**University of Macau**

**Faculty of Science and Technology**



**Detecting Software Defects Using Verification Tools**

Final Report

***by***

**IP TONG, DB625785**

Final Project Report submitted in partial fulfillment  
of the requirements of the Degree of   
Bachelor of Science in Computer Science

Project Supervisor

Prof. Qiwen XU

18 May 2020

**DECLARATION**

I sincerely declare that:

I and my teammates are the sole authors of this report,

All the information contained in this report is certain and correct to the best of my knowledge,

I declare that the thesis here submitted is original except for the source materials explicitly acknowledged and that this thesis or parts of this thesis have not been previously submitted for the same degree or for a different degree, and

I also acknowledge that I am aware of the Rules on Handling Student Academic Dishonesty and the Regulations of the Student Discipline of the University of Macau.

Signature : IP TONG

Name : IP TONG

Student No. : DB625785

Date : 18 May 2020

ABSTRACT

As the society is developing rapidly since the 21st century, software application has been expanding further in terms of scale and complexity. The security and reliability of the software application become the major problem. Avoiding those problems and guaranteeing the quality of software application is now in the priority to that need to solve. As the result, software defect and detection technology has raised. Which can be further separated into dynamic detection technology and static analysis technology. Dynamic detection technology is the verification of the executing program. Static analysis technology is to check and analyze the program code while not running the program, and by using different techniques in static analysis technology, shortcoming could be discovered in the earlier stages of the program development. Therefore, it has drawn many attentions from the software development field and working as a significant part in the software defect detection.

Within the software development, software verification is the major section among the others. The amount of time and funding spent in any project are mainly based on three factors, which is the functional security of the project, level of the business risk, and the quality culture of the organization. However, regardless of any factors cause the organization to perform quality control, secure the high-quality and safe software application are more important than the determination of the program itself.

In this paper, we will introduce the common software application problems and the methods of the detection of those problems. And a complete verification of a software code will be performed afterwards as a case study.

**TABLE OF CONTENTS**

**CHAPTER 1.** **INTRODUCTION 6  
1.1 Background 6  
1.2 Benefit 6  
1.3 Problems 7  
1.4 Target 8**

**CHAPTER 2.** **KNOWLEDGE OF VERIFICATION 8  
2.1 Introduction of Formal verification 9**2.1.1 Formal verification principle  **10  
2.2 Introduction of static analysis 11**

**CHAPTER 3.** **VERIFICATION TOOL 12  
3.1 Introduction 12  
3.2 Introduction of Frama-C 13**3.2.1 Framework 13  
3.2.2 Plug-In 14  
3.2.3 Memory Model 15  
3.2.4 Annotations 15

**CHAPTER 4.** **RELATED WORK 18**

**4.1 Formal verification 18**

**4.2 Run-time Error Problems 18**

4.2.1 Buffer overflow 18

**CHAPTER 5.** **STUDY CASE 19  
5.1 Functional requirements**  **19  
5.2 Relate Code**  **21  
5.3 Formal verification**  **23**

**CHAPTER 6.** **A FLOW CHART OF THE SCHEDULE 27**

**CHAPTER 7.** **CONTRIBUTIONS 29**

**CHAPTER 8.** **CONCLUSION AND FUTURE WORK 31**

**CHAPTER 9. REFERENCES 32**

**LIST OF TABLES**

[Table 1: Frama-C Plug-in 14](#_tyjcwt)

**CHAPTER 1. INTRODUCTION**

**1.1 Background**As the first impression of the field of computer science, most of the people will consider that there are a high proportion of people within the field are programmers. However, not all the programmers know the existence of code detector, which also known as software detection. Even some of the programmers do know the exist of it, they may consider software detection as a dispensable, but in fact, software detection can help the program to be finished in a much easier fashion when excluding possible errors.  
  
Programmer also can be a code detector. They are not contradicting with each other. Most of computer science students only knows about programmer since programming is the fundamental of computer science. Therefore, we will be lack of the competitiveness in many technological organizations if we only know about programming. However, we can also learn about software detection, not only increase our employment opportunities but also increase the opportunity to get a promotion and a pay raise.  
  
Our interim task for this project is that learn how to use Frama-C in the first semester.   
And final task is that all the process of the verification.

**1.2 Benefit**Software verification is an essential part of software development. The effort and budget invested in a particular project depend on many factors, such as the functional safety of the project, the level of business risk, or the quality culture of the organization. A code in software development likes a skeleton, and a good software verification like a muscle. We think that cannot be ignored, but nowadays in China, we can not find a lots of things about this.  
  
We found some of the example about the accident happened due to the lack of a good verification in before publishing:  
 a) On June 4, 1996, the Ariane-v launch vehicle of European Space Agency had a catastrophic explosion accident just 37 seconds after launching, resulting in a direct economic loss of 2.5 billion. After found that the aerospace accident is caused by the software failure of the inertial reference system.  
  
 b) On the eve of the new era in 2000, due to the use of two digits to represent the year in some older computer systems, and the inability to recognize the year 2000 and the years after it, the "The Millennium Bug" problem that shocked the world occurred, resulting in much time-dependent banking, tax, power, customs, and other institutions unable to work normally. As a result, it took hundreds of billions of dollars of human and material resources to control the problem.  
  
 c) On November 1, 2005, because there was no information on the display screen of the Tokyo Stock Exchange, and the failure was difficult to repair for a while, the stock exchange of that morning was forced to suspend until 1:30 p.m., which had a serious impact on Tokyo stock market.  
  
 d) On June 1, 2009, an Air France flight AF447 crashed on its way from Rio de Janeiro, Brazil, to Paris, France. There were 228 people on board. No one survived! Some experts speculate that the failure of the tester may be the real cause of the accident.  
  
 e) On July 23, 2011, the D301 bullet train from Beijing south to Fuzhou and the D3115 bullet train from Hangzhou to Fuzhou South had a serious rear-end collision in the concave segment section of Shuangyu, Wenzhou, Zhejiang Province, and four carriages of the rear car fell off the viaduct. The accident resulted in 40 deaths and about 200 injuries. According to the investigation, the traffic accident of the Yongwen railway is caused by the serious design defects of the equipment data acquisition unit of the train control center.   
  
 f) On August 16, 2013, the arbitrage strategy system of Everbright Securities strategic investment department failed due to design defects, resulting in the wrong purchase of 23.4 billion yuan, causing the Everbright Wulong index event, which had a huge impact on China's A-share market on that day, and brought losses of 190.4 million yuan to Everbright.  
  
As in the case study below, we had an example which is about automobile driving system to show. We successfully verify every functional requirement in that code. It seems like a simple step, but it is very important in future development of the system. We ensure that the code can execute and work in any case, we will introduce it in the next part of the report.

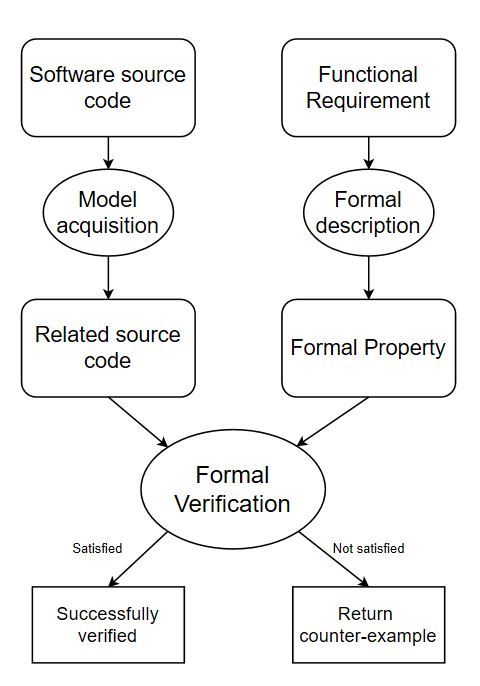
**1.3 Problems**At the beginning of the project, there are branch of knowledges that we need to learn and understand about the verification tool, which will be use in the detection process. We originally have a plan to cooperate with Huawei mobile OS department to perform a verification on their provided source codes of their phone driver system with related functional requirements.   
  
However, due to the epidemic situation of COVID-19, Huawei mobile OS department decided to end the cooperation, so we changed the original plan and look for some other codes for the verification.

**1.4 Target**  
  
This paper aims to determine the application degree of modern formal verification tools in real life, in order to verify the actual requirements for safety critical code. This includes identifying which types of requirements and code can be inferred, potential problems that may arise, and how to avoid them.

**CHAPTER 2. KNOWLEDGE OF VERIFICATION**

**2.1 Introduction of Formal verification**

Formal verification is a technique to find out every single states of a software system. By looking through all state spaces of the migration system, the detection module will automatically prove the looked through state spaces and generates examples for the cases that not meeting the property of verification. Verify the formal model of a specific code meets its requirement is the essential part of the model checking. The basic concept is to use the migrating system(S) to show how the verifying system should work and use the modal logic or the sequential logic formula(F) to describe the property of the system. In this way, verification process is changed from if the system has the wanted properties, to if the state migration system (S) is a model that fulfill the sequential logic formula (F). In another word, try to prove the requirements in a mathematical way. Early stages model detection technology is used in abstract system model and hardware system model widely. However, the model detection technology has been used in software systems like C programs and Java programs widely.

  
***Figure 1: Verification flow diagram***

**2.1.1 Formal verification principle**

Now we would taking about formal verification detail which is it principle. In our verification tool Frama-C, we used it plug-in named WP Plug-In. That means is use weakest precondition to verify the functional requirement if or not satisfied. So we would lead to a principle: ***Hoare Logic***.  
  
***Hoare logic*** is a formal system, which is used to verify the correctness of the computer system. The ***Hoare Triple*** is the most basic formula expression in the ***Hoare Logic***. ***Hoare Triple*** are symbolic representations that accurately describe what functions a program is intended to perform:

{P}S{Q}

P: Precondition (State)  
S: Instructions  
Q: Postcondition (State)

**2.1.1.1 Hoare Triple algorithm**

**2.1.1.1.1 Combination**

{P}S{Q} Λ {P}S{R} ⇒ {P}S{Q Λ R}  
{P}S{Q} Λ {R}S{Q} ⇒ {P Λ R}S{Q}

If the state described by 'condition P' can be described as 'True' by 'instruction S', then the state described by 'condition P' can be described as 'True' by 'instruction S'.

For example:  
 {x= -3} x:= -x {x>0} Λ {x= -3} x:= -x {x:even}  
 ⇒  
 {x= -3} x:= -x {x>0 Λ x:even}

**2.1.1.1.2 Strengthening/Weakening Conditions**

{P}S{Q} Λ R ⇒ P   
⇒  
{R}S{Q}

If the state described by 'condition P' can be described by 'condition Q' through 'instruction S', and if 'condition R' is 'True', then 'condition P' is also 'True', then the state described by 'condition R' through 'instruction S' can be described by 'condition R' as well as 'True'.

For example:  
 {x<0} x:= -x {x>0} Λ (x= -2) ⇒ (x<0)  
 ⇒  
 {x=-2} x:= -x {x>0}

**2.1.1.1.3 Concatenation**

{P}S0{Q} Λ {Q}S1{R} ⇒ {P}S0;S1{R}

If the state described by 'condition P' reaches the state described by 'condition Q' through 'instruction S0' and the state described by 'condition Q' reaches the state described by 'condition Q' through 'instruction S1' and the 'state R' described by 'condition Q' is 'True', then the state described by 'condition P' reaches the state described by 'instruction S0;S1' can be described by 'condition R' as well as 'True'.

For example:

{x>4} x:=x+1 {x>5} Λ {x>6} x:=-x {x<-6}  
 ⇒  
 {x>4} x:=x+1; x:=-x {x<5}

**2.1.1.2 Weakest pre-condition**

Weakest pre-condition, we would called it WP in the next. That is very important for our software verification in Frama-C. We used it principle to verify the code if not satisfied the functional requirements. We would intros it in the next. Now we would intro it principle.

Suppose 'S' is a statement and 'Q' is a predicate, which describes a certain relationship determined after' s' is executed. Define another predicate from 'S' and 'Q', and record it as 'WP (S, Q)', which means:

{wp(S,Q)} ⇒ S{Q}

The set of all such states, 'S' starts from any state, and will terminate in a limited period of time to satisfy the state of 'Q'.

This function is also central to formal software verification, as tools usually verify a function f by calculating the weakest pre-condition W from the function body and the post-condition Q, and then try to voluntarily prove that the pre-condition P implies W.

In use of Frama-C, we use an annotation, ‘requires’ means pre-condition, ‘ensure’ means post-condition. We would detail intro it after.

**2.2 Introduction of static analysis**

Starting from early way to verification based on lexical analysis, string match and string matching, the static analysis technology is now evolving into program analysis and the combination of data mining technology. Static analysis tool of source codes looks for the hidden errors and vulnerable parts in the source codes before executing the program as an important way and method of discovering the vulnerabilities in the software. There are two crucial indicators to grade the quality of a static analysis tools:

1. False negatives: the proportion of the program which have security issues but not been discovered by the tool yet.
2. False positions, the rate that the verification tool reports safety issues incorrectly in a program which does not already exist.

For the static analysis of existing source code, according to the different analysis mechanism and functions, it can be divided into the following four categories:

1. Based on the annotation analysis tools, using the functional requirement provided by the programmer to help to analyze the program, we can find defects such as those variables that are not initialized and array that is out of their declared bounds.
2. Lexical analysis related tool is used to pre-process the source file into token stream, and then look for the related structure in the library.
3. For the semantic analysis tools, by analyzing the program data flow and control flow, think of the basic semantics of the program, we can verify the string usage defects and lock those in C program.
4. Tool based on memory leaks, which detects problem like buffer overflow, leak of the memory, mismatched memory unit allocation in the program.

In this project, we would chose Frama-C as our verification tool, because it include a lot of different analysis mechanism and functions, we would introduce it in the next, and then we would major in annotation analysis, because it would fix most of problems in nowadays software applications. Other analysis is useful but not necessary, at least in this project.

**CHAPTER 3. VERIFICATION TOOL**

**3.1 Introduction**

There are many tools with different purposes and programming languages for formal validation. Since our research mainly focus on studying the verification of real-time security critical code, due to the portability and performance which is usually C language based and this is limited the choice of tools that support this language.

Within the area of verification tools of C code, the most famous and with the highest use rate is Frama-C, which is based on combination verification. Combination verification required the function’s caller must first satisfy under some assumptions and noted as a contract that identifies the function’s properties. After that, they will use deductive reasoning based on the weakest precondition calculus to build up the proving obligation. Finally, it will be completed by the automatic theorem prover to examine the related required conditions specified as annotation in the C code.

The program will first verify for any error getting from the standard compiler before performing reasoning part. Then, it will convert an internal logical representation, which often referred to as ghosting state. This internal state contains a logical representation of any variables from the program, also any others of the abstract variables and functions. As same as to provide a more effective model for program reasoning. Ghost states can only effective on static validator, but not compilers.

These tools used the compositional features of the underlying theory when they fulfil their modular validation, which means the functionality is proved individually. When checking function those calling another function, the function contract of the function caller will only be supposed to be justifiable and continues to check the calling function based on the result state by the verification tool. Therefore, a lot of valid expression in C are also valid as in the comments.

Also, both tools claim to be healthy (as in no false negative). This assumes that there are no bugs in the implementation.

**3.2 Introduction of Frama-C**

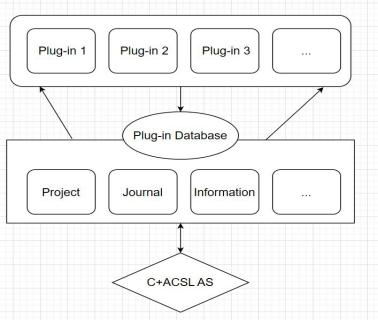
Frama-C is a free software tool developed two French research institutions called CEA and INRIA. It is a platform to analyze and verify programs based on different theories. Its analysis object is C program source code. Frama-C can be used for Linux and MAC also can work on windows through a stimulator. It is still developing and able to get business support from external organization. In addition to functional validation, Frama-c also supports multiple types of analysis, such as value analysis and program slicing, and due to its modular structure plug-ins, there can have more functions.

There are two standard deductive verification plugins that come with Frama-C, Jessie, and WP (weakest precondition). WP focuses on parametrization with regards to the memory model, and was, therefore, the plugin that was used. It is based on weakest precondition calculus and generates proof obligations (i.e. mathematical first-order logic formulas) from the annotations, which is then submitted to an automatic theorem prover. The standard theorem prover used by WP is Alt-Ergo, but it also supports other ones. Frama-C also supports the use of interactive proof assistants such as Coq, which can be used as a complement to an automatic theorem prover.

Frama-C only supports the analysis of strictly sequential code.

**3.2.1 Framework**

The basic framework of Frama-C is shown in Figure 1 is based on the same kernel and uses the same formal specification language ACSL called ANSI C specification language, which is used to do annotation on a C program. Because of that, many plug-ins are developed based on the Frama-C platform which can be used for static analysis, derivation, demonstration, and testing of software. At the same time, Frama-C also provides some extensible APIs to facilitate user development plug-ins.

****

***Figure 2: Frama-C Framework***

**3.2.2 Plug-In**

Within the plug-ins which based on deductive verification, abstract interpretation, and specific symbol execution test. There are three basic plug-ins were built on the Frama-C platform with some derived plug-ins. Table 1 shows some common framework-c plug-ins.

***Table 1: Frama-C Plug-in***

|  |  |  |
| --- | --- | --- |
| **Plug-in** | **Principle** | **Describe** |
| **Value Analysis**  **plug-in** | Abstract  interpretation | Calculating the value range of variables for variables in a program |
| **WP plug-in**  **(Weakest Precondition)** | Deductive  verification | Proving obligation for generating ACSL comments in C program |
| **Jessie plug-in** | Deductive  verification | Comments on ACSL in deductive verification C program |
| **Slicing plug-in** | Flow analysis | Dependency based program slicing |

The Frama-C plug-ins can be used together due to the Shared use of ACSL, a formal code of conduct language provided by Frama-C in platform-based plug-ins. The user describes the functional characteristics of the program. And we will describe more of the plug-ins below.

**3.2.2.1 Value analysis plug-in based on Abstract Interpretation**

The value of Frama-C analysis plug-ins based on the principle of abstract interpretation, which is able to calculate the possible values of program variables in different program points automatically and give an alert for potential run-time errors. Similar commercial software such as Polyspace and Astree also have these functions. However, the value analysis plug-in of Frama-C not only has the benefit of open source, but also have the more important difference is that the value of Frama-C analysis plug-ins can be applied to all source codes that meet the ISO C99 standard, not to a particular application field.

As an example, for any integers, which need to be bound with INT\_MIN and INT\_MAX. Eva plug-in will check and limit the variable to prevent overflow when the software is running. In addition, the value plug-in is unique in that it is rooted in Frama-C's system framework, and its results can be used again by another Frama-C plug-ins.

**3.2.2.2 Jessie and WP plug-ins based on derivation verification**

These two plug-ins are based on the weakest precondition technology to prove that C code meets the program specification described by ACSL.

The method of derivation and verification is based on model. For a C program, the algorithm model of abstract expression is used to describe the integer or floating-point number, and the memory model is used to abstract the storage of the program.

The use of WP plug-in can slow down the validation conditions, independent of the specific model. For the algorithm model, users can choose any number types such as integer, floating-point machine integer, real number and floating-point number. The memory model of WP plug-in is directly inspired by the weakest precondition operator, which the program supposed to be not contain pointers.

For instance, a function to return the maximum number within two variables. The user, as an example, can determine the cases of the function: a< b, and a>b. Wp plug-in will then try to prove the source code having a same output as the user determined conditions. If the Wp plug-in find out any conditions that is not determined by the user, it will then output a message to inform the user. In the previous example, a=b is not determined by the user but in fact this case will happen when the function is running. User can then change the code to meet all possible outcomes and prevent unexpected results.

**3.2.2.3 Slicing plug-in based on program flow analysis**

Frama-c's slicing plug-in outputs a subset of the original code. The slicer is generated based on the slicing criteria provided by the user. And the output slicer is still a compiled C code with the same behavior as the original program.

**3.2.3 Memory Model**

WP supports several memory models, with the two main ones currently being the Hoare model and the Typed model.

The Hoare model is the simplest model, directly influenced by the classic weakest precondition calculus. In this historical model, programs do not have pointers, and each local and global variable corresponds to a distinct logical variable. While the values of pointers can be represented, the model cannot handle reads and writes through them, which means the heap cannot be represented at all. However, this model is very efficient and produces simple proof obligations, which can still make it useful for certain programs or functions.

**3.2.4 Annotations**

Frama-C uses a specification language called ANSI/ISO C Specification Language (ACSL) for its annotations. ACSL supports many features, such as function preconditions, postconditions and invariants over loops and data. The annotations are specified in traditional C comments, starting with the special character @.

/\*@ requires \valid(n) && \valid(m);

assigns \*n, \*m;

ensures \*n == \old(\*m) && \*m == \old(\*n);

\*/

void swap(int \* n, int \* m);

Example of function contract in Frama-C.

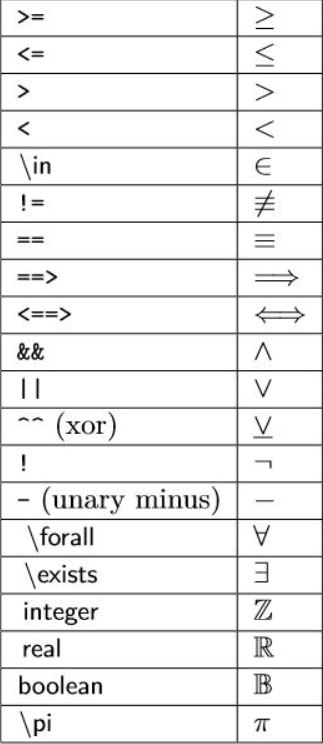
In this example, the only precondition is that both pointers n and m point to a valid memory location. In function contracts, the keyword \old represent the value of the expression before the program runs, which is always true of expressions in a precondition, while expressions in a postcondition based on the value after running the program in default. Therefore, the postcondition secures any variables like A will contain the value that B contained before and after running the program. Also, when a function has side effects, you have to specify which memory may be changed with the assigns keyword so that calling functions can infer which memory may have been changed and if it ensures to be the same as before the call.

While Frama-C does not support the full ACSL specification, it still has a rich feature set. It supports invariants over loops and data structures and the specification of global lemmas. Ghost code (both variables and functions) can be used to simplify verification, but such code can not have any side-effects, i.e. it may not affect memory other than the ghost state. Frama-C also supports natural arithmetic (as opposed to the unsigned and signed integer arithmetic in C). Because of its type of unsafe memory model, some additional annotations such as separated are needed to prove that pointers are not aliased.

By default, Frama-C will also try to prove that every function terminates. For functions that are recursive or contain loops, the keyword decreases can be used to specify an integer measure, a sequence that will decrease during the execution of the function, to help prove the termination. Alternatively, for non-terminating functions, the keyword terminates can be used to specify under what conditions the function will halt.

**CHAPTER 4. RELATED WORK**

Static analysis is procedure that assess and analyze the code without running it and is the critical section of software quality and security. It can help us to expose many trick situations that we may meet during the execution by several methods, such as control flow analysis, data flow analysis , lexical analysis, semantic analysis.

**4.1 Formal verification**In software code verification, the most important is formal verification. It can verify the code in the flow whether have any wrong or not. I take a simple example, like one past one equal to two, if now I need to verify this equation, I need to give it a annotation, also called weakest precondition, like two minus one equal to one, that means the sum minus subtrahend equal to minuend. If verification had success, the original equation would had been verify. In the software code also like this, in our job, we need to verify each function one by one, to detect which one the verification was not success, if not success, it must any wrong into the function. After we would do a troubleshooting, to check which part have bug that let our final goal not achieved.  
  
In theory, it maybe simple, but in the operation in Frama-C, that would be a challenge. We learned to using Frama-C very long time. And try to do many exercises, finally, we leaned it how to do a formal verification.   
  
The important part is in using ACSL language as annotation, it need to give Frama-C read. After Frama-C do a verification by annotation our given. It is worth mentioning that, the rules symbols not like our use in usual. They were changed, like this:  
  
 

***Figure 3: ACSL Rules Symbol***

**4.2 Run-time Error Problems**Static analysis of code able to determinate multiple types of defects and vulnerabilities. Starting from alert level like discovered some unused variables, till error lever with discovering bugs.

Here are some common, more serious and statically detectable problems. We would take one simple examples that the software code common risks. And we would show how can we use Frama-C to detect the risks. In different risks, there are may be use different plug-ins. But in our study case, most of the plug-in is WP Plug-in, because their engineers were very careful. As below 4.1.

**4.2.1 Buffer overflow**

This is a buffer overflow example, in this case, the arry buf[] only have 10 space,there would be 0,1,2,3,4,5,6,7,8,9, but in the next, if x greater than 10, equal to 10 or less than 0, the buffer would be overflow. Unfortunately, the system would not report error when x greater than 0, equal to 10 and less than 10. That is a serious risk when the input x worse.

void foo(int x)

{

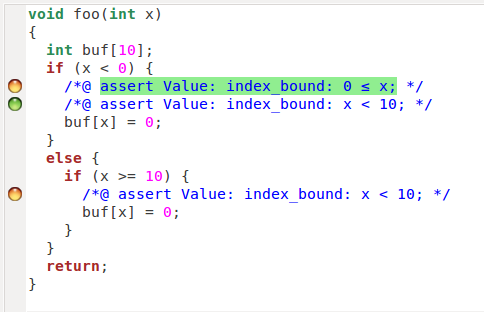
int buf[10];

if (x < 0 || x >= 10)

   buf[x] = 0;

}

Therefore, we use value analysis plug-in to detect it. There had two yellow lights means had two detection wrongs. The first is show you to know that the index bound must be greater than or equal to 0. The second is show you to know that the index bound must be less than 10.



***Figure 4: Buffer overflow example***

**CHAPTER 5. STUDY CASE**

Recently, the vehicle is controlled by the embedded computer system even more. Such systems are safety critical, which may cause serious consequences by the calculation errors. A common way to secure that software works correctly is by testing. However, as these develops and become more and more complicated, the difficulties to secure a far enough testing range also increases. As a solution, we carry out a case study of the steering module in Scania. Part of the requirements have been verified successfully.

**5.1 Functional requirements**

This module controls the steering system of vehicles. The following is a list of requirements it should adhere to. This is a vehicle steering system controlling module.  
The requirements below are the module should be stick to.  
 1. If there is no flow in the primary circuit or there is a short circuit, then the primary circuit will not able to provide power steering.  
  
 2. If the signal of wheel based speed is over 3 km/h, then the vehicle should be consider as moving.  
  
 3. If the vehicle is consider moving and the primary circuit is not able to provide power steering, then the vehicle should be considered as moving without primary power steering.  
  
 4. If the vehicle is considered as moving without primary power steering, then the secondary circuit have to take over the power steering.  
  
 5. If the power steering is provided by the secondary circuit and the parking brake is off, then activation of the electric motor is necessary.

**5.2 Relate Code**

#define VEH\_MOVING\_LIMIT 3  
#define TRUE 1  
#define FALSE 0  
  
typedef enum  
{  
 WORKING,  
 NO\_FLOW,  
 SHORT\_CIRCUIT,  
} SENSOR\_status;  
  
typedef struct  
{  
 int wheel\_speed;  
 int parking\_brake;  
 int prim\_low\_flow;  
 int prim\_high\_volt;  
 int second\_circ\_handles\_stee;  
 int elec\_motor\_act;  
} VEHICLE\_INFO;  
  
typedef enum  
{  
 PARKING\_BRAKE\_APPLIED,  
 PRIMARY\_CIRCUIT\_LOW\_FLOW,  
 PRIMARY\_CIRCUIT\_HIGH\_VOLTAGE,  
 WHEEL\_BASED\_SPEED,  
 SECONDARY\_CIRCUIT\_HANDLES\_STEERING,  
 ELECTRIC\_MOTOR\_ACTIVATED,  
 NUM\_SIGNALS  
} SIGNAL;  
  
int status[NUM\_SIGNALS];  
/\*Reads the specific signal from the status.\*/  
  
int read(SIGNAL i)  
{  
 if(i < NUM\_SIGNALS)  
 {  
 return status[i];  
 }  
}  
  
/\*Writes the specific signal to the status.\*/  
  
void write(SIGNAL i, int value)  
{  
 if(i < NUM\_SIGNALS)  
 {  
 status[i] = value;  
 }  
}

/\*Get the current status of the system.\*/

void get\_system\_status(VEHICLE\_INFO \*veh\_info)  
{  
 veh\_info->wheel\_speed = read(WHEEL\_BASED\_SPEED);  
 veh\_info->parking\_brake = read(PARKING\_BRAKE\_APPLIED);  
 veh\_info->prim\_low\_flow = read(PRIMARY\_CIRCUIT\_LOW\_FLOW);  
 veh\_info->prim\_high\_volt = read(PRIMARY\_CIRCUIT\_HIGH\_VOLTAGE);  
 veh\_info->second\_circ\_handles\_stee = read(SECONDARY\_CIRCUIT\_HANDLES\_STEERING);  
 veh\_info->elec\_motor\_act = read(ELECTRIC\_MOTOR\_ACTIVATED);  
}

/\*Check the status of the primary steering circuit sensors.\*/

void check\_prim\_sensor\_status(VEHICLE\_INFO \*veh\_info, SENSOR\_status \*sensor\_status)  
{  
 if(veh\_info->prim\_high\_volt == TRUE)  
 {  
 \*sensor\_status = SHORT\_CIRCUIT;  
 }  
 else if(veh\_info->prim\_low\_flow == TRUE)  
 {  
 \*sensor\_status = NO\_FLOW;  
 }  
 else  
 {  
 \*sensor\_status = WORKING;  
 }  
}

/\*Check if the secondary circult need to handle the steering. \*/

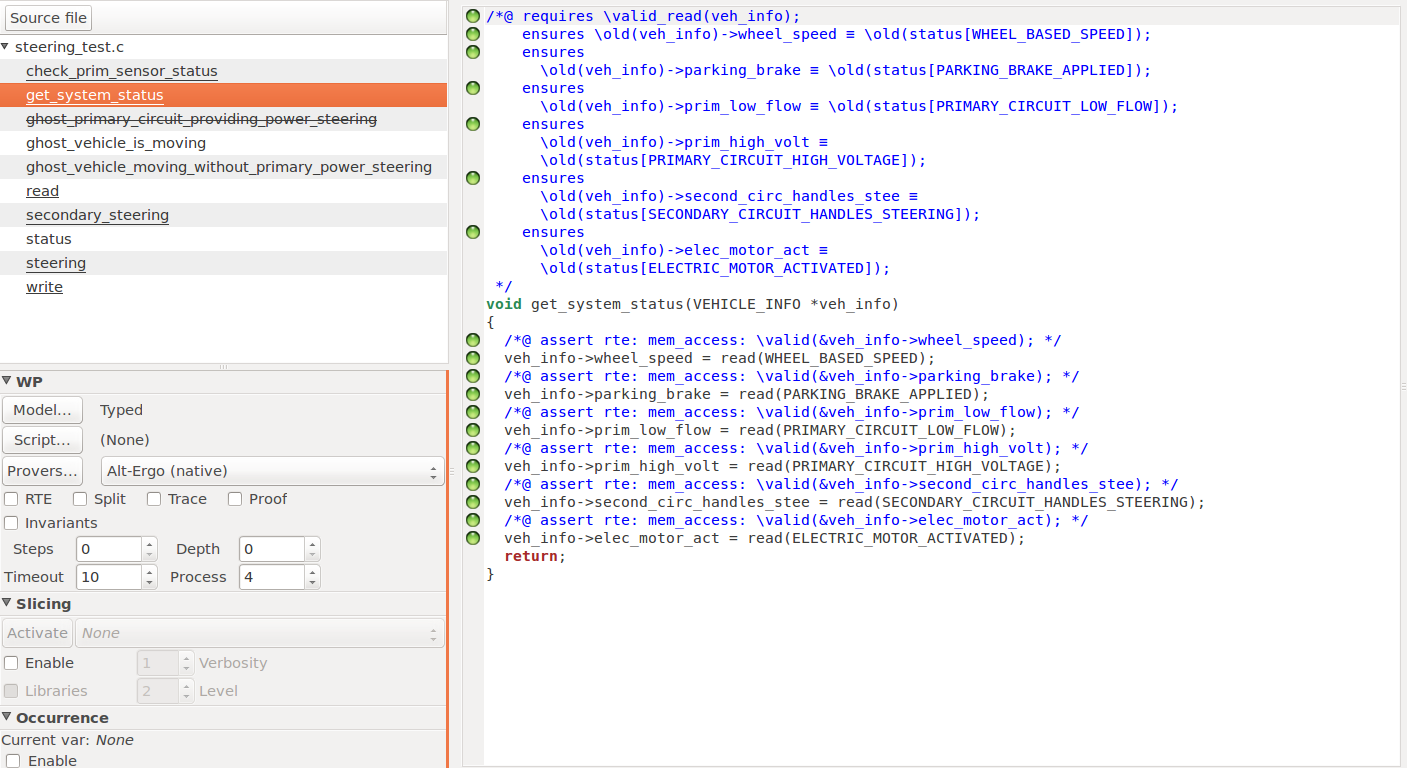
void secondary\_steering(VEHICLE\_INFO \*veh\_info,SENSOR\_status \*sensor\_status)  
{  
 char vehicleIsMoving;  
 char vehicleIsMovingWithoutPrimaryPowerSteering;  
  
 // Check whether the vehicle is moving.  
 if(veh\_info->wheel\_speed > VEH\_MOVING\_LIMIT)  
 {  
 vehicleIsMoving = TRUE;  
 }  
 else  
 {  
 vehicleIsMoving = FALSE;  
 }  
  
 // Check whether vehicle is moving without primary power steering.  
 if(vehicleIsMoving == TRUE && (\*sensor\_status == NO\_FLOW || \*sensor\_status == SHORT\_CIRCUIT))  
 {  
 vehicleIsMovingWithoutPrimaryPowerSteering = TRUE;  
 }  
 else  
 {  
 vehicleIsMovingWithoutPrimaryPowerSteering = FALSE;  
 }  
   
 // Let secondary circuit handle steering if necessary.  
 if(vehicleIsMovingWithoutPrimaryPowerSteering == TRUE)  
 {  
 veh\_info->second\_circ\_handles\_stee = TRUE;  
 }  
  
 // Activate the electric motor.  
 if(veh\_info->second\_circ\_handles\_stee==TRUE&& veh\_info->parking\_brake == FALSE)  
 {  
 veh\_info->elec\_motor\_act = TRUE;  
 }  
}

/\*Entry point of steering module.\*/  
void steering()  
{  
 VEHICLE\_INFO veh\_info;  
 SENSOR\_status prim\_sensor;  
 get\_system\_status(&veh\_info);  
 check\_prim\_sensor\_status(&veh\_info, &prim\_sensor);  
 secondary\_steering(&veh\_info, &prim\_sensor);  
 write(SECONDARY\_CIRCUIT\_HANDLES\_STEERING,veh\_info.second\_ci rc\_handles\_stee);  
 write(ELECTRIC\_MOTOR\_ACTIVATED, veh\_info.elec\_motor\_act);  
}

**5.3 Formal verification**For this study case, we can easy see that there had many variables and related function, and that all related code was worked for the automobile operation status, ensure the automobile in diffident status keep working.   
  
About we how to use Frama-C for that, we need type the code to command in Linux:  
  
frama-c-gui -wp -rte -val -main=steering steering\_test.c

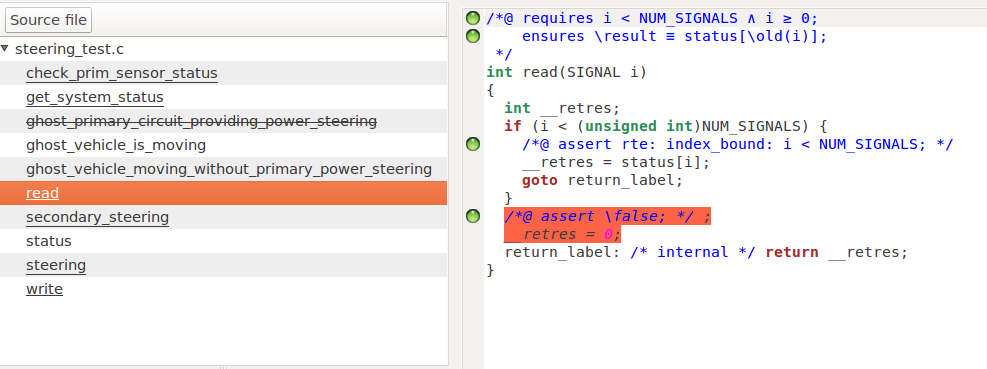
Now I explained what their means, ‘-wp’ is the call of the WP (weakest precondition), ‘-rte’ is the called RTE (runtime error, which used in generating annotations) and ‘-val’ is the called VAL(value analysis) plug-ins and -main=steering is directing the steering function in the code as the main function in order to let the VAL plug-in works.

Before we begin, let we introduce the graphical user interface of Frama-C. The upper left section is all the functions and ghosting variables identified in steering\_test.c by the slicing plug-in. The session in the lower left corner is the control panel for other plug-ins, such as WP and slicing. Finally, the panel on the right is how Frama-C reads the source code using our clauses and generated comments. The circle next to the line of code represents the state of each clause and comment. Green means verified, and yellow means the plug-in cannot be verified.

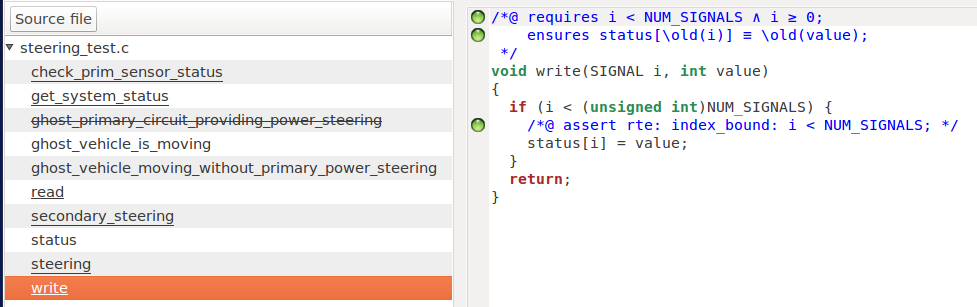
In the screen shot below (the get\_system\_status function in the code), we set the prerequisites for a valid memory allocation with veh\_info, which looks up the value from the signal through the read function of the requires clause (mentioned in the next section). The next assurance clause is the postcondition, which verifies that the value is properly assigned by calling the read function. Finally, the assert-rte that generates the correct memory address allocation annotation for the veh\_info variable.  
  


***Figure 5: Frama-C Application***

Here are the write and read functions mentioned in the previous function. In the same way, this function will only work in the range of I from 0 to the total number of data type signals to get the correct value. And make sure it returns the correct value from the state array. The assertion comment validates the index range of the state array again. Writing clauses in a function is like reading a function.

Note that there is a section highlighted in red, which is an error detected by the Val plug-in. In the source code, Val considers a part missing from the function and automatically adds it.  


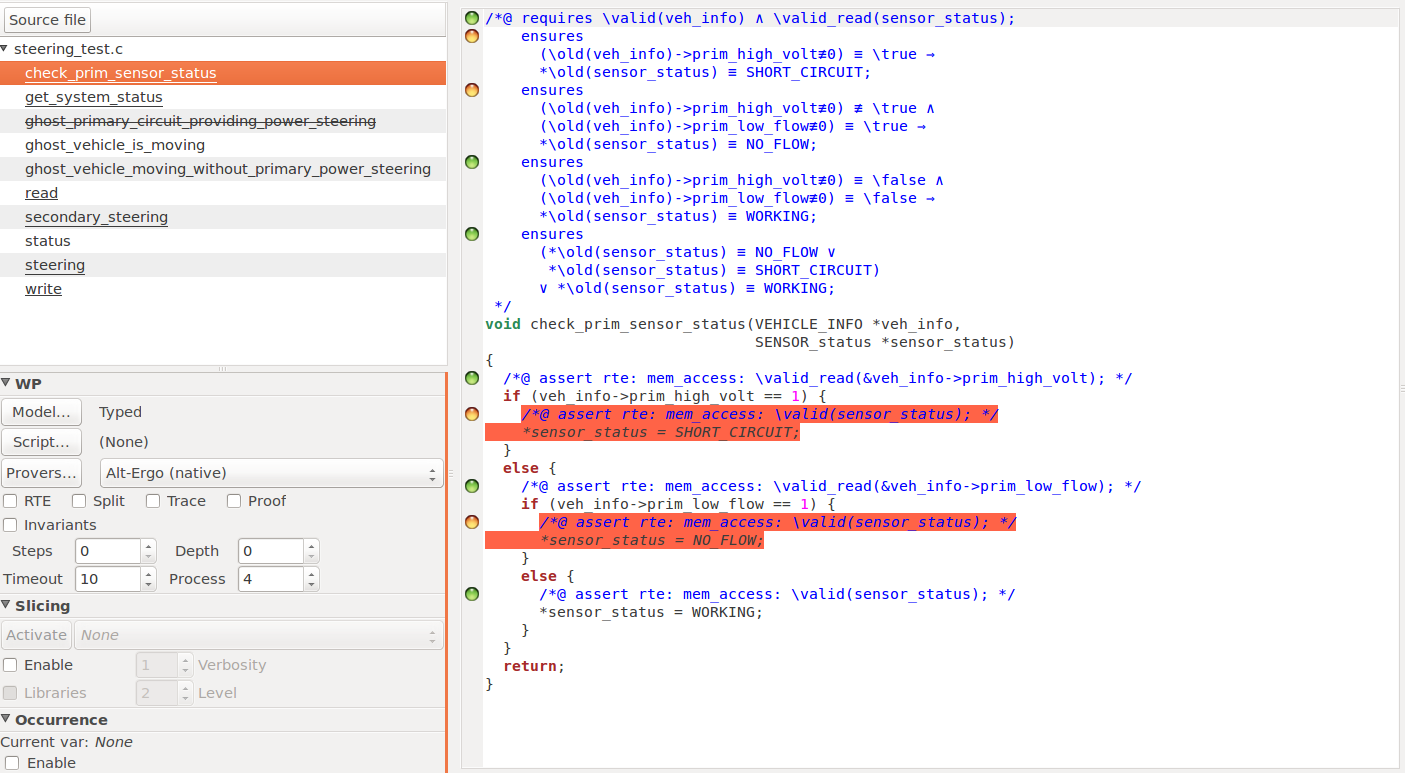
***Figure 6: Frama-C Application***



***Figure 7: Frama-C Application***

The next function is check\_ prim\_ sensor\_ Status function. The assert annotation requires and ensures that clauses are similar to those in the previous and upcoming functions, so interpretation will be skipped.

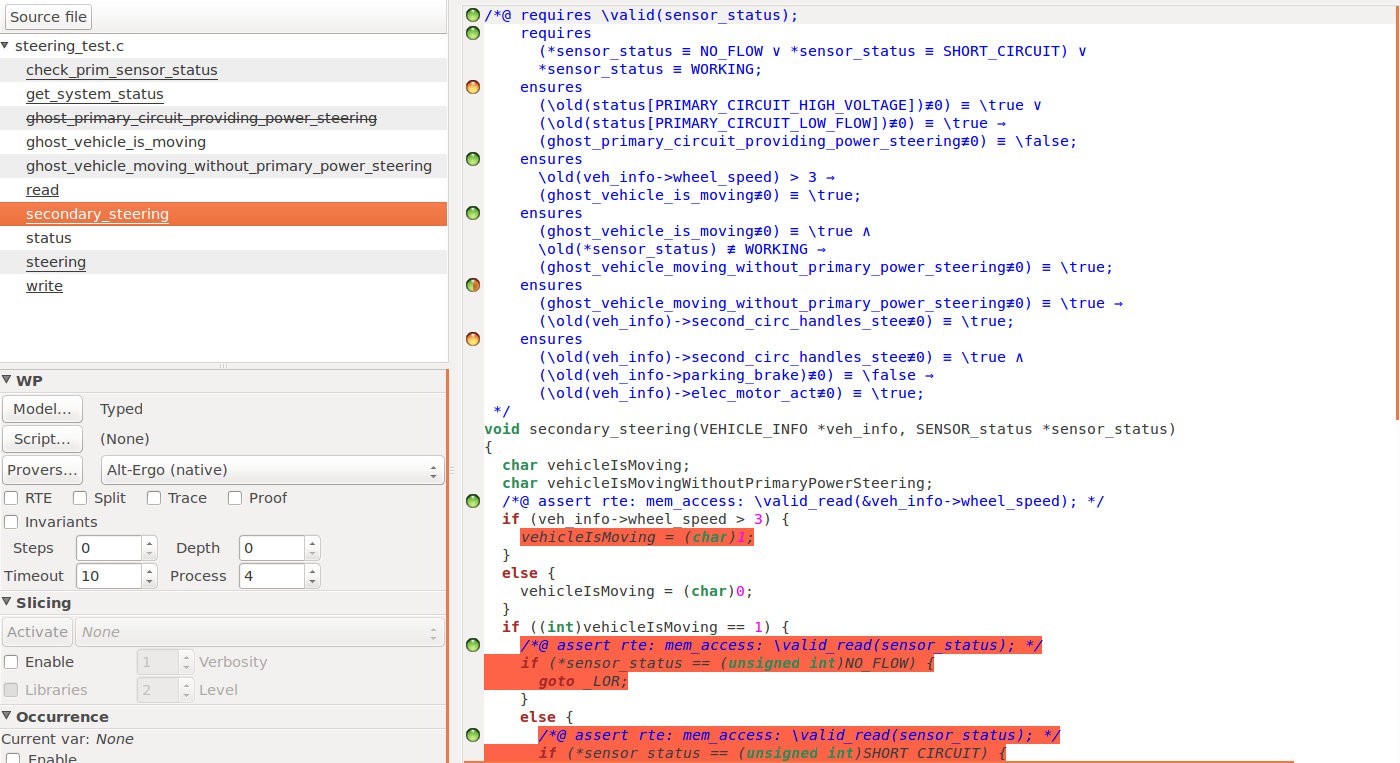
Pay attention to the highlighted parts in red. They are the code that the tool thinks is dead and has not been executed in the verification of Val plug-in. The value sent when this function is called. Run only the last if - else part of the code, and val issues a warning to notify the user. For similar reasons, codes with orange circles refer to both cases and results as undefined proofs because they are dead and cannot ensure the allocation of values.



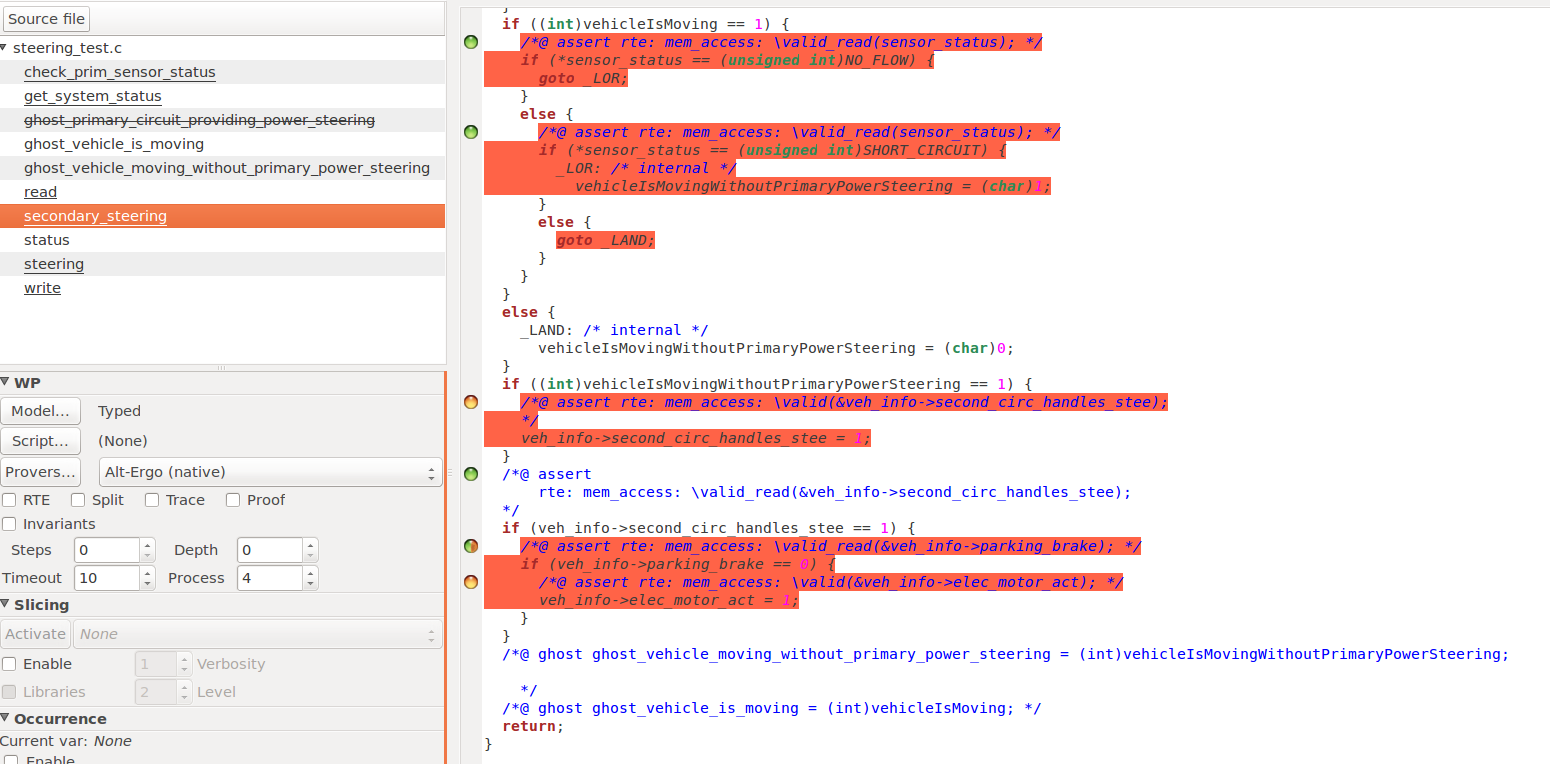
***Figure 8: Frama-C Application***

The next two screenshots are the secondary\_steering function. The guarantee clause in this function is for her to verify the five functional requirements we received. This is the entire module of the curial, as these five requirements keep the vehicle working properly and prevent possible serious accidents.

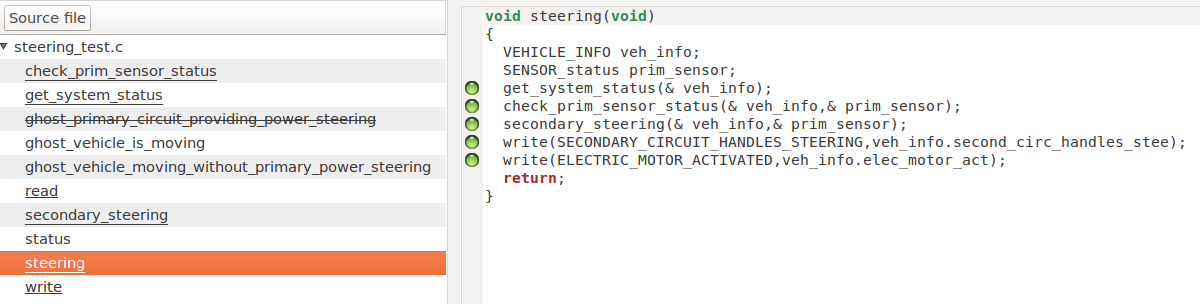
The ghost variable declared at the beginning of validation is also used in this function to double-check the original code. Because this function is based on if-else conditions, like check\_ prim\_ sensor\_ Like the status function, the Val plug-in will notice some dead code. Therefore, these ensure that clauses and comments cannot be successfully validated by the WP plug-in.

Note that some circles are half green and half orange, which means that validators in the WP plug-in can only partially validate. In this case, we have to look at Frama-C's console to find out which molecular sentence failed and find out the reason for the failure.

***Figure 9: Frama-C Application***

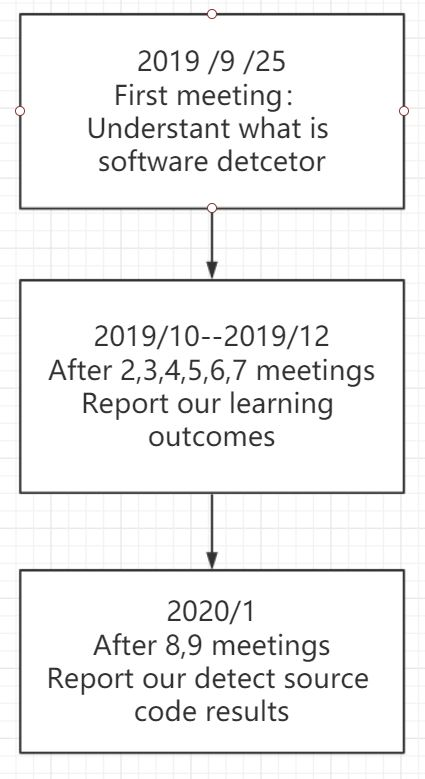


***Figure 10: Frama-C Application***

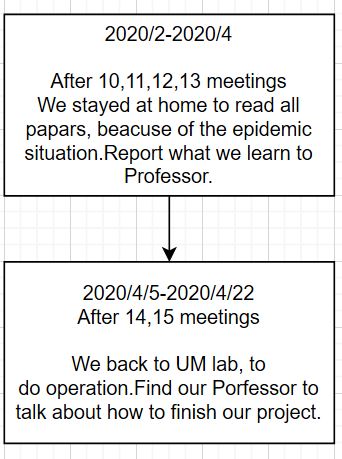
Finally, this is the last piece of functionality and the entire steering module that calls the other functions. This function is also set as the main function of the VAL plug-in. If this step is not performed, the validation of the vehicle information and prim\_ sensor will fail, because VAL will automatically validate the code in a top-down fashion by default. Then the declaration of these two variables will be in the last step. Therefore, in the functions explained earlier, most annotation validations fail because the memory location cannot be accessed. Note that all the functions shown here are green circles because the WP plug-in has successfully executed this function in the stream, although partial validation in the function has not yet been determined.   


***Figure 11: Frama-C Application***

**CHAPTER 6. A FLOW CHART OF THE SCHEDULE**

****

***Figure 12: Flow chart of the first semester schedule***

******

***Figure 13: Flow chart of the second semester schedule***

**CHAPTER 7. CONTRIBUTIONS**

Most of time, we learned together. We found some tutorial each other in home and after we put it in to our FYP computer. And we would find any time we not busy to learn together.

About paper work, we would have our contributions:

TONG: Contact the professor, part of meeting notes, part of Final report.

ANSON: Check meeting notes, part of Interim report,part of Final report.

**CHAPTER 8. CONCLUSION AND FUTURE WORK**

In conclusion, this project was not easy. It was cost a lot of time to learn the all kinds of knowledge of software verification. Because of it involved many aspects. As a year four computer science student, we did not learn more about that before. But it was always important in computer science even more occupation.

This project just show a little parts of these. In the future, we may learn more about another language verification, like Java, Python. In our society, their would be more language can be our challenges. There would be a working never stop. Everything is progressing, the bugs and errors also. Therefore, it need more software versifier in the future. Now in China, I can not see there pay attention to this and make serious, but in foreign countries, they were already running. We need to spent more time for this.

**CHAPTER 9. REFERENCES**

[1] “Towards Secure Things,or How to Verify IoT Software with Frama-C” [Tutorial at ZINC 2018].[Online]. Available: [http://nikolai-kosmatov.eu/publications/tutorial\_2018\_05\_30\_zinc2018.pdf](http://www.endnote.com/)

[2] Allan Blanchard. “Introduction to C program proof with Frama-C and its WP plugin” June 12, 2019(Beta)[Online]. Available: [http://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf](http://www.endnote.com/)

[3 ]Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye. “Frama-C / WP Official Tutorial”. Available: <https://frama-c.com/download/frama-c-wp-manual.pdf>

[4] Nikolai Kosmatov. “Combined Static and Dynamic Analyses in Frama-C: An Overview” March 20 2018. Available: <https://nikolai-kosmatov.eu/publications/tutorial_2018_03_20_costrv2018.pdf>