FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Predicting Stack Use in Embedded Software Applications

Carlos Daniel Alves Garcia

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Mário Jorge Rodrigues de Sousa

Second Supervisor: Eng. Carlos Manuel Rodrigues Machado

February 19, 2019

Abstract

On mass market products the production cost is one of the main concerns. This constraint also applies to electronic controller units (ECU) used in the automotive industry, leading to a choice of computerized systems with limited resources (such as Code Flash, Data Flash, RAM, Stack usage and CPU load). It therefore becomes essential to monitor the resources used by the embedded software during its development.

The monitoring ensures that the architecture, the implementation and functionality all fit within the hardware limitations. The resource monitoring may be done at compile time using static analyses techniques, or during runtime. The former predicts the use of resources by analyzing the source code. The latter focuses on analysis at runtime.

This dissertation describes the architecture and implementation of a tool to monitor the use of the stack resource by all the embedded software in an ECU. A prediction for stack use is obtained during the software development process using a static analysis approach. With the help of the tool, embedded software developers can determine an upper bound for the use of Stack through a deterministic way.

Acknowledgments / Agradecimentos

A realização desta dissertação de mestrado tornou-se possível graças a várias pessoas que impulsionaram o meu trabalho. Assim, não posso deixar de mostrar o meu eterno agradecimento pelo apoio que recebi.

Gostaria de agradecer ao meu orientador da faculdade, Professor Mário Jorge Rodrigues De Sousa, pela orientação, disponibilidade e contributo prestado ao longo da realização desta dissertação.

Ao meu orientador na empresa, Engenheiro Carlos Manuel Rodrigues Machado, pela paciência e auxílio no enriquecimento do trabalho desenvolvido.

À Faculdade de Engenharia da Universidade do Porto, pela cultura, oportunidade, condições e recursos, possibilitando o desenvolvimento desta dissertação. Tendo sido também, uma segunda casa ao longo do meu percurso académico.

Um enorme obrigado aos meus colegas de trabalho que me acolheram da melhor maneira desde o meu primeiro dia na empresa.

À minha namorada pela infinita paciência e flexibilidade que teve durante a realização da dissertação.

À minha família pela confiança e apoio que tiveram em mim desde o meu primeiro dia na faculdade.

Ao meu avô pelos valores que sempre me tentou transmitir.

A todos os meus amigos que fizeram com que este percurso se tornasse inesquecível.

Carlos Garcia

"You should be glad that bridge fell down.
I was planning to build thirteen more to that same design"

Isambard Kingdom Brunel

Contents

List of Figures

List of Tables

Abbreviations and Symbols

CFG	Control Flow Graph
CPU	Central Processing Unit
ECU	Electronic Control Unit
EPB	Electric Park Brake
GPL	General Public License
GUI	Graphical User Interface
NN	Neural-Network
RAM	Random-access memory
RTOS	Real-Time Operation System
SRS	Supplemental Restraint System
SSA	Static Stack Analyzer
SW	Software
UML	Unified Modeling Language
WCSU	Worst Case Stack Usage

Chapter 1

Introduction

This master dissertation was developed under industrial environment. The work aims to find a practical solution attending to the company's needs.

This dissertation started on September 3rd, having a duration of 4 months. The work was held in a company considered leader in the automotive industry and technologies. Due to confidentiality terms, the company will not be identified.

The company's products are produced in large scale. On mass market the production cost is the core concern. This constrain leads to a use of computerized systems, which have limited resources, for being less expensive. Being the automotive industry a demanding, competitive and constantly evolving sector, it is essential that companies focus on innovation and productivity in order to create top notch products and captivate new clients. The company is developing a product that will be produced in large scale. The solution proposed in this dissertation intends to make the architecture, implementation and functionality of the company's product adequate to the resource limitations.

The next sections of this chapter provide information to contextualize and explain the work done during these four months.

1.1 Problem Definition

Embedded systems have a wide variety of resources. However, for this particular case, the resources that most concern the company are RAM (random access memory) and ROM (read-only memory). More concretely regarding the RAM, the concerns relate only to the data/BSS segment (allocation of static variables) and to the stack segment. There is no usage of dynamic allocation and therefore, the Heap will not be considered.

Monitoring those resources is essential to improve the reliability of the system. Hence, this dissertation proposal emerged. Regarding the work done within the dissertation, the focus is just the stack. The main goal with the monitoring of the stack is to find the WCSU (worst case stack usage) scenario. Thus, the engineers would be able to set a minimal bound for the stack and therefore avoid unexpected defects at runtime created by stack overflow failures. The task to set

a bound for the stack is not trivial. If the stack is oversized, there will be more wasted system resources than necessary. On the other hand, an under sizing will create a risk of stack overflow.

For a better understanding of the problem, a brief characterization of the system follows. The company is developing a safety-critical embedded system, based on a microprocessor that runs under a RTOS (Real-Time Operating System). The task model is periodic and uses a fixed-priority preemptive scheduling. The system has a CPU (Central Processing Unit) with two cores and a single stack for each core. There is no memory isolation and therefore, the tasks use shared stacks.

To tackle this problem two methods were identified. The first and very common one is by testing the software. This method works like a watermark in a river. So, the entire stack is filled up with a predefined value. Then, after executing the program, it should analyze its values and check where the mark stills. That amount of stack is the worst case scenario for those specific inputs and runtime, which will probably differ if the method is repeated. The other method considered consists in analyzing the code without actually executing it. For example, by using a tool capable of static stack analysis. Basically, the tool would be capable of collecting information about the stack usage per function and all the possible sequence of function calls (control flow graph) throughout the entire SW (software). Thereby, the tool could combine that information in order to find the WCSU scenario.

In order to protect the intellectual property of the company, a scenario with similar characteristics to the company system was created. From now on, consider the following scenario as if it were the company's. Imagine a real-time embedded system based on a microprocessor with a single stack. The system has a fixed-priority scheduling designed to control an intelligent semaphore. The semaphore has input buttons for the pedestrians and sensors to detect the presence of vehicles (Input_Task). Besides that, the main difference between this semaphore and the traditional ones is that it has an incorporated Neural Network (NN). This NN collects data from the vehicles traffic and the pedestrians and tries to find patterns (learning) aiming to mitigate the waiting time per car on the semaphore. Then, it must evaluate the learning (NN_Task) and according to that it corrects (SmartCorrection_Task) the state of the semaphore (LightStatus_Task). The CFG (control flow graph) of the semaphore system is represented on the figure 1.1.

1.2 Goals and Motivation

At this moment, the company's project is developed under continued delivery and integration using the Jenkins platform [2]. This platform allows the developers to do some of the resource's monitoring. However, it is incomplete because it only includes dynamic monitoring (testing the software) of the resources, which has some drawbacks that will be enumerated later.

Strategies to predict in compile time the stack usage of all the embedded software in the ECU (Electronic Control Unit) should be established. The strategies defined should gather useful information to make stack management decisions and guide code optimization in order to prevent and avoid stack overflow failures. The goal is to define strategies to find the worst case stack usage scenario during the software development process and therefore ensure stack safety.

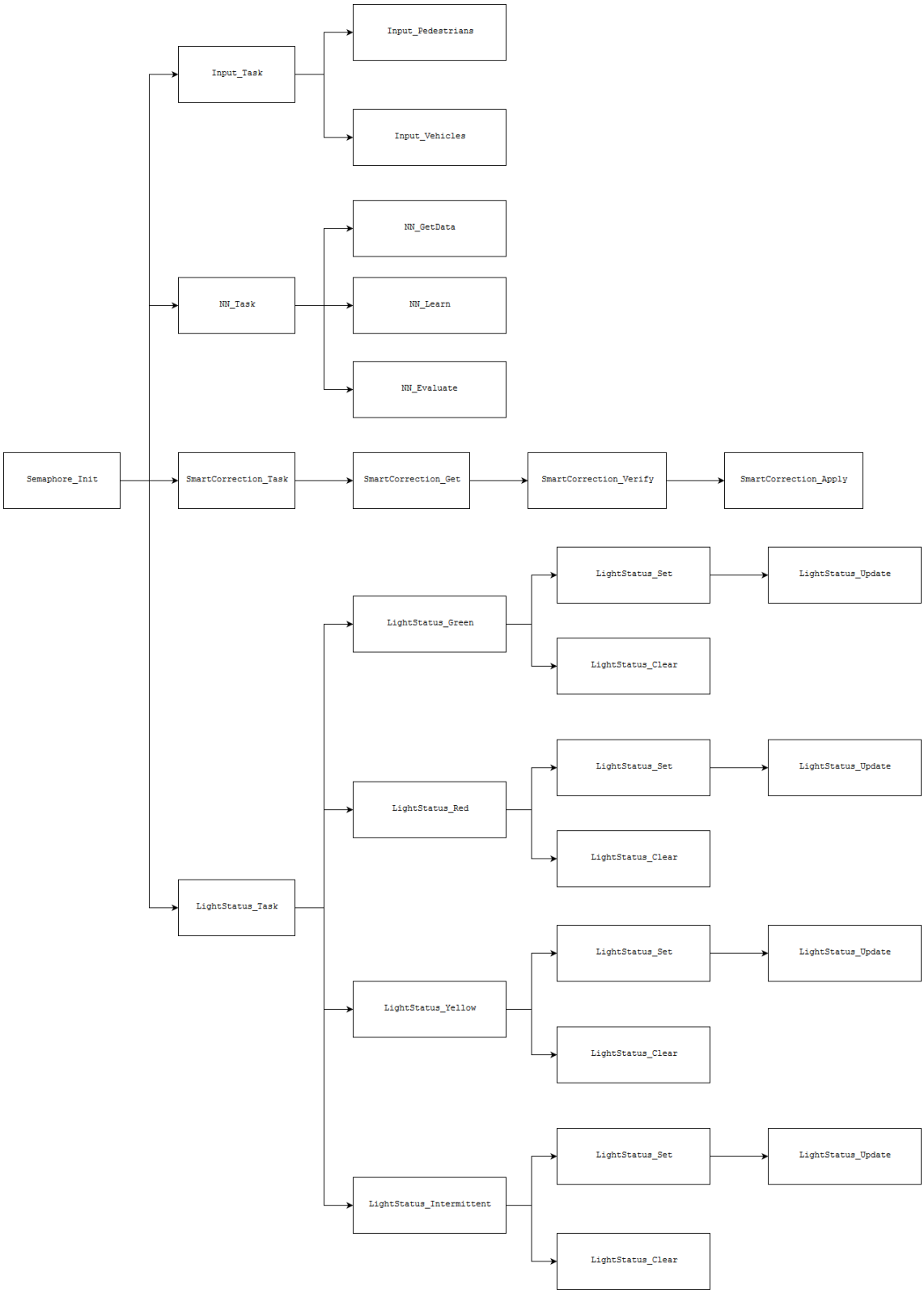

Figure 1.1: Semaphore System Control Flow Graph

1.3 Dissertation Structure

Regarding the structure of this dissertation, in addition to the introduction chapter, there are four more chapters.

In chapter 2 the literature review is provided. A research of background knowledge regards this chapter.

In chapter 3 a description of the tool is presented. Besides that, the requirements of the software and description of the architecture is identified. It also enumerates some challenges faced by the static stack analyzer tool founded during the implementation stage. Furthermore, there is a description of the implementation and validation phase of the software, as well as an explanation for how to use the SSA (Static Stack Analyzer) tool.

Chapter 4 is about the analysis and report of the results. In addition, ways to get around with the over consumption of the stack along with its advantages and disadvantages are presented.

Lastly, chapter 5 provides a conclusion. Improvements to the SSA tool are proposed. A self-evaluation of the work is also performed considering the fulfillment of the Software requirements. Moreover, possible future works are identified.

Chapter 2

Literature Review

This chapter regards the study of the state of the art in relation to the subjects of this dissertation. The background research allowed finding a route to follow to perform the work.

In section 2.1 some important basic concepts are defined. Then, in 2.2 different approaches for stack safety are presented as well as their advantages and disadvantages. After that, a comparison between them is done in order to better understand the study subject and to define a direction for this dissertation.

2.1 Basic Concepts

2.1.1 Embedded Systems

First of all, it is important to define what an embedded system is. Obviously, there are many different ways to define it present in the most varied literature. However, there is consensus that an embedded system can be considered as system based on a microprocessor, designed to perform one or more specific functions [3]. These systems differ from a computer because they are not intended to be programmed by the user. Generally, embedded systems have hardware limitations and do not accept the installation of software [4].

Embedded systems are often used in hard real-time systems and safety-critical systems environments. Hence, there are time constrains. Embedded systems must be able to process and fulfill certain tasks within a certain known time.

Nowadays, industries like automotive rely heavily on embedded systems.

2.1.2 Safety-Critical Systems

Embedded software are often designed to be used in safety-critical systems. Safety-critical systems are characterized for having low fault tolerance. A failure on those systems can lead to serious consequences like loss of life, huge damages or even destruction of the system.

Stack overflow failures, due to bad coding (e.g. incorrect use of dynamic memory allocation) or hardware limitations of the embedded microcontroller, may cause unexpected runtime errors

which are hard to debug. Also, a runtime overload is critical for systems with real-time characteristics.

There is a wide variety of safety-critical applications that uses embedded software, such as the SRS (Supplemental Restraint System), the EPB (electric park brake), medical devices like defibrillators and so on. Because undetected bugs can be very costly, it is crucial to attempt to find software defects before delivering.

Thus, the resource usage and execution time monitoring of embedded systems become essential and strategies must be defined to avoid and prevent stack overflow and runtime overload failures.

2.2 Approaches for Stack Safety

One of the quality criteria for embedded software is stack safety [5]. Stack safety can be implemented through an established upper bound for the worst case stack usage scenario and so, avoid stack overflow failures. Therefore, stack safety is the assurance that the software will not suffer a stack overflow failure at any time.

Obviously, if there is not enough stack, you can resolve it by a hardware upgrade. However, this alternative is not preferred when the goal is to keep the cost of production as low as possible.

The contribution of this dissertation is to provide a practical strategy through a static analysis tool in order to size the stack and guarantee stack safety in an interrupt-driven embedded software.

Most of the times, SW can be too slow, too buggy and too insecure [6]. For Matt Might, SW is so poorly constructed these days that we can't engineer software. He also asserts that the field of software engineering is more or less fraud. He explains himself with the cartoon present in Figure 2.1. Observing the cartoon people know that the Calvin's dad is wrong, that this is not how civil

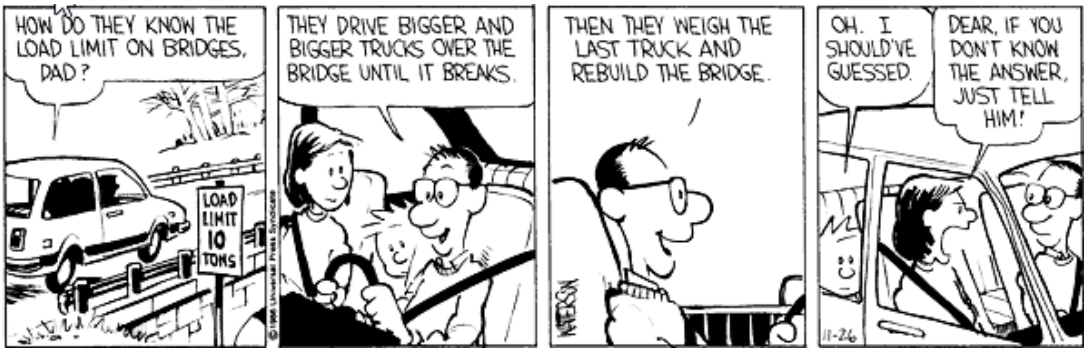

Figure 2.1: Dad teaches Calvin the wonders of science

engineers build bridges. Before starting constructing the bridge, civil engineers must analyze the scenario and study the dynamics and therefore deduct an accurate prediction model that describes the behaviour and gives the load limit of the bridge. Well, if Calvin would be asking about how software engineers know the limits of software instead of that of the bridge, the same answer of his father wouldn't be as wrong as it was in the bridge case.

So, when software engineers build SW, they submit SW to a lot of different test cases and try to get the SW to break. Other fields in engineering have prediction models for the behaviour of the system, for SW, however, it is not that linear.

In Sections 2.2.1 and 2.2.2 different approaches to achieve stack safety are described.

2.2.1 Testing Based Approach

Testing based approaches consist exactly of what is described above. Software engineers measure, during runtime or by simulating the application, generally through the watermark technique, the maximum amount of memory actually used from a specific test case. This watermark technique consists of the initialization of all the reserved memory for the stack with a unique value. Then, after running the application you can observe the values of the stack and realize which was the worst case of the stack usage for that case just by simply checking the area of the stack that remained unchanged and, therefore, with the unique value defined before runtime. Figure 2.2 illustrates the procedure for this technique.

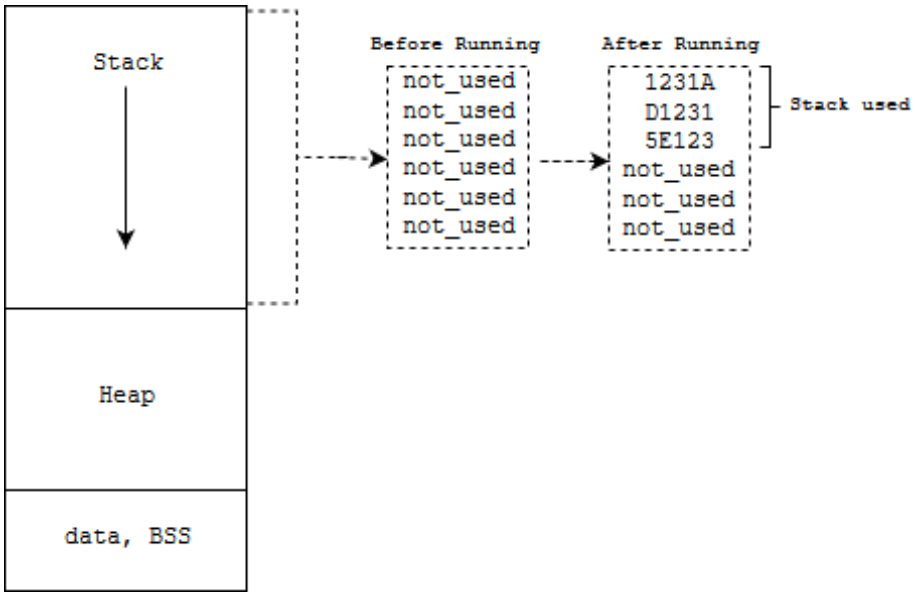

Figure 2.2: Watermark Technique

Another method to estimate the stack usage under a testing based approach is by using a debugger capable of measuring the stack in runtime. The developers can stop the execution of the application at key points of the source code and check the value of the stack usage.

2.2.1.1 The advantages and disadvantages of Testing Based Approach

Testing based approaches provide useful data for a specific execution case. However, this approach has several disadvantages comparing with a static analysis based approach. First of all, the test cases normally do not cover all the possible execution scenarios and therefore they do not

demonstrate the real worst case stack usage case. This weakness is stronger the more complex the SW is. In addition, for interrupt-driven embedded software, where the system's worst case stack usage scenario is very unique in time and depends highly on the input and the environment, it is extremely unlikely and challenging that the software is going to be submitted to a test that contains the worst case or simulates that scenario and produces that situation.

Without lowering the importance to put the SW under as many execution test possibilities as possible, the testing based approach carries significant costs and requires a lot of resources. Furthermore, the measurement during runtime may influence the behavior of the SW regarding time constrains and, therefore, it might create consequences in the case of a real-time system and mislead the observer. Besides that, sometimes it can be impossible to realize such tests due to lack of resources of the system target, which is actually often the case in typical embedded systems where they are used to have limited hardware [7].

So, the disadvantages of the testing based approach enumerated before restrict the test possibilities and the main problem of this approach is that often it will not include the WCSU scenario. As such, developers should not rely on these tests only in order to guarantee stack safety. Last but not least, the testing based approach treats the system as a black box, so it does not provide information to the developers of where they should focus to optimize the stack usage [5].

2.2.2 Static Analysis Based Approach

As explained above, it is very risky to bet on stack safety by a testing based approach since it does not guarantee the worst case stack usage scenario. Therefore, the software somewhere in the future may go according to Murphy's Law and lead to a stack overflow scenario, causing catastrophic failures in real-time systems and runtime errors that make the diagnose of the stack overflow very hard. Thus, the alternative is the static analysis based approach.

> *"Anything that can go wrong will go wrong."*
> *Murphy's Law*

In fact, for industries like automotive, where software is often used in hard real-time systems and safety-critical systems, there is an international standard for functional safety of electronic systems, ISO 26262 [1], which states the static analysis of the software as a prominent goal of the software design and implementation.

The static analysis based approach consists of computing valid information regarding the software stack usage without executing the application in opposition to a testing based approach.

The term of static analysis often includes the analysis performed by a software tool and manual review.

[1] The ISO 26262 is a derivative standard of IEC 61508

2.2.2.1 Manual Review

Manual source code review is a way to perform static analysis without using a tool. However, the review can be time consuming and the auditors must understand and be familiarized with the code to be aware of what type of vulnerabilities they are supposed to seek.

For a more efficient reviewing, it should be done at an early stage of the project, since the cost and risk of detecting and correcting security vulnerabilities and quality defects later in the software development process can be higher. Furthermore, if those vulnerabilities or bugs were to escape into the market and be experienced by customers, the consequences would be fatal for the project and damage reputations [8].

The reviewing process does not rely only on the source code but also on all the documentation, the software requirements and design, since errors can appear at any time of the software development process, from the definition of the architecture up to the testing stage. The reviewing does not exclude other stack safety approaches. At the implementation stage of the software development process, each developer should proceed with a self-review of its written code, where the developer could find out some vulnerabilities or bugs in the code he has written and correct it. In addition, a peer review can be done, where the developer asks a colleague to review its own written code. In the moment of merging branches of the project, a team review should also be done. Each developer should have perception and responsibility and try to seek for common possible bugs or vulnerabilities in the code for a win-win situation. Generally, there are guidelines to follow for the reviewing process which prevent the same defects to appear in further situations of the software development process (*e.g.* not reviewing the code right after it is written).

2.2.2.2 Using Tools for Static Analysis

The static analysis can be performed through the usage of tools. Those tools interpret the source code, or the object or the executable file and realize how the stack usage varies along the different paths of the CFG. Then, the stack usage can be combined with the CFG in order to compute the worst case stack usage. The estimated WCSU of a software can be expressed as:

$$WCSU \quad = \quad max\,(SU_{main}) + \sum_{i=0}^{n} max\,(SU_i) \tag{1}$$

where the **SU** stands for the stack usage of a function and **n** is the depth of calls from the main function.

Using tools for static analysis compared with reviewing is much more feasible. The more complex the software is and the software development process advances the more the manual reviewing can become a difficult and exhausting task. Generally, using tools for static analysis is much faster than a review and therefore, it can be used more often during the software development process. In addition, and in opposition to a manual review, which requires the auditor to understand the code and to have enough technical knowledge to track vulnerabilities, by using a tool it is not necessary that the auditor presents the same level of education and therefore can be done by a

wider variety of people. A static analyzer tool can highlight critical parts of the software that can be hard to identify in a review.

A good static analyzer tool should present clear results, this means that the results should be understandable and unbiased for the user, even if the user doesn't know much about common bugs and security vulnerabilities. Then, the unbiased interpretation of the results must lead to the mitigation of software defects and to a consequent boosting of the software quality.

The static analysis tool provides very important information to achieve stack safety although it does not treat the different parts of the code with different degrees of importance, since it does not know how the software works and which part or feature is more important. Hence, the output results of the tool must be studied in order to decided where to perform some optimization of the software or which parts do not need much concern because they do not represent much risk for the software.

Once in a while, those static analysis tools struggle with several challenges in interrupt-driven embedded software when not instructed to handle them. Some challenges faced by those tools can be loops in the CFG, calls with unknown stack usage or function pointer calls. Besides, generally a static analyzer tool does not have any information about the task model of the system and therefore, can not predict the usage for a preemptive scenario. Those challenges are going to be discussed and studied in section 3.5 of this dissertation.

In addition, it should also be said that a static analyzer tool can as well produce wrong output results due to the existent recursivity in the code. A static analyzer tool aiming to define an upper bound for the stack reporting wrong outputs can be very hazardous on safety-critical systems. The tool can not demonstrate results that show that there will be no possibility for stack overflow failures when in fact there may be. The tool should be aware of these cases and not report wrong information. Instead, it should say that it is not able to do its job properly. The same, although not so critical, in the opposite case. The tool should find the balance between being optimistic and pessimistic. A well-designed tool is the one that will never be so optimistic as to report false scenarios of stack safety, even though it may report false stack overflow scenarios [9].

Some compilers already have features capable of static stack analysis. For example, the GCC compilers have the "-fstack-usage" compilation parameter. This feature tells the compiler to output an ".su" file in the current directory. That file contain information about the stack usage per function. However, this feature does not provide information about the WCSU for the entire call chain of each function. The stack usage per function should be combined with the CFG information to determine the WCSU. Yet, for large and complex programs, the output becomes hard to analyze by a human. So, the output must be processed and formatted to be suitable for visualization. The visualization of the output can be done with tools like the **FreeMind**, **MindGenius**, **Conzilla**, etc.

2.2.2.3 The advantages and disadvantages of Static Analysis

Static analysis can only gain meaning if a study of the output results is made. Static analysis alone does not ensure that the software is failure proof. This characteristic can be considered a

disadvantage in many situations. On the other hand, if a good static analysis is performed there is no better way to guarantee stack safety.

Furthermore, static analysis has the advantage of reducing the amount of debugging and software tests. It is done without actually executing the software and thus, it cannot compete with the testing based approach regarding the accomplishment of the functional requirements of a project.

In interrupt-driven embedded software, generally the static analyzer tool does not identify the nature of the scheduling of the tasks. The tool does not know if the tasks are preemptive or non-preemptive, as well as the priority of each task. Therefore, the tool is unable to distinguish which task can be interrupt by whom. Surely, the tool can be instrumented to identify those aspects. However, there is a value for the stack usage associated with the task switch that will not be considered and so it will create an error.

In addition, complex software projects generally use third-party software where the source code is often not provided. Hence, the stack usage of such third-party software will not be considered and the accuracy of the results will be lower.

Figure 2.3 shows a brief comparison between the testing based approach (*e.g.* watermark) and the static analysis based approach (using a tool). The figure shows the reality of bounding the stack with a testing based approach vs a static analysis approach. As explained before, with the testing based approach it is very likely that the WCSU is going to be worse than it was expected. So, the developers are always insecure and questioning if they actually observed the WCSU scenario in tests. On the other hand, with a static analysis, the task of defining an upper bound for the stack can be more deterministic.

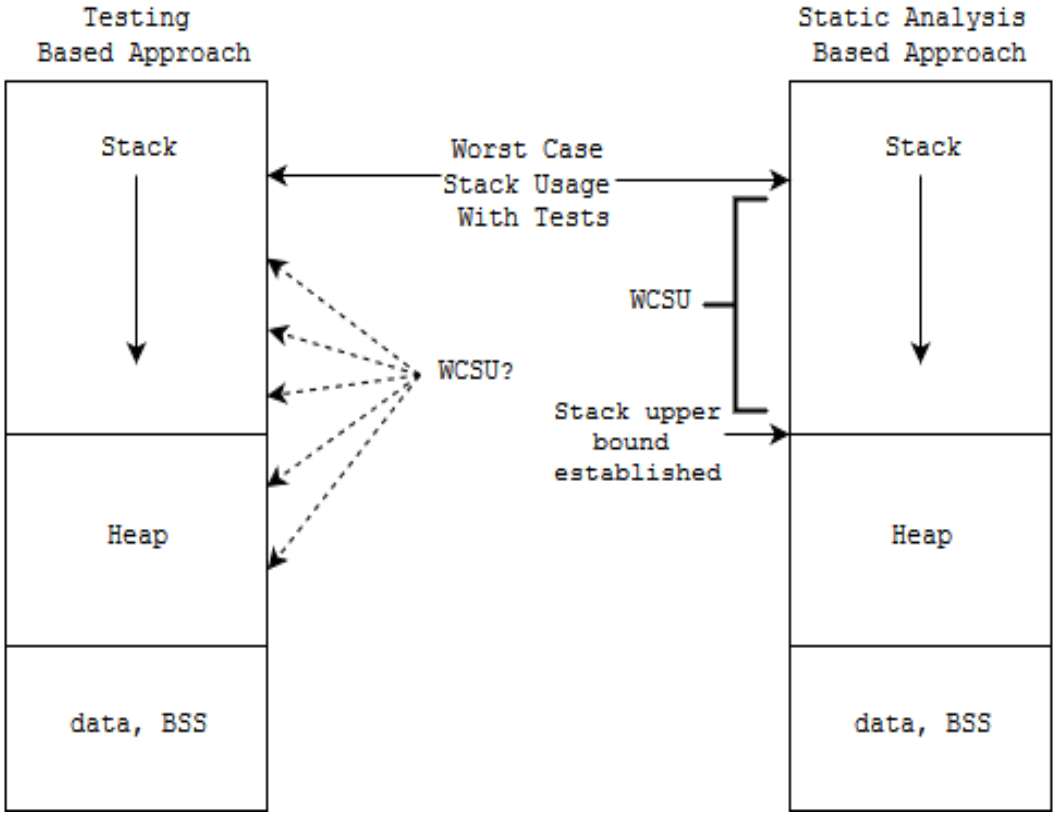

Figure 2.3: RAM layout showing a comparison between the WCSU by Testing vs WCSU by Static Analysis

Chapter 3

Static Stack Analyzer Tool

This chapter aims to present and explain the Static Stack Analyzer Tool developed throughout this dissertation.

3.1 Tool Description

The Static Stack Analyzer Tool was implemented to aid static stack analysis for embedded software applications written in C.

The Static Stack Analyzer (SSA) simplifies the work of developers. The tool allows them to quickly track a problem on the stack and identify its root. This is useful because developers can now quickly decide where to reduce the stack usage and mitigate the amount of runtime tests required (which take time and are expensive) to set a bound for the stack.

The SSA is a program that receives as an input a text file and gives as an output a graphical representation of the entire CFG regarding the application under analysis. The tool does not take into account the information regarding the task model of the system, and therefore the output file provides information about a non-preemptive scenario of the system. Despite that, for the developers familiarized with the software, they know where an interruption can occur and which tasks can preempt others. In this way SSA allows them to find the worst case stack usage scenario that can occur at runtime.

3.2 Requirements

Before defining the architecture of the software, it is important to define the software requirements for the final version of the SSA. Thereby, this section aims to enumerate and describe the defined requirements. The software requirements were established in order to attend the company's needs. The requirements were divided in two types: The non-functional requirements, and the functional requirements. They are enumerated in section 3.2.1 and 3.2.2 respectively.

3.2.1 Non-Functional Requirements

The non-functional requirements relate to the use of the application and not directly to its output results. These types of requirements usually relate to concepts such as security, availability, efficiency, performance, reliability and so on. Hence, the fulfillment of these requirements are going to be discussed further on section 5.1 of this dissertation based on the use of the SSA.

The Static Stack Analyzer was intended to meet the non-functional requirements identified in table 3.1.

Table 3.1: Static Stack Analyzer Tool Non-Functional Requirements

i	Name	Description	Priority
1	Cautious	The Tool should never underestimate the WCSU scenario	High
2	Deterministic	The Tool should be trustworthy in a way that does not provides incorrect information to the user and it must give the same result on successive trials for the same input	High
3	Accurate	The Tool should take the real worst case scenario into consideration. Thereby, the stack upper bound should be as small as possible aiming to save system resources usage	High
4	Usable	The Tool must give feedback information to the user while running, such as error and warning messages when something goes wrong as well as success information in case of success. In addition, the output must be generated in reasonable time (less than ten seconds)	High
5	Modular	The architecture should be modular. As an example, in case of necessity to rework the software for it to be compatible with a different compiler, that should not affect the parts of the software, regardless of the compiler	Medium

3.2.2 Functional Requirements

The functional requirements refer directly to the output results of the SSA. Hence, the fulfillment of these requirements is going to be discussed further on section 5.1 of this dissertation just by analyzing the output results.

The SSA was intended to meet the following functional requirements identified in table 3.2:

Table 3.2: Static Stack Analyzer Tool Functional Requirements

i	Name	Description	Priority
1	Folding	The tool should present the output where the CFG paths are hidden and allow the user to decide which branch of the CFG to open rather than presenting all the CFG paths. It should not present the same CFG path twice, instead it contains a link to the first occurrence of the same path in the software.	Medium
2	Statistical features	The tool should provide statistical information	Medium
3	Color Convention	The tool must identify different levels of severity for each problem on the CFG	High
4	Navigable	It must be easy to go directly to the origin of the problem that we are trying to analyze	Medium

3.3 Architecture

In order to start the implementation, an architecture was defined in a way to be suitable and capable to face problems found further on in the implementation part. The software architecture is going to be described in this section as well as the relations with other software.

The block diagram in figure 3.1 shows a high level view of the architecture. Yet, only the rectangles with a yellow background color relate directly to the SSA. However, the remaining rectangles are essential for the SSA to produce an output.

As we can observe, the flow is quite simple. The application under analysis comes from the developer's work. Then, it must be compiled with Green Hills compiler which outputs a text file besides the executable file. That file will serve as an input to the SSA tool. In turn, the SSA creates a MindMap file (XML format) which must be opened with FreeMind [10] to be visualized. If everything goes accordingly, the MindMap file must present the entire CFG of the source code along with additional information to perform the static analysis.

Regardless the SSA tool, Green Hills is the compiler used by the developers of the company to build software. The SSA takes advantage of an utility of the compiler to collect some data. The utility used is the gstack which analyzes statically an application and reports the maximum stack size that will be needed during execution. The gstack output also provides information about the CFG from different entry points of the source code. The gstack output is nothing else than a text file using a **.gse** file extension. The information reported by the gstack can be more than enough for the majority of the code. However, for large and complex software, the gstack output becomes unmanageable for human reading. In addition, the gstack may require the developer to

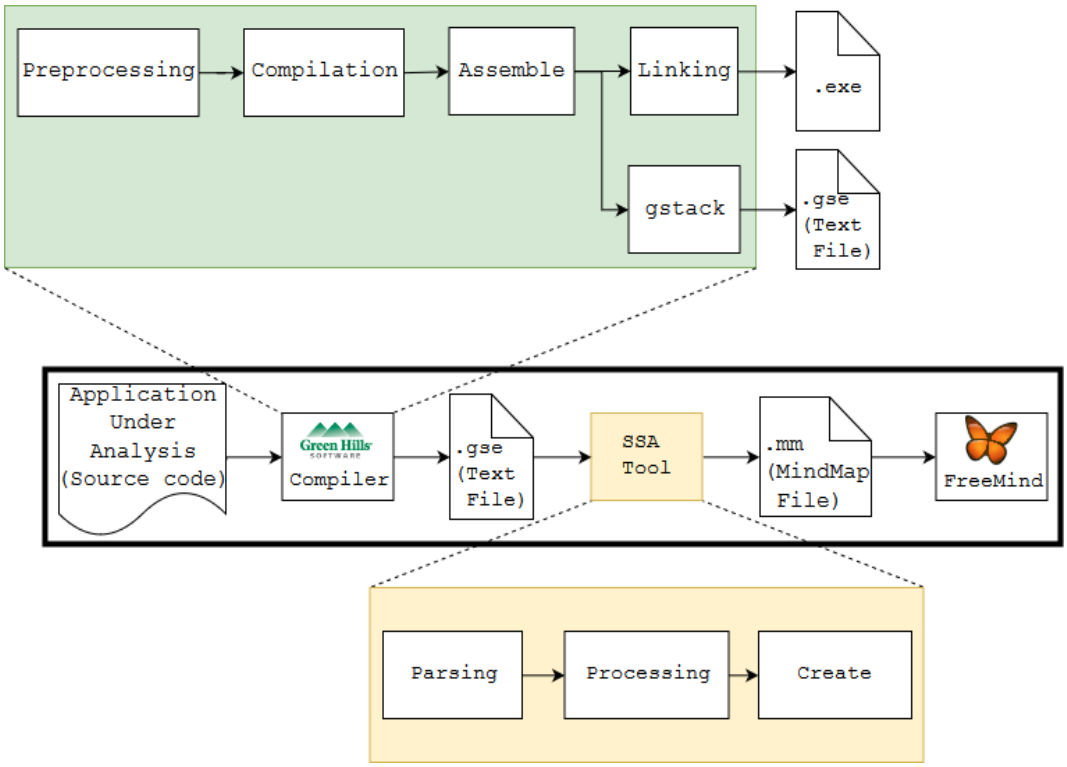

Figure 3.1: Software Architecture

provide additional information in order to produce valid results. More concretely, the developer may have to instrument the source code for the gstack to know when there is recursion or function pointers. In the same way, there is no big drawback for small software. However, considering the company's needs, this way to address the problem is not feasible. Imagine the following scenario. Let's simplify the problem and assume that there are no recursive calls in the source code, since it is not a good practice in programming embedded software. Consider a program that only contains seven function pointers. Well, in this case the developer just needs to add seven lines of code to instruct the program to be used with the gstack. So far so good. Now, consider that there are eighty function pointers in the program. The instrumentation of the code would be more exhaustive and slow but still possible. In the scenario of the company, there is an amount of more than nine hundred function pointers. Instrumenting the code in this situation would take a lot of time. In addition, cases where the project is developed under CI/CD (continuous integration/continuous delivery), where the software teams produce software in short cycles and the SW is conducive to changes of all kinds, the struggle would be bigger. For each build, there would be another tons of changes in the code just to prepare the program to be submitted to a static stack analysis. For the given reasons, performing a static stack analysis just by using the gstack would be impractical.

In order to better understand what the gstack does report and what it does not report, below are shown two parts of the output file generated by the gstack utility regarding the semaphore CFG model presented in figure 1.1.

Part 1:

```
------------------------------------------------------------------
|  Max Stack Usages For All Nodes                                |
------------------------------------------------------------------

Individual and total stack usages for all nodes
(values given in bytes):

Individual       Total      Function
----------       -----      --------
         3        8327      Semaphore_Init
         7         230      Input_Task
         4         150      Input_Pedestrians
       130         223      Input_Vehicles
      3223        8324      NN_Task
       850        1280      NN_GetData
      2867        5101      NN_Learn
 <unknown>   <unknown>      NN_Evaluate
       427        5384      SmartCorrection_Task
       388        4957      SmartCorrection_Get
       520         842      SmartCorrection_Verify
       128         128      SmartCorrection_Apply
        24          48      LightStatus_Task
         7          17      LightStatus_Green
         7          10      LightStatus_Set
         3           3      LightStatus_Update
         0           0      LightStatus_Clear
         7          17      LightStatus_Red
         7          17      LightStatus_Yellow
        14          24      LightStatus_Intermittent
```

Part 2:

```
------------------------------------------------------------------
|  Call Graph                                                    |
------------------------------------------------------------------

Call Graph Starting From Entry Points:

000 Semaphore_Init
001 . Input_Task
002 . . Input_Pedestrians
003 . . Input_Vehicles
004 . NN_Task
005 . . NN_GetData
006 . . NN_Learn
007 . . NN_Evaluate
008 . SmartCorrection_Task
009 . . SmartCorrection_Get
010 . . . SmartCorrection_Verify
011 . . . . SmartCorrection_Apply
012 . LightStatus_Task
013 . . LightStatus_Green
014 . . . LightStatus_Set
015 . . . . LightStatus_Update
```

```
016 . . . LightStatus_Clear
017 . . LightStatus_Red
018 . . . LightStatus_Set                          [RPT: 014]
019 . . . LightStatus_Clear                        [RPT: 016]
020 . . LightStatus_Yellow
021 . . . LightStatus_Set                          [RPT: 014]
022 . . . LightStatus_Clear                        [RPT: 016]
023 . . LightStatus_Intermittent
024 . . . LightStatus_Set                          [RPT: 014]
025 . . . LightStatus_Clear                        [RPT: 016]
```

The text file contains more information than the two parts above, but they were not presented because they are not used by SSA.

The file format is simple to understand. Part 1 of the file provides information regarding the stack usage for each function (column "Individual") as well as their maximum consumption throughout the call chain (column "Total"). Part 2 relates to CFG. The dots on the lines indicate the depth (later referred to as level) of the call in the CFG. The "[RPT]" tags on some lines indicate that the function is being called again. To minimize the file, gstack chooses that the call chains of those repeated entries are not displayed again.

Once the text file is generated by gstack, the SSA receives it as an input of the tool and collects information about the CFG and the individual stack usage per function. To do so, the SSA uses an algorithm (explained later on section 3.4.1) to parse the text file and ignore everything that will not be useful for the SSA. The lines of the text file that contain the stack usage per function (part 1) and the CFG (part 2) are the only useful data in the text file that the SSA needs.

The following is the processing block. In this part, the SSA will compute the maximum stack usage for each branch of the CFG and isolate the tasks in a strategic place to simplify the posterior analysis.

After the CFG is processed, the SSA starts to create the output file (MindMap file) with specific attributes and color conventions for each node. The attributes and the color conventions are described in further sections of the dissertation.

As soon as the MindMap file is created it must be opened in FreeMind in order to be visualized. The reasons for choosing FreeMind as a tool to show the results were several. FreeMind is a free software and an open source software, licensed under GNU GPL (General Public License). Basically, this means that anyone is free to use FreeMind for its own purpose without entailing any costs. Besides that, FreeMind is a light tool in terms of hardware requirements. Thus, it can be used in the majority of the current personal computers. Hence, the developers don't need a powerful machine to analyze the results. In addition, FreeMind is a very complete tool with all the features needed to fulfill the requirements of the SSA. Features such as the possibility to fold branches of the CFG, add attributes to the nodes, choose the color of the nodes and add graphical links between nodes.

The architecture was designed to be modular. This means that if in future works the necessity to change any block emerges, the other ones do not need to be modified. For example, if the

compiler changes, only the Parsing block will be modified but the processing and the create blocks will remain unchanged. The same for the other two blocks.

3.4 Implementation

The SSA Tool is a software written in C++ with about 1700 lines of code. This section will describe the implementation of the tool. Thereby, an individual description for the Parsing, the Processing and the Create block is going to be presented.

3.4.1 Parsing

The Parsing block is responsible for reading the gstack's output file and store all the essential information (part 1 and 2) for the SSA.

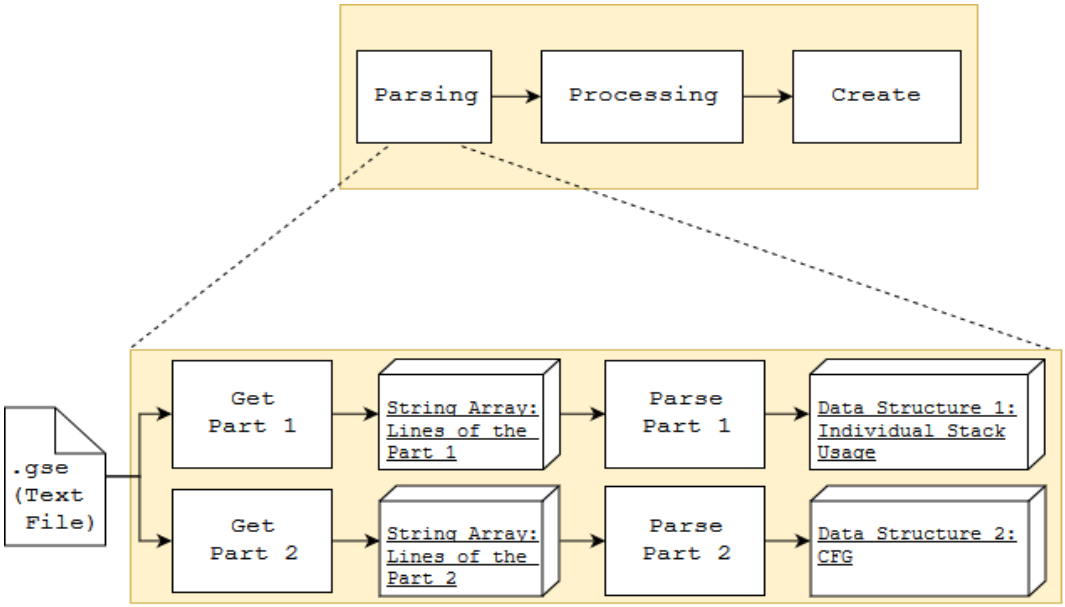

Figure 3.2: Parsing Block Diagram

As it can be observed in figure 3.2, the Parsing starts with two blocks. One block to get part 1 (stack usage per function) and another one to get part 2 (CFG) of the gstack's output file. Figure 3.3 illustrates through an UML (Unified Modeling Language) activity diagram how both blocks work. So, each function starts to open the gstack's output file and reads it line by line. Once it finds the beginning line of the respective part, it will store the lines until it finds the end of each part. Hence, part 1 and part 2 are stored in two arrays of strings. Henceforward, the SSA does not need the input file anymore. Then, the SSA should parse part 1 and 2 by line in order to delete useless information. The parsing of part 1 aims to attach a value (stack usage) to a function and therefore, a C++ data structure with two members (name and value) was used. Because the gstack

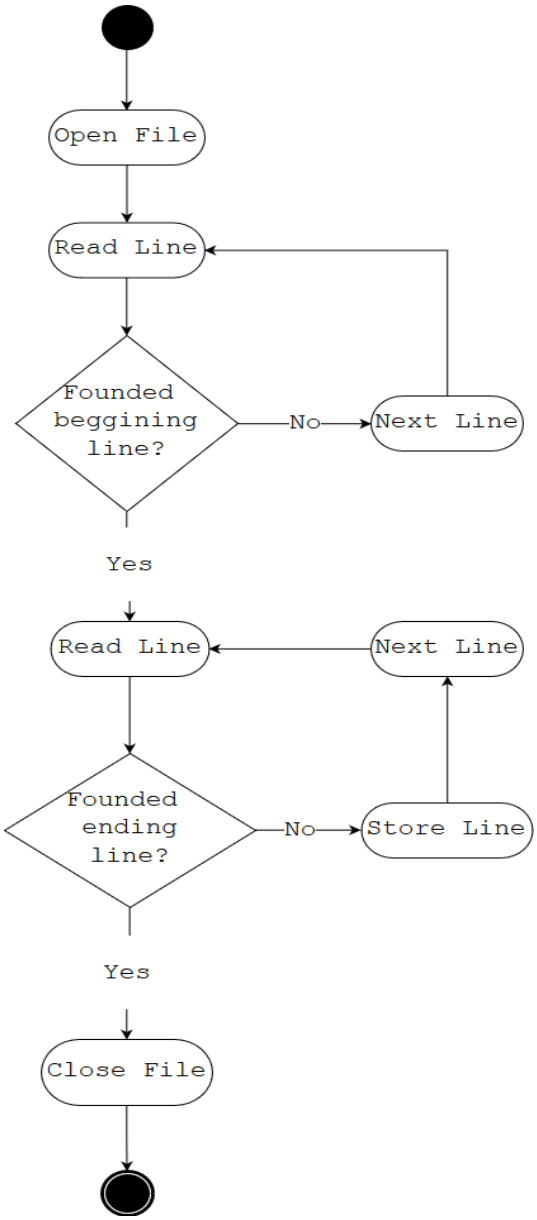

Figure 3.3: Get Part 1/2 of the Gstack's Output

output file has a solid format, it was observed for each line of part 1 that the individual usage was always between the first and the eleventh element of the line and the name of the function was after the twenty-eighth element. Hence, the parsing of part 1 consists in getting a sub string for those intervals in order to get the value and the name. Several were the reasons for the tool to ignore the information of the column "Total" on Part 1 of the gstack output. Regarding the company's software, the gstack output was reporting values on the "Total" column three times bigger than the total stack of the embedded system. Such values do not really help the developers to determine an upper bound for the stack. The origin of those values is inherent in calls of recursive nature, calls with unknown stack usage or the use of function pointers. The way the SSA handles those particular calls is described in section 3.5. For part 2, a different data structure was used intending to store the [RPT] (in case of existence), the level (depth on the call chain, which is the number of dots in the line), the line number and the name of the function for each line of the CFG.

The Parsing block returns two data structures. One containing the individual use of the stack by function and another with the information of the CFG. Next comes the processing block.

3.4.2 Processing

The processing block's goal is to prepare the parsed information and process it in a way to create the output MindMap file.

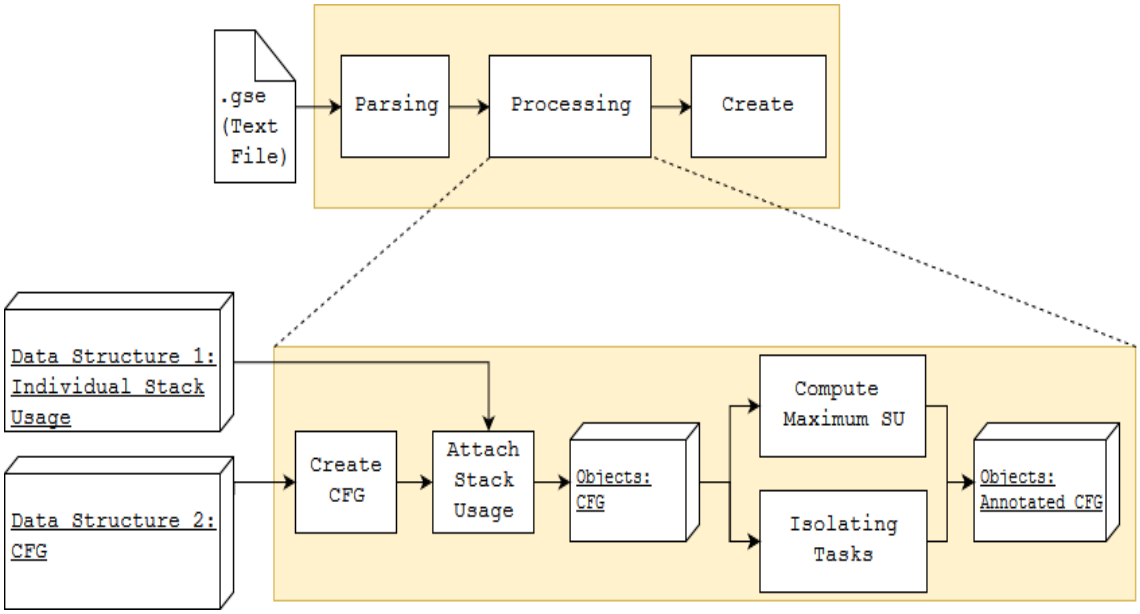

Figure 3.4: Processing Block Diagram

The processing block receives two data structures resulting from the parsing block as shown in figure 3.4. The processing block starts by copying the data structure with the CFG information into C++ objects in order to make it easy to process the data further and to separate what is parsing and what is processing. Hence, the CEntry class was created and for each entry point (nomenclature

used in the gstack output file to reference the functions as starting points for the analysis) on the CFG there will be an object from the class CEntry. Then, each entry is linked to the entries that belong to the same call chain through objects from the class CEntry_list. The way to know if an entry is called by another one is with the level (number of dots on the line from the gstack output) information collected on the parsing block. For example, if entry 'A' has 2 dots (level 2) and 'B' is the next one and it has 3 dots (level 3), that means the first one is calling the second one. Hence, entry 'A' will use the method AddEntry() in order to add entry 'B' to the call chain of entry 'A'.

Figure 3.5 shows an UML class diagram of the class CEntry and CEntry_list. As it can be seen in the figure, the CEntry class has several attributes. Some of those attributes (level, name, line) are defined based on the information collected by the Parsing block.

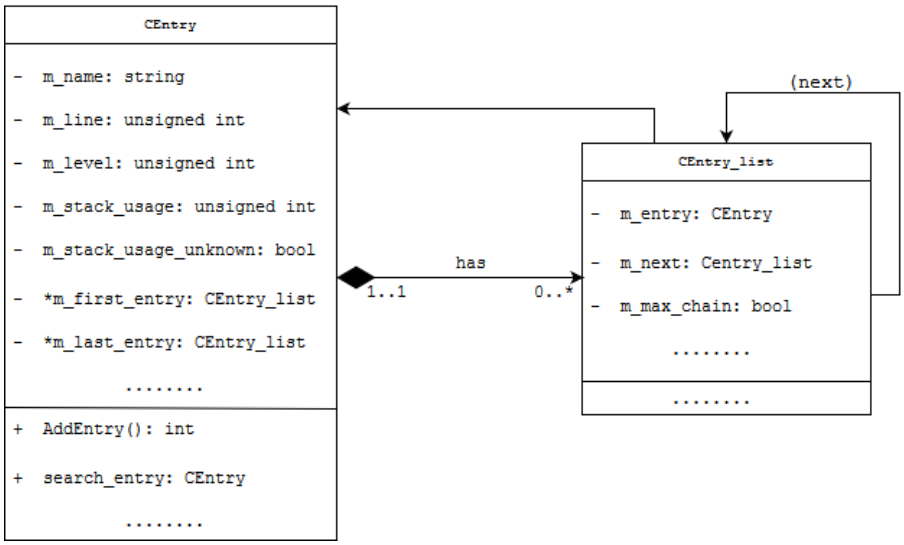

Figure 3.5: CFG Data Structure

To better understand how the entries are related with others, figure 3.6 presents an instance (zone bounded with the red line) of a particular point in time through an UML object diagram. Besides that, the figure shows the way the SSA manipulates the data collected from the gstack output for a portion of code like the one present in the figure. As we can observe, for each call on the code there is an entry on the CFG. That does not mean all the functions are going to be called, it is just a possibility. Thereby, the entry Semaphore_Init may call four other entries. The Semaphore_Init object has a pointer to the first and last list. The pointer for the first list was enough to do all the processing. However, when the SSA wants to add a new entry to a call chain, it is convenient to have a pointer to the last entry instead of going through all the lists and when it finds the last one add the entry. Besides, each entry points to the first and the last list, each list also points to the respective entry and to the next list on the CFG. This model allows the SSA to manipulate the data throughout the entire CFG.

Once the CFG information is stored in objects, the stack usage information collected on the parsing block must be inserted as well. To do that, for each element of Data_Structure_1, the

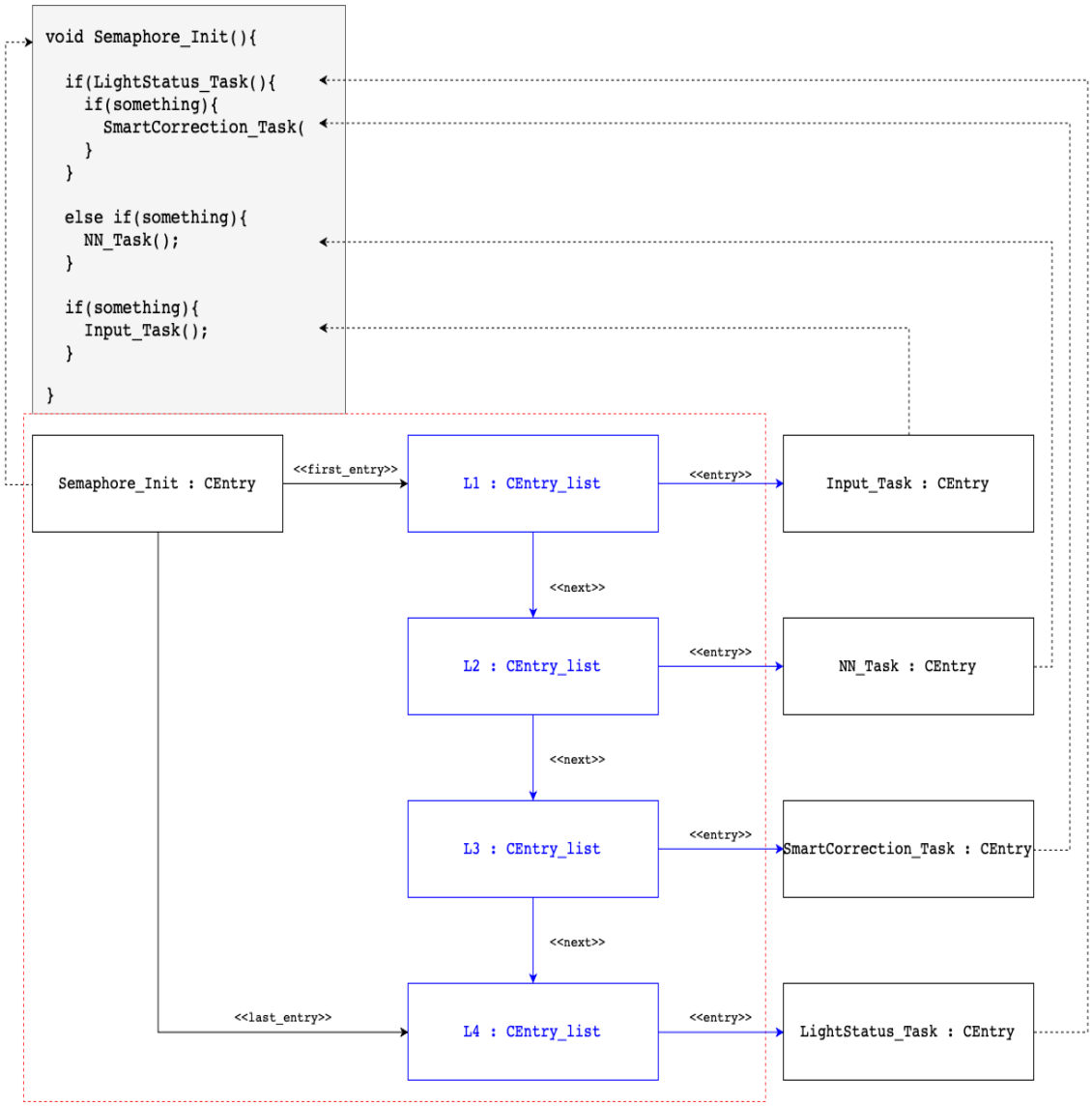

Figure 3.6: Instance of the CFG for the .gse File

object on the root of the CFG will use the method search_entry(). This method is a recursive search (it seeks for the wanted name/line) on the entire CFG of the entry and returns the CEntry desired object. In case of finding the entry on the CFG, the stack usage of the respective function is copied to the respective attribute of the object related to the stack usage.

Then, for each entry the WCSU will be computed based on the expression presented on 1. The algorithm to compute the maximum stack usage is described below with a pseudocode:

Algorithm 1 Compute Maximum Stack Usage (Simplified Version)

```
function Compute_Entry_Max_Stack_Usage (entry)
max ← 0
called_entry ← first_called_entry
while there are called entries do
    usage ← Compute_Entry_Max_Stack_Usage(called_entry)
    if usage > max then
        max ← usage
    end if
    called_entry ← next_called_entry
end while
return max + entry_stack_usage
```

The algorithm presented above is a simplified version of the one implemented on the SSA. This one works for a well behaved CFG. That is for cases where there are no recursive calls and no entry is repeated on the CFG. So, as we can conclude from the pseudocode, to compute the maximum stack usage for each entry the algorithm is recursive. Considering the CFG on 1.1, let the algorithm be explained. For each task of the semaphore system there are other called entries. Regarding the Input_Task, to compute the maximum stack usage the algorithm will compare between the Input_Pedestrians and the Input_Vehicles which consume more stack. Then, the maximum between both will be accumulated to the Input_Task. This procedure is done recursively for the entire CFG and therefore the maximum stack usage of the entire CFG computed.

Furthermore, aiming to have a better visualization of the CFG, the tasks were isolated, that is, they were copied to the root of the CFG. The purpose of isolating the tasks on the root is because normally the tasks are the code which consumes more stack and therefore the SSA user might want to analyze that part of the application. Generally, in large software the tasks can appear in a high level on the CFG. Then, to avoid the struggling of the user on finding the tasks, they are copied to the root of the CFG.

The processing block returns a set of objects with the CFG as well as information regarding the WCSU for each call chain. This is called the annotated CFG.

3.4.3 Create

Lastly, the Create block is responsible for creating the output MindMap file. Before starting to implement the Create block it was necessary to be familiarized with FreeMind and therefore

understand the syntax of a **.mm** extension file. Thereby, the "Hello World" MindMap represented on figure 3.7 was created manually through the FreeMind GUI (Graphical User Interface).

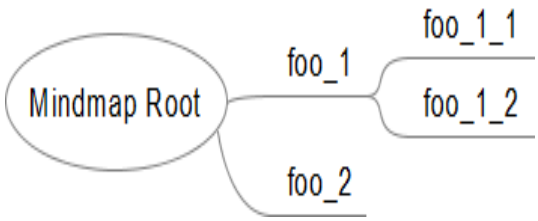

Figure 3.7: MindMap Hello World

Then, after saving the MindMap, the **.mm** file was opened in a text editor in order to understand the file format. The corresponding **.mm** file to the MindMap shown above is the following:

```
<map version="1.0.1">
<!-- To view this file, download free mind mapping software
    FreeMind from http://freemind.sourceforge.net -->
<node CREATED="1544088094153" ID="ID_1276422861" MODIFIED
    ="1544088216759" TEXT="Mindmap Root">
     <node CREATED="1544088100643" ID="ID_296576183" MODIFIED
         ="1544088117922" POSITION="right" TEXT="foo_1">
          <node CREATED="1544088141761" ID="ID_67485772" MODIFIED
              ="1544088155369" TEXT="foo_1_1"/>
          <node CREATED="1544088184405" ID="ID_1555033878"
              MODIFIED="1544088190264" TEXT="foo_1_2"/>
     </node>
     <node CREATED="1544088109965" ID="ID_1097349476" MODIFIED
         ="1544088113511" POSITION="right" TEXT="foo_2"/>
</node>
</map>
```

Deleting tag by tag, it was realized that several information were useless for our purpose. More specifically, the "CREATED" and the "MODIFIED" tags were not doing anything and would not be necessary. Since the **.mm** file for the company's application has more than one hundred thousand lines of text, the less pointless information it has the smoother will the visualization be. Thus, those tags were ignored because without them FreeMind was showing the same result. By using the GUI several features of FreeMind relevant for the purpose were discovered. Thereby, each feature considered relevant had been tested. Hence, through an observation of the .mm file it was easy to get familiarized with the syntax for each feature.

Once the syntax of the .mm file was known, the implementation of the Create block could be started. Figure 3.8 shows the flow of the Create block.

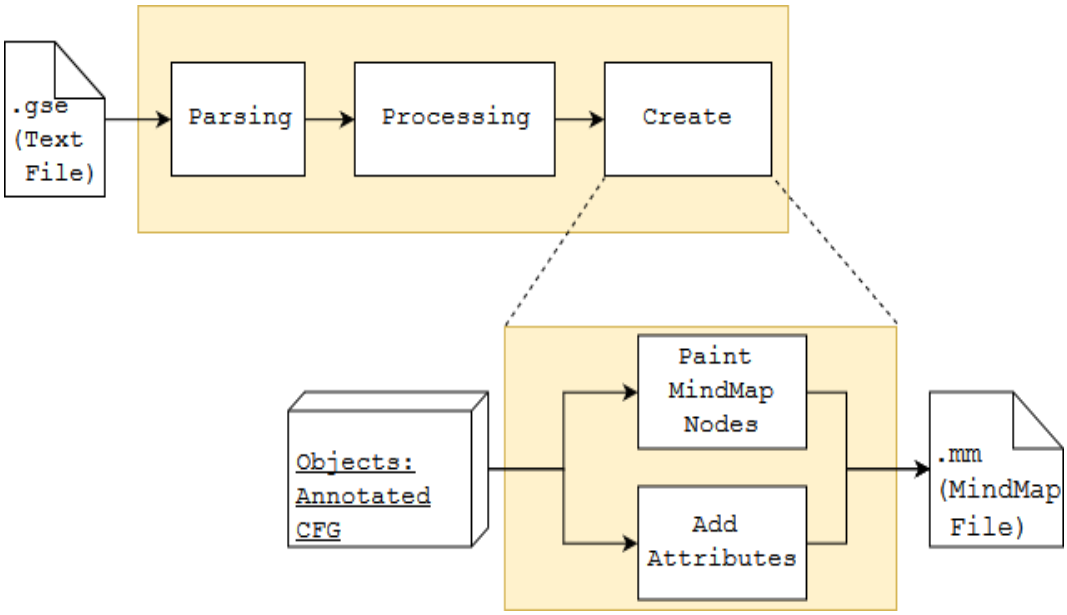

Figure 3.8: Create Block Diagram

First of all, the header of the file must be written on the output MindMap for FreeMind recognize it has a MindMap file. The header of the file is the following:

```
<map version="1.0.1">
<!-- To view this file, download free mind mapping software
    FreeMind from http://freemind.sourceforge.net -->
```

After the header, the body of the MindMap file containing the nodes of the CFG should be written on the output file. To do so, the Create block receives the annotated CFG from the Processing block. Then, for each object of the CFG recursively information similar to the following one will be printed:

```
<node BACKGROUND_COLOR="#ff6666" FOLDED="true" ID="LINE_0"
    POSITION="right" TEXT="Semaphore_Init [MAX: 6093 ] [USE: 3 ]
     ">
     <attribute NAME="Usage" VALUE="3"/>
     <attribute NAME="# Calls" VALUE="4"/>
     <attribute NAME="Max Use" VALUE="6093"/>
     <attribute NAME="Average" VALUE="1805"/>
     <attribute NAME="Std Dev" VALUE="2498"/>
     <attribute NAME="Acc Max" VALUE="7222"/>
     <attribute NAME="Dbg Line" VALUE="0"/>
     <attribute NAME="R_ID" VALUE=""/>
```

As we can see, for each entry on the CFG several attributes are added, as well as the respective color of the node. The criteria to choose the color for each node is explained on section 3.7 of this dissertation. The algorithm to write the entries on the .mm file is the following:

Algorithm 2 Write Nodes on the MindMap File

```
function Write_Entries()
called_entry ← first_called_entry
new node(Root_entry)
node.Print_Tags
node.Print_Attributes
if there are no sub nodes OR the entry is repeating then
    node.Close
    return 0
end if
while there are called entries do
    Write_Entries(called_entry)
    called_entry ← next_called_entry
end while
return 1
end function
```

From the pseudocode above, the SSA first starts to write on the output file a root for the MindMap. Then, it will recursively write the tags and the attributes for all the called entries present on the annotated CFG. The node's tags refer to the base information of each node, such as the background color, the position relatively to the MindMap root, the node's name ("TEXT") and so on.

Once all the nodes of the CFG are written on the output file the next step is to write the footer of the file to inform FreeMind where the end of the MindMap is. The footer of a MindMap file is shown below.

```
</map>
```

At this point, the SSA work is done. The last step is to open the file in FreeMind in order to visualize the entire CFG and take conclusions about the stack safety of the application in runtime. The MindMap file structure created by the SSA for the semaphore system can be seen in the appendix A of this dissertation.

3.5 Challenges

In fact, a static analyzer tool can tackle several drawbacks concerned by using a testing based approach. However, static analyzer tools aim at getting the maximum possible accurate results and thus they face a common set of challenges. In this section, we are going to describe the strategies adopted to handle those challenges for the specific company's needs through the SSA tool respecting the software requirements.

3.5.1 Loops in the Call Flow Graph

In the presence of loops in the CFG, a method to calculate the WCSU should be defined. Since if there isn't an iteration counter to halt the loop, the stack usage will be infinite and the SSA will not be able to handle those loops and produce valid results.

According to the nature of the embedded software, recursive calls are not recommended. However, if there is any possibility for recursive calls to exist, the code must be well instrumented so the bounds of the loops are known. Generally, developers use flags to avoid those scenarios. Then, supported by a source code observation, it was realized that for such loop scenarios reported on the gstack output, there was in fact a loop in the CFG. However, this would not happen when executing the application due to the resources having already been reserved before by another part of the code. This phenomenon is called circular dependencies and not loops or recursive calls. Hence, to tackle the presence of loops in the CFG, it was considered that the stack usage for the loops in the worst case is the consumption for one iteration.

Those loops in the CFG have an arrow link to represent the loop and to navigate to the beginning of the loop. They also have a "<RECURSIVE>" tag to attract the attention of the developers in the moment of the output analysis. In addition, each node of those calls has the attribute R_ID. This attribute pretends to differentiate loops that might exist on the entire CFG. Thereby, the user that is performing the analysis can distinguish multiple loops. Figure 3.9 shows an example with two loops in the CFG where the nodes from each loop have a different R_ID.

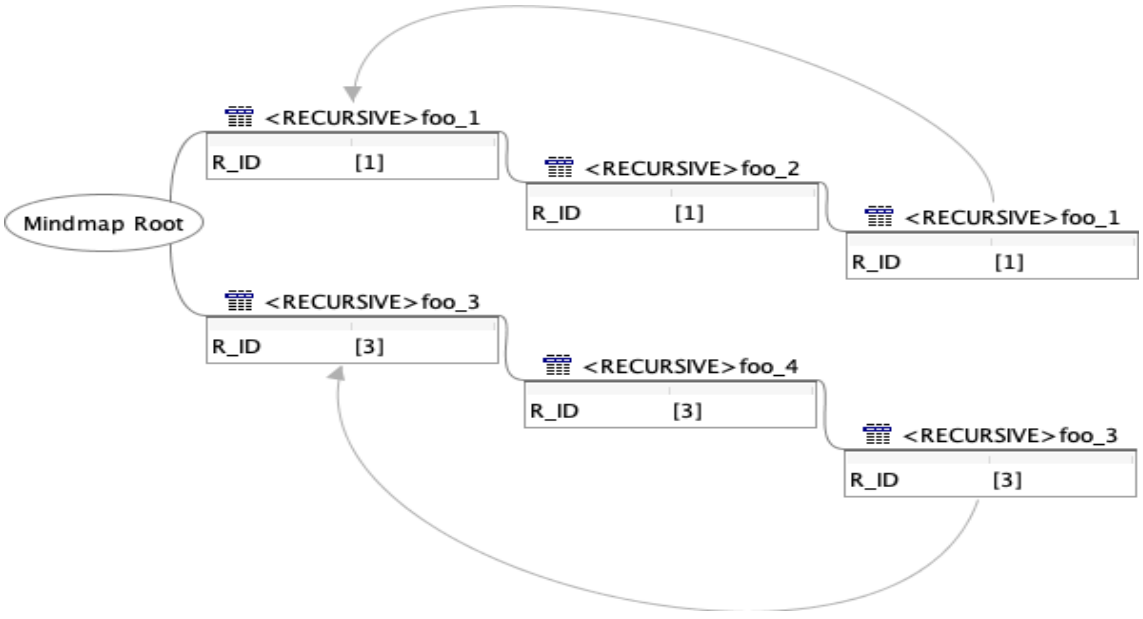

Figure 3.9: Loops in the CFG

The way the SSA identifies loops on the entire CFG is described on algorithm number 3. Basically, it will search recursively in a call chain for an entry with a specific name. The name to search is the name of the entry which the SSA tries to identify whether it is recursive or not. If it

Algorithm 3 Identify Loops

```
function Search_Entry_Ex(entry, searching_name)
resp ← NULL
if entry.name = searching_name then
   return entry
end if
called_entry ← first_called_entry
while there are called entries do
   resp ← Search_Entry_Ex(called_entry, searching_name)
   if NULL! = resp then
      break
   end if
   called_entry ← next_called_entry
end while
return resp
```

finds an entry with the specific name, which means that the call is recursive and returns that entry as the response (resp). In case of failure, the function returns NULL.

Accordingly, going back to the simplified algorithm to compute the maximum stack usage explained on number 1, it is possible now to explain the complete algorithm implemented on the SSA. Thus, algorithm number 4 represents the one implemented on the SSA. The small difference between both algorithms is, the latter accepts the presence of loops in the CFG. When a loop is found, the algorithm does not need to compute the maximum stack usage because it has already been done before. Thus, the algorithm only needs to get the value for the maximum stack usage already stored in the computer memory. Besides, algorithm 4 sets all the nodes on the loop as recursive entries.

In future versions, the SSA should ask the user for an iteration bound for those loops that may appear in CFG rather than assume one iteration for all the loops to compute the maximum stack usage. In addition, the tool could output a file listing all the loops existent in the software.

3.5.2 Calls with Unknown Stack Usage

Calls with unknown stack usage are functions from which the SSA didn't collect any information regarding the stack usage. These can occur when a third party software is used on the software and the source code is not available. Hence, they are treated like a black box where there is no information about stack usage.

It is a consequence that the level of uncertainty will increase in case these calls appear. However, the output can be analyzed to evaluate how critical it is for possible occurrences of stack overflow failures. In addition, a measure at runtime of that third party software can be done in order to get some idea of the stack usage.

Regarding the implementation of the SSA, calls with unknown stack usage information were not considered to compute the WCSU scenario, since there is not much usefulness to say that stack overflow will happen because there is a function with an unknown stack usage. This approach

Algorithm 4 Compute Maximum Stack Usage

```
function Compute_Entry_Max_Stack_Usage (entry)
max ← 0
called_entry ← first_called_entry
while there are called entries do
  if the entry is repeating then
    if Search_Entry_Ex(called_entry, called_entry.name) returns NULL then
      usage ← called_entry.Get_Max_Stack_Usage {value already computed before on the
      first appearance}
    else
      called_entry.Set_Entry_Recursive
      usage ← called_entry.Get_Stack_Usage
    end if
  else
    usage ← Compute_Entry_Max_Stack_Usage(called_entry)
  end if
  if usage > max then
    max ← usage
  end if
  called_entry ← next_called_entry
end while
return max + entry_stack_usage
```

to handle the unknown is for sure an optimistic way to pass away the challenge, yet that does not exclude a future analysis of the output by the developer and so those nodes in the CFG are coloured yellow.

In future versions, a feature should be implemented that asks what kind of approach the user desires. For example, the optimistic or the pessimistic, and the SSA should take a position according to the user's wishes and thus compute the WCSU without ignoring calls with unknown stack usage if the user wants a pessimistic approach. One of the possible approaches would be to estimate the stack usage of these calls using a debugger. The application under analysis should be stopped at a key point before the "unknown" function is called. Then the stack usage should be registered. Thus, it would be possible to calculate the difference in stack usage relative to the point at which the function is called. The stack usage of those "unknown" calls could be added to SSA. Thereby, the SSA would report more accurate results.

3.5.3 Function Pointer Calls

The existence of function pointers introduce difficulties in the calculation of the stack usage for the CFG paths in which they appear. In the company's scenario those function pointer calls exist as well as a strategy that must be established to face those cases. In CFG the function pointer calls were painted in blue following the color convention requirement.

Each time there is a function pointer call, the SSA reports and attaches to the branch of the CFG all the function pointer calls present in the entire software and assumes the one that consumes

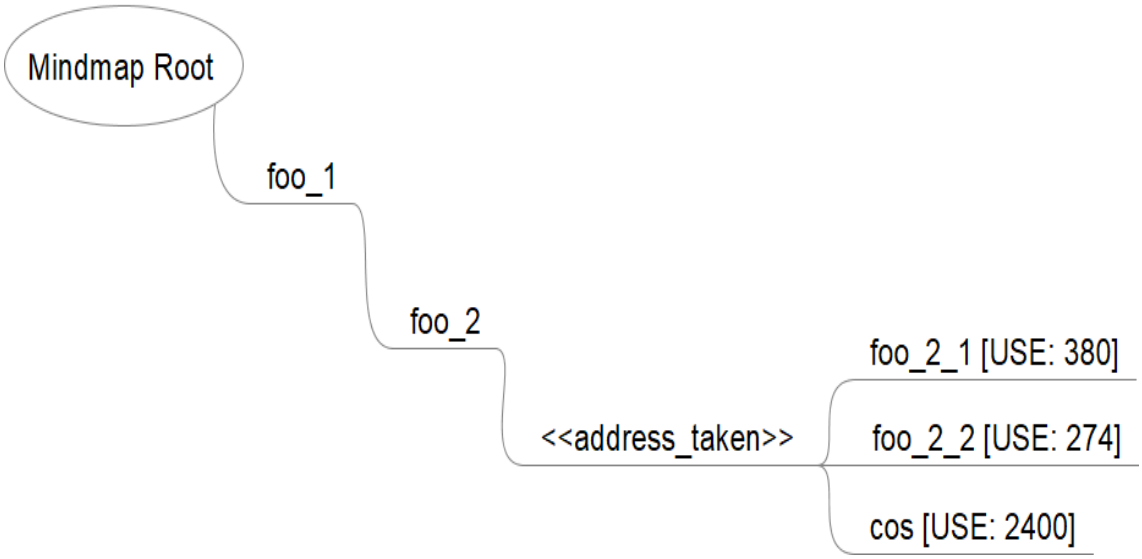

Figure 3.10: Function Pointer call in the CFG

more stack as the WCSU scenario. This approach is very pessimistic and sometimes does not make even sense, because in the source code there aren't as many calls as the CFG reports and so it does not consume that amount of stack.

Thus, after analyzing the source code of the software of the company, it was realized that the best strategy to adopt is to consider the call with the maximum stack usage and also with the same prefix of the name of the function which uses the function pointer. Let's explain this strategy with an example. Figure 3.10 represents a scenario which has a function pointer call in the CFG. Just by an observation of the source code, there is actually a function pointer call, although there is just one call while the CFG reports three possible calls at that point. This is because the entire software has three function pointers and the CFG was not capable of understanding which one was being called at that certain time. It was realized that for this case the functions that might be being called through a function pointer were the foo_2_1 or the foo_2_2 because it has the same function's name prefix "foo_2". That is due to the name convention respected by the company's developers. Hence, it does not make any sense to consider the WCSU of 2400 bytes (cos function) as figure 3.10 shows because that actually does not happen in runtime. Then, only 380 bytes should be considered. This is a huge interpretation error and therefore, besides the developer being able understand the context and figure out that the scenario is not real in runtime, the software provides some statistical information similar to the ones presented in figure 3.11 that can aid the analysis.

In future versions, the SSA should only present the function pointers with the same prefix of the function that is calling it and therefore, consider only the maximum of those calls and not all the function pointer calls in the software to calculate the WCSU.

<<address_taken>>	
Usage	0
# Calls	3
Max Use	2400
Average	1018
Std Dev	1198
Acc Max	3054
Dbg Line	2

Figure 3.11: MindMap node attributes

3.6 Performance and Validation

For the validation of the SSA, the V-model development process was adopted. The V-model derives from the waterfall model but, instead of progressing down in a linear way like a waterfall, for each implementation phase there is an associated validation phase. In figure 3.12 we can observe the V-model and the phases of the project development in function of the completeness of the project with time and level of complexity.

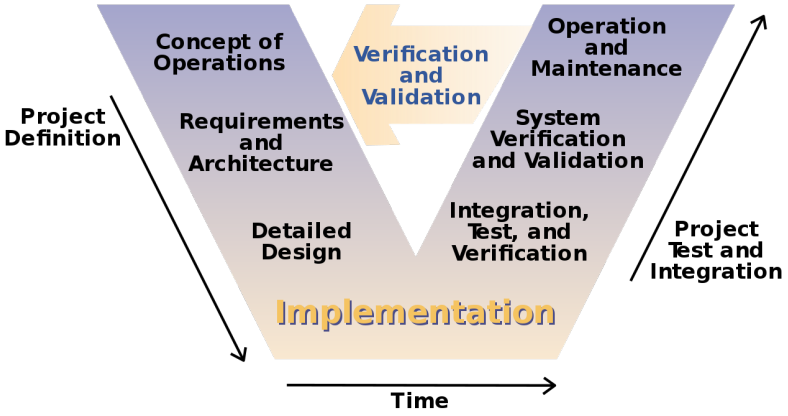

Figure 3.12: V-model of the System's Engineering Process [1]

Within validation there are typical phases according to the V-model. The validation phase should include Unit Tests, Integration Tests, System Tests and User Acceptance Tests. Those test names may vary among different bibliographic references.

As far as Unit Tests are concerned, they are designed to mitigate bugs for small portions of code, such as a feature or function which works independently of other parts of the software. The Unit Tests certify that those small portions of code work correctly when isolated from the rest of the software.

Integration Tests guarantee that Unit Tests are in accordance among themselves and so, when the portions of the code are tested along with others they function correctly. Regarding the implementation of the SSA, whenever a new feature was implemented the SSA was subjected to

the whole set of tests aiming to find undetected bugs in the software. Hence, as new features or functions were added, unique .gse files were created to test if the new code worked among the remaining already implemented code. Whenever a new code was implemented, the SSA was subjected to the complete set of tests used before for other functions. This ensured that the different parts worked independently and between them.

System Tests, unlike Unit and Integration Tests, also include the client's business team. System Tests verify that functional and non-functional requirements have been fulfilled, as well as whether the software performs acceptably or not.

User Acceptance Tests are performed in a real environment with realistic data. These tests verify that delivered software meets the client's needs and if it is capable of being used in real time.

More Specifically on the SSA implementation, the validation phase was done mostly through Integration Tests alongside with the debugging and analysis of the output to verify if the software functioned as expected. Moreover, a function was implemented that prints out in the console the information collected from the .gse file in order to check if the data collected is valid. Besides, when analysing the company's application, the stack usage values reported by the SSA were compared with real values measured during runtime (watermark technique) to verify the veracity of the results. This comparison was done for three different builds of the application under analysis. In general, the values reported by the SSA where higher than the measured ones since the SSA take into account the WCSU scenario, which does not always occur at runtime. Table 3.3 shows a comparison between the values given by the tool and the measured ones for each of the three builds for the core 1 and core 2 of the microprocessor CPU.

Table 3.3: SSA vs Measured Values (in bytes)

Build	**Measured**		**SSA**		**Error (%)**	
	Core 1	**Core 2**	**Core 1**	**Core 2**	**Core 1**	**Core 2**
SW_1	5509	34068	9738	34540	43	1.4
SW_2	6188	38285	9639	38421	36	0.4
SW_3	6562	38285	9639	38421	32	0.4

As it can be seen, regarding the core 2, the SSA reported very close results. On the core 1, the error is bigger because the WCSU scenario did not happened at runtime. However, the error can work as a confidence interval. Due to confidentiality terms, the values on the table were normalized and therefore, do not represent real values.

To measure the performance of the SSA tool the execution time only was considered. The software was instrumented to display the execution time and was executed 10 times with an input comprising a total of 15,471 calls. The average execution time for the 10 attempts was of 5,849 seconds with a computer equipped with an Intel Core i5 4300U CPU. The same test was done with a more powerful CPU, an Intel Xeon E5-1620 v4. In the latter, an average time of 3.542 seconds was recorded. In fact, the SSA software was not programmed and designed to take advantage of

multi core processors and therefore, does not exist much difference between both tests. Furthermore, the performance of the tool is not really relevant to its purpose of use. Since the developers will only use the tool once in a while, taking 5 seconds or 20 seconds is not very significant.

3.7 Using the Static Stack Analyzer Tool

This section discusses how to use the SSA tool. Before running the software, it is required to prepare the input file. This preparation relies on a utility of the compiler which reads the source code for all the files of the software and returns a text file that contains the CFG represented in a solid text structure. However, for large and complex software it is quite hard to interpret the CFG generated in that way by the compiler utility. As such, the SSA gives a huge contribution for a better understanding of the CFG by representing it graphically.

Once the input file is created, the user, in case of using windows, should open the command prompt at the directory of the software. Next, the user should execute the application and insert the program input parameters, them being the name of the input file and the desired name for the output file. The SSA has also a '–help' functionality which shows a usage example of the tool. Below this procedure with a windows command prompt is presented.

```
C:\CarlosGarcia\StaticStackAnalyser\Release>StaticStackAnalyser.exe --help

Welcome to Static_Stack_Analyser Tool  Ver(1.0.1.0)
--------------------Help!-------------------------
Usage Example:
        SSA.exe <InputFile.gse> <OutputFile.mm>

-------------------------------------------------
```

If everything is in conformity with the tool, the console may show something similar to the following example:

```
C:\CarlosGarcia\StaticStackAnalyser\Release>StaticStackAnalyser.exe
..\TestSamples\semaphore.gse semaphoreMindMap.mm

-Welcome to Static_Stack_Analyser Tool  Ver(1.0.1.0)-

[Information] Input file: semaphore.gse
[Input File] (Reading Stack Usage per function)

[Input File] (Reading Call Graph)
The beginning of the call graph was found! (line 37)
The End of the call graph was found! (line 62)

[Information] Number of Entry Points: 20

[Prepare] (Inserting the Call Graph)
[Prepare] (Inserting the Stack Usage per function)

[Processing] (Computing Max)

[Information] Max Stack Used -> 6093 bytes
```

```
[Processing] (Isolating Tasks)

[Output File] (Creating MindMap file)

[Information] The output MindMap file was saved as semaphoreMindMap.mm
[Information] Done.
```

Once the output MindMap file has been created, it should be opened in **FreeMind** in order to be displayed graphically.

Figure 3.13 shows the MindMap file generated by the SSA for the semaphore system. The following list provides a detailed legend and description of figure 3.13:

- The "[MAX: x]" represents the WCSU in bytes of the entire chain starting from each node.
- The "[USE: x]" represents the stack usage in bytes of the function itself.
- The red colored node point the WCSU path for each branch on the CFG.
- The yellow colored node is to signalize an unknown stack usage for the function itself.
- The orange colored node means that the node has a stack usage greater than fifty percent of the red node stack usage of the same group of calls (Functions called by the same function).
- The grey colored node represents a no need for alert due to the stack usage of the function and the WCSU from there equals zero.
- The warning icon attached on the NN_Learn function (for example) is to inform that the function itself is the major responsible for the node to belong to the WCSU path since only the stack usage of the function is greater than fifty percent of the max stack usage of the call chain.
- The arrow link is to simplify and minimize the CFG. Furthermore, it points to the first appearance of the function in the CFG.
- Each node has similar statistics to the ones presented in figure 3.11.

Although it is not shown in figure 3.7, the SSA also provides the following additional information:

- All existent functions in the source code under analysis which are not called, the SSA attaches them to a node named "Not Called" on the left side of the CFG root.
- The SSA recognizes loops in the CFG and marks them with a tag. See 3.9.
- In case there is a function pointer call the node is colored in blue.

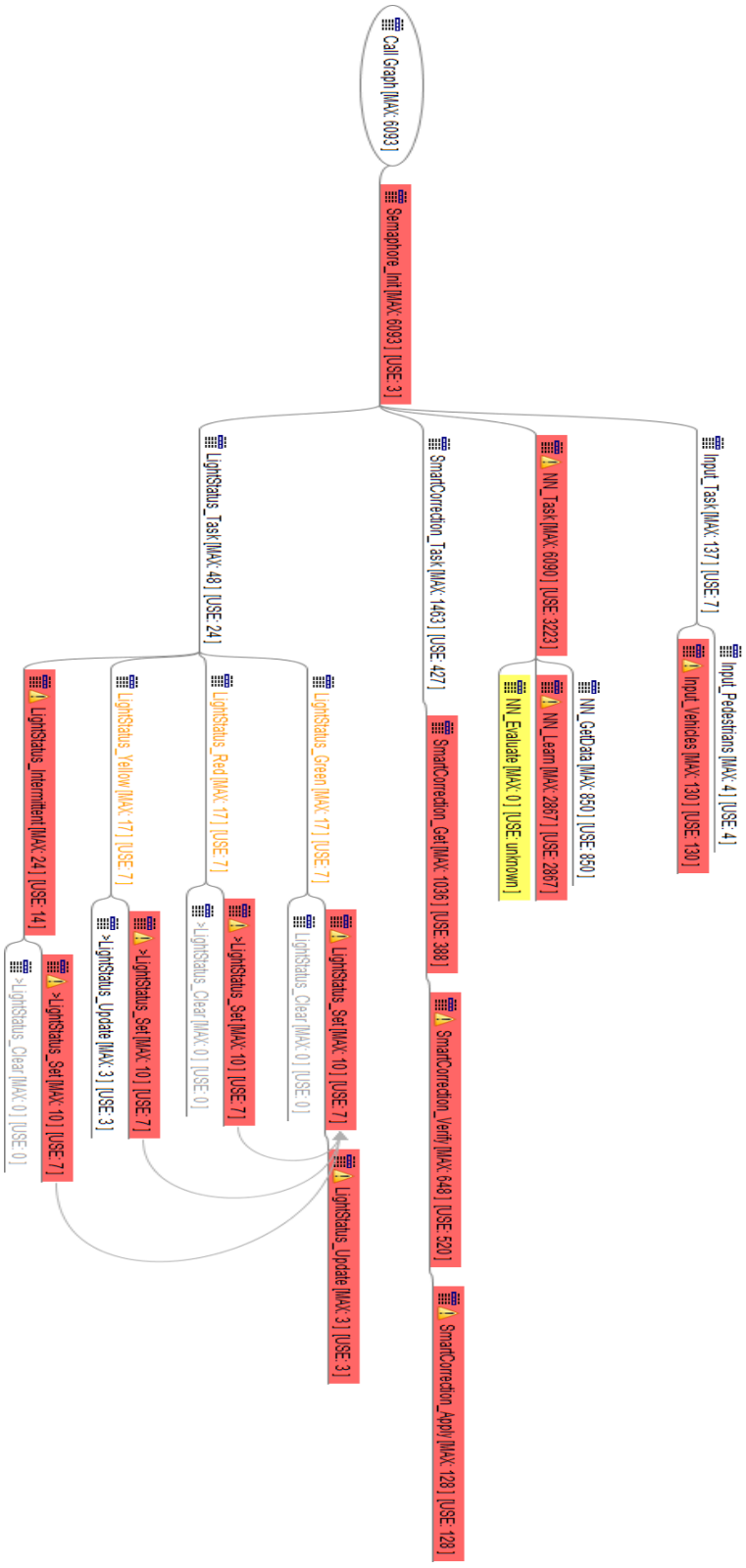

Figure 3.13: Semaphore System MindMap

Chapter 4

Analysis and Results

The SSA was used in the industrial environment to aid the developers of the company in finding a solution for the over usage of the stack. Without the SSA, the developers would be struggling seeking for parts of the code that needed optimization and reduce the amount of stack usage. In the company's case, where the analyzed software is quite large and complex, the gstack output file is almost impossible to be analyzed in useful time by a human. That was the main practical contribution of the SSA for the company's project.

In this chapter we are going to enumerate some possible strategies to minimize the stack usage for worst case scenario.

4.1 Solutions to Stay Within the Stack Limits

First of all, let's remind the semaphore system idealized on section 1.1 of this dissertation. Regarding the semaphore system, consider the following task model represented on table 4.1.

Table 4.1: Semaphore System Tasks

i	Task Name	Priority	Period (ms)	Stack Usage (bytes)
1	LightStatus_Task	1	100	48
2	Input_Task	2	200	137
3	SmartCorrection_Task	3	500	1463
4	NN_Task	4	(Idle)	6090

In typical real-time systems with preemptive scheduling the tasks compete between them to be executed on the processor. Generally, this competition is priority-driven and therefore, any time a lower priority task is being executed and a higher priority task becomes ready, the lower priority task is preempted. Although, before that the lower priority task context is saved by the RTOS and the higher priority task context is restored. When the higher priority task finishes, the other one continues [11]. From the table, the LightStatus_Task is the task with the highest priority and therefore, it can interrupt any task. Besides, it is the lightest task in terms of stack usage. On the other hand the NN_Task is the task with the lowest priority and the heaviest one. As far as the

time period of each task is concerned, the logic applied was, the higher the priority of the task, the shorter the period of it. Regarding the stack usage column, it corresponds to the maximum stack usage which was withdrawn from the SSA output results presented on figure 3.13. The stack is used not only by executing the tasks to store local variables or return addresses, but also to save tasks context when switching among them. Furthermore, the interrupt handler has its own memory allocation on the stack. Any time a lower priority task is interrupted by a higher priority task the stack accumulates the consumption of both tasks. Then, the WCSU will be higher than in cases where there is no preemption.

In order to simplify the problem, let's assume that the interrupted service and the context switch do not consume any stack. For the semaphore system the WCSU would be the sum of all tasks stack usage which results in a total of 7738 bytes. Supposing that the system has only 8 KB (8192 bytes, for base two), only 454 bytes remain available on the stack. Figure 4.1 shows how the SSA output report this optimistic value for the WCSU. As it can be observed, the Semaphore_Init node on the MindMap has an attribute called "Acc Max". That attribute stands for the sum of all the adjacent nodes. This number represents the case when all the tasks of the system are interrupted by another with a higher priority (except the task with priority 1) and therefore, the stack usage is the highest possible.

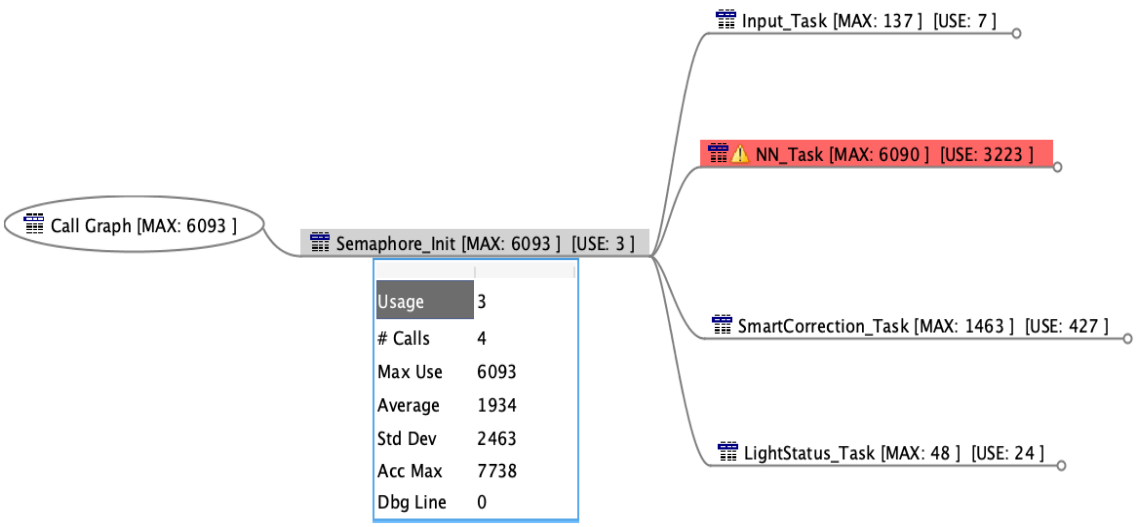

Figure 4.1: Semaphore Worst Case Stack Usage

Some way out should be discussed in order to ensure that the stack stays within the limits. The more common found solutions where the following ones:

- Using Non-Preemption Groups - This solution is basically limiting which tasks can interrupt each other. A non-preemption group is a set of tasks that cannot interrupt each other.

To do so, for each task two static priorities are associated. The base priority and the dispatch priority. The first is the priority of a ready task which competes with others to be executed on the processor. As far as the second one is concerned, at runtime, when a task begins to run, the active priority is set as its dispatch priority. Hence, the running task can only be interrupted by tasks with a higher base priority than their dispatch priority. This mechanism could prevent a group of heavy tasks preempting each other and therefore, the WCSU scenario would be less dangerous for the system. To better explain how does the non-preemption groups solution works, consider the table 4.2:

Table 4.2: Semaphore System Tasks With Non-Preemptive Groups

i	Name	Base Priority	Dispatch Priority
1	LightStatus_Task	1	1
2	Input_Task	2	2
3	SmartCorrection_Task	3	3
4	NN_Task	4	3

Accordingly with the defined dispatch priority for each task, is possible to deduce the following. At runtime, if the NN_Task is being executed, its priority changes from four to three and therefore, it can not be preempted by the SmartCorrection_Task. Since those tasks are the heaviest ones, this technique will minimize the WCSU significantly. Basically, this workaround swaps stack by runtime.

- Join Tasks - Joining tasks is basically what the name means. In fact, this is a variant of the first solution presented. Although, it is much simple to implement. Joining two heavy tasks can mitigate the WCSU. Considering the semaphore system, joining the SmartCorrection_Task with the NN_Task, without transposing time constraints characteristic of a real-time system, the stack would decrease in 1463 bytes for the WCSU. In order to join those tasks, the source code from both should be merged. Consider a first stage, where the source code is similar to the following:

```
void NN_Task() {
    NN_GetData();
    if(data has been collected) {
        NN_Learn();
    }
    if(data has been learned) {
        NN_Evaluate();
    }
}

void SmartCorrection_Task() {
        SmartCorrection_Get();
```

```
}
```

For this model, where the NN_Task is the task with the lower priority among all the tasks of the system, it is plausible to say that, in a certain point at runtime, the NN_Task will be preempted by the SmartCorrection_Task. This event will pile the stack memory. For the worst case scenario, the preemption occurs when the NN_Task is using the maximum stack usage (6090 bytes). Hence, is very likely that the stack use of the system will reach at least 7553 bytes (NN_Task + SmartCorrection_Task). This solution can attenuate this value only by preventing the two tasks from preempt each other and so, the stack does not accumulate. In this way, a total of 1463 bytes (SmartCorrection_Task stack usage) can be saved. In order to join both tasks, the code should be modified to something similar to the following:

```
void Intelligent_Task() {
    NN_GetData();
    if(data has been collected) {
        NN_Learn();
    }
    if(data has been learned) {
        NN_Evaluate();
    }

    SmartCorrection_Get();
}
```

- Using Static Variables - By using static variable allocation, the compiler will reserve memory on the RAM, but not on the stack segment. When there is more available space on RAM than on the stack, this simple strategy can be adopted to minimize the stack usage. If the variable is initialized, it is allocated in the data segment. In case of the variable not being initialized or having value zero, the memory allocation refers to the .BSS segment. With regard to programming, this solution is quite basic to implement. Lets demonstrate with an example. Consider the following code chunk in C language:

```
#include <stdio.h>

void foo() {
        int x = 0;
        x++;
        printf("%d\n", x);
}

int main() {
        foo();
```

```
        return 0;
}
```

The variable **x** in this example is stored in the stack segment of the RAM memory. Although, if the same variable is declared with the keyword **static** before the type (**int** for this example), the compiler will allocate the variable in the data segment of the RAM. Thereby, using the static keyword will mitigate the stack consumption. The code chunk below shows the way of using the static keyword in C.

```
#include <stdio.h>

void foo() {
        static int x = 0;
        x++;
        printf("%d\n", x);
}

int main() {
        foo();
        return 0;
}
```

4.2 Advantages of Using the Tool

Using the SSA tool can be very beneficial during the development of large and complex embedded software applications. Regarding the company's software, which has over fifteen thousand calls, it can be very challenging to find a solution to stay within the stack limits. The trump of the SSA is the clear way it shows the entire CFG of the software. Thus, it helps and simplifies the developer's work aiming to choose a way out to ensure that the stack stays within the limits. Even for small changes in the software, the developers can analyse and predict the consequences at runtime.

Chapter 5

Conclusion

In this chapter the conclusions of this dissertation will be stated.

5.1 Fulfillment of the Goals

The SSA was created in order to attend the company's needs, as well as to be suitable and useful in order to make the project more trustworthy. The SSA met the company's needs. The tool was used in the company's project aiming to minimize the percentage of stack usage. The SSA supported the developers to find parts of the company's software responsible for the high usage of stack and therefore, a solution was established faster than if they hadn't had a tool like the SSA. As such, it can be considered that the SSA was validated with real data in a real environment.

Regarding the fulfillment of the SSA software requirements defined before the implementation phase of the project, the majority of them were achieved. However, for the current version of the tool, it is not ensured that the results reported by the SSA are not underestimated. For cases in which the application under analysis uses third party software, and where the source code is not provided, it is not possible to guarantee that the results are not being underestimated.

In addition, thanks to the SSA, a developer can give a fast explanation to a new contributor on the project of how the software does behave due to the friendly graphical representation of the SSA output.

Considering the tests done to the SSA software, it's reasonable to say the tool is robust and does not have any known bug.

5.2 Improvements for the SSA

The SSA was useful for the company. However, there are some aspects which can be improved.

As explained, the tool reports information for a non-preemptive scenario. Thereby, one possible improvement for the tool could be receiving information regarding the task model of the system aiming to estimate the WCSU for a preemptive scenario.

For the particular company's needs, it was considered by the SSA that loops which might appear on the CFG only consume stack equivalent to one iteration. For future versions of the SSA, the tool could ask the user, when it finds a loop, how many iterations the SSA should consider in order to compute the stack usage.

At the moment, the SSA depends on the gstack utility. Furthermore, a feature which interprets the source code and gets the CFG as well as the stack usage per function could be implemented on the SSA. This way, the SSA could be used for a wide variety of projects independently of the compiler used in each project. Although with less impact, the same could be applied to FreeMind. A tool to visualize the output could be created in substitution of FreeMind and therefore, the SSA would be totally independent. Considering that this improvements actually happen in the future, small rearrangements would need to be done to the parsing block and the create block but there would not be any changes needed regarding the processing block.

The current version of the SSA is actually very optimistic. The tool just ignores when a function on the source code has unknown stack usage. Obviously, there is not much to do when those unknown values come from third party software. However, it would be more correct if the SSA would report the entire chain as unknown rather than only the function marked unknown. Hence, the SSA could ask if the user desires an optimistic or pessimistic computation of the WCSU. This functionality is quite simple to implement although it was not considered during the implementation phase because the company uses third party software and therefore the SSA would report always useless outputs.

Regarding the challenge faced by the existence of function pointer calls on the code, the output of the SSA could be improved in order to be more suitable for the company. Hence, the SSA should only present the function pointers with the same prefix of the function that is calling it and therefore, minimize the size of the output file and avoid showing pointless information.

Aiming to aid the analysis and finding a solution for a possible problem by the user, the SSA could include information regarding the runtime for each function in the output file. This information would help the developer to evaluate the consequences of possible additional runtime overheads caused by changes in the scheduling of the system (*e.g.* join tasks).

Lastly, a feature can be implemented to report, in a different file than the .mm file, a list containing all the loops or recursive calls in the entire CFG.

5.3 SSA Future

Since the utility of the SSA tool to solve a practical problem faced by the company was demonstrated throughout the period of this dissertation, one of the directions to take in this project could be the certification of the software for the company's usage. Once the SSA would be certified by the company, the tool could be widely used in many projects currently in development by the company.

Appendix A

Semaphore System MindMap File

```
<map version="1.0.1">
<!-- To view this file, download free mind mapping software FreeMind
    from http://freemind.sourceforge.net -->
<attribute_registry SHOW_ATTRIBUTES = "hide" />
<node FOLDED="true" POSITION="right" ID="LINE_-2" TEXT="Call Graph [MAX
    : 6093 ] " >
   <attribute NAME = "Usage" VALUE = "unknown" />
   <attribute NAME = "# Calls" VALUE = "3" />
   <attribute NAME = "Max Use" VALUE = "6093" />
   <attribute NAME = "Average" VALUE = "2031" />
   <attribute NAME = "Std Dev" VALUE = "2872" />
   <attribute NAME = "Acc Max" VALUE = "6093" />
   <attribute NAME = "Dbg Line" VALUE = "-2" />
   <attribute NAME = "R_ID" VALUE = "" />
        <node FOLDED="true" POSITION="right" ID="LINE_0" TEXT="
            Semaphore_Init [MAX: 6093 ]  [USE: 3 ] " BACKGROUND_COLOR =
             "#ff6666" >
           <attribute NAME = "Usage" VALUE = "3" />
           <attribute NAME = "# Calls" VALUE = "4" />
           <attribute NAME = "Max Use" VALUE = "6093" />
           <attribute NAME = "Average" VALUE = "1934" />
           <attribute NAME = "Std Dev" VALUE = "2463" />
           <attribute NAME = "Acc Max" VALUE = "7738" />
           <attribute NAME = "Dbg Line" VALUE = "0" />
           <attribute NAME = "R_ID" VALUE = "" />
                <node FOLDED="true" POSITION="right" ID="LINE_1" TEXT="
                    Input_Task [MAX: 137 ]  [USE: 7 ] " >
                   <attribute NAME = "Usage" VALUE = "7" />
                   <attribute NAME = "# Calls" VALUE = "2" />
                   <attribute NAME = "Max Use" VALUE = "137" />
                   <attribute NAME = "Average" VALUE = "67" />
                   <attribute NAME = "Std Dev" VALUE = "63" />
```

```
    <attribute NAME = "Acc Max" VALUE = "134" />
    <attribute NAME = "Dbg Line" VALUE = "1" />
    <attribute NAME = "R_ID" VALUE = "" />
        <node FOLDED="true" POSITION="right" ID="LINE_2
            " TEXT="Input_Pedestrians [MAX: 4 ]  [USE:
            4 ] " >
            <attribute NAME = "Usage" VALUE = "4" />
            <attribute NAME = "# Calls" VALUE = "0" />
            <attribute NAME = "Max Use" VALUE = "4" />
            <attribute NAME = "Average" VALUE = "0" />
            <attribute NAME = "Std Dev" VALUE = "0" />
            <attribute NAME = "Acc Max" VALUE = "0" />
            <attribute NAME = "Dbg Line" VALUE = "2" />
            <attribute NAME = "R_ID" VALUE = "" />
        </node>
        <node FOLDED="true" POSITION="right" ID="LINE_3
            " TEXT="Input_Vehicles [MAX: 130 ]  [USE:
            130 ] " BACKGROUND_COLOR = "#ff6666" >
            <icon BUILTIN = "messagebox_warning" />
            <attribute NAME = "Usage" VALUE = "130" />
            <attribute NAME = "# Calls" VALUE = "0" />
            <attribute NAME = "Max Use" VALUE = "130" />
            <attribute NAME = "Average" VALUE = "0" />
            <attribute NAME = "Std Dev" VALUE = "0" />
            <attribute NAME = "Acc Max" VALUE = "0" />
            <attribute NAME = "Dbg Line" VALUE = "3" />
            <attribute NAME = "R_ID" VALUE = "" />
        </node>
</node>
<node FOLDED="true" POSITION="right" ID="LINE_4" TEXT="
    NN_Task [MAX: 6090 ]  [USE: 3223 ] "
    BACKGROUND_COLOR = "#ff6666" >
    <icon BUILTIN = "messagebox_warning" />
    <attribute NAME = "Usage" VALUE = "3223" />
    <attribute NAME = "# Calls" VALUE = "3" />
    <attribute NAME = "Max Use" VALUE = "6090" />
    <attribute NAME = "Average" VALUE = "1239" />
    <attribute NAME = "Std Dev" VALUE = "1202" />
    <attribute NAME = "Acc Max" VALUE = "3717" />
    <attribute NAME = "Dbg Line" VALUE = "4" />
    <attribute NAME = "R_ID" VALUE = "" />
        <node FOLDED="true" POSITION="right" ID="LINE_5
            " TEXT="NN_GetData [MAX: 850 ]  [USE: 850 ]
             " >
            <attribute NAME = "Usage" VALUE = "850" />
```

```
                <attribute NAME = "# Calls" VALUE = "0" />
                <attribute NAME = "Max Use" VALUE = "850" />
                <attribute NAME = "Average" VALUE = "0" />
                <attribute NAME = "Std Dev" VALUE = "0" />
                <attribute NAME = "Acc Max" VALUE = "0" />
                <attribute NAME = "Dbg Line" VALUE = "5" />
                <attribute NAME = "R_ID" VALUE = "" />
            </node>
            <node FOLDED="true" POSITION="right" ID="LINE_6
                " TEXT="NN_Learn [MAX: 2867 ]  [USE: 2867 ]
                 " BACKGROUND_COLOR = "#ff6666" >
                <icon BUILTIN = "messagebox_warning" />
                <attribute NAME = "Usage" VALUE = "2867" />
                <attribute NAME = "# Calls" VALUE = "0" />
                <attribute NAME = "Max Use" VALUE = "2867"
                    />
                <attribute NAME = "Average" VALUE = "0" />
                <attribute NAME = "Std Dev" VALUE = "0" />
                <attribute NAME = "Acc Max" VALUE = "0" />
                <attribute NAME = "Dbg Line" VALUE = "6" />
                <attribute NAME = "R_ID" VALUE = "" />
            </node>
            <node FOLDED="true" POSITION="right" ID="LINE_7
                " TEXT="NN_Evaluate [MAX: 0 ]  [USE:
                unknown ] " BACKGROUND_COLOR = "#ffff66" >
                <attribute NAME = "Usage" VALUE = "unknown"
                    />
                <attribute NAME = "# Calls" VALUE = "0" />
                <attribute NAME = "Max Use" VALUE = "0" />
                <attribute NAME = "Average" VALUE = "0" />
                <attribute NAME = "Std Dev" VALUE = "0" />
                <attribute NAME = "Acc Max" VALUE = "0" />
                <attribute NAME = "Dbg Line" VALUE = "7" />
                <attribute NAME = "R_ID" VALUE = "" />
            </node>
        </node>
        <node FOLDED="true" POSITION="right" ID="LINE_8" TEXT="
            SmartCorrection_Task [MAX: 1463 ]  [USE: 427 ] " >
            <attribute NAME = "Usage" VALUE = "427" />
            <attribute NAME = "# Calls" VALUE = "1" />
            <attribute NAME = "Max Use" VALUE = "1463" />
            <attribute NAME = "Average" VALUE = "1036" />
            <attribute NAME = "Std Dev" VALUE = "0" />
            <attribute NAME = "Acc Max" VALUE = "1036" />
            <attribute NAME = "Dbg Line" VALUE = "8" />
```

```
<attribute NAME = "R_ID" VALUE = "" />
    <node FOLDED="true" POSITION="right" ID="LINE_9
      " TEXT="SmartCorrection_Get [MAX: 1036 ]  [
      USE: 388 ] " BACKGROUND_COLOR = "#ff6666" >
      <attribute NAME = "Usage" VALUE = "388" />
      <attribute NAME = "# Calls" VALUE = "1" />
      <attribute NAME = "Max Use" VALUE = "1036"
        />
      <attribute NAME = "Average" VALUE = "648" />
      <attribute NAME = "Std Dev" VALUE = "0" />
      <attribute NAME = "Acc Max" VALUE = "648" />
      <attribute NAME = "Dbg Line" VALUE = "9" />
      <attribute NAME = "R_ID" VALUE = "" />
        <node FOLDED="true" POSITION="right" ID
          ="LINE_10" TEXT="
          SmartCorrection_Verify [MAX: 648 ]
           [USE: 520 ] " BACKGROUND_COLOR =
          "#ff6666" >
          <icon BUILTIN = "messagebox_warning"
            />
          <attribute NAME = "Usage" VALUE =
            "520" />
          <attribute NAME = "# Calls" VALUE =
            "1" />
          <attribute NAME = "Max Use" VALUE =
            "648" />
          <attribute NAME = "Average" VALUE =
            "128" />
          <attribute NAME = "Std Dev" VALUE =
            "0" />
          <attribute NAME = "Acc Max" VALUE =
            "128" />
          <attribute NAME = "Dbg Line" VALUE =
            "10" />
          <attribute NAME = "R_ID" VALUE = ""
            />
            <node FOLDED="true" POSITION="
              right" ID="LINE_11" TEXT="
              SmartCorrection_Apply [MAX:
               128 ]  [USE: 128 ] "
              BACKGROUND_COLOR = "#ff6666
              " >
              <icon BUILTIN = "
                messagebox_warning" />
```

```
                                        <attribute NAME = "Usage"
                                            VALUE = "128" />
                                        <attribute NAME = "# Calls"
                                            VALUE = "0" />
                                        <attribute NAME = "Max Use"
                                            VALUE = "128" />
                                        <attribute NAME = "Average"
                                            VALUE = "0" />
                                        <attribute NAME = "Std Dev"
                                            VALUE = "0" />
                                        <attribute NAME = "Acc Max"
                                            VALUE = "0" />
                                        <attribute NAME = "Dbg Line"
                                             VALUE = "11" />
                                        <attribute NAME = "R_ID"
                                            VALUE = "" />
                                    </node>
                                </node>
                            </node>
                        </node>
                        <node FOLDED="true" POSITION="right" ID="LINE_12" TEXT
                            ="LightStatus_Task [MAX: 48 ]  [USE: 24 ] " >
                            <attribute NAME = "Usage" VALUE = "24" />
                            <attribute NAME = "# Calls" VALUE = "4" />
                            <attribute NAME = "Max Use" VALUE = "48" />
                            <attribute NAME = "Average" VALUE = "18" />
                            <attribute NAME = "Std Dev" VALUE = "3" />
                            <attribute NAME = "Acc Max" VALUE = "75" />
                            <attribute NAME = "Dbg Line" VALUE = "12" />
                            <attribute NAME = "R_ID" VALUE = "" />
                                <node FOLDED="true" POSITION="right" ID="
                                    LINE_13" TEXT="LightStatus_Green [MAX: 17 ]
                                      [USE: 7 ] " COLOR="#ff9900" >
                                    <attribute NAME = "Usage" VALUE = "7" />
                                    <attribute NAME = "# Calls" VALUE = "2" />
                                    <attribute NAME = "Max Use" VALUE = "17" />
                                    <attribute NAME = "Average" VALUE = "5" />
                                    <attribute NAME = "Std Dev" VALUE = "5" />
                                    <attribute NAME = "Acc Max" VALUE = "10" />
                                    <attribute NAME = "Dbg Line" VALUE = "13" />
                                    <attribute NAME = "R_ID" VALUE = "" />
                                        <node FOLDED="true" POSITION="right" ID
                                            ="LINE_14" TEXT="LightStatus_Set [
                                            MAX: 10 ]  [USE: 7 ] "
                                            BACKGROUND_COLOR = "#ff6666" >
```

```
        <icon BUILTIN = "messagebox_warning"
            />
        <attribute NAME = "Usage" VALUE =
            "7" />
        <attribute NAME = "# Calls" VALUE =
            "1" />
        <attribute NAME = "Max Use" VALUE =
            "10" />
        <attribute NAME = "Average" VALUE =
            "3" />
        <attribute NAME = "Std Dev" VALUE =
            "0" />
        <attribute NAME = "Acc Max" VALUE =
            "3" />
        <attribute NAME = "Dbg Line" VALUE =
             "14" />
        <attribute NAME = "R_ID" VALUE = ""
            />
          <node FOLDED="true" POSITION="
              right" ID="LINE_15" TEXT="
              LightStatus_Update [MAX: 3
              ]  [USE: 3 ] "
              BACKGROUND_COLOR = "#ff6666
              " >
              <icon BUILTIN = "
                  messagebox_warning" />
              <attribute NAME = "Usage"
                  VALUE = "3" />
              <attribute NAME = "# Calls"
                  VALUE = "0" />
              <attribute NAME = "Max Use"
                  VALUE = "3" />
              <attribute NAME = "Average"
                  VALUE = "0" />
              <attribute NAME = "Std Dev"
                  VALUE = "0" />
              <attribute NAME = "Acc Max"
                  VALUE = "0" />
              <attribute NAME = "Dbg Line"
                   VALUE = "15" />
              <attribute NAME = "R_ID"
                  VALUE = "" />
          </node>
    </node>
```

```
                <node FOLDED="true" POSITION="right" ID
                    ="LINE_16" TEXT="LightStatus_Clear
                    [MAX: 0 ]  [USE: 0 ] " COLOR = "#
                    A0A0A0" >
                    <attribute NAME = "Usage" VALUE =
                        "0" />
                    <attribute NAME = "# Calls" VALUE =
                        "0" />
                    <attribute NAME = "Max Use" VALUE =
                        "0" />
                    <attribute NAME = "Average" VALUE =
                        "0" />
                    <attribute NAME = "Std Dev" VALUE =
                        "0" />
                    <attribute NAME = "Acc Max" VALUE =
                        "0" />
                    <attribute NAME = "Dbg Line" VALUE =
                         "16" />
                    <attribute NAME = "R_ID" VALUE = ""
                        />
                </node>
            </node>
            <node FOLDED="true" POSITION="right" ID="
                LINE_17" TEXT="LightStatus_Red [MAX: 17 ]
                [USE: 7 ] " COLOR="#ff9900" >
                <attribute NAME = "Usage" VALUE = "7" />
                <attribute NAME = "# Calls" VALUE = "2" />
                <attribute NAME = "Max Use" VALUE = "17" />
                <attribute NAME = "Average" VALUE = "5" />
                <attribute NAME = "Std Dev" VALUE = "5" />
                <attribute NAME = "Acc Max" VALUE = "10" />
                <attribute NAME = "Dbg Line" VALUE = "17" />
                <attribute NAME = "R_ID" VALUE = "" />
                    <node FOLDED="true" POSITION="right" ID
                        ="LINE_REF_14" TEXT=">
                        LightStatus_Set [MAX: 10 ]  [USE: 7
                         ] " BACKGROUND_COLOR = "#ff6666" >
                    <arrowlink DESTINATION = "LINE_14" />
                        <icon BUILTIN = "messagebox_warning"
                             />
                        <attribute NAME = "Usage" VALUE =
                            "7" />
                        <attribute NAME = "# Calls" VALUE =
                            "1" />
```

```
                <attribute NAME = "Max Use" VALUE =
                    "10" />
                <attribute NAME = "Average" VALUE =
                    "3" />
                <attribute NAME = "Std Dev" VALUE =
                    "0" />
                <attribute NAME = "Acc Max" VALUE =
                    "3" />
                <attribute NAME = "Dbg Line" VALUE =
                     "14" />
                <attribute NAME = "R_ID" VALUE = ""
                    />
            </node>
            <node FOLDED="true" POSITION="right" ID
                ="LINE_REF_16" TEXT="">
                LightStatus_Clear [MAX: 0 ]  [USE:
                0 ] " COLOR = "#A0A0A0" >
                <attribute NAME = "Usage" VALUE =
                    "0" />
                <attribute NAME = "# Calls" VALUE =
                    "0" />
                <attribute NAME = "Max Use" VALUE =
                    "0" />
                <attribute NAME = "Average" VALUE =
                    "0" />
                <attribute NAME = "Std Dev" VALUE =
                    "0" />
                <attribute NAME = "Acc Max" VALUE =
                    "0" />
                <attribute NAME = "Dbg Line" VALUE =
                     "16" />
                <attribute NAME = "R_ID" VALUE = ""
                    />
            </node>
        </node>
        <node FOLDED="true" POSITION="right" ID="
            LINE_20" TEXT="LightStatus_Yellow [MAX: 17
            ]  [USE: 7 ] " COLOR="#ff9900" >
            <attribute NAME = "Usage" VALUE = "7" />
            <attribute NAME = "# Calls" VALUE = "2" />
            <attribute NAME = "Max Use" VALUE = "17" />
            <attribute NAME = "Average" VALUE = "6" />
            <attribute NAME = "Std Dev" VALUE = "3" />
            <attribute NAME = "Acc Max" VALUE = "13" />
            <attribute NAME = "Dbg Line" VALUE = "20" />
```

```
<attribute NAME = "R_ID" VALUE = "" />
    <node FOLDED="true" POSITION="right" ID
        ="LINE_REF_14" TEXT="">
        LightStatus_Set [MAX: 10 ]  [USE: 7
         ] " BACKGROUND_COLOR = "#ff6666" >
    <arrowlink DESTINATION = "LINE_14" />
       <icon BUILTIN = "messagebox_warning"
            />
       <attribute NAME = "Usage" VALUE =
           "7" />
       <attribute NAME = "# Calls" VALUE =
           "1" />
       <attribute NAME = "Max Use" VALUE =
           "10" />
       <attribute NAME = "Average" VALUE =
           "3" />
       <attribute NAME = "Std Dev" VALUE =
           "0" />
       <attribute NAME = "Acc Max" VALUE =
           "3" />
       <attribute NAME = "Dbg Line" VALUE =
            "14" />
       <attribute NAME = "R_ID" VALUE = ""
           />
    </node>
    <node FOLDED="true" POSITION="right" ID
        ="LINE_REF_15" TEXT="">
        LightStatus_Update [MAX: 3 ]  [USE:
         3 ] " >
       <attribute NAME = "Usage" VALUE =
           "3" />
       <attribute NAME = "# Calls" VALUE =
           "0" />
       <attribute NAME = "Max Use" VALUE =
           "3" />
       <attribute NAME = "Average" VALUE =
           "0" />
       <attribute NAME = "Std Dev" VALUE =
           "0" />
       <attribute NAME = "Acc Max" VALUE =
           "0" />
       <attribute NAME = "Dbg Line" VALUE =
            "15" />
       <attribute NAME = "R_ID" VALUE = ""
           />
```

```
        </node>
</node>
<node FOLDED="true" POSITION="right" ID="
    LINE_23" TEXT="LightStatus_Intermittent [
    MAX: 24 ]  [USE: 14 ] " BACKGROUND_COLOR =
    "#ff6666" >
    <icon BUILTIN = "messagebox_warning" />
    <attribute NAME = "Usage" VALUE = "14" />
    <attribute NAME = "# Calls" VALUE = "2" />
    <attribute NAME = "Max Use" VALUE = "24" />
    <attribute NAME = "Average" VALUE = "5" />
    <attribute NAME = "Std Dev" VALUE = "5" />
    <attribute NAME = "Acc Max" VALUE = "10" />
    <attribute NAME = "Dbg Line" VALUE = "23" />
    <attribute NAME = "R_ID" VALUE = "" />
        <node FOLDED="true" POSITION="right" ID
            ="LINE_REF_14" TEXT="">
            LightStatus_Set [MAX: 10 ]  [USE: 7
             ] " BACKGROUND_COLOR = "#ff6666" >
        <arrowlink DESTINATION = "LINE_14" />
            <icon BUILTIN = "messagebox_warning"
                />
            <attribute NAME = "Usage" VALUE =
                "7" />
            <attribute NAME = "# Calls" VALUE =
                "1" />
            <attribute NAME = "Max Use" VALUE =
                "10" />
            <attribute NAME = "Average" VALUE =
                "3" />
            <attribute NAME = "Std Dev" VALUE =
                "0" />
            <attribute NAME = "Acc Max" VALUE =
                "3" />
            <attribute NAME = "Dbg Line" VALUE =
                 "14" />
            <attribute NAME = "R_ID" VALUE = ""
                />
        </node>
        <node FOLDED="true" POSITION="right" ID
            ="LINE_REF_16" TEXT="">
            LightStatus_Clear [MAX: 0 ]  [USE:
            0 ] " COLOR = "#A0A0A0" >
            <attribute NAME = "Usage" VALUE =
                "0" />
```

```
                                        <attribute NAME = "# Calls" VALUE =
                                            "0" />
                                        <attribute NAME = "Max Use" VALUE =
                                            "0" />
                                        <attribute NAME = "Average" VALUE =
                                            "0" />
                                        <attribute NAME = "Std Dev" VALUE =
                                            "0" />
                                        <attribute NAME = "Acc Max" VALUE =
                                            "0" />
                                        <attribute NAME = "Dbg Line" VALUE =
                                             "16" />
                                        <attribute NAME = "R_ID" VALUE = ""
                                            />
                                    </node>
                                </node>
                            </node>
                        </node>
                        <node FOLDED="true" POSITION="left" ID="LINE_-1" TEXT="Not
                            Called
Functions [MAX: 0 ] " BACKGROUND_COLOR = "#ffff66" >
                            <attribute NAME = "Usage" VALUE = "unknown" />
                            <attribute NAME = "# Calls" VALUE = "0" />
                            <attribute NAME = "Max Use" VALUE = "0" />
                            <attribute NAME = "Average" VALUE = "0" />
                            <attribute NAME = "Std Dev" VALUE = "0" />
                            <attribute NAME = "Acc Max" VALUE = "0" />
                            <attribute NAME = "Dbg Line" VALUE = "-1" />
                            <attribute NAME = "R_ID" VALUE = "" />
                        </node>
                        <node FOLDED="true" POSITION="right" ID="LINE_-1" TEXT="RTAOS
                            TASKS [MAX: 0 ] " COLOR = "#A0A0A0" >
                            <attribute NAME = "Usage" VALUE = "0" />
                            <attribute NAME = "# Calls" VALUE = "2" />
                            <attribute NAME = "Max Use" VALUE = "0" />
                            <attribute NAME = "Average" VALUE = "0" />
                            <attribute NAME = "Std Dev" VALUE = "0" />
                            <attribute NAME = "Acc Max" VALUE = "0" />
                            <attribute NAME = "Dbg Line" VALUE = "-1" />
                            <attribute NAME = "R_ID" VALUE = "" />
                                <node FOLDED="true" POSITION="right" ID="LINE_-1" TEXT
                                    ="MASTER CORE [MAX: 0 ] " BACKGROUND_COLOR = "#
                                    ff6666" >
                                    <attribute NAME = "Usage" VALUE = "0" />
                                    <attribute NAME = "# Calls" VALUE = "0" />
```

```
                    <attribute NAME = "Max Use" VALUE = "0" />
                    <attribute NAME = "Average" VALUE = "0" />
                    <attribute NAME = "Std Dev" VALUE = "0" />
                    <attribute NAME = "Acc Max" VALUE = "0" />
                    <attribute NAME = "Dbg Line" VALUE = "-1" />
                    <attribute NAME = "R_ID" VALUE = "" />
                </node>
                <node FOLDED="true" POSITION="right" ID="LINE_-1" TEXT
                    ="SLAVE CORE [MAX: 0 ] " BACKGROUND_COLOR = "#
                    ff6666" >
                    <attribute NAME = "Usage" VALUE = "0" />
                    <attribute NAME = "# Calls" VALUE = "0" />
                    <attribute NAME = "Max Use" VALUE = "0" />
                    <attribute NAME = "Average" VALUE = "0" />
                    <attribute NAME = "Std Dev" VALUE = "0" />
                    <attribute NAME = "Acc Max" VALUE = "0" />
                    <attribute NAME = "Dbg Line" VALUE = "-1" />
                    <attribute NAME = "R_ID" VALUE = "" />
                </node>
        </node>
</node>
</map>
```

References

[1] Wikipedia contributors. V-model (software development) — Wikipedia, the free encyclopedia, 2018. [Online; accessed 9-November-2018]. URL: https://en.wikipedia.org/w/index.php?title=V-Model_(software_development)&oldid=865190501.

[2] Jenkins. URL: https://jenkins.io.

[3] Maria Soto, Marc Sevaux, and Andre Rossi. *Memory Allocation Problems in Embedded Systems: Optimization Methods*. ISTE LTD, 2012. URL: https://www.ebook.de/de/product/19908472/maria_soto_marc_sevaux_andre_rossi_memory_allocation_problems_in_embedded_systems_optimization_methods.html.

[4] Andrew S. Tanenbaum, Herbert Bos, and Andrew S. Tanenbaum. *Modern Operating Systems: Global Edition*. Prentice Hall, 2014. URL: https://www.ebook.de/de/product/24544513/andrew_s_tanenbaum_herbert_bos_andrew_s_tanenbaum_modern_operating_systems_global_edition.html.

[5] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In Rajeev Alur and Insup Lee, editors, *Embedded Software*, pages 306–322, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[6] Matt Might. What is static analysis? by matt might, 2014. URL: https://www.youtube.com/watch?time_continue=5&v=POvX4hYIoxg#t=3m51s.

[7] Eric Botcazou, Cyrille Comar, and Olivier Hainque. Compile-time stack requirements analysis with gcc motivation, development, and experiments results, 11 2018.

[8] Klocwork. Early bug detection, comprehensive coverage., 2008. URL: http://www.klocwork.com/solutions/defectDetection.asp.

[9] B. Chess and G. McGraw. Static analysis for security. *IEEE Security Privacy*, 2(6):76–79, Nov 2004. doi:10.1109/MSP.2004.111.

[10] Jörg Müller, Daniel Polansky, Petr Novak, Christian Foltin, Dimitry Polivaev, et al. Freemind - free mind mapping software, 2014. URL: http://freemind.sourceforge.net/wiki/index.php/Main_Page#License.

[11] Robert Davis, Nick Merriam, and Nigel Tracey. How embedded applications using an rtos can stay within on-chip memory limits, 01 2000.