Abstract

Today, traditional mechanical and electrical systems are replaced with special ICT (Information and communication technology) based solutions and with the invention of new technologies; this trend is increasing further more. This special ICT-based domain is called Real-time systems and today’s drive-by-wire, electronic stability programs in car, control software in vehicles are just a few examples of real time systems. The task is a fundamental element of the software in a real-time system, and it is always necessary to know the longest execution time of a task, since missing a task’s deadline is a not allowed in a time critical hard real-time system.

The longest execution time of a task or the Worst Case Execution Time (WCET) is estimated by WCET analysis. This estimation should be tight and safe to ensure the proper timing behavior of the real time system. But this WCET analysis is not always easy to perform, as the execution time of a task can vary by software characteristics like program flow or input data and also by hardware characteristics like speed of CPU, cache, pipeline and others.

There are several methods and tools for WCET analysis. Some of them are commercial products and other are research prototypes. To verify and validate WCET analysis tools, evaluations of the tool’s properties are important, and thus WCET benchmark programs has emerged in recent years. These are intended for comparison between these tools properties and associated methods.

The Mälardalen WCET benchmark suite has been maintained to evaluate the properties of various tool sets. In this thesis these benchmarks programs have been analyzed by SWEET (Swedish WCET Analysis Tool), the main tool used in this thesis. SWEET is a research prototype for WCET analysis. The main goal of this thesis work was to extend existing benchmark programs for WCET tools. It was obvious that most work load will be on benchmark program extension and at the beginning the work has been started by analyzing different small WCET benchmark programs.

The evaluation of SWEET’s properties has been taken into a further extent by analyzing another benchmark program which is called PapaBench, a free real-time benchmark from Paparazzi project that represents a real-time application, developed to be embedded on different Unmanned Aerial Vehicles (UAV). Lots of time was required to complete the analyzing of PapaBench. The main reason behind this extensive work was that we decided to participate with SWEET in WCET Challenge 2011 (WCC 2011).

So the purpose of the thesis ultimately turned into analyzing PapaBench instead of extending the WCET benchmark programs. The result of the thesis work is therefore mainly the analysis results from the analysis of PapaBench, which were reported to WCC 2011. The results from WCC 2011 are included in a paper presented at the WCET 2011 workshop, which took place in July 2011 in Porto, Portugal.

Another part of the work was to examine real-time train control software which was provided by Bombardier. The main reason behind getting these industrial codes was to possibly add new benchmark programs to the Mälardalen WCET benchmark suite. A thorough manual study of this code has been performed to find out whether new benchmark programs could be found. However, due to its structure and size, we decided that this code was not suitable to add to the Mälardalen WCET benchmark suite.
Acknowledgements

This thesis was a part of my Master degree in Software Engineering program at IDT in Mälardalen University. I would like to thank my thesis supervisor Jan Gustafsson for his help in all the way of the whole thesis work. I would also like to thank Andreas Ermedahl in the Mälardalen WCET group for helping me in various ways, including fixing a Linux script. Filip Sebek, who is an employee from SE division of Bombardier Transport, helped me a lot with the code of train control software.

I would also like say thanks to my wife Sonia for supporting me with the entire thesis work.
Contents
1. Real-time Systems and Timing Analysis...........................................................................................................8
  1.1. Introduction..........................................................................................................................................................8
    1.1.1. Hard and soft real-time systems......................................................................................................................8
    1.1.2. Event-triggered and time-triggered real-time systems......................................................................................8
    1.1.3. Interaction with the environment via sensors and actuators .........................................................................9
    1.1.4. Task and task instances ..................................................................................................................................9
  1.2. Time critical systems and the need of software timing analysis .................................................................9
  1.3. WCET analysis ................................................................................................................................................10
  1.4. Dynamic WCET analysis ................................................................................................................................11
  1.5. Static WCET analysis ........................................................................................................................................11
    1.5.1. Flow analysis and flow facts ........................................................................................................................12
    1.5.2. The low level analysis phase ........................................................................................................................13
    1.5.3. Calculation phase ...........................................................................................................................................14
    1.5.4. Hybrid WCET analysis ..................................................................................................................................16
2. WCET analysis tools ..............................................................................................................................................17
  2.1. Static analysis tools..........................................................................................................................................17
    2.1.1. Commercial static analysis tools ..................................................................................................................17
    2.1.2. Research prototype for static analysis ........................................................................................................18
  2.2. Hybrid analysis tool ..........................................................................................................................................18
3. Description of SWEET ..........................................................................................................................................19
  3.1. The ALF language...........................................................................................................................................19
    3.1.1. Syntax ............................................................................................................................................................19
    3.1.2. Memory model .............................................................................................................................................19
    3.1.3. Program model .............................................................................................................................................19
    3.1.4. Data model ....................................................................................................................................................19
    3.1.5. Values ...........................................................................................................................................................19
    3.1.6. Type system ..................................................................................................................................................20
    3.1.7. Operators ....................................................................................................................................................20
    3.1.8. Statements ...................................................................................................................................................20
    3.1.9. Semantics .....................................................................................................................................................21
    3.1.10. ALF grammar ............................................................................................................................................21
    3.1.11. Example of an ALF program ....................................................................................................................21
3.1.12. Flow analysis using ALF .......................................................... 21
3.2. Translators to ALF ........................................................................ 22
  3.2.1. The melmac tool ...................................................................... 22
3.3. SWEET flow analysis .................................................................... 22
  3.3.1. Abstract execution .................................................................. 23
  3.3.2. Generated graphs .................................................................... 23
  3.3.3. Input annotations .................................................................... 24
  3.3.4. Flow facts generation and the flow fact language ..................... 25
3.4. Early timing analysis using SWEET ............................................... 26
3.5. Low level analysis (low-SWEET) .................................................. 26
3.6. SWEET WCET calculation ............................................................ 27
4. The Bombardier Train Control System ............................................. 29
  4.1. Brief description of the system .................................................... 29
  4.2. Task generation and code properties .......................................... 29
  4.3. Tasks with loops ........................................................................ 30
  4.4. Discussion ................................................................................. 30
5. WCET Benchmarks .......................................................................... 31
  5.1. Mälardalen WCET Benchmarks .................................................. 31
    5.1.1. Examples of analysis using SWEET ....................................... 32
  5.2. PapaBench .................................................................................. 33
    5.2.1. Analysis of PapaBench using SWEET ................................... 33
6. WCET Challenge 2011 .................................................................... 35
7. Analysis of PapaBench for WCC 2011 .............................................. 36
  7.1. Cross compilation of PapaBench program ................................... 36
  7.2. Compiling the PapaBench program ............................................ 37
  7.3. Processing Fly by Wire ............................................................... 37
    7.3.1. Error and correction ............................................................. 38
    7.3.2. Converting the source code to .alf format ......................... 38
    7.3.3. Problems and solutions during C to ALF conversion ............ 38
  7.4. Processing Autopilot ................................................................... 39
    7.4.1. Converting the source code to .alf format ......................... 40
    7.4.2. Corrections .......................................................................... 40
  7.5. Automation of PapaBench program analysis ............................... 40
  7.6. Analyzing PapaBench with SWEET ........................................... 40
  7.7. Problems encountered .................................................................. 41
7.8. Analysis of the problems in WCC 2011 and our results

7.8.1. Problem: AutoPilot A1 – UAV driving according to the flight plan

7.8.2. Problem: AutoPilot A2a – navigation management in HOME mode

7.8.3. Problem: AutoPilot A3 – send the GPS position to ground

7.8.4. Problem: Fly-By-Wire F1b: AutoPilot command transmission to servos (no command to transfer)

8. Conclusions and Future Work

8.1. Conclusions

8.2. Future work

9. References
Overview of the Thesis

Chapter 1 presents the theory of real-time systems and timing analysis and the different types of WCET analysis. Chapter 2 describes the different types of WCET tools that exist today. Chapter 3 goes into depth in describing SWEET, the main tool used in this work. The chapter also describes ALF (ARTIST2 Language for WCET Flow Analysis), which is an intermediate code format used by SWEET during the analyses, and the translators to the ALF format. The flow analysis of SWEET is described, as well as the low-level analysis and WCET calculation.

Chapter 4 describes the Bombardier Train Control System. This software was studied to see if it could offer additional benchmark programs. It is a giant with a huge number of lines of codes.

Chapter 5 describes examples of WCET benchmarks, which are used to evaluate WCET tools and methods. The Mälardalen WCET benchmark programs were first taken into account as a starter for WCET analysis. WCET estimation and program flow analysis for these programs was performed with the help of SWEET at the beginning of this thesis work. All the needed files exist in the Mälardalen benchmark website. This analysis was performed to get familiar with the SWEET WCET tool and at the same time to get familiar with the code properties of these programs. Without any knowledge of code properties of existing programs, it is not possible to add new benchmark programs with different code properties. So, consequently no detailed results were presented in this report, as they have previously been analyzed for various purposes. Chapter 6 also introduces PapaBench, another WCET benchmark, and an example showing the analysis of the code with SWEET.

Chapter 6 presents the WCET Challenge 2011 (WCC 2011). Since we (the author, the supervisor and the examiner) decided to go for participation in participation of SWEET on WCC 2011, this actually became the main goal of the thesis work.

Chapter 7 describes the analysis of PapaBench for WCC 2011. Most of the efforts in this thesis work were actually spent on this analysis. It took lots of time and efforts because of mainly two reasons. One is preparation of source codes of PapaBench for analysis with SWEET, which was a large task. We also encountered errors in the ALF files, due to problems in the C to ALF translator, and explored bugs in SWEET. At the end of the chapter, we present our solutions to the analysis problems defined in WCC 2011.

Chapter 8 presents conclusions and future work, and Chapter 9 lists the references used in the thesis report. Appendix A contains the report to WCC 2011.
1. Real-time Systems and Timing Analysis

1.1. Introduction

A system is called a real-time system if,

1. It reacts upon outside events and performs function based on those and gives response within a certain time.
2. The correctness of the function not only depends on the accurate results but also on the time when those results are produced.

For achieving predictable timing behavior a real-time system should have

1. Timeliness: As results have to be correct not only in functional domain but also in temporal domain, the RTOS must provide kernel specific mechanism for time management and for handling tasks with explicit time constraints and different criticalness.
2. Predictably fast handling of events: the system must be able to predict the consequence of any scheduling decision to ensure minimum level of performance.
3. Possibility to prioritize among tasks: The real time system must be able to prioritize among tasks in importance order (e.g. which task is more important).

In real-time systems, late data is or delayed perception information is considered as bad data. For example, a real time system in self guided missile is to sample various data at the same time to ensure the attack on a specific target. So there will be several obstacles in various moments (time). Depending on the obstacle in a specific moment (time), the missile must change its direction. If at time $t$ the dimension of an obstacle was 4 square meter, then the missile should get this obstacle dimension at the time $t$. But if it get the dimension information at time $t+2$ instead of $t$ then there would be a clash at time $t$.

A real-time system is not equivalent to a fast system. Fast computing implies to the fact that the average response times for a given set of tasks would be minimized. For example, a computing system in the SETI (Search for Extra Terrestrial Intelligence) project must be very fast to shorten the average response time for several tasks. However, the response time of the individual task are not crucial. In a real time system however, individual task constraints are important.

Every task has both a functional and a temporal domain. So individual timing constraints must be met and the functionality must fulfill the system specification. So, rather than being fast a real time system,a real-time system should have predictability.

1.1.1. Hard and soft real-time systems.

Missing deadlines is not acceptable in a hard real time system. As it controls the environment, missing a deadline could cause serious damage. For example, if a real time system in a submarine is unable to detect the pressure of water, then the submarine may go down in deep water where water pressure will destroy the submarine. The designer must be able to predict the peak-load performance and ensure that the system does not miss the predefined deadlines.

If a deadline is missed in a soft real time system, only the quality of the service will be reduced. Missing a deadline won’t cause serious or catastrophic damage, only some computation will be useless. For example, if a task misses its deadline in a VOIP (voice over internet protocol) system, there will only be some low quality voice in a specific moment.

1.1.2. Event-triggered and time-triggered real-time systems.

Event triggered real-time systems: the events that are external to the system are responsible to trigger some tasks. Usually sensors are used to generate interrupts that trigger tasks when temporal control is enforced from the environment onto the system. System with sufficient computing power will make
this system work well and system with heavy load conditions may make them fail. Clock interrupts will occur may be at regular or irregular intervals. For example, an automatic door will only be opened when there are some objects in the range of door sensors.

*Time triggered real-time systems:* all activities are initiated at predetermined points in time. Real time systems of this kind are time triggered in a sense that all activities are controlled by a recurring clock tick. Usually a certain scenario is repeated over a regular interval of time. This means that sensors will only sample objects when the clock ticks. For example, in car manufacturing industry all parts are assembled by industrial robots which have sensors that sample the presence of different parts of cars in an assembly line at a regular interval of time. In this case sensors samples only by a clock tick.

1.1.3. Interaction with the environment via sensors and actuators.

*Sensor:* As computer system can also process digital data, it is sensors which transform physical data into digital format. There are many kinds of sensors available for various activities. For example a light sensor will initiate some activities when it detects light. The process of taking physical data is called sampling.

*Actuator:* Actuators transform digital data into physical action. For example, the real time system in a robot cannot act on its environment without having any actuator (e.g. arms, legs). The process of performing actions on the environment is called actuation.

One example of a real-time system that uses both sensors and actuators is a multi story building elevator. The real time system is used to ensure the maximum security for the passengers. In this system, sensors are used for various purposes (like speed, floor detection, and overweight). For example, if the limit of a lift is exceeded by people, the pressure sensor can detect the overweight. Then there will be some warning by a speech or any other technology. If the system is using an alarm to warn the passengers, then this alarm is an actuator.

1.1.4. Task and task instances.

Task and task instances are fundamental elements of real-time systems.

*Task:* a sequential program in real-time systems is called a task. A task performs a specific activity and possibly communicates with other sequential programs (tasks) in the system.

*Task instance:* a periodic task is executed periodically, with a certain time interval between two consecutive invocations and in every period or each invocation the portion of a periodic task is called a task instance.

Real-time timing constraints for a real-time application are usually imposed by some physical constraints of the controlled system. An example can be a GPS (Global Positioning System) based navigation system is used in vehicles to get the real-time mapping of its positioning. For these GPS systems it is necessary to get the current location at current time. For example the vehicle is in position x,y,z at time t. After a few moments it will be in position x1,y1,z1 at time t1, as it is moving. So the sampling of the GPS system should be very frequent to get the current location. For example if a navigation system’s sampling rate is 20 times per second, then to get the current location the deadline for each sample is 50 ms. If the sampling misses its deadlines the current location won’t be accurate.

From above it can be concluded that embedded real time systems are usually used to control the environment. Therefore, failure of these real time systems may cause serious damage. To guarantee the functionality of these systems, predictability is the basic requirement of a real time system. There must be guaranteed response time for all tasks in a real time. So let’s have a look on how timing is studied by WCET analysis.

1.2. Time critical systems and the need of software timing analysis

To ensure the proper timing behavior, timing analysis is performed in today’s industry. This is necessary to schedule the tasks in a time critical real time system to guarantee the right or precise
To meet the correct deadline, tasks are analyzed for prediction of the worst case execution time of respective task.

When the timing analysis is done by performing worst case execution time analysis, results are used to design and schedule the tasks of respective time critical system which is a reliable system with proper timing guarantee and correct precision value.

In today’s industry, real-time systems are complex and the complexity is increasing more and more with the evaluation of new invention of hardware and software architectures. Without proper timing analysis, it is difficult to get a reliable time critical hard real time system. Worst case execution time analysis is performed to ensure the reliability of the system.

Many scheduling algorithms and schedulability analysis in real-time system require having some form of knowledge about the worst-case timing of a task. WCET analysis has a much broader application area. WCET analysis is a natural tool to apply in any product development where timeliness is important. By using WCET analysis, designing and verifying hard real-time systems (i.e. a system where a missed deadline is unacceptable) can be simplified instead of using extensive and expensive testing. WCET analysis may be used to assist in selecting appropriate hardware. The designers of a system can take the application code they will use and perform WCET analysis for a range of target systems, selecting the cheapest processor that meets the performance requirements.

1.3. WCET analysis

The execution time of a task varies because of varying input data for the system and also generally, the environment of the task (like the system state). The shortest execution time is called best-case execution time (BCET) and the longest execution time is called worst-case execution time (WCET). The actual worst-case execution time (WCET) is the longest possible execution time of a program or task when the program is run on its target hardware. The BCET is usually used in connections with control application where output must be sent to the controlled object neither too soon, nor too late.

Figure 1 depicts the WCET and BCET of a task and the real-time properties of a task. In figure 1 we can also see the variations of task’s execution times by probability distribution of execution times.

The goal of the WCET analysis is to produce bounds of the WCET. The bound is valid for using in hard real-time system when it is safe i.e. guaranteed not to be less than actual WCET. The bound is even more useful when it is tight, i.e. provides small overestimation compared to the actual WCET.

Often, the longest time of a task execution is measured by timing the execution with different test data. For example, if we have test data that will cover all the paths and branches of our sample code, we can get the longest time for a task to be taken for its execution. But it is often impossible to do an exhaustive testing of all inputs, since the set of input value combinations can be huge.

Another problem is that the execution time for each instruction will depend on the contents of registers in a processor that we are using. So the same code will have different task execution time in different processors. So it is important to consider each low level instruction generated by the high level sample code.

Yet another problem is that we have cache and pipelining in the processors and these will also affect the execution time of sample code. So we may have different execution time for the same task with same test data.
Thus, finding the bounds on execution times of a task is not so simple. There are different methods and tools for this purpose.

1.4. Dynamic WCET analysis

Today in industry, a common method used to derive WCET estimates is by measurements and this measurement based approach is called dynamic WCET analysis. There are variety of tools that are employed for measurement based analysis i.e. emulators, logic analyzers, and oscilloscopes. The principle behind this method is that running the same program many times and try different “bad input” values to provoke the WCET.

It can easily be understood that the method is time consuming and very difficult. It is hard to guarantee that the actual WCET has been found, and each test run only gives the execution time for a single execution path. In figure 1 it can be observed that measurement based approach produces bounds in the unsafe range which is less than or equal to the actual WCET. So a safety margin is usually added to obtain safe bounds. If too much margin is added, resources will be wasted and if there is too small margin, the result will be a potentially unsafe system.

1.5. Static WCET analysis

Static timing analysis is an alternative method to derive WCET estimates which does not require to perform any actual execution of the program. It relies on models and analysis of characteristics of the software and hardware involved. If the models are correct, static analysis will derive a safe WCET which will be equal or greater than actual WCET. As both software and hardware characteristics are involved in deriving WCET, both the properties of the hardware and software are considered when calculating the execution time of a program.

Static analysis is divided into three phases. The first one is the flow analysis phase, which is actually derivation of information on the possible execution paths through the program. The second one is the low level analysis, which is used to determine the timing behaviour of instructions in the program, given the architectural features of the target hardware. The final phase is the calculation phase, where information from the two other phases is used to find out costliest execution path of the program.
1.5.1. Flow analysis and flow facts

A program can be executed in several different ways, i.e. through different paths. To identify these paths is a task for flow analysis. Flow analysis can be divided into three phases [10].

1. Flow information extraction: Flow information can be obtained by manual analysis (given by manual annotations) or by automatic flow analysis.
2. Flow representation: The flow information must be represented in an uniform manner.
3. Conversion for calculation: The control flow must be converted to a format suitable for the final WCET calculation.

The flow analysis phase is made on source code, intermediate code or machine code. The purpose of the flow analysis is to derive bounds on the dynamic execution behaviour of the program. This includes information on which functions get called, loop bounds (maximum number of times loops are iterated), dependencies between conditions or branches, paths that are feasible through the program, and execution frequencies of code parts.

The flow analysis does not know the execution path which corresponds to the longest execution time and for this reason the collected information must be safe over-approximation, i.e., the flow information must contain all possible paths. Flow analysis should rely on automatic flow analysis as much as possible [10]. However, automatic flow analysis may fail to find some flow facts. Then the programmer may provide flow information manually.

Scope graphs can be used to present the flow analysis result of program [3]. A control graph is partitioned into scopes in scope graph. A scope is a loop or a function which can be repeated. For example, a loop nest will be represented by a chain of scopes, where the scopes for inner loops are below scopes for outer loops. The purpose of the use of scopes is to structure the flow analysis of the program and also to structure the generated flow constraints in such a way that the execution of repeating constructs can be analyzed and constrained.

Flow information for scopes can be expressed as flow facts [8]. Flow facts are used to check or constrain virtual execution counters for the nodes in CFG (control flow graph) in a scope. If we think about a scope hierarchy and each time a scope is entered from above, for every node N, its corresponding counter #N is initialized to zero and is incremented at each execution of node N.

Flow facts have the format scope : context : linear constraint. Here context can have two possibilities. It can be a forall context [range] which specifies that the linear constrain should hold for all iterations of the scope. It can also be a foreach context <range> which specifies that linear constraint should hold for each individual iteration of scope. The constraint should hold for all possible iterations, if range is left out. It is easy to express loop bound and infeasible path constraints as flow facts.
For example, a loop “L1” with header node H and nodes B1,…., Bn will look like this when expressed as flow fact,

\[ L1: [ ] : \#H < k \]

where the flow fact restricts the number of loop iterations to at most k-1.

The flow fact

\[ L1: <3...7>: \#B1 +. .. + \#Bn < n \]

expresses that for each of the individual loop iterations 3 to 7, all the nodes B1,…,Bn cannot be executed.

1.5.1.1. Value analysis

Value analysis is a method for determination of the value range for variables at different program points. The results of this analysis are used, e.g., for determination of infeasible paths and loop bounds [16].

1.5.1.2. Loop bounds

Flow analysis is mostly focused on loop bound analysis. This is because upper bounds on the number of loop iterations must be known in order to derive WCET estimates at all. Similarly, recursion depths also are bounded as those are not known in a program (explicitly given by the programmer). To give these bound manually is often laborious and possibly error-prone. So, automatic loop bound analysis is preferred. Paper [2] is recommended for further study.

1.5.1.3. Infeasible paths

Infeasible paths are not required to derive the WCET bound. They are only of interest when one wants to tighten the WCET estimation. In paper [2], the definition of infeasible paths is given 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”. Two types of infeasible paths can be identified:

- Infeasible paths caused by semantic dependencies
- Input-data limitation dependent infeasible paths.

1.5.2. The low level analysis phase

The purpose of the low level analysis is to calculate the effect of hardware timing on execution times. A timing model is used to derive the processor timing for the individual instruction executions. Hardware features includes various hardware enhancing features i.e. branch prediction, pipelines, memory system, presence of cache etc [10]. The timing model contains the timing properties for both the processor and system hardware that affect the instruction execution timing.

This low level analysis phase is performed on object code or binary code. The reason is that hardware features are explicitly visible in these levels of code. The low level analysis is divided into two sub phases:

- Global low level analysis
- Local low level analysis.

1.5.2.1. Global low level analysis

This analysis phase handles hardware features that reach across the entire program. Instruction caches, data caches, and branch predictors are examples of features that cause global effects for the entire program. The result from this global low level analysis is passed onto a local low level analysis and these results are called execution facts which are facts that tell whether specific instructions hits or misses the instruction cache, whether a branch is correctly predicted or not etc.
To tighten the performance gap between main memory and relatively slow access speed to main memories, caches are installed between the processors and main memory. The CPU looks for the next instruction in cache before searching in main memory. A successful search is called a cache hit which results in fast access speed. On the other hand, an unsuccessful one will result in a cache miss which will result in slow access speed. Instruction cache, data cache and unified cache are the examples of different types of cache. Depending on the type of the cache to analyze, the WCET tools use different approaches to analyze the program’s timing behavior. More details can be found in [8].

1.5.2.2. Local low level analysis

This phase handles the hardware features that do not reach across the entire program. This analysis handles effects of machine timing which depends on a single instruction and others that are located in its immediate neighborhood. Pipeline overlap between instructions and basic blocks are typical examples of local effects. The presence of pipeline influences the execution times of instructions in sequence. Besides, memory speed and instruction alignment are also to be considered as local effect on low level analysis. Access times of the present memories (e.g. on-chip ROMs/RAMs, flash memories, ordinary RAMs, etc.) directly influence the execution time of an instruction. The greater part of research in local low-level analysis has been directed at pipeline analysis [8].

Instructions are processed into several pipeline stages. The number of stages can vary from two to ten pipeline stages. If a typical pipeline has five stages (IF, ID, EX, MEM, WB), instructions go through these stages while being processed. If each stage cost one cycle to complete and no pipeline is present, an instruction has to wait for the one that precedes it to be complete, before it is processed by the CPU. Thus, there will be at most five clock cycle waste of time between each two instructions neighboring each other. We can have the basic idea about this from figure 3.

1.5.3. Calculation phase

The results from flow analysis and low level analysis are combined at this phase and the final calculation is performed for WCET estimation. There are three main techniques to calculate the WCET upper bound:

- Tree-based calculation
- Path-based calculation
- Implicit Path Enumeration Technique

![Figure 3. Example of instruction processing with and without pipeline [33].](image-url)
1.5.3.1. Tree-based calculation

When a syntax tree is used to represent the program flow, then tree based calculation is used. Program structures (i.e. loops, sequence, conditional statements etc) are presented as nodes in the tree. Basic blocks are represented as leaves in the tree. There are some given rules for calculation by traversing the tree from bottom up. Each node in the tree is interpreted to an equation which expresses its timing based on its child nodes.

By looking at figure 3 (from [10]) we can get a basic idea. The main disadvantage of this method is that it cannot handle unstructured code.

1.5.3.2. Path-based calculation

When flow information is represented as a flow graph, this method is used to derive the timing estimation. From the figure 3 we can get the basic idea. Each node or basic block in a path has an execution time and these execution times are added together to get the execution times of the path. Bounds on loops on a path are defined as maximum number of possible iterations. Resultant bounds are multiplied by the sum of execution times of each node or basic block contained in the loops.

![Diagram](image.png)

Figure 4: Flow representations and calculation methods [10].

1.5.3.3. Implicit Path Enumeration Technique (IPET)

In this method, both the program flow and basic-block execution time bounds are combined together and the combination is represented as sets of arithmetic constraints. These arithmetic constraints are basically logical and algebraic restrictions which are extracted from the program’s structure and also from the possible program flow. A time variable (t_entry) is assigned to a node in the CFG which is the representation of the execution time of the corresponding node. The number of times the node is being visited is represented as a count variable (x_entry). This count variable is also assigned to the number of times edges in the CFG are being visited. The WCET is derived by the maximization of the sum:
The IPET calculation finds an upper timing bound and a worst-case count for each execution count variable. From figure 3c we can see the basic calculation where constraints and formulas generated by an IPET-based bound calculation method for the task depicted in figure 3a. This method has the ability to handle different types of flow information.

1.5.4. Hybrid WCET analysis

Both the static and dynamic methods are combined in the hybrid WCET estimation method. In this method, static analysis of the source code is performed, and a model of the program is created. The program is prepared to run on real hardware by partitioning it and adding measurement points. Then those partitions are executed on real hardware and resultant timing measurements are brought back to the static analyzer to be used for derivation of WCET estimation of the program.

The main advantage is that a timing model for the target hardware is not needed. The timing model is replaced by a structured measurement of the program partition using real hardware. The main disadvantage of this method is that there is no guarantee of getting safe WCET estimates. The main reason behind this disadvantage is that the timing of the partitions is obtained by using measurements, meaning that the worst case behavior can be missed.
2. WCET analysis tools

In recent years, several tools for WCET analysis have been developed, both as academic research prototypes and also as commercial products. Some tools are based on hybrid WCET analysis (measurement based estimation), and some tools based on static analysis. Some important features of WCET tools can be described according to four questions:

1. What is the functionality of the tool?
2. What methods are employed in the tool?
3. What are the limitations of the tool?
4. What hardware platform does the tool support?

Typically, WCET analysis tools are evaluated by the metric of accuracy of the WCET estimate. But other properties such as performance (i.e. scalability of the approach) and general applicability (i.e. ability to handle all code construct found in real-time system) are also considered for evaluation of WCET analysis tools.

2.1. Static analysis tools

Some WCET tools use static analysis methods. They exist both as commercial products and research prototypes. For example, aiT, Bound-T are fully commercial products. On the other hand there are research prototypes that exist in academia such as OTAWA, Heptane, SWEET, Chronos etc. Some of them are described shortly below with the help of those four questions that have already been discussed.

2.1.1. Commercial static analysis tools

2.1.1.1. aiT

This tool is produced by the German company AbsInt [5]. The purpose of this tool is to obtain upper bounds for the execution times of code snippets (like subroutines) in program executables, or tasks in a real-time application. Instead of working on source code, aiT works on executables, because source code does not contain information that are important for cache analysis and memory areas with different timing behavior [11].

User input is needed for this tool to derive WCET for given subroutines. This user input (or manual annotation) can be upper bounds for loops or flow facts which are provided by the user. Some of the annotations are not necessary for WCET derivation, but may improve the precision of WCET results. aiT also supports input or annotations specifying the value of registers and variables. More details on user annotations and task analysis can be found on Daniel Sehlberg’s thesis [3]. The paper written by C. Ferdinand and R. Heckmann [4] can be read to know more about the analysis techniques used in aiT.

The aiT uses several phases for determining upper bounds, and the phases use different methods. Value analysis and cache/pipeline analysis are realized by abstract interpretation. Path analysis is implemented by ILP (Integer Linear Programming). Reconstruction of the control flow is performed by bottom-up analysis. AbsInt’s graph browser aiSee is used to visualize the call graph and the control-flow graph [11].

Automatic analysis is used by aiT to determine the upper bounds of iterations of loops and the targets of indirect calls/branches. However, this analysis method does not work in all cases. Manual annotations are needed in case of failure of automatic the analysis. aiT relies on standard calling convention which might fails in some cases and additional annotations are needed for resolving this problem [11].
Versions of aiT have support for some of Motorola PowerPC, Motorola ColdFire, Renesas and Infineon TriCore 1.3 [11].

2.1.2. Research prototype for static analysis

2.1.2.1. SWEET

SWEET is a research prototype from Mälardalen University with main research focus on the flow analysis. SWEET performs flow analysis on intermediate code in order to be language independent. The new version of SWEET (ALF-SWEET) has been used in this thesis work. Details can be found in section 3.

2.2. Hybrid analysis tool

The only hybrid analysis tool that exists as a commercial product is RapiTime. However; there are research prototypes that exist in academia like tools from TU Vienna, FORTAS etc. The commercial product RapiTime is described shortly below.

2.2.1.1. RapiTime

RapiTime is developed by Rapita Systems Ltd., York, UK [7]. This tool aims at medium to large real-time embedded systems on advanced processors. Its target application areas are automotive electronics, avionics and telecommunications. RapiTime is a measurement-based tool. This tool not only computes WCET estimate of a program as a single value, but also the whole probability distribution of the execution time of the longest path in the program. The input to RapiTime can either be a set of source files (C or Ada) or an executable. The user has to provide test data from which measurements will be taken. The output is a HTML report with WCET estimations and actual measured execution times, split for each function and sub-function. Timing information of the running system is captured by either a software instrumentation library or even traces from CPU simulators. The user can guide the instrumentation and analysis process by adding manual annotations [11].

The RapiTime tool is structured based and works on a tree representation of the program. This structure is derived from either the source code or direct analysis of the executables. The timing of individual blocks is derived from extensive measurements extracted from the real system. The WCET estimates are computed using algebra of probability distributions. Timing analysis can be performed on different contexts which allow individual analysis. The level of details and how many contexts are analyzed is controlled by annotations [11].

The RapiTime does not rely on a model of the processor and it can model any processing unit (i.e. out-of-order execution, multiple execution units etc). But the limitation is put on the need to extract execution traces, which require some code instrumentation and a mechanism to extract these traces from the target system. RapiTime cannot analyze programs with recursion and with non-statically analyzable function pointers [11].

The supported hardware platforms are Motorola processors (including MPC555, HCS12, etc), ARM, MIPS, and NecV850 [11].
3. Description of SWEET

SWEET uses one or more ALF file(s), possibly together with an input annotation file, as input. The ALF file is created from the program to be analysed. We start with a short description of ALF before we continue with the basics of SWEET, followed by a description of input annotations.

3.1. The ALF language

ALF (ARTIST2 Language for WCET Flow Analysis) is a language which is used for flow analysis for WCET calculation. It is an intermediate language which is designed for flow analysis rather than code generation. It is furthermore designed with the ability of representing code on source-, intermediate- and binary level (both linked and unlinked) through relatively direct translations. In this way information in the original code is maintained for precise flow analysis. ALF is basically a sequential imperative language which has a full textual representation. It can thus be seen as an ordinary programming language. However, it is intended to be generated by tools rather than written by hand [12].

3.1.1. Syntax

The syntax of ALF is similar to the LISP programming language, which makes it easy to parse and read. The ALF syntax uses prefix notation which is similar to LISP and its use of parentheses (,{}) is similar to the Erlang programming language. The following example denotes the unsigned 32-bit constant 0.

\{dec\_unsigned\ 32\ 0\}

3.1.2. Memory model

The memory model in ALF distinguishes between program and data addresses. It is a memory model which can relocate unlinked code. Both the program and data addresses have a symbolic base address and a numerical offset. If two addresses have same base address and same numerical offset, then those addresses are considered as equal. But the address spaces for code and data are disjoint [11].

3.1.3. Program model

The program model of ALF is similar to the C programming language. An ALF program is a sequence of declarations and its executable code is divided into a number of function declaration. Each function contains a linear sequence of instructions and normally statements are sequential. ALF has ‘jump’ instructions, which are used to jump to a statement with a certain label. This mechanism is used for representing program control in low-level code. For representing high-level code, ALF has a useful structured function calls. When an ALF program is run, a function named “main” will be executed. ALF programs without a main function cannot be run, but still be analyzed [12].

3.1.4. Data model

The data memory of ALF is divided into frames. Each pointer has a symbolic base pointer (frameref) and a size. The size is specified in the least addressable unit (LAU) of the ALF program, which is generally chosen as a byte (8 bits). Frames can also be given an unbounded size which is useful for modeling dynamic data areas such as heaps and stacks. Data addresses are formed from a symbolic part and an offset, like labels. The symbolic part of a data address is a frameref and the offset is a natural number in 16 LAU. Frames can be either statically or dynamically allocated.

3.1.5. Values

Values can be:
- Numerical values: signed/unsigned integers, floats, etc
- Framerefs
• Data addresses (f,o), where f is a frameref and o is an offset (natural number)
• Code addresses (labels)(f,n), where f is an identifier and n is a natural number

There is a special value ‘undefined’. This provides a fallback in situations where an ALF producing tool, e.g., a binary-to-ALF translator, cannot translate a piece of the binary code into sensible ALF.

3.1.6. Type system

ALF has a simple, static, monomorphic type system with subtyping. All types but one is parameterized with respect to a size, which can be natural number or unlimited size. There are two classes of types (reflected in the system): bitstring types, whose binary representation is fully known and symbolic types whose data have symbolic contents. The type system has the following basic types: size, anytype(n), bitstring(n), symbolic(n), int(n), unsigned(n), signed(n), float(m,n), fref(n), address(n), lref(n), label(n).

3.1.7. Operators

Operators in ALF are of five types.
• Operators on data of limited size: neg, add, cadd, sub, csub, u_mul, s_mul, u_div, s_div, u_mod, s_mod, f_neg, f_add, f_sub, f_mul, f_div, f_to_f, f_to_u, f_to_s, u_to_f, s_to_f.
• Operators on data of unbounded size: exp2
• Operators on bit strings: l_shift, r_shift, r_shift_a, s_ext, not, and, or, xor, select, conc, repeat.
• A conditional: eq, ne, u_lt, u_ge, u_gt, u_le, s_lt, s_ge, s_gt, s_le, f_eq, f_ne, f_lt, f_ge, f_gt, f_le, if.
• A conversion function – b2n

An example is
{add W VEXPR1 VEXPR2 CEXPR }
where W is an integer constant specifying the bit width of the arguments, and the result, VEXPR1, VEXPR2 are expressions for the arithmetic operands in, and CEXPR is an expression specifying carry in [13].

3.1.8. Statements

ALF has the following statements with their informal semantics given below [13].
• { null } – Do nothing
• { store ADDRESS EXPR+ with EXPR+ } - evaluate the address expressions in ADDRESSEXPR+ into a1, . . .,an, in left-to-right order, then evaluate the expressions in EXPR+ into e1, . . .,en (same order), and concurrently store each ei at address ai
• { switch NUM EXPR {target INT NUM VAL0 LABEL EXPR0 } . . .
• { target INT NUM VALn–1 LABEL EXPRn–1 } } - NUM EXPR is evaluated, and then compared to each constant INT NUM VALi in order. If the computed value is equal to the j:th constant INT NUM VALj, then execution continues at the label given by evaluating the label expression LABEL EXPRj
• { jump LABEL EXPRleaving n } - Evaluate LABEL EXPR and jump unconditionally to the resulting address. n is a nonnegative integer constant: it specifies how many scope nesting levels the jump may exit from the current scope
• { freeFREF EXPR } - Evaluate FREF EXPR, and deallocate the dynamically (with dyn alloc) allocated memory pointed to by the result.
• { call LABEL EXPR EXPR LIST result ADDR EXPR LIST } - Evaluate LABEL EXPR (the function to be called), the expressions in EXPR LIST (the arguments), and ADDR EXPR LIST (the addresses where to store the results), in that given left-to-right order, write each
evaluated argument in EXPR LIST to the corresponding formal argument for the procedure which LABEL EXPR evaluated into, and then call this procedure.

- {return EXPR LIST} - Values and control are returned using a return statement, which takes a list of expressions to be evaluated, in left-to-right order, when the statement is reached by the execution.

3.1.9. Semantics
ALF is an imperative language with standard semantics based on state transitions. The state is comprised of the contents in data memory, a program counter (PC) holding the label of the current statement that is going to be executed, and some representation of the stacked environments for function calls [14].

3.1.10. ALF grammar
The details of ALF grammar are beyond the scope of this thesis and thus readers are referred to [14].

3.1.11. Example of an ALF program
The following C code: if(x > y) z = 42;
can be translated into the ALF code below:
```
{ switch { s_le 32 { load 32 { addr 32 { fref 32 x } { dec_unsigned 32 0 } } } }
  { load 32 { addr 32 { fref 32 y } { dec_unsigned 32 0 } } } }
  { target { dec_unsigned 1 1 } }
  { label 32 { lref 32 exit } { dec_unsigned 32 0 } } } }
{ store { addr 32 { fref 32 z } { dec_unsigned 32 0 } } with { dec_signed 32 42 } }
{ label 32 { lref 32 exit } { dec_unsigned 32 0 } }
```
The if statement is translated into a switch statement jumping to the exit label if the (negated) test becomes true (returns one). The test uses the s le operator (signed less-than or equal), taking 32 bit arguments and returning a single bit (unsigned, size one). Each variable is represented by a frame of size 32 bits [12].

The above piece of code is not a complete ALF program. Every ALF program starts with a ALF start symbol which is followed by a macro definition with its formal definition, if there is any macro definition. Every ALF program should have a main function which is run when ALF programs are run. The body of the function is called scope. The argument declarations are done inside the function body and statements are specified inside the scope.

3.1.12. Flow analysis using ALF
There are wide set of sources, like linked binaries, source code, and compiler intermediate formats that can be selected as input to a translator for conversion to ALF format. The flow analysis in SWEET is then performed on ALF code. It will be simpler to implement different flow analyses, if there is one standard format like ALF. ALF also offers an advantage to compare between different analysis methods. Different analyses can be chosen to be the most effective ones for different kinds of translated input formats and thus facilitates the analysis of heterogeneous programs. This is useful in situations where some parts of programs are available as source code and some other parts are available as binaries. In the same program it is possible to have different parts with different characteristics which require different flow analysis methods. Flow analysis methods can be used as “plug-ins” and it is even possible to share results with other methods. It is possible to use several flow
analysis methods in parallel and best result can be selected. So, it is clear that ALF will be able to offer high degree of flexibility.

ALF itself does not carry any flow data. Therefore, the analysis results must be mapped back as flow constraints on the code from which the ALF code was generated. The ALF generator can use conventions for generating level names in order to facilitate this mapping back to program points in the original code. The Program Flow Fact (PFF) format was developed to accompany ALF. PFF is used to express the dynamic execution properties of a program, in terms of constraints on the number of times different program entities can be executed in different program contexts [15].

3.2. Translators to ALF

SWEET cannot perform its analysis directly on C source or executable files. Therefore, SWEET uses the ALF format to perform its analysis. ALF is the only input format for program to be analyzed by SWEET. Figure 4 shows the uses of ALF with conjunction of SWEET with different steps and representations. Details can be found in [15].

![Figure 5 The use of ALF with SWEET [15].](image)

A number of translators to ALF are being developed and all three types of sources (i.e. source code, intermediate code and binary code) will be handled by these translators [15].

3.2.1. The melmac tool

The C to ALF translator which is has been used in this thesis work is called “melmac” (the name of the planet where ALF came from in the TV series) and can be found on the melmac website [17]. To avoid complexity and save time a shell script “c_to_alf_using_christers_machine” has been used in this thesis which converts source files to ALF format when run in a shell environment (i.e. Cygwin). This script takes C source file as input and generates, using melmac, the corresponding ALF file.

The melmac tool is open source software. Its main use is to generate ALF programs from C programs. Thus, it provides a C frontend. It generates ALF programs from an intermediate representation called Termite. SATIrE [34] includes a program called "c2term" which generates Termite from C. The melmac distribution contains a few small C files, and the corresponding Termite files, in the example_terms directory. The tool is started from the command line. It takes exactly one command telling it what to do, and possibly a number of options telling it how to do it. The main commands all convert an input file into an output file.

Several parameters related to ALF code generation (mostly sizes of data types) are controlled by a configuration file. When melmac is invoked, it will look for this file in a number of default places. It is possible to tell it which file to use by using the --config command line flag.

3.3. SWEET flow analysis

SWEET uses a multiphase approach for flow analysis. A program slicing is used to restrict the flow analysis to only those parts of the program that may affect the program flow. The main method within SWEET for flow analysis is abstract execution [2].
3.3.1. Abstract execution

The idea behind abstract execution (AE) is to extract properties of the run-time behavior of a program by making an “interpretation” of the program using abstractions of values instead of concrete values. This method is based on classical abstract interpretation [18]. The main difference between classical abstract interpretation and abstract execution is that in abstract execution all possible executions at a certain program points are analyzed separately [2]. Abstract execution uses abstract operators and abstract values for program variables when executing the program in an abstract domain [2].

In SWEET, the abstract domain is the domain of intervals. This means that each variable will hold an abstract value (an interval) instead of a single value. Each operation will calculate a new interval from the intervals in the operation. The new interval will then represent the concrete values the variable could hold at that point. This ensures that abstract values will always hold the set of possible values during the execution of a program. This also guarantees that no execution paths will be missed by the analysis. However, the analysis may produce flow constraints which are not tight because of possible overestimation of the value range. As a consequence, the result of WCET calculation will be less tight because the inclusion of infeasible paths [18].

In some condition nodes, the abstract execution will consider both true and false branches as possible nodes in the execution path. Two abstract states must then be created in order to hold the result of both possible paths. This forces the abstract execution to handle many abstract states in parallel. The states have a tendency to grow exponentially with the length of the execution paths. Thus SWEET uses a merging mechanism to save space and analysis time. Merging of states is basically done at program points where different paths join. There are several kinds of merging strategies where the user has the options for choosing the actual merge points. Thus it is possible for user to choose between fast analysis and precise analysis. The SWEET algorithm [2] merges the abstract states which belong to the same scope iteration and program point. Figure 5 shows the flow analysis with abstract execution. More details about analysis using examples could be found in [2] [10] [18].

3.3.2. Generated graphs

SWEET generates a set of graphs to represent the flow analysis results. Depending on selected options, SWEET produces CG (call graph), SGH (Scope graph hierarchy), RSG (Reduced scope
3.3.3. Input annotations

The main goal of input annotations is to define the input values (abstract values) at a certain point of the program. Annotations are stored in a text file (.ann), where each line in the file corresponds to one annotation. A typical example of a SWEET run using input annotations would be like this:

```
sweet -i=filename.alf annot=filename.ann func=function name -ae ffg=uhss lang=ff
```

where ae = abstract execution and lang = ff, which means output language will be the default SWEET type of flow facts. The following shows an example of an .ann file.

```
STMT_ENTRY main BB0 0 ASSIGN x INT -1 5;  // Annot 1
STMT_EXIT foo BB72 5 ASSIGN x INT 3 4 || y INT 2;  // Annot 2
FUNC_ENTRY bar ASSIGN s 0 32 INT 1 67 || s 32 16 INT -12 14 ||
        s 48 32 ADDR x;  // Annot 3
PROG_ENTRY ASSIGN g ADDR d 0 d 32 d 64 d 96 d 128;  // Annot 4
```
Explanations:

- Annot 1 adds an annotation before the statement labeled <BB0,0> in function main which assigns the integer interval -1..5 to variable x.
- Annot 2 adds an annotation after the statement labeled <BB72,5> in function foo which assigns the integer interval 3..4 to variable x and value 2 to variable y.
- Annot 3 adds an annotation after entry of function bar which updates a larger aggregate data structures with several values. It assigns the integer interval 1..67 to the first 32 bits of s, interval -1..14 to the following 16 bits of s, and a pointer value holding the addresses of variable x to the following 32 bits.
- Annot 4 adds an annotation at the global program scope which assigns a set of pointer values to the global variable g, all pointing at different parts of a larger global aggregate data structure.

3.3.4. Flow facts generation and the flow fact language

The execution behavior (static and dynamic) of program or program flow representation in SWEET is defined with a formalism that is powerful enough to describe the complex flows found in embedded real-time software. The flow representation is flexible enough to capture the output from a variety of flow analysis methods and manual annotations [10]. This flow representation consists of a scope graph which is a graph representation capturing the dynamic execution behavior of program, and a flow fact language which is an annotation language providing additional constraints on the program flow [1]. Each scope in a scope graph corresponds to a certain repeating or differentiating execution context in the program (i.e., a loop or a function call), and describes the execution of the object code of the program within the context [10]. The scope graph is acyclic and there is a containment relation between scopes which means that a loop nest will be represented by a chain of scopes for inner loops are below scopes for outer loops [2]. This scope graph also describes how scopes are invoking other scopes. If a function or loop is called many times in the program, each call would be represented with its own scope. The context sensitivity in the scope graph makes the analysis more accurate but costlier since each context will be analyzed separately [2].

Each scope has a set of associated flow information facts (flow facts). Each flow information fact consists of three parts: the name of the scope where the fact is defined, a context specifier, and a constraint expression. Figure 7 shows the conversion of the program into scope graph with attached flow facts.

A flow fact has the following syntax [22]:

\[ \text{scope: context: linear constraint} \]
3.3.4.1. Scope
This field is the definition of 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.

3.3.4.2. 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].

3.3.4.3. Linear constraint
The flow fact in SWEET represents either loop bounds or infeasible paths. If LOOP1 is a loop header, and the loop iterates maximum four times, then SWEET expresses this as: LOOP1 < 5. If LOOP1 is an infeasible node, the expression would be: LOOP1 = 0. More detailed information about scope graphs and flow facts can be found in [1] and [2].

The command sweet -h topic=ffg will show details about the flow fact language and the command sweet -h topic=annot will show details about annotations to specify constraints on possible input values of the program.

3.4. Early timing analysis using SWEET
Timing properties are normally verified late in development process, provided that hardware is available and source codes are compiled. Costly system redesign is needed when timing properties are not met. Timing estimates are therefore very useful also during the early stage of system development. Early timing analysis is done on source codes rather than binary or object codes. An automatic method has been presented in [26] to identify a source level timing model for a given combination of hardware configuration, and compiler. The models are identified from measured execution times for a set of synthetic “training programs” compiled for intended platform with execution of each training program for a variety of inputs. Training program suites are designed for both simple architectures and advanced architectures. A number of “virtual instructions” are decided for the source level language to be analyzed for WCET estimation. An abstract machine is defined by those virtual instructions and the abstract machine can execute the source code in reasonably direct manner. The execution time for each virtual instruction is recorded by executing the training programs on an emulator. The timing model is automatically identified with an execution time for each virtual instruction. SWEET is then used to do an approximate static WCET analysis on source level using the resulting timing models. The resulting WCET deviated 0 - 20% from the real WCET estimates [26].

3.5. Low level analysis (low-SWEET)
The timing analysis and final WCET estimation is currently done with low-SWEET (the low level analysis part of SWEET). Currently, this part of SWEET only supports ARM9 and NEC V850E. The tool low-SWEET works in two phases: global low level analysis and local low level analysis. Instruction cache, data cache and branch predictions are examples of global low level analysis. Global analysis does not generate any concrete execution times and thus the analysis results are passed on second phase or local analysis phase as “execution facts”. The icache fact in figure 8 is an example of execution facts from global analysis result.

The processor behavior analysis in SWEET is decoupled from flow analysis and this analysis is based on a two phase approach. At first phase determination about different instructions access to memory areas is performed. An instruction cache analysis is performed, if there is any instruction cache in target hardware. The analysis result is a set of execution facts which are used in the pipeline analysis in second phase or local low level analysis. These facts specify the memory area’s reference to an
instruction and also whether that instruction has hit or missed the cache. Moreover, assumptions on branch prediction outcomes (whether a branch is correctly predicted or not) can also be specified by execution facts.

The local low level analysis is mainly related to handling machine timing effects that depends on a single instruction and its immediate neighbors. Pipeline analysis is an example of this phase. This level operates on timing graph, which is a graph for whole program. The nodes and edges in the timing graph correspond to nodes and edges in the scope tree (without the scope structure which is not relevant with this level). Each node in the timing graph has an associated execution fact which is generated during global low level analysis (figure 8).

The pipeline analysis generates times for the nodes and edges in the timing graph. Times for nodes correspond to the execution times of basic blocks (with associated execution facts) in isolation (e.g. \( t_Q \)) and times for edges, (e.g. \( t_{QR} \)), to the pipeline effect when the two successive nodes are executed in sequence (usually an overlap) [10]. The individual nodes are run in the simulator at first and then the sequence. Execution times are compared and finally timing effects for sequence of nodes are calculated. Figure 9 illustrates the process of timing graph. Trace-driven cycle-accurate CPU model is used for pipeline analysis by simulating object code sequences. The simulator takes instructions together with execution facts. The execution facts can correctly be accounted for in each instruction. To enforce the worst-case timing behavior of the instructions, execution facts are used. Worst-case timing is assumed for data dependent instructions. To allow standard CPU simulators to be used as CPU models, pipeline analysis has been explicitly designed. But there is a requirement for this and that is the simulator should be clock-cycle accurate. The simulator can be forced to perform its simulation according to given instruction sequence and the corresponding execution facts, and does not suffer from timing anomalies [10,11].

Figure 8. Timing Effect Calculation [10]  
Figure 9. Example of Timing Graph [10]

3.6. SWEET WCET calculation

The calculation phase supports three types of calculation techniques: a fast path-based technique [23] [1] [24], a global IPET technique [1], and a hybrid clustered technique [25][1]. The clustered calculation can perform both local IPET and/or local path-based calculations [11]. IPET is often chosen as calculation technique, as it best satisfies the modularity of SWEET and also allows for expressions of the most complex flows (including unstructured flow) [11].
In the IPET method, restrictions on program flow are given as algebraic and/or logical constraints. Each basic block and/or basic program flow edge in the program is given a time variable ($t_{\text{entity}}$) which denotes the execution time of the node or edge and also a count variable ($x_{\text{entity}}$) which denotes the number of times node or edge is executed. For example, in figure 10 node C has timing $t_c=7$ and the corresponding count variable $x_c$. The count variable is considered global for the program part for which WCET is calculated and its value represents the total number of executions of the node for the complete execution of the program. For example $x_c$ holds the total number of times that node C is executed over the complete program execution.

An example of how IPET works is shown in figure 10b. The execution count variables of the start and exit nodes are both set to one which is constraining the program to only start and exit once.

To model the possible program flows, structural constraints are used. To do that, the number of times a node can be executed are set to be equal to the sum of execution counts of its incoming and outgoing edges. For example, the following constraints are generated for node B,

$$x_B = x_{AB} = x_{BC} + x_{BD}$$

The estimated WCET is found by maximizing the sum:

$$\sum_{i \in \text{entities}} x_i * t_i$$

The maximizing problem can be solved using either ILP (integer linear programming) or constraint programming. Constraint programming can handle more complex constraints than ILP, and ILP can only handle linear constraints but is usually faster. An advantage with IPET is that complex flow information can be expressed using constraints, but on the other hand it can also result in longer computation times and the result is implicit. For example, $X_c=80$ and $X_d=6$ means that C executes 80 times and D executes 6 times, but it is not possible to know in which order they execute.

In SWET, extended IPET is used as calculation technique. This is an extension to IPET allowing full expressive power of SWET’s flow fact language. The first input to the extended IPET calculation method is a scope graph with flow facts that together represent possible program flows. Scopes are allowed to contain one or more header nodes, and to have several in-nodes and out-edges, which is sufficient to handle most types of unstructured code. More details on the extended IPET method can be found in [1].
4. The Bombardier Train Control System

The Bombardier Train control system is an embedded real time system which is running in one of their latest trains. Bombardier offered parts of their code that we could examine in more detail and perhaps use as WCET benchmark program and to test our static code analyzer on. We got the code that should be investigated with NDA-protection.

4.1. Brief description of the system

At first we will describe the software briefly and also a little bit about the functions in a train. Later on this chapter we will discuss the tasks in the system.

This software has been implemented for the CPU PPC603/MPC 5200 which belongs to Power PC family (PPC). The embedded operation system VXWORKS is used in the system.

Most of the codes is drawn with the help of in-house tools (developed in Canada) called MTPA. The drawings can be compared to paired circuits in a wiring diagram. Then C codes are generated from the drawings with the help of the MTPA tool. The generated code often behaves like the following:

(a) Reading a lot of inputs (signals)
(b) Execute any conditions map loaded input (for example, makes X for Y and Z are true)
(c) Produces a lot of outputs (signals).

The only conditional executions of the generated codes are if-statements (no loops, etc.). Some codes are written manually by hand with the help of C codes and those codes sometimes contain one loop. All generated / handwritten code merges together into one big C file ccode.c running on all nodes in the system which actually implies that each node will have its own copy. Other codes can be written in another way. Third-party code + OS code have often no access to the source format. Real time task concepts have been implemented in the system. Several generated or handwritten functions can be assembled into a task. They often use static task schedule with offset. The tasks have priorities. The OS allows 16 tasks and several tasks with the same priority can merge into a single task. Task or watch dogs also exist to check that deadlines are not missed. There is redundancy in the system.

The train system consists of a number of wagons. There are actually several nodes and one node per wagon is used. The nodes are connected via networks. The same code runs on any VCU. A node now has both an old 68K processor which is newer and much faster PPC. New functionality is often added to PPC while the old code preferably kept left on the old but proven 68K.

The Zealcore system debugger is used for this software. System performance is checked rather than checking the code.

4.2. Task generation and code properties

The tool MTPE actually generate the whole project named as LTAAF and this LTAAF then generate two devices. One is CCUO and another device is CCUD. Tasks are then generated for both of the devices. For every device, task and task body is also generated. Task instances are processed into a task body.

Some of the tasks have only a few lines of codes and some of them have more than 7000 lines of codes. The tasks are created without any loops or recursion. Only if-else conditional statements are used and most of the tasks have “macro” definitions. OS functions are called in almost all tasks. All of those tasks have “structs”.

29
4.3. Tasks with loops

We have already seen that most tasks lacks any loop. But there is a task that has loops. This function “sci_md” is "Serial Communication Interface for Message Data”. This module implements the functionality of SCI-MD, the message data part of the Serial Communication Interface (SCI). The other parts are SCI-PD (the process data part) and SCP layer. At the INIT level, the TRACS element SCIMDF calls the function sciInitMD to initialize this module. The main part of SCI-MD is a replier task, which implements a number of MVB replier function instances. The task handles the message data communication between MVB caller functions and the SCP layer. The main function (start address) of the task is sciReplierMain. The replier task waits for events at a queue and when an event occurs, the task performs the appropriate actions. Three basic events can occur:

- A call request message is received from an MVB caller function.
- A reply message (from a slave device) or an error code is received from the SCP layer.
- A reply message or an error code has been sent (completely) to the MVB caller function.

The task also handles time-out supervision for the instances, that has active message data conversations.

The module Scp.c provides communication between TRACS applications and third party equipment (slaves) connected to COMC serial ports. Communication between the TRACS application and the SCP layer is either asynchronous or synchronous. In asynchronous mode the TRACS application makes a request for service and continues processing. It receives notification from the SCP layer after the requested service has been performed. In synchronous mode the SCP layer is invoked as a function and executes in the calling task's time slice. The mode is channel independent and is defined at compile time. Communication between the SCP layer and the slave is synchronous; the SCPlayer transmits a message to a slave and waits for a reply. Asynchronous communication between the SCP layer and slave devices is possible only when a channel is opened in slave mode. This module implements the functionality of SCI-PD, the process data part of the Serial Communication Interface (SCI). The other parts are SCI-MD (the message data part) and SCP layer.

4.4. Discussion

This code from Bombardier was collected to possibly offer new benchmarks in the Mälardalen benchmark program suite. As the real-time OS was missing with the provided software, implementations of stubs were needed to completely compile the entire software. Compilation of the entire software was not so difficult. But the estimated time for analyzing the code was very long, due to its size. To make best use of time, a thorough manual code observation was performed to assess whether there is new code structure in this software which could be added to benchmark program suite. This actually took less time than analyzing the software with tool. One task was also analyzed with SWEET to get the control flow diagram of the task. As no new code properties were found in the software to add to the benchmark program suite, a detailed report of this analysis is excluded from this thesis report.
5. WCET Benchmarks

A number of WCET analysis tools have emerged both as commercial tools and academic research prototypes. By comparing obtained WCET estimates (assuming that actual worst case execution time is available), an idea of the precision of a particular WCET tool can be found. A common set of benchmarks is required to compare the WCET tools, and the associated methods and algorithms. Though the evaluation metric is the accuracy of the WCET estimate, other properties like performance (scalability of the approach) and general applicability (ability to handle all code constructs found in real-time systems) has high importance [27].

5.1. Mälardalen WCET Benchmarks

The Mälardalen WCET benchmarks have been assembled with the goal that the set of benchmarks will be easily available, thoroughly tested, and well documented. This set of benchmarks, written in C, was collected in 2005 from several researchers within the WCET field. A set of different program constructs are included in the benchmarks to support testing and evaluation of WCET tools [27]. The Mälardalen WCET benchmarks are available on a web page [19]. Benchmarks can be categorized as Well-structured code, Unstructured code, Array and matrix calculations, Nested loops, Automatically generated code, Recursive code, Nested if-statements etc. Some of the benchmarks are shown in the following table.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>janne_complex</td>
<td>Nested loop program.</td>
<td>The inner loops number of iterations depends on the outer loops current iteration number.</td>
</tr>
<tr>
<td>jfdctint</td>
<td>Discrete-cosine transformation on 8x8 pixel block.</td>
<td>Long calculation sequences (i.e., long basic blocks), single-nested loops.</td>
</tr>
<tr>
<td>lcdnum</td>
<td>Read ten values, output half to LCD.</td>
<td>Loop with iteration-dependent flow.</td>
</tr>
<tr>
<td>lms</td>
<td>LMS adaptive signal enhancement. The input signal is a sine wave with added white noise.</td>
<td>A lot of floating point calculations.</td>
</tr>
<tr>
<td>ludcmp</td>
<td>LU decomposition algorithm.</td>
<td>A lot of calculations based on floating point arrays with the size of 50 elements.</td>
</tr>
<tr>
<td>matmult</td>
<td>Matrix multiplication of two 20x20 matrices.</td>
<td>Multiple calls to the same function, nested function calls, triple-nested loops.</td>
</tr>
<tr>
<td>sinver</td>
<td>Inversion of floating point matrix.</td>
<td>Floating value calculations in 3x3 matrix. Nested loops (3 levels).</td>
</tr>
<tr>
<td>ndes</td>
<td>Complex embedded code. A lot of bit manipulation, shifts, array and matrix calculations.</td>
<td>A lot of bit manipulation, shifts, array and matrix calculations.</td>
</tr>
<tr>
<td>ns</td>
<td>Search in a multi-dimensional array.</td>
<td>Return from the middle of a loop nest, deep loop nesting (4 levels).</td>
</tr>
<tr>
<td>nsichneu</td>
<td>Simulate an extended Petri net.</td>
<td>Automatically generated code with more than 250 if-statements.</td>
</tr>
<tr>
<td>qsort-exam</td>
<td>Non-recursive version of quick sort algorithm.</td>
<td>The program sorts 20 floating point numbers in an array. Loop nesting of 3 levels.</td>
</tr>
</tbody>
</table>

Both the properties (contains loops, contains recursion, uses include files etc.) and meta-data (inputs for some of the benchmarks, number of loop iterations etc.) are included in the benchmark web page. Multiple input values has have been defined for some of the benchmark programs for being able to test and evaluate WCET tools that can handle programs with input dependency. This is because most realistic programs are run with different inputs at different invocations. A program’s WCET is usually highly dependent on inputs, if the inputs affect the control flow. For an embedded program (or task) written in C or a similar language, the input variables may be values read from the environment using primitives such as ports or memory mapped I/O, parameters to main or the particular function that invokes the task, and data used for keeping the state of tasks between invocations or used for task communication, such as external variables, global variables or message queues. These inputs are provided as intervals, i.e., limits to the inputs. The inputs are stored as “input annotations” (.ann files) [27].
5.1.1. Examples of analysis using SWEET

To show the result of an analysis of a benchmark program with SWEET, the program “fibcall” has been selected. If we run SWEET for this program, SWEET can produce four types of graphs (call graph, scope hierarchy graph, full scope graph and reduced scope graph) which are used for flow analysis. The command

```
sweet -i= fibcall.alf -d g=cg,sgh,fsg,rsg
```

in a shell command prompt will generate four graphs as DOT [20] files. By using the Graphviz [21] program those graphs can be viewed.

In figure 11, we can see that call graph is showing the call between two functions main_1 and fib_0. The function’s names are actually the ALF names, as SWEET analyzed intermediate code or ALF code of the program.

Figure 12 shows the reduced scope graph of fibcall. The scope graph is context sensitive graph showing call to functions and entries to loops. The root scope main_1 is typically the main function, or the top function in a sub graph. The scope can either be a function or loop. The scope “main_1_fib_01_L1” is a loop and the scope “main_1_fib_01” is a function. The arrow from one scope to another below represents a call in the case of a function scope, or a loop invocation in the case of a loop scope. For example in the scope “main_1_fib_01 L1”, the back edge arrow from the ALF node _4 to _3 is a loop invocation. The scope hierarchy graph is context sensitive, which means that each call site to a function creates a unique scope of the called function. The graph gives a possibility to find all, possibly looping, scopes (functions and loops) in the program [27].
5.2. PapaBench

PapaBench [29] is a WCET benchmark derived from the Paparazzi [30] UAV controller. This controller has been developed in the ENAC School in Toulouse and targets a low-cost UAV, that is, model airplane embedding a microprocessor. This controller has been successful to drive a lot of different models in complex autonomous missions and has won several prices in this domain.

Basically, the UAV is made of several actuators (motor, flaps, etc) and a very light set of sensors including a GPS (connected by a serial port) and an infrared sensor to control slope. The system may be controlled from ground using a classical wireless link or may fly in an autonomous mode performing a pre-programmed mission. In this case, the wireless descending link is only used to transfer flight log or video if the payload is composed of a little camera.

In its original configuration, the computing hardware was composed of two ATMEL AVR microprocessors communicating by a SPI link. The first one, fbw (fly-by-wire), was responsible for the control of actuators and sensors and for the stabilization of the flight. It was also used to perform commands emitted by the wireless link. The second microprocessor, autopilot, was a bit more powerful and was concerned by the realization of the mission, that is, the choice of the flight plan. The system had several emergency mode activated according to the whole system state. In a first mode, it tries to return to its “home” base. In another one, it tries to save the integrity of the model plane by ensuring a minimal landing drive. And in a very bad case, it puts the actuators in a configuration ensuring it will simply plane gracefully in the hope it may land without breaking anything.

To perform a flight, the first task is to program a flight plan and to generate automatically a piece of code included in the embedded software system. Then, the full system is compiled and composed of two binaries programs: fbw and autopilot. In the next step, the programs are transferred to the controller and the plane is launched (by hand) and the controller starts to drive the plane. If all is ok, the flight plan ends by letting the model plane landing at its start point.

PapaBench is fully written in C language, so that it can be compiled using any C compiler.

5.2.1. Analysis of PapaBench using SWEET

PapaBench consists of two programs; autopilot and fly-by-wire. Both of the programs has several real time tasks including interrupt routines. WCET estimates and flow analysis can be performed both for the tasks and interrupts routines.

To show an example of analysis of PapaBench with SWEET, we consider the task “course_pid_run” from the autopilot program. If we run SWEET to generate graphs, SWEET will generate the rsg (reduced scope graph) which is shown in figure 13.
In the above figure, we can easily see the scopes (loops, functions etc.) and find the loops. If we run SWEET with the “ae” (abstract execution) option, a .gen file will be generated showing the loop bounds of the loops (i.e. course_pid_run_L1). This .gen file format is used by default. SWEET has also options for generating flow facts in a format that can be used as input for other WCET tools (i.e. aiT, Rapita etc.). If we look inside the .gen file, the result looks like the following:

: (course_pid_run, alf_label_5_8) : [] : alf_label_5_8 <= 1 ; %% uhss - Upper header bounds in scope context.

: (course_pid_run, alf_label_7_8) : [] : alf_label_7_8 <= 1 ; %% uhss - Upper header bounds in scope context."

This result is showing the loop bound for the for two loops in the task course_run_pid. Loop bounds are shown for the first loop “course_pid_run_L1” with the alf code header label “_5_8” and “course_pid_run_L2” with the alf code header label “_7_8”.

Figure 13: Reduced scope graph for task course_pid_run.
6. WCET Challenge 2011

The purpose of the WCET Tool Challenge (WCC 2011) is to be able to study, compare and discuss the properties of different WCET tools and approaches, to define common metrics, and to enhance the existing benchmarks and collect and develop new benchmarks. The WCET Tool Challenge has been designed to find a good balance between openness for a wide range of analysis approaches, and specific participation guidelines to provide a level playing field. This should make results transparent and facilitate friendly competition among the participants. However, participants and other interested parties should be aware that results from different WCET tools will still be hard to compare directly, and there is not yet an established classification or set of performance metrics in this field. Therefore, the purpose of this Challenge is not to establish “winning tools” [31].

The first WCET Tool Challenge event took place in 2006. The results were presented at the ISoLA 2006 conference, and a full report is available [31].

A new Challenge took place in 2008, with a revised set of benchmarks. The status and initial results were presented and discussed at the WCET Workshop 2008 on July 1. The slides of the presentation are available. The final report was published in the proceedings of the WCET Workshop 2008. The 2008 Wiki contains detailed information about participating tools, target processors, benchmark code, etc [31].

A third Challenge took place during the spring of 2011. For this Challenge, industrial real-time production code has been used for the evaluation and more complex target processor architectures have been considered. Details are provided in the 2011 Wiki [31].

A number of analysis problems or questions are defined for each WCC benchmark for the participants to analyze and answer. There are basically two kinds of problems; WCET analysis problems for WCET estimator tool and flow analysis problems for tools that are intended for flow analysis tools like SWEET. These tools do not have the “low-level” analysis for computing WCET bounds for the selected processors. Flow analysis problems can also show differences in the flow analyses of different WCET tools, and this may explain some differences in the WCET bounds computed by the tools [31].

By participating in WCC 11, SWEET entered the WCET tool challenge for the second time. Source-level program flow analysis has only been performed by SWEET as it does not support the target processors in the WCC 11. There were three benchmarks programs in WCC 11 and the analysis by SWEET was restricted to the PapaBench program.
7. Analysis of PapaBench for WCC 2011

This section describes the analysis of the PapaBench program with SWEET for WCC 2011. The main hardware used in this thesis was a PC with Windows XP professional. No emulators or simulators were used for this project. SWEET was the main software used in this thesis. The front end part of SWEET remained unchanged throughout the whole thesis. SWEET was modified during the thesis by fixing bugs found in it. In order to generate the necessary intermediate ALF code for using as input for SWEET, melmac [17] has been used. SWEET is command based tool without any GUI and because of this reason some scripts were used in the project setup and analysis. We will discuss them shortly later in this chapter. Cygwin [32] has been used in this thesis as a shell environment for running SWEET and performing all analysis. The method for project set up is described below.

7.1. Cross compilation of PapaBench program

This section describes the steps for cross compilation of PapaBench program. Our host system is a Windows machine with Intel Pentium processor. Since the PapaBench program is written for ARM microcontroller processor, we had to prepare our host system ready for compiling the PapaBench program. To make our host ready for cross compilation, we used the GNU ARM toolchain for Cygwin. The toolchain consists of the GNU Binutils; compiler set (GCC) and debuggers (Insight for Windows and Linux). Newlib is used for the C library. The toolchain includes the C and C++ compilers.

The following steps were followed for making ready the host Windows machine for cross compilation.

Step 1

In this step 1 we installed Cygwin with necessary packages. While we were installing Cygwin we had to take care of some important packages that are required for cross compilation. The “make” package is required for building source from Makefiles.

We used GCC-4.1 toolchain which actually includes the following packages:

- binutils-2.17
- gcc-4.1.1
- newlib-1.14.0
- insight-6.5

Step 2

This step includes installation of ARM-GCC Toolchain.

After we have got the binaries we built them in several steps. The toolchain were installed in the /usr/cross/arm directory. We used the command “export PATH=$PATH:/usr/cross/arm/bin” which sets a path to the /usr/cross/arm/bin: in the .bashrc file. As we will compile program for ARM processor, our target is “arm-elf”. For testing the installation we used the command “arm-elf-gcc -v” which among the other things also shows the GCC version number. The process of compilation is described below.

1. cd /usr/cross/arm/binutils-source
2. binutils-source/configure --target=arm-elf --prefix=/usr/cross/arm/bin --enable-interwork --enable-multilib
3. make all install
4. export PATH="$PATH: /usr/cross/arm/bin 
5. cd /usr/cross/arm/gcc-source
6. gcc-source/configure --target=arm-elf --prefix=/usr/cross/arm/bin --enable-interwork --enable-multilib --enable-languages="c,c++" --with-newlib --with-headers=/usr/cross/arm/newlib-source/newlib/libc/include

7. make all-gcc install-gcc

8. cd /usr/cross/arm/newlib-source

9. newlib-source/configure --target=arm-elf --prefix=/usr/cross/arm/bin --enable-interwork --enable-multilib

10. make all install

11. cd=/usr/cross/arm/gcc-source

12. make all install

13. cd=/usr/cross/arm/insight-source

14. insight-source /configure --target=arm-elf --prefix=/usr/cross/arm/bin --enable-interwork --enable-multilib

15. make all install

**Step 3**

In this step the Windows Path was set up by editing the Windows Path variable. After the above process we made sure our path includes the following:

C:\cygwin\bin;C:\cygwin\usr\local\bin;C:\program files\gnuarm\bin;

The first two are obviously for the Cygwin apps, the last is for the ARM-GCC toolchain.

**Step 4**

To test our installed cross compiler, we used the following command by running a hello.c program:

```
$ arm-elf-gcc -v -Wall -g -o hello hello.c
```

### 7.2. Compiling the PapaBench program

After we have successfully built the cross compiler, we continued with compilation of the PapaBench program. As the PapaBench consist of two programs, there should be two executable files in .elf format after compilation. Though there is nothing to do with .elf files in process of PapaBench analysis, we did it mainly because of to make sure that nothing is missing in the source files provided by the developers. To compile the PapaBench we followed the following three steps.

- Extracted the archive
- By looking at "conf/Makefile.local", we checked the configure the ARCH (architecture) and MACH (machine) variables. Configuration for compilation on local host are checked in this file.
- Finally in the top level directory of PapaBench program is observed for “Makefiles”. After the execution of the make command, it took some seconds to compile the whole program and the compiler generated two file like fbw.elf and autopilot.elf.

### 7.3. Processing Fly by Wire

After we have made sure that cross compiler could generate two executable files by compiling the PapaBench program, we continued with preprocessing of the source files of the PapaBench program. Without preprocessing the files, it is not possible to convert all source files to .ALF format. The preprocessing of source files is described below.

To preprocess the file, we had to find out the file dependency among the source files. The connected files for fly by wire was found out by looking at make file in top level directory. We found that there are a total of five files those are connected with main.c file in fly by wire program. The name of the files are: spi.c , ppm.c , adc_fbw.c, servo.c, uart.c. The procedure for preprocessing is same for all
those files except the change in file name and library file inclusion. So we will only discuss the procedure for main.c file, as described below.

By running the basic preprocessing command we had to search out the library files to include in preprocessing and at the same time we had to find out the definition of target hardware which was Atmega 128. The preprocessing command main.c file is:

cpp -I../../../../../avr/include/ -I../../../../../avr/include/ -I../../../../../avr/include/ -I../../../../../sw/include/ -I../../../../../sw/var/include/ -D__AVR_ATmega128__ main.c main_preprocessed.c

In the above command, -I is used for including library files for main.c and -D is used for target processor type definition. The preprocessed file is saved as main_preprocessed.c.

7.3.1. Error and correction

The above preprocessing command was working for all files except the uart.c file. Errors were generated and the definition of hardware, AVR Atmega 128 was not working for uart.c files. According to the header files definition, only two device could be defined. Either AVR ATmega 128 or AVR ATmega8. After we have added definition for AVR ATmega 8, the preprocessing was errorless.

7.3.2. Converting the source code to .alf format

The above preprocessed files were converted to .alf file which is the format used by SWEET. The converter is installed in another machine and to convert file we had to run a script which is named as “c_to_alf_using_christers_machine.sh”. We had total six files for the fly by wire program and we had to run the same command for all those files. The typical command for converting the main.c file is:

c_to_alf_using_christers_machine.sh main_preprocessed.c

It should be mentioned that after the preprocessing of the file main.c, the main.c file is saved as main_preprocessed.c which is actually converted to .alf file and after conversion, the new converted file name is main_preprocessed.alf. The command for other files can be obtained from the author upon request.

7.3.3. Problems and solutions during C to ALF conversion

After we have got .alf files for Fly by wire program, it was natural to analyze them by running SWEET directly using a shell command. But unfortunately this was not possible due to some errors that had to be corrected by directly editing the .alf files. The corrections were made by removing the “digits” and “::” from the .alf files. Instead of editing those manually, we used a “sed” (source editor) command to automate the whole process. The following command removes the errors from a typical main_preprocessed.alf file. The command for editing main_preprocessed.alf with “sed” is

```
  sed 's/::[0-9]*//g' main_preprocessed.alf > main_preprocessed_unmangled.alf
```

The same command was followed for other alf files (the full commands can be obtained from the author upon request).

After we have got new edited files, we run SWEET for all those six files to check consistency of all files and also to generate a call graph for the specific task. The command is:

```
sweet -i=main_preprocessed_unmangled2.alf,spi_processed_unmangled.alf,ppm_processed_unmangled.alf,servo_processed_unmangled.alf,uart_processed_unmangled.alf,adc_fbw_processed_unmangled.alf sed_unmangled.alf func=main -c -d g=cm
```

But after running this command we have found that there are still many errors in the main_preprocessed_unmangled.alf file due to the ALF language translator. The remaining errors that we got from the .alf file are mentioned below.
alf static check found problems:
In file main_preprocessed.alf, on line 191:
{ label 64 { lref 64 alf_label_30 } { dec_unsigned 64 0 } }
{ call
  { label 64 { lref 64 uart_init_tx::2147483647 } { dec_unsigned 64 0 } }
  result
}

From the above, it is clear that without the basic knowledge of ALF language it is not possible to find out the source of error. So we had to edit the file at specific line by checking the error and also by analyzing the ALF language semantics. We had to put lots of efforts at this points to understand the ALF language and to be able to edit the lines. For instance, in the file main_preprocessed_unmangled.alf, there was an error at line number 203 which was creating problem for SWEET to analyze files. The command that we used for solving the problem is:
```
sed -e '203 s/{ addr 64 { fref 64 alf_temp_var_1 } { dec_unsigned 64 0 } }//'
-e '207 s/{ addr 64 { fref 64 alf_temp_var_2 } { dec_unsigned 64 0 } }//'
''main_preprocessed_unmangled.alf > main_preprocessed_unmangled2.alf
```

In the above command the strings at line 203 and 207 are removed by using sed command. To connect two commands, -eis was used with the sed command. The same procedures was followed for other lines which are creating errors (the full commands can be obtained from the author upon request). After the error correction, the errorless file is saved as main_preprocessed_unmangled2.alf. After corrections of the file, the fly by wire program is ready for to be analyzed with SWEET.

7.4. Processing Autopilot

The list of files were figured out in the same way like fly by wire and the list of source files those were needed to be prepared are: main.c, gps_ubx.c, nav.c, modem.c, link_fbw.c, spi.c, adc.c, infrared.c, pid.c, uart.c, estimator.c, if_calib.c, mainloop.c and math.c.

Preprocessing of files in Autopilot is almost the same as for fly by wire with some exceptions. The main.c file in autopilot could not be directly preprocessed because of a static inline function problem. So “inline” words were commented. We have specified line numbers with sed command to get rid of this problem and also to avoid “nested comments” problems. We have used following command to solve the problem.
```
sed -e '95 s_inline_/*inline*/_
-e '144 s_inline_/*inline*/_
-e '165
s_inline_/*inline*/_ main.c > main_changed.c
```

In the above command line numbers have been specified to avoid nested comments problem for replacing “inline”. All “inline” are replaced with “*inline*”. The main.c file was now saved as main_changed.c, ready for preprocessing. The command for preprocessing is the same like fly by wire files except the definition “DUBX”. The command is:
```
cpp -I../../include/ -I../../var/include/ -I../../../avr/include/ -I../../../avr/include/avr/ -I../../../sw/airborne/autopilot/ -D__AVR_ATmega128__ -DUBX main_changed.c
main_changed_processed.c
```

The above command was used for preprocessing of the other files except the inclusion of library files and file name. The math.c file was edited to prepare it for preprocessing. Actually we had to edit some definition and also in math.c file and the command is below.
```
sed '24 c\
#define pp_fabs(x) ((x)< 0. ? -(x) : (x))
' math3.c > math4.c
```
7.4.1. Converting the source code to .alf format

Like fly by wire, a similar procedure had been followed for converting the autopilot program files. The full commands can be obtained from the author upon request.

7.4.2. Corrections

This is the point where we had to put lots of efforts to make the .alf files ready for analyzing with SWEET. When we tried to check files consistency by running SWEET with all the files, some errors remained. As the errors were found in .alf file, we had to put efforts on learning the .alf language semantics to be able to do the necessary corrections. SWEET was also upgraded by fixing bugs in it.

7.5. Automation of PapaBench program analysis

The most interesting part of this thesis was making the analysis fully automated. To make the analysis of PapaBench program easy and less time consuming, we have fully automated the analysis by creating two scripts. The script `script_for_auto.sh` was created to analyze the autopilot program which will actually preprocess files, convert to .alf format, remove the errors and finally analyze files with SWEET command. Though there were many errors in autopilot program files, the automation made it easy to analyze them with desired results. The script `script-for_fbw.sh` was created for fly by wire program and like the autopilot script it is also fully automated. So when we analyzed the PapaBench for WCET challenge, we only had to write the necessary SWEET parameters to get the desired results. This actually saved a lot of time and we were able concentrate more on analyze with SWEET rather than repeating instructions. The full commands can be obtained from the author upon request.

7.6. Analyzing PapaBench with SWEET

The following steps were involved for analyzing PapaBench with SWEET.

1. Run the scripts for both programs
2. Deriving the annotation file for desired task
3. Run SWEET with required options along with annotation file

After converting all files for both programs (Autopilot and Fly_by_wire) the following command is used to generate SWEET graphs for a specific function:

```
sweet -i=file1.alf,file2.alf,file3.alf,file4.alf,…,fileN.alf func=function -c -d
g=sgh,fsg,rsg,cg,cfg
```

This is an example of a simple command in SWEET. But there many more advanced commands for more deep and specific analysis with SWEET. For instance, the “-ae” option is used for abstract execution. The command `sweet -help` will show details in the help file for more options that can be used with SWEET.

The typical command we used to check the ALF code, generate graphs, compact alf file and generate map file was:

```
sweet -i=file1.alf,file2.alf,…,fileN.alf func=course_run -d g=sgh,fsg,rsg,cg,cfg
file=course_run -fi=course_run p=alf -mpal=course_run.map
```

By the above command, all .alf files were made compact into one file. Then it is easy to perform abstract execution using this single file.

The typical command for the abstract execution used in our experiment is:

```
sweet -i=course_run.alf func=course_run -x ffg=uhss merge=all -f lang=gen,rapita
```

By the above command we directed the output of the flow facts to a format that can be used by other tools like Rapitime from Rapita.

The command for performing abstract execution by using an annotation is:
sweet -i=navigation_update.alf annot=navigation_update.ann func=navigation_update
-D noglobinit -x ffg=uhss merge=all -f lang=gen,rapita

In the above command the annotation file is navigation_update.ann is used for performing abstract execution with output for the Rapita tool.

For creating flow facts for the aiT tool, we used the following command for navigation_update :
sweet -i=navigation_update.alf annot=navigation_update_case2.ann func=navigation_update
-ae ffg=uhss merge=all -f o=navigation_update_case2
lang=ff,ais

In the above command, flow facts were generated as .ais format which is the input format for the aiT tool.

We have automated the above three phases in two different scripts; one for autopilot and one for the fly_by_wire program. So by simply typing script_for_auto.sh or script_for_fbw, we will run all those phases at once.

7.7. Problems encountered

The c_to_alf translator was installed on another machine, and we had to set up the local workingmachine to for running the script “c_to_alf_using_christers_machine.sh” and because of some user administrative problems it stopped working. It was fixed later on.

To get the full executable file (.elf) for both the autopilot and fly_by_wire programs, we had to perform cross compilation for the target ARM hardware. As we were running Cygwin on host windows computer (i686), there was a problem regarding the native compiler and the standard C compiler. Thus as a consequence, we could not get an executable file at once. Also, there were sometimes too few details in the PapaBench documentation. Finally after correction of errors regarding the cross compilation tool-chain installations, we could manage to have executables for both the autopilot and the fly_by_wire programs.

The c_to_alf translator we used in this experiment had some problems with some programming keywords and thus we had to tweak those. Also, some of the errors we discovered were corrected later on after the converted .alf files were analyzed by SWEET. For example, when we were analyzing a single .alf file it was ok. But after we have put all files together, many errors were found. Then we had to study the ALF language grammar and went through the errors, and later on we were able to solve the errors with a kind of “pattern matching”. This actually saved us lots of time at the later stage of SWEET analysis. Also, bugs were found in SWEET which was kind of expected, as it is a research prototype. But it took time to solve the bugs.

There were problems with some piece of codes in PapaBench programs. It took little time to discover those problems with code after manual observations. We reported those problems to the responsible person of the benchmark.

7.8. Analysis of the problems in WCC 2011 and our results

In WCC11, PapaBench had a number of problems to be solved by the participants, both for WCET estimation and for flow analysis results. Our report was based on performing flow analysis to solve the specific problems. There were six problems for flow analysis and we were able solve all of them. Loops were involved in those problems, and SWEET was able to automatically derive the bounds for the loops, and the bounds were precise. There was an infinite loop in one problem, but the .alf codes were patched to limit the bound on this loop. Another problem was connected with absolute addresses defined in some places of PapaBench. However, this problem was solved by changing the analysis mode of SWEET. The full report to WCC 11 is attached with this thesis report as appendix A.

To understand those six problems in WCC 2011, some background about the real-time tasks and sub programs of PapaBench should be known.
PapaBench was extracted from an actual embedded real-time system, called Paparazzi, which drives an UAV (Unmanned Aerial Vehicle). There are two programs, called fbw (for Fly By Wire) and autopilot, each running on its own processor with its own local separated memory space and its own input / output peripherals. The communication between both processors is performed thanks to a Serial Peripheral Interconnect link with no straight effect on the memory space of each other. Each program implements an infinite control loop that calls the tasks performing the UAV drive. In PapaBench, the entry point of a task is a C function which is called from the control loop. In turn, each task function may develop its own program call tree to achieve its work. Usually, analysis is applied on a whole program. With PapaBench, analysis must be made on the functions which are tasks of the application. Every task has its own matching C function entry point. For example, C Entry Point Function for the task which receives MCU0 values is check_mega128_values_task.

For every problem in WCC 2011, there are corresponding functions that were to be analyzed and flow analysis result were to be reported. More details are described below.

7.8.1. Problem: AutoPilot A1 – UAV driving according to the flight plan

This task ensures that the UAV is following the required course angle, changing the course if required. It contains two loops that ensure that an angle is in the interval [-PI, PI]. This angle is determined as the difference between the current course (obtained from GPS) and the required one (from the flight plan). In fact, variables used to compute this angle are assigned in different tasks. The task for this problem is course_run which is actually a subprogram of main.c in Autopilot program. There are actually four inputs for this task and these are estimator_hspeed_dir, gps_fcourse, ubx_msg_buf+24 and desired_course. Flow analysis is to be reported for F1 (Bound of loop pid.h:31) and F2 (Bound of loop pid.h:32). This pid.h is a header file and the macro NORM RAD ANGLE is defined at line number 31 and 32. The loops are called in a function course_pid_run which is found in pid.c file. The PID (Proportional Integral Derivative) controller (roll, pitch, climb, altitude, course) is implemented in this pid.c file. We found the expression NORM_RAD ANGLE(err) in pid.c file. This err is actually a float value defined in the same function and we have found three cases after investigation. These are -PI <= err <= PI, PI < err < 2*PI and -2*PI < err < -PI. The value of err is calculated from the differences of two inputs variables estimator_hspeed_dir and desired_course. Loop bounds are estimated by changing the value of these two inputs. For all those three cases we used three different annotation files. One example is case one, where we have used FLOAT values of estimator_hspeed_dir 0.0 and desired_course 0.0 in the annotation file. Then we performed abstract execution by running SWEET for this case and got the desired results for loop bounds which is correct according to PapaBench’s documentation. For the two problems F1 and F2 we also got the correct loop bounds.

7.8.2. Problem: AutoPilot A2a – navigation management in HOME mode

If a problem occurs, the UAV controller moves from normal mode to HOME mode. This means that it will try to go back to its starting point. The subprogram for this problem is navigation_update and this task has one input named pprz_mode. Flow analysis for this problem is to be reported for F1 which is loop bounds for line number 75 in the nav.c file in the autopilot program. At line 75 of the nav.c file, a macro NormCourse is defined. This macro NormCourse is used macro CircleXY and then macro CircleXY is used in macros Circle and Goto3D. Macro Circle is used in the function nav_home.

Now the question is how we should actually continue with flow analysis and loop bound for macro at line 75? At first we Check the ALF code, generate graphs, compact alf file and map file by running SWEET. As navigation_update is the function for this flow analysis problem, we get the mapping file for navigation_update by running SWEET. To set the input value for this flow analysis, we use an annotation file where we set pprz_mode to PPRZ_MODE_HOME (value 3). This pprz_mode is a global variable used in macro Goto3D and to continue with flow analysis, its value should be bounded by some value. After calculation we found that a variable called estimator_hspeed_mod should be
bounded with some values and we have found three cases depending on those values. Like the previous problem we use three annotation files to get the loop bound by running SWEET.

7.8.3. Problem: AutoPilot A3 – send the GPS position to ground

This task performs two things. One is effectively sending the GPS position to the ground station and another is estimate the pitch according to the infrared sensor. Estimate the pitch is driven by a condition. At first run, an initialization is performed and for other runs, the initialization is marked done and not performed. The sub program for this problem is send_gps_pos which is task for sending GPS position to ground station. No input variable is defined for this task and this function is called two times. Flow analysis should be reported for F1 and F2 in case of this problem. Both of them are indicated at line number 181 in estimator.c file in Autopilot program. By looking at that specific line we see that a function estimator_update_ir_estim is called.

The question for problem F1 is whether the function is executed at in first run or not, and for F2 whether the function is executed or not in the second run. So we go for relation of this function to other function calls and variables. We get a little problem in this case with hard coded addresses. This was solved by updateing SWEET.

The function timer_periodic() in main.c file returns a value which actually could be true or false. This creates a problem with an infinite loop. So we had to terminate the loop by adding a bound on the upper limit of the loop. Finally by analyzing with SWEET, we found that flow fact which means that the call to estimator_update_ir_estim, which is this position in the source code, is never called in any of the two iterations of the outer loop in which this call is made. So we got the correct answer for both F1 and F2.

7.8.4. Problem: Fly-By-Wire F1b: AutoPilot command transmission to servos (no command to transfer)

The command transmission to servos from the Autopilot is called periodically. Sometimes, no command has come and nothing need to be transferred to the servos. This task enables the interrupt to send information to the servos. This scenario persists as long as the system stays in this mode. The sub program for this problem is check_mega128_values_task which is a task function with input of a global variable spi_was_interrupted with value 0. Flow analysis is to be reported for the question whether line 221 in main.c file in Fly-By-Wire program is executed or not. By observing at line number 221 in main.c file and by following the program flow we discovered a problem with an infinite loop in the main.c file. The function timer_periodic writes to hard-coded addresses in memory and thus cause infinite loop. We had to terminate the loop and set the interruption to 0. So by then, after setting the global variable spi_was_interrupted to 0, SWEET gave us the flow facts which actually means that the line number 221 was never executed. This was the correct result for problem number 6.
8. Conclusions and Future Work

8.1. Conclusions

The main purpose of this thesis has been to analyze the benchmark program PapaBench with SWEET. This is the second time SWEET participates in WCC 11 and all six flow analysis problems in WCC 2011 were solved successfully. Due to lack of time, it was not possible to analyze the other two benchmark programs included in WCC 11.

The preparation work for converting source files to ALF format was time consuming and problematic because of the current c_to_alf compiler. A better translator is needed for the source code conversion.

One of the most excellent features of SWEET is its own flow fact language which can easily be directed for output to another WCET tools like aiT and RapiTime. However, a time consuming feature of SWEET is the automatic flow analysis using option abstract execution.

By performing manual observation on source codes from Bombardier, we have discovered that most of the tasks are simple programming constructs without any loops or recursions. Only a few tasks contain some, simple loops. The source codes are mainly machine generated and our current c_to_alf translator can’t translate most of these source files because of not having necessary operating system services. As we did not get the OS (operating system) from Bombardier, the necessary operating system services are not available in our experiment. Because of this reason we could not cross compile the whole software to get executable and stub implementation were needed for compiling all source files separately to analyze with WCET tools. This process will take long time for this giant software. But by performing through manual observation on all codes we found the fact that there is no new code properties that could be added to Mälardalen WCET Benchmark suite. As the main reason behind getting industrial code was to add new benchmark and considering this factor, we have excluded the analysis result from this thesis report.

During this thesis work, we had to go through a variety of benchmark programs and we learned a great lesson which we must follow for extension of more benchmark programs in MDH website. The lesson is “if we are going to provide a good set of benchmarks we also have to provide good support for setting them up in an easy way on a variety of target platforms.”

8.2. Future work

The current MDH benchmark programs are too small for cache analysis and most of them are not real-time industrial applications. The WCET tools along with their associated methods and algorithms should be evaluated by testing real-time industrial codes with larger size with mixed types of code constructs. Hardware features are also should be considered for benchmark extension. Some types of code constructs, like code with highly context sensitive execution behavior, programs with complex low-level code (like bit-operations and shifts), use of dynamic memory, mode-specific behavior, tasks with multiple roots, tasks wrapped in a loop, and programs using function pointers are also to be considered for future benchmark program extension.

The current MDH benchmark program suite should be updated by changing the current website into a open wiki and also by making precompiled binaries for more types of compilers and processors. The website should also be updated by upgrading the information about SWEET so that information about SWEET can easily be accessed for further research.

The current c_to_alf translator should be corrected or replaced by a new one.
9. References


[13] Saranya Natarajan, Developing an ALF interpreter for the SWEET WCET analysis tool, EURECA Exchange Student, Master’s thesis, Department of Computer Science and Engineering, Mälardalen University, Sweden

[14] Andreas Ermedahl, Jan Gustafsson, and Björn Lisper, ALF (Artist2 Language for Flow Analysis) Specification, Dept. of Computer Science and Electronics, Mälardalen University, Västerås, Sweden

[15] Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg, School of Innovation, ALF – A Language for WCET Flow Analysis, Design and Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden


http://www.mrtc.mdh.se/projects/wcet/benchmarks.html


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

[22] Dani Barkah. Evaluating program flow analysis for WCET calculations at Volvo CE, Master’s Thesis, Department of Electronic, Computer, and Software Systems (ECS), School of Information and Communication Technology (ICT), Royal Institute of Technology, Sweden

[23] Friedhelm Stappert, Andreas Ermedahl, Jakob Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. CASES ’01 Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems


[26] Peter Altenbernd, Andreas Ermedahl, Björn Lisper, Jan Gustafsson. Automatic Generation of Timing Analysis of High-Level Code


[28] WCET TOOL CHALLENGE 2011: REPORT


[31] WCET Tool Challenge.URL : http://www.mrtc.mdh.se/projects/WCC/


[34] http://www.complang.tuwien.ac.at/satire/
Appendix A

Report from SWEET analysis for WCET Challenge 2011

Jan Gustafsson and Nazrul Mohammad Islam
School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden.

jan.gustafsson@mdh.se, mim08002@student.mdh.se

1. Introduction

This report describes the results from using SWEET in the WCET Challenge 2011.

SWEET (Swedish WCET Analysis Tool) is a prototype tool for WCET analysis developed by the Mälardalen WCET research group. SWEET can handle full ANSI-C programs including pointers, unstructured code, and recursion. Unlike most WCET analysis tools, SWEET is integrated with a compiler and performs its flow analysis on the intermediate representation ALF (ARTIST2 Language for Flow Analysis) code of the compiler, after structural optimizations.

ALF is a language intended to be used for flow analysis in conjunction with WCET (Worst Case Execution Time) analysis. ALF is designed to be possible to generate from a rich set of sources: linked binaries, source code, compiler intermediate formats, and possibly more. ALF is designed to be amenable to program analysis rather than code generation. In that respect, it is therefore different from most compiler intermediate formats. ALF is designed to be possible to generate from a rich set of sources: linked binaries, source code, compiler intermediate formats, and possibly more. This has certain implications for ALF’s program model, which must encompass both high- and low-level constructs while being as amenable to program analysis as possible.

Thus, the control structure of the ALF code and the object code is similar, and the flow analysis for the ALF code is valid for the object code as well. The low-level analysis of SWEET currently supports the NECV850E and ARM9 processors. The tool currently supports three different calculation methods: a fast path-based method, a global IPET method, and a hybrid clustered method. SWEET currently uses abstract execution with intervals, for the flow analysis. It allows the user explicit control over the placement of merge points, to control the tradeoff between precision and analysis time.

Since SWEET does not support the target processors in the WCET Challenge 2011 (MPC5553/MPC5554, ST10, and ARM7), we only perform flow analysis this time. We decided to analyse only the PapaBench code, for the following reasons:

• We excluded debiel due to lack of time.
• We also excluded the Daimler code, since we anticipated problems for the students to use our tool. Especially the translator from C to ALF (see below) is an unstable prototype.

The translator from C to ALF was developed during the ALL-TIMES project 2007 - 2010. However there are still some bugs in the translator which cause some trouble for certain C constructs. We are currently in the process of writing a new translator for C to ALF, based on llvm.

We will describe the main workflow and results of the analysis in this report. Detailed information, like commands, files etc. will be sent upon request.

2. Flow analysis problems and their solutions

SWEET has been used in the WCET Challenge 2011 only for generating flow facts, i.e., doing flow analysis. Flow analysis using SWEET is done in a number of steps:

1. Preprocess the C source file, using cpp.

2. Translate the preprocessed code to ALF code using a C to ALF translator.

3. If necessary, generate input annotation files for the analysis. This is needed to limit the input variable values.

4. Analyse the ALF code with SWEET. SWEET has a large set of different analysis possibilities; linking of ALF files, mapping between C and ALF files, syntax check, graph generation, many types of flow fact generation, and more.

2.1. Problem: AutoPilot A1 – UAV driving according to the flight plan

Flow-analysis results to be reported:

<table>
<thead>
<tr>
<th>Number</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Bound of loop pid.h:31</td>
</tr>
<tr>
<td>F2</td>
<td>Bound of loop pid.h:32</td>
</tr>
</tbody>
</table>

We have found three cases, depending on the value of the variable ”err” at line 67 in file ”pid.c”:

\[
\text{NORM\_RAD\_ANGLE}\text{(err)};
\]

We get the flow facts (loop bounds) below in the AIS format. The mapping from ALF to C is created by the translator from C to ALF, and used by SWEET. The line number is number 67 in file ”pid.c”, and the column number refers to the pre-processed code (note that NORM RAD ANGLE is a macro which is defined at line 31-32 in pid.h, but is used at line 67 in pid.c).

Case1: -\Pi \leq err \leq \Pi

#AISflow facts generated using mapping file
#Label specifications
#Label:pid.c_67_20 refers to file:
pid.cfunction:course_pid_run line: 67 column:20 and
alf_label:alf_label_5_8

48
Case 2: $\pi < \text{err} < 2\pi\pi$

The line and column numbers are the same as in case 1.

```c
#Loopbounds
LOOP "pid.c_67_20" MAX 0 END ; LOOP "pid.c_67_104" MAX 0 END ;
```

Case 3: $-2\pi < \text{err} < -\pi$

The line and column numbers are the same as in case 1.

```c
#Loopbounds
LOOP "pid.c_67_20" MAX 1 END ; LOOP "pid.c_67_104" MAX 0 END ;
```

2.2. Problem: AutoPilot A2a – navigation management in HOME mode

Flow-analysis results to be reported:

<table>
<thead>
<tr>
<th>Number</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Bounds of loop at nav.c:75</td>
</tr>
</tbody>
</table>

The line 75 in nav.c looks like this:

```c
#defineNormCourse(x) {while (x <0) x+=360; while (x >=360) x-360;}
```

which means this it is a macro definition. The macro is used in macro Circle which in its turn is used in macro CircleXY. This macro is used at line 175, which can be seen in the flow fact below.

We have found three cases, depending on the value of the variable "estimator_hsspeed_mod".

Case 1: $5.0 < \text{estimator\_hspeed\_mod} < 100.0$

```c
#AIS flow facts generated using mapping file
#Label specifications
#Label:nav.c 175_441 refers to file: nav.c function:nav_home line: 175 column: 441 and
alf_label:alf_label_484 $1
LABEL FILE "nav.c" LINE 175="nav.c_175_441" ;
...
Case 2: \(0.01 < \text{estimator\_hspeed\_mod} < 5.0\)

The line and column numbers are the same as in case1.

```c
#Loopbounds
LOOP "nav.c_175_441" MAX0 END ; LOOP "nav.c_175_469" MAX0 END ;
```

Case 3: \(-1000.0 < \text{estimator\_hspeed\_mod} < -0.1\)

The line and column numbers are the same as in case1.

```c
#Loopbounds
LOOP "nav.c_175_441" MAX 1 END ;
LOOP "nav.c_175_469" MAX 0 END ;
```

2.3. Problem: AutoPilot A2b – navigation management in normal mode

No flow analysis results to be reported.

2.4. Problem: AutoPilot A3 – send the GPS position to ground

<table>
<thead>
<tr>
<th>Number</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>In first run, is estimator_c:181 executed?</td>
</tr>
<tr>
<td>F2</td>
<td>In second run, is estimator_c:181 executed?</td>
</tr>
</tbody>
</table>

Line 180-181 in estimator\_c looks like the following:

```c
if (estimator\_flight\_time)
estimator\_update\_ir\_estim();
```

When analysing this problem, we had to remove the ”inline” keyword at lines 94, 144 and 165 in main\_c, since our C to ALF translator did not accept this use of the keyword.

There was also a problem with analyzing the following loop at line 73-75 in main\_c:

```c
(init\_cpt) {
  if (timer\_periodic())
    init\_cpt--;
}
```

Since timer\_periodic() reads from a hard-coded address to a timer, it returns an unknown value which could be true and false, and therefore there is no upper limit on the number of iterations of the loop! We had to tweak the analysis so that the loop had to terminate.

SWEET then produced the following information (in AIS format) by using the ”unsp” option (upper bounds for nodes in program context):

```c
#Label: estimator\_c_181_7 refers to file:estimator\_c function: estimator\_update\_state\_gps
line: 181 column: 7 and alf\_label: alf\_label_54\_S10
LABEL FILE "estimator\_c" LINE 181="estimator\_c_181_7" ;
...
```
This flow fact means that this node, i.e. the call to estimator update ir estim, which is this position in the source code, is never called in any of the two iterations of the outer loop in which this call is made.

2.5. Problem: AutoPilot A4 – parse a message from the GPS

No flow analysis results to be reported.

2.6. Problem: AutoPilot A5 – send log information to the ground

No flow analysis results to be reported.

2.7. Problem: AutoPilot A6 – control input from the radio link

No flow analysis results to be reported.

2.8. Problem: Fly-By-Wire F1a: AutoPilot command transmission to servos (nominal case)

No flow analysis results to be reported.

2.9 Problem: Fly-By-Wire F1b: AutoPilot command transmission to servos (no command to transfer)

No flow analysis results to be reported.

<table>
<thead>
<tr>
<th>Number</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBW.1.b.F</td>
<td>main.c:221 is not executed</td>
</tr>
</tbody>
</table>

This program contains an infinite loop at line 154 in main.c. We had to assure that this loop terminated (was executed only once). Then, after setting the global variable "spi was interrupted" to 0, SWEET gave the following result:

```
#Label:main.c_221_2 refers to file:
main.c function:check_mega128_values_task line:
221column: 2and alf_label: alf_label_91
LABEL FILE"main.c" LINE221="main.c_221_2" ;
...
FLOWSUM("main.c_221_2") <=0;
```
This means that this line was never executed.

2.9. Problem: Fly-By-Wire F2:SPI port reception

No flow analysis results to be reported.

2.10. Acknowledgment

We are grateful for the support given by Andreas Ermedahl, Linus Källberg, and Björn Lisper.

Disclaimer: the instruction on what to achieve and how to do the flow analysis was not always fully clear, and we apologize for any misunderstandings.