**University of Macau  
Faculty of Science and Technology**



**Detecting Software Defects Using Verification Tools**

Final Report

***By*  
ANSON CHIU, DB626242**

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 : ANSON CHIU

Name : ANSON CHIU

Student No. : DB626242

Date : 29 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. Within all of the software applications, Internet of Things (IoT) applications are now the most important type of applications since it will link with machines and vehicles that could affect the reality and deal with the user’s personal information. Hence, securing the applications and avoiding any bugs and errors are crucial in these years. Software detection is a subject for verifications and finding out the possible errors that may occur in any software, such as IoT applications. Especially for those in critical fields such as medical systems and the railway transportation systems which always have human-beings’ life involved.

**TABLE OF CONTENTS**

1. **INTRODUCTION 6  
   1.1 Background   
   1.2 Advantages   
   1.3 Target  
   1.4 Problem Encountered**
2. **SOFTWARE DETECTION METHOLOGIES 8  
   2.1 Formal verification  
   2.2 Criteria for formal verification  
   2.3 Techniques of formal verification  
   2.4 Introduction of static analysis**
3. **VERIFICATION TOOLS – FRAMA-C 11  
   3.1 Introduction of Frama-C  
   3.2 Weakest per-condition  
   3.3 Value analysis plug-in based on Abstract Interpretation  
   3.4 Slicing plug-in based on program flow analysis  
   3.5 Annotations**
4. **CASE STUDY: VERIFICATION OF STEERING MODULE IN SCANIA 13  
   4.1 Verified Code  
   4.2 Functional requirements   
   4.3 Verified result**
5. **REFERENCES 22**
6. **Introduction**
   1. **Background**

As the society is now rapidly developing, our life is more relied on software applications. Applications on everyone’s mobile phone, computers, which may deal with our sensitive personal information and data and causing serious security problems when there is a leak happened. Safety problems on the Internet of Things (IoT) applications are another potential problem. IoT application are widely used in our daily life, such as in your vehicles, the lock on your door and even in some critical system like the railways and medical devices. Most of the data in IoT applications would process through a complex algorithm. If these systems are insecure and having errors while running, our life may experience a serious threat, causing serious safety problems.

In order to solve the potential safety and security problems in the software applications, these problems must be tested before the application is published. However, with the complex algorithm within the codes, testing through all of the possible ways of the code become extremely inefficient and even impossible. As a result, formal verification is found and become the solution for solving safety and security problems in the application.

* 1. **Advantages**

Within the process of software development, effectiveness and efficiency are the most important factors. With the limited budget for the development process, it is difficult to test all the possibilities of the program before publishing. However, by using formal verification tools, the verification process has become easy and can effectively evaluate the security and safety issues in the program and avoid them before the publishing. Although the process may take more time if the program has to pass through the verification process, securing the program to function properly is more important.

* 1. **Target**

This report is aimed to describe different methods to verify a program and study the method to verify a source code by using some modern formal verification tools. Finally perform a verification on an actual code of some system under their own requirements, try to discover the potential problems and avoiding them.

* 1. **Problem Encountered**

At the beginning of the project, since both of us did not know much about software detection and verification. We need to study the related knowledge which may be used in the detection process in order to perform a verification on a real code. 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 in order to perform the verification.

1. **SOFTWARE DETECTION METHOLOGIES**
   1. **Formal verification**

Formal verification is brunch of specifications that are decided by the programmer of how a specific program should work. These specifications work just like a blueprint of the program, not only the programmer could follow it and build up the codes, but also the verification tools to evaluate is the code meting the specifications. In many cases, the specification and the code itself would not match up which may later cause defects that lead to breakdowns, malfunctions and cyberattacks, and causing serious problem if the software is processing our personal information. Within the verification process, the programmer could first translate the specification into coded instruction for the verification tools to read and evaluate. Then we can analyze the properties of the code and secure there have no data loss during the translation and execution, and make sure the software is doing exactly what it should do.

During the verification process, both the codes and the specifications are translated into mathematical representations by the verification tool, which are then check against each other by mathematical proof. Then if any mismatching parts are discovered, means that the related codes are not functioning properly. And by correcting the mismatching parts, the code could then be mixed to what it should do. These seems easy to find out the possible flaws in a small program. However, as the software applications are now growing both in scale and complexity, looking through all the flaw will become as difficult as walking through all the routes in a complicated maze. If we use conventional testing on the verification process, it may take us thousands of years to run through all the possible defects. But with formal verification, we could evaluate all possible scenarios in the entire state space all at once, as a result, we could eliminate the entire classes of defects hence dramatically improving the safety and security of the software. Also decrease the cost and time for the software development.

* 1. **Criteria for formal verification**

There are four criteria which formal verification could be classify into:

1. Coding standards  
   Which are set of rules or good habits the programmer should follow in the process of the development, such as formatting style, constructs and programming language. These standards could be changed within different programing platforms, industries and organizations. Some well-known coding standards are CERT C Coding Standard for C language and MISRA C (Motor Industry Software Reliability Association).
2. Software metrics  
   Which is the standard of measure of the extent to some measurable property that the software have, such as efficiency, complexity, maintainability, reliability and testability. There are also other metrics which some object-oriented languages should have, such as the amount of source code elements, levels of inheritance and interactions between the objects and methods.
3. Fault patterns  
   Which are group of software defects, errors, or faults the usually seen, such as memory related errors, path resolution errors and user interface (UI) errors. These kinds of errors could be verifying by using static analysis tools, which will be further describe in the following.
4. Runtime failures  
   Which are the errors detected when the code is executing, such as array index out-of-bound, division by zero, dead code and null pointer.
   1. **Techniques of formal verification**

There are three techniques for verify a program.

1. Manual Review.  
   This technique is the easiest among three techniques but this the first step to verify a developing program. By looking through the codes to find out the section that is not meeting the first criteria of formal verification – coding standard. Since there are various of standards among industries and organizations. Using verification tool to look for them is not practical. During the process, every programmer could also check the developing program to look for some simple mistakes like some common errors. Programmers could also examine the boundary for any errors and the data flow through the program to look for any simple mistakes. However, this would take a lot of time and effort to run through all the possibilities and verify through the whole program. And here come the efficient methods to perform it.
2. Static analysis  
   This technique is to examine the code for any possible errors but without executing the codes. With static analysis, examine of code metrics, fault patterns and possible runtime errors could be much easier and saving a huge amount of time compare to manual review. As an example, memory allocations are those problems that programmers could not discovered easily since we often do not know how the processer will assign them. But with the help of verification tools, the memory assignment could be track easily and as a result, possible error could be discovered and avoid by changing the code.
3. Dynamic analysis  
   This technique scan through the program source code and testing the program with different input values with analysis tool. The main objectives are to check for potential errors that may occur at runtime, examine the behavior of each variable and monitoring the performance. This technique will need more time than static analysis due to the actual execution of the code is needed. However, there are limited amount of problems with the static analysis performed only. Some of the example tools of performing dynamic analysis are VB Watch (used for Visual Basic). Iron.js (used for JavaScript) and Purify (used for memory issues).
   1. **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.

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.

1. **VERIFICATION TOOLS – FRAMA-C**

**3.1 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.

Frama-C is a verification tool for C codes which have many available plug-ins for any advanced verification. Within all the available plug-ins, we mainly focus on two of them since these are the very essential elements in the verification of software: Eva (Evolved Value analysis), which is useful in the verification of the effective domain for the variables. 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. The Second plug-in is Wp (weakest preconditions), which is aimed to prove the users’ determinate precondition and postconditions for any functions. 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 Weakest per-condition**

Weakest per-condition (WP) is very important plug-in for software verification in Frama-C. We used its principle to verify the code if not satisfied the functional requirements.

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 per-condition W from the function body and the post-condition Q, and then try to voluntarily prove that the per-condition P implies W.

In the operation of Frama-C, we use an annotation ‘requires’ representing per-condition, ‘ensure’ representing post-condition.

**3.3 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.

**3.4 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.

**3.5 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 cannot 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.

1. **CASE STUDY – VERIFICATION OF STEERING MODULE IN SCANIA**

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.

* 1. **Verified 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);

* 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.

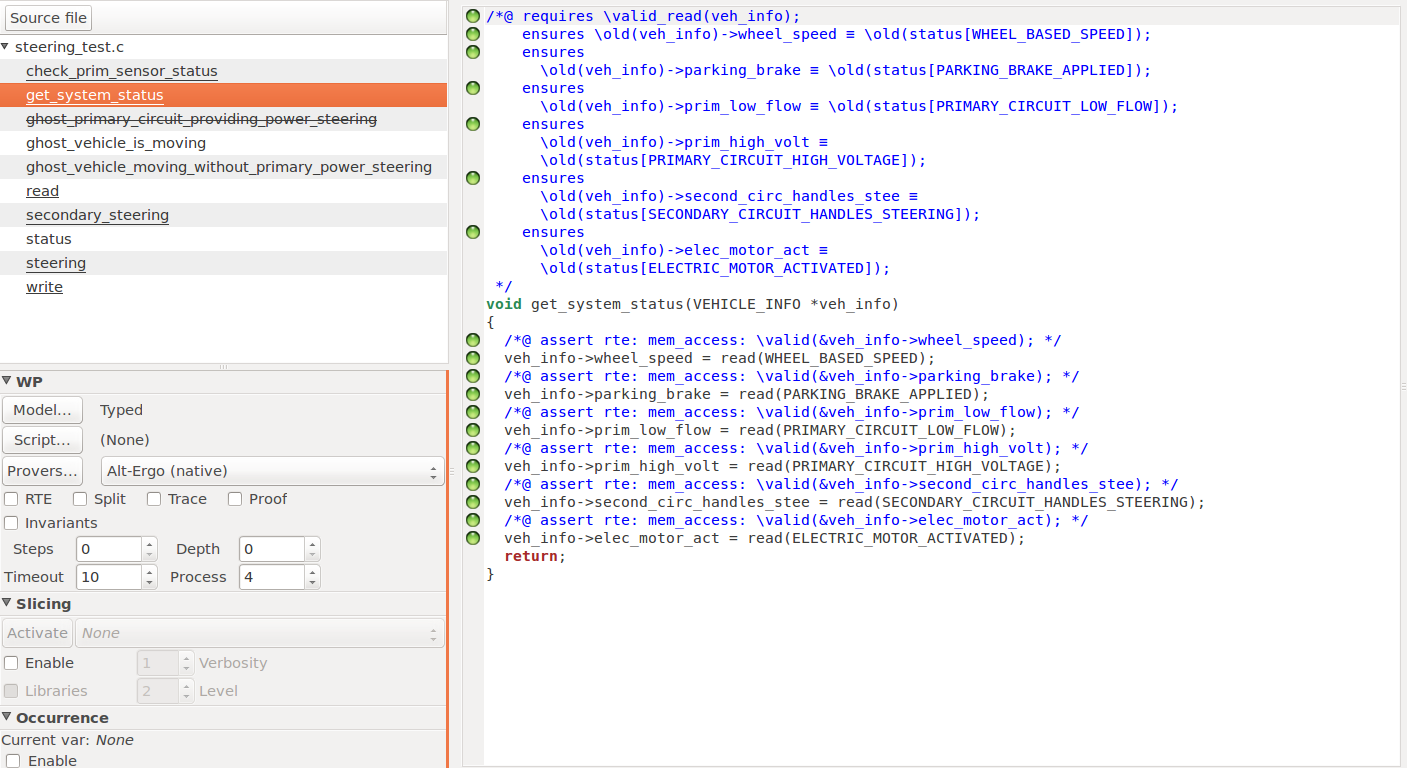
The requirements below are the module should be stick to.

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.
   1. **Verified result**

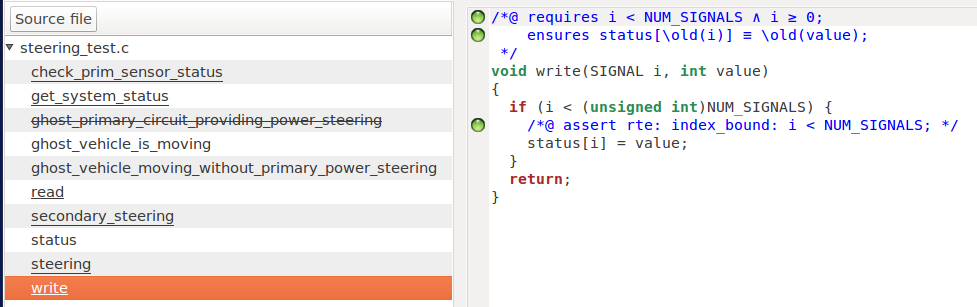
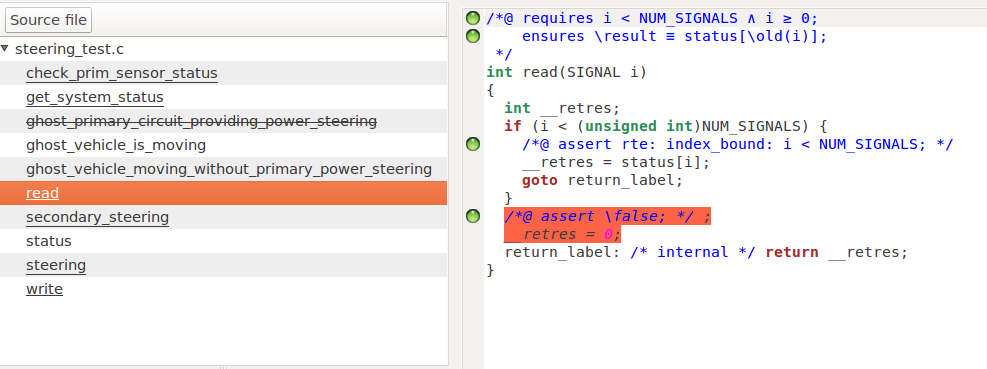
This code is processed through the command in Linux cmd:  
frama-c-gui -wp -rte -val -main=steering steering\_test.c

-wp, -rte, -val is the call of the WP (weakest precondition), RTE (runtime error, which used in generating annotations) and 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 start to express our understandings, let us introduce about the GUI of Frama-C first. The upper left section is all function and ghost variables identified in steering\_test.c by the slicing plug-in. The bottom left session is the control panel of other plug-ins such as WP and Slicing. Finally, the right panel is how the Frama-C reads the source codes with our clauses and the annotations generated. The circles next to the line of code represent the state of each clauses and the annotations. Green in color means passes the verification, while yellow in color represents that the plug-ins could not verify.

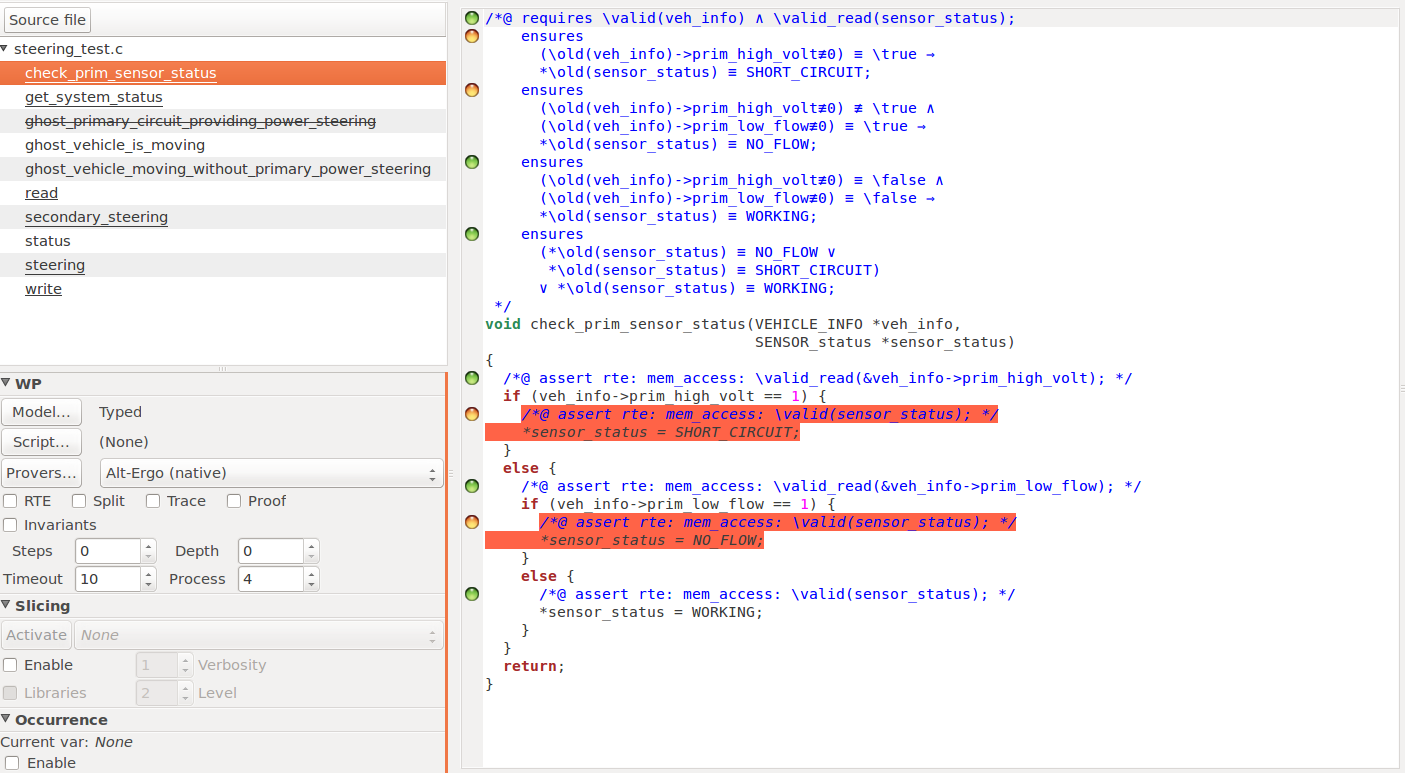
In the screenshot below, which is the get\_system\_status function from the code, we set up a precondition for a valid memory allocation with veh\_info, which this function will look for the value from the signal through the read function (that would be mention in the next part) by the requires clauses. Next the ensures clauses is the post conditions, which is verify the value is correctly assigned through the call of read function. Finally, for the assert rte that generates the annotation of the correctly assign of the memory address for the veh\_info variables.

Below is the write function and read function that mention in the pervious function. With the same fashion, this function will only work under the precondition that i is within the range of 0 to the total number of signals in the data type SIGNALS in order to obtain a correct value. And make sure it will return the correct values form the status array. Assert annotation to the verify the index range of the array of status again. And clauses in the write function is just like the read function.

Noted that there is section highlighted in red, which is the error detected by the VAL plug-in. In the source codes, VAL consider that a part is missing in the function and add it in automatically. 

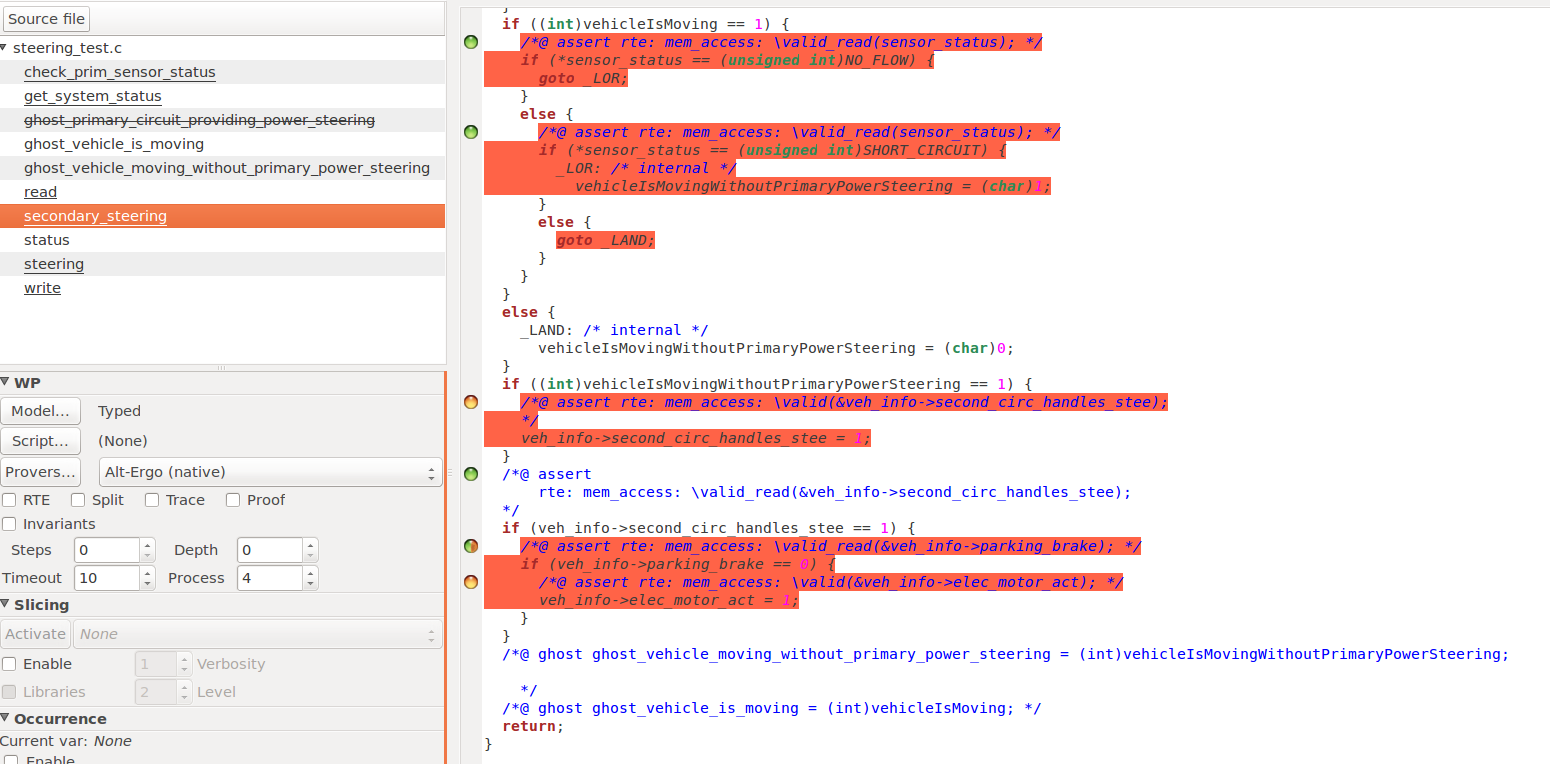
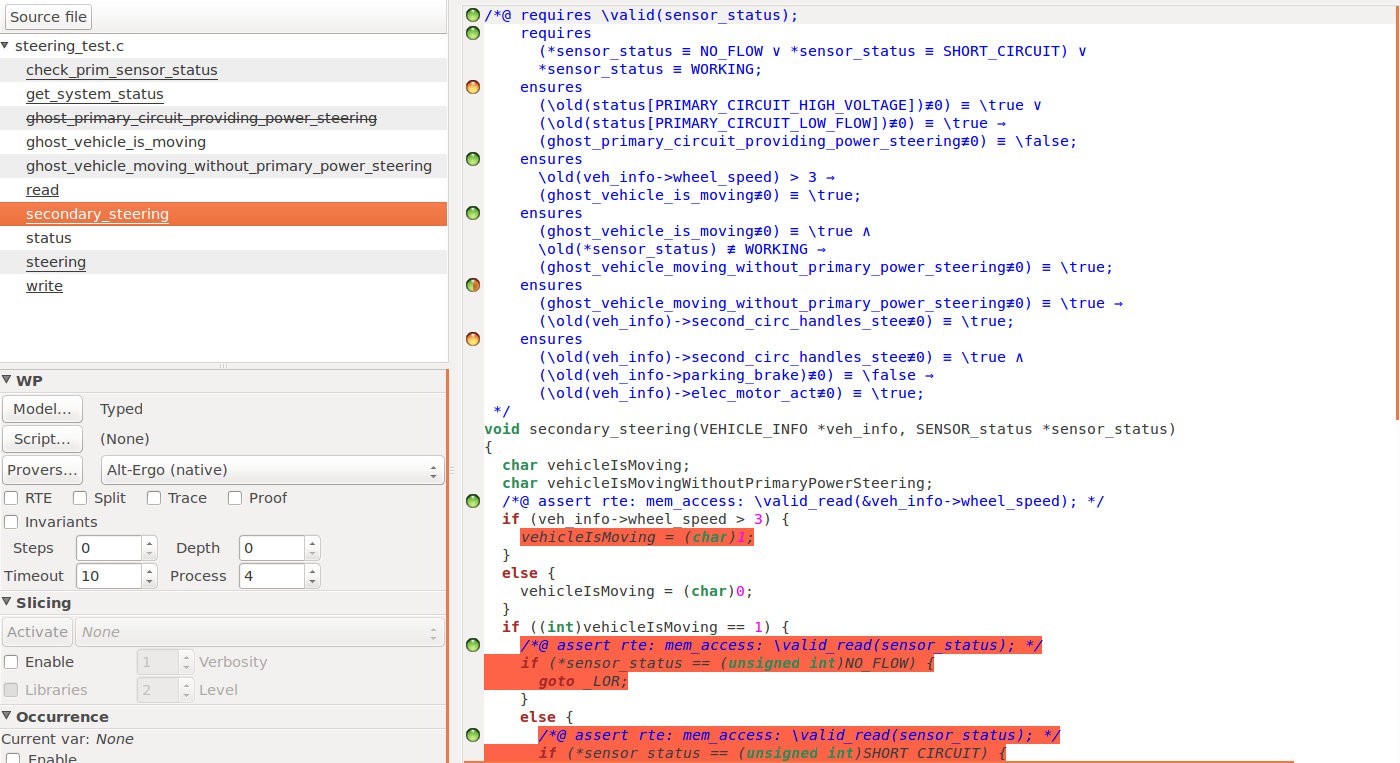
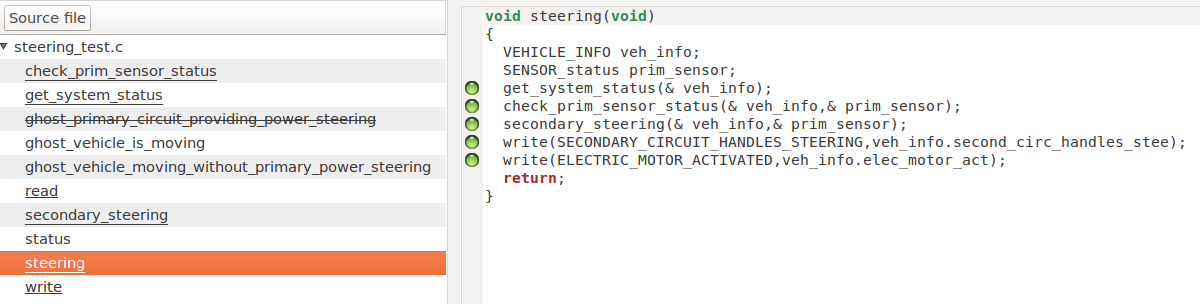
Next function is the check\_prim\_sensor\_status function. The assert annotations, requires and ensures clauses is in a similar fashion in the previous and coming functions, so explanation will be skipped.

Focus on the red highlights, those are the code that the tool considers dead, which has not been executed in the verification by the VAL plug-in. With the value sent in as this function is called. Only the last if- else section in the code runs and VAL sending an alert out to note the users. With the similar reason, those code with orange circles refers to those two cases and results as undefined proof since those code are dead and cannot ensures the assign of values.



Following two screenshots are the secondary\_steering function. The ensures clauses in this function are her to verify the five function requirements that we received. This is curial in the whole module since that five requirements maintains the vehicle to work properly and prevents serious accident which may result in.

The ghost variables declared in the beginning of the verification also been used In this function to make a double check with the original code. As this function is based on if-else conditions, just like the check\_prim\_sensor\_status function, there are some dead codes noted by the VAL plug-ins. Hence, those ensures clauses and annotations cannot be successfully verified by the WP plug-in.

Noted that there are some circles are half in green and half in orange, which means the prover in WP plug-ins can only partially verified. In this case, we have to look into the console of Frama-C to find out the which part of the clauses are failed and look for the reason behind.   
  
Finally, this the last part of the function, also the caller of the other functions of the whole steering module. This function is also set as the main function for the VAL plug-in. Without this step, the verification of veh\_info and prim\_sensor will be failed as VAL will automatically verify the code in a top-down approach in default. Then the declare of these two variables will be at the last step. Hence, failed most of the annotation verification in the previous explained function due to failed to access the memory locations. Noted that all the function shown here are in green circles because the WP plug-in has successfully executed the function in a flow, despite the fact that part of the verification within the function are undetermined.   


**5 REFERENCES**

1. Whatever Happened to Formal Methods for Security? J. Voas and K. Schaffer, Computer (Long Beach Calif). 2016 Aug; 49(8): 70–79.
2. Ghosts for Lists: A Critical Module of Contiki Veriﬁed in Frama-C. Allan Blanchard1, Nikolai Kosmatov2, and Fr´ed´eric Loulergue3
3. Formal Veriﬁcation of a Memory Allocation Module of Contiki with Frama-C: a Case Study. Fr´ed´eric Mangano1, Simon Duquennoy2,3, and Nikolai Kosmatov1
4. Frama-C’s RTE plug-in for Frama-C 20.0 (Calcium). Philippe Herrmann and Julien Signoles
5. Introduction to C program proof with Frama-C and its WP plugin. Allan Blanchard