D. Sehlberg. Static WCET Analysis of Task-Oriented Code for Construction Vehicles. Master's Thesis. Dept. of Computer Science and Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden 2005.

[4] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a Flow Analysis for Embedded System C Programs, The 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'05), Sedona, USA, February 2005.

[5] Arcticus Systems Website. URL: http://www.arcticus-systems.com/

[6] Computer Encyclopedia. URL: http://www.answers.com/topic/real-time

[7] K. Hänninen, T. Riutta: Optimal Design. Master's Thesis. Dept. of Computer Science an Engineering, Märlardalen University, Box 883, S-721 23 Västerås, Sweden (February 2003).

[8] SWEET manual, under development, copy on request from the WCET group.

[9] The Worst Case Execution Time Project (WCET) website.
URL: http://www.mrtc.mdh.se/projects/wcet/research.html

[10] Execution Time Analysis for Embedded Real-Time Systems, A. Ermedahl

[11] The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools

[12] Wikipedia, URL: http://en.wikipedia.org/wiki/Basic_block

[13] S. Bygde: Abstract Interpretation and Abstract Domains. Dept. of Computer Science an Engineering, Märlardalen University, Box 883, S-721 23 Västerås, Sweden (May 2006).

[14] The Dot Language. URL: http://www.graphviz.org/doc/info/lang.html

[15] Graphviz, Graph Visualizing Software. URL: http://www.graphviz.org/

[16] C. Ferdinand and R. Heckmann. Worst-Case Execution Time Prediction by Static Program Analysis. In 18th IEEE International Proceedings on Parallel and Distributed Processing Symposium, 2004.

[17] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper. Faster WCET Flow Analysis by Program Slicing. In Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'06).

[18] AbsInt Angewandte Informatik GmbH website. URL: http://www.absint.com

[19] AbsInt Angewandte Informatik GmbH, *aiSee* website. URL: http://www.absint.com/aisee/

[20] Ubuntu Operating System website. URL: http://www.ubuntu.com

[21] A Retargetable Compiler for ANSI C website. URL: http://www.cs.princeton.edu/software/lcc/

[22] SML Compiler website. URL: http://www.smlnj.org/

[23] CIL-Infrastructure for C Program Analysis and Transformation. URL:http://hal.cs.berkeley.edu/cil/

[24] The Caml Language website. URL: http://caml.inria.fr/

[25] GCC, the GNU Compiler Collection website. URL: http://gcc.gnu.org/

[26] TASKING - Embedded software development tools website. URL: http://www.tasking.com/

[27] Emulator, Figure 2.4(b). URL: http://www.data-action.com/images/R3K_H_tn.jpg

[28] Logic-Analyzer, Figure 2.4(a). URL: http://www-search.sonoma.edu/users/b/bloomal/graphics/1680_logic_analyzer.JPG

[29] Rapita Systems Ltd. website. URL: http://www.rapitasystems.com/

[30] O .Eriksson and Y .Zhang. Evaluation of Static Time Analysis for CC Systems. Dept. of Computer Science an Engineering, Märlardalen University, Box 883, S-721 23 Västerås, Sweden (August 2005)

[31] Tidorum Ltd. website. URL: http://www.tidorum.fi/bound-t/

[32] Jan Gustafsson: Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation. PhD thesis, Department of Computer Systems, Information Technology, Uppsala University, May 2000.