DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2016

Verification of Functional Requirements of Embedded Automotive C Code

CHRISTIAN LIDSTRÖM

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

KTH Computer Science
and Communication

Verification of Functional Requirements of Embedded Automotive C Code

Master of Science in Engineering in Computer Science and Engineering
Master of Science in Computer Science
KTH Royal Institute of Technology
School of Computer Science and Communication

CHRISTIAN LIDSTRÖM
clid@kth.se

Master's Thesis at KTH CSC
Supervisor: Dilian Gurov
Examiner: Mads Dam
Principal: Scania CV AB
Course: DA222X

Abstract

Today's vehicles are increasingly controlled by embedded computer systems. Such systems are of safety-critical nature, where an error in the computation could have dire consequences. A common way to ensure that software works is testing, but as the complexity of these systems grows larger it gets harder to ensure enough coverage in the tests.

Another way to ensure that software fulfills its requirements is formal verification, where properties of the code are proven mathematically to hold under certain conditions. Formal verification gives a higher level of confidence in the correctness of code than testing alone, but it is not as widely used within the industry. This project has examined whether current state-of-the-art tools for formal verification are ready to be used to verify real-life safety-critical code.

To answer this, a case study on a module running in Scania's vehicles was performed. Several of the requirements were successfully verified. The thesis also identifies for what type of code and requirements this is possible, and describes a process for how it can be done.

Referat

Verifiering av funktionella krav på inbyggd C-kod i motorfordon

Idag kontrolleras fordon allt mer av inbyggda datorsystem. Sådana system är säkerhetskritiska, där ett fel kan ha ödesdigra konsekvenser. Ett vanligt sätt att försäkra sig om att mjukvaran fungerar är testning, men när komplexiteten av dessa system växer blir det allt svårare att försäkra sig om att testen har tillräcklig täckning.

Ett annat sätt att försäkra sig om att mjukvaran uppfyller dess krav är formell verifiering, där egenskaper hos koden bevisas matematiskt att hålla under vissa villkor. Formell verifiering ger ett högre förtroende för kods korrekthet än vad enbart testning skulle göra, men används ännu inte i lika stor utsträckning inom industrin. Detta projekt har undersökt huruvida moderna verktyg för formell verifiering är mogna att användas för att verifiera riktig säkerhetskritisk kod.

För att svara på detta har en fallstudie av en modul i Scanias fordon genomförts. Flera av dess krav lyckades verifieras. Rapporten identifierar också för vilka typer av kod och krav detta är möjligt, och beskriver en process för hur det kan utföras.

Contents

Listings

Chapter 1

Introduction

This chapter introduces the problem, objective and method of this thesis.

1.1 Background

Formal verification refers to the process of proving properties of code, and it is an active and important area of research. There are many tools available for this purpose, both of commercial and academic nature. Most of these tools apply some form of Hoare logic to specify annotations in the source code. These annotations can then be verified by performing a static analysis of the code, often with techniques building on weakest precondition calculus. Two such tools are VCC, developed by Microsoft Research, and Frama-C, developed by French research institutes CEA and Inria. These tools have been used to prove functional properties of software in a number of projects, and this thesis aims to study their application in real life scenarios further.

Scania is a well known company, mostly associated with the trucks and buses that they manufacture. Functions in these vehicles are increasingly being controlled by advanced computer systems, some of which are of safety-critical nature and are expected to adhere to strict safety standards. The embedded programs running in these systems are written in C. Being able to formally prove that they functionally follow all requirements is of great interest to the company making them, especially as the industry is moving towards autonomous vehicles.

In particular, road vehicle manufacturers will soon be expected to adhere to the international standard for functional safety *ISO 26262* [22], which recommends formal verification of safety-critical components.

1.2 Problem

The question that was examined in this thesis is:

How can state-of-the-art verification tools be utilised to verify that embedded C code adheres to the functional requirements imposed by safety standards? Under what limitations on the code and its requirements is this possible?

1.3 Objective

The thesis aimed to identify the extent to which modern tools for formal verification can be used in real-life situations, to verify actual requirements imposed on safety-critical code. This includes identifying what type of requirements and code that can be reasoned about, as well as potential problems that might occur, and how to avoid them.

1.4 Method

To answer the question a case study was performed, where state-of-the-art verification tools were used to try to verify the requirements of a module in a safety-critical embedded system that is used in real life scenarios. The code base used in the case study was the module controlling steering systems in Scania's vehicles, which is part of a larger embedded system written in C.

The tools Frama-C and VCC were initially selected as a starting point, since they are two of the most known and mature tools for verifying C code. Then, a preliminary comparison was performed, in order to choose the tool most suitable for the code base, its requirements, and workflow of the company.

The formal verification process was performed by selecting a set of appropriate requirements and annotating the code such that a tool could verify its correctness. The selection of requirements and annotation process was continuously evaluated as more was learnt about the capabilities of the verification tool.

1.5 Delimitations

The thesis is focused only on the type of code, requirements and issues that are present in the software that is the subject of the case study. This includes only examining the possibilities of verifying sequential code with simple program flows, as well as only describing processes for formal verification of a single C module. Additionally, verification will only be attempted for requirements that can interpreted as functional.

1.6 Contribution

The thesis describes the process of annotating a code base in order to formally verify its requirements, in a way that is compatible with current tools, including methods for decomposing requirements on a module to the level of functions. It also proposes guidelines for how to write C code that is more easily verified, and identifies what limitations the requirements must adhere to to be verifiable.

The results are of value to developers producing safety-critical software, as it will serve as guidance for writing code that can be integrated in a process of continuous formal verification.

It is also of interest to developers of the verification tools themselves, as it provides insight into how the tools are used and what their limitations are in real scenarios.

1.7 Ethics and Sustainability

As embedded automotive software become more complex, the methods for verifying their correctness must become more sophisticated. Formal verification is a more rigorous approach, that provides larger coverage than other commonly used methods, such as testing. By providing insight into how formal verification can be used to ensure that automotive systems adhere to their requirements, this thesis will help make vehicles safer for all road-users, which has both social and economic benefits. Better functioning vehicles will also have less impact on the environment.

1.8 Collaboration

The practical work in the project was done in collaboration with another student, John Eriksson, whose thesis will focus on formalising a requirements model for embedded systems. A summary of the results can be found in section 6.4.

1.9 Structure

Chapter 1 gives a brief introduction to the problem and the methods used.

Chapter 2 presents the relevant theoretical background.

Chapter 3 gives an overview of related work done within the area.

Chapter 4 presents the results of the preliminary comparison of the two verification tools.

Chapter 5 describes the code base and requirements on which the case study was performed.

Chapter 6 discusses what type of requirements can be verified by current tools, and gives a brief overview of a model for formalizing requirements.

Chapter 7 describes the process of producing an annotated code base for verification.

Chapter 8 gives some guidelines for writing easily verifiable code.

Chapter 9 presents the quantitative results of the case study.

Chapter 10 contains the conclusions of the thesis.

Appendix A presents a full example module with requirements and annotated source code.

Chapter 2

Background

This chapter presents the background theory of the thesis.

2.1 Formal Verification

The first serious attempt at providing a basis for being able to formally define the meaning of a program was done in [14], which provided a calculus for reasoning about arbitrary flowcharts.

This calculus was then built upon in [17], which introduced a formal system usually known as Hoare logic that enables reasoning about correctness of programs written in any well-defined language. Central to Hoare logic is the triple:

$$\{P\}S\{Q\}$$

where P and Q are logical expressions and S is a set of statements. The notation states that whenever P (the precondition) holds, executing S will result in a state where Q (the postcondition) is true, if S terminates. This formal system also includes axioms for arithmetic and assignment, as well as inference rules for how to transform and combine logical expressions with the execution of different statements. Thus, this provides a basis for logically proving functional properties of programs.

Most modern verification tools use a notation similar to the above triple to specify function contracts, which then define what conditions will be met after executing a function depending on what state it was executed from.

One of the inference rules presented in the paper is the Rule of Composition:

$$\frac{\{P\}S_1\{R\} \quad \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$$

This rule allows deduction of a new assertion for the execution of S_1 followed by S_2 by proving two partial assertions over S_1 and S_2 separately.

This system was then further expanded upon with the introduction of Weakest Precondition calculus [11] (or Predicate Transformer semantics). We say that if

$P \Rightarrow Q$ then Q is weaker then P. The weakest precondition $W = \text{wp}(S, Q)$ is a condition that is necessary and sufficient to reach Q after S has been executed, or in other words, for all P such that $\{P\}S\{Q\}$ holds then $P \Rightarrow W$. Since only one weakest precondition for any pair S, Q can exist this calculus is functional, while the Hoare calculus is merely relational, and the following expression always holds:

$$\{\text{wp}(S, Q)\}S\{Q\}$$

This technique is also central to formal program verification, as tools usually verify a function f by calculating the weakest precondition W from the function body and the postcondition Q, and then try to automatically prove that the precondition P implies W. Formalised as a Hoare rule it can be stated as:

$$\frac{(P \Rightarrow W) \quad \{W\}f\{Q\}}{\{P\}f\{Q\}}$$

These simple rules and transformer semantics are not efficient enough to directly be used to automatically modern complex programs, and much progress has been made since their introduction [20].

Another problem is that these formal systems only capture mathematical properties, which do not correlate with the real world systems that the formal model is applied to. In real systems a stronger model is needed, that capture issues such as invalid memory and aliasing.

Even ignoring the above complication, not all properties that hold in an execution can be proven to do so. A common example is the halting problem, which was introduced in [23]. It is the problem of determining whether a given program will at some point stop running. The paper proved that the halting problem is undecidable, that is no algorithm can exist that solves the problem for all inputs.

2.2 Embedded Systems

The code base this case study is performed on is part of an automotive embedded system. An embedded system is usually a micro-processor based system designed to perform a specific function or set of functions, as opposed to a general purpose system that is programmable by the user [16]. Embedded systems are commonly used to control real-time and safety-critical software, such as an automotive system. Specifically the case study will be performed on the module controlling the steering system of automotive vehicles, which is both safety-critical and real-time.

Such an embedded system may also be described as cyber-physical system (CPS). A CPS is an engineered system where the software is tightly and seamlessly integrated with physical components [19]. In the system that is the subject of this case study the software directly controls physical components in the steering system, as well as depends upon data from physical sensors.

An embedded system can often be viewed at as a purely functional relation between the variables of its environment, in this case a vehicle. It reads a set

of inputs, usually sensors that measures different properties of its surroundings, such as speed or pressure. From this a set of outputs are computed, commonly different types of actuators, which makes the vehicle behave in different ways. This is visualised in figure 2.1.

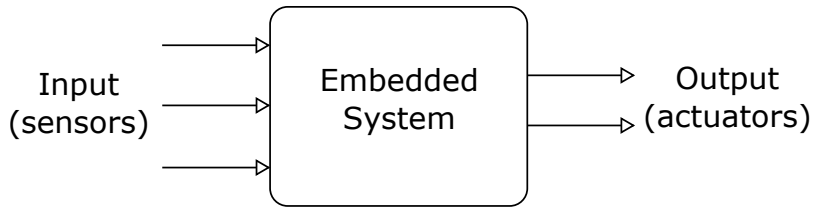

Figure 2.1: Blackbox view of an embedded system as a function computing a set of outputs from a set of inputs.

2.3 The C Language

The software of embedded systems is usually written in C [18], a general-purpose imperative programming language available on almost any platform. It is commonly referred to as a low level language, because it considers objects similar to those of hardware. The fundamental types supported are characters, integers and floating point numbers, which can be organised by arrays, structures, unions and pointers.

Unlike most higher level languages, C only has a weak type system that can be easily circumvented. C also has explicit memory allocation (and deallocation), and memory address arithmetic, and the language allows pointers to different types to be aliased (i.e. chunks of memory pointed to may arbitrarily overlap) [6, 8]. All of this means that that memory safety must be explicitly verified. Most verifiers take advantage of the fact that safety-critical C code is usually written in a type-safe manner, and can leverage this by reasoning about the code in much more strongly typed model. Listing 2.1 contains a small C function that has two aliased pointers of different types referring to memory that is manually allocated and deallocated, all features that are usually not be available in languages with stronger type systems.

```
void main() {
    int * i = malloc(sizeof(int));
    char * c = (char *) i;
    free(i);
}
```

Listing 2.1: C function with aliased pointers of different types.

More current versions of C supports multi-threading, and the embedded system that was used in the case study can as a whole be considered to be concurrent, as many of the physical components are mapped directly to memory. However, intermediate layers exist in the software, so that the individual modules such as the one that was focused on in this project can be considered to be strictly sequential.

2.4 Verification Tools

A wide range of tools for formal verification exist, each for different programming languages and purposes. Since the thesis is focused on verification of safety-critical real-time code, which is usually written in C for performance and portability reasons, the choice of tools was limited to ones supporting this language.

Two of the most well known and widely used tools for formal verification of C code are Frama-C [10] and VCC [6]. Both of these tools are based on compositional verification, i.e. functions are annotated with contracts which specify the properties of the function under certain assumptions that the callers of the function must fulfill. They then use deductive reasoning based on weakest precondition calculus to construct proof obligations, which are then discharged by automated theorem provers, to verify the requirements specified as annotations in the C code.

Before any reasoning is performed the program is first checked for compilation errors by a standard compiler, and then translated into an internal logical representation (usually referred to as a ghost state). This internal state includes logical representations of all real program variables, but also many other abstract variables and functions, in order to provide a stronger model in which to reason about the program. The ghost state is only known to the static verifier, and not to the compiler.

Both tools take advantage of the compositional nature of the underlying theory, in that they perform modular verification. This means that functions are verified separately. When a function that calls another function is verified, the tools just assume that callee's function contract holds, and continues to verify the calling function according to the resulting state.

The annotation languages of both tools are similar in syntax to the C language itself. Many valid C expressions are therefore also valid when used inside annotations, which will be shown in the following sections.

Furthermore, both tools claim to be sound, i.e. that there are no false negatives. This is assuming the absence of any bugs in the implementation.

2.4.1 Frama-C

Frama-C [10, 9] is a free software tool developed by the two French research institutes CEA and Inria. It is available for Linux and Mac, and can be made to work on Windows via emulation. It is under active development, and commercial support is available from external companies. Apart from functional verification Frama-C

also supports many types of analysis, such as value analysis and program slicing, and because of its modular architecture plugins exist for even more functionality.

There are two standard deductive verification plugins that come with Frama-C, Jessie and WP. WP focuses on parametrisation 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, e.g. z3. 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 analysis of strictly sequential code.

Memory Model

WP supports several memory models, with the two main ones currently being the Hoare model and the Typed model [2].

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 correspond 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.

The Typed model is the default memory model. In this model global and local variables each correspond to a single logical variable just as before. However, there is now also support for reads and writes through pointers, which means a distinction must be made between purely local variables and local variables whose addresses have been taken. The latter type must be proven with more care since it is now susceptible to being changed by side-effects in other function calls.

Pointers in this model is generally represented by a pair of a base address and an offset. The heap is now represented by three memory variables, corresponding to arrays of integers, floating point numbers, and addresses. Pointer values are then translated to a location in one of the arrays. This division is the reason for its name, and simplifies the process of distinguishing between separated memory variables. It also has the consequence that casting between pointers of different types cannot be translated.

Annotations

Frama-C uses a specification language called ANSI/ISO C Specification Language (ACSL) [3] for its annotations. ACSL supports many features, e.g. function pre- and postconditions and invariants over loops and data. The annotations are specified in ordinary C comments, starting with the special character @. An example of a small function contract in Frama-C can be seen in listing 2.2.

```
/*@ requires \valid(a) && \valid(b);
    assigns *a, *b;
    ensures *a == \old(*b) && *b == \old(*a);
 */
void swap(int * a, int * b);
```

Listing 2.2: Example of function contract in Frama-C.

In this example, the only precondition is that both pointers `a` and `b` point to a valid memory location. In function contracts, the keyword `\old` denotes the value of the expression before execution, which is always true of expressions in a precondition, while expressions in a postcondition refer to the value after execution unless otherwise specified. Thus, the postcondition ensures that after execution `a` will contain the value that `b` contained before execution, and vice versa. Also, when a function has side effects you must specify what memory may be changed with the `assigns` keyword, so that calling functions can deduce what memory may have been changed and what is guaranteed 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 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 unsafe memory model, some additional annotations (compared to VCC) 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 contains loops the keyword `decreases` can be used to to specify an integer measure, a sequence that will decrease during 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.

2.4.2 VCC

VCC (Verified Concurrent C) [6] is an open source tool developed by Microsoft Research. It is only available for Windows, and provides integration with Visual Studio. Development of the tool was stopped in 2013.

The deductive reasoning in VCC is performed by a program called Boogie. The proof obligations generated by Boogie from annotations are then passed to an SMT solver, the default being z3.

As the name suggests, VCC focuses on verification of concurrent code.

Memory Model

The VCC memory model [8] is stronger but similar to Frama-C:s Typed model. In this model, system memory is seen as a set of typed structs, or objects, which is maintained in ghost memory as a type state keeping track of locations of valid objects. The model guarantees that valid objects are always separated, which has the result that objects of different types will never overlap at all, and VCC is thus able to efficiently take advantage of well-written code.

Because of its support for verification of concurrent code, VCC also has an advanced ownership model [7], so all objects contain information about their ownership and closedness. Threads are only allowed to perform sequential writes to memory of which it is the owner, and sequential reads to memory that it either owns or can be proven to not change. The heap is actually organised just not as a set of objects, but a set of of trees, where each root represents a thread and every edge indicates ownership by the source vertex. Threads then own their local variables, structs own their fields, and so on. For strictly sequential programs this introduces some extra proof obligations concerning memory ownership and mutability.

Annotations

VCC uses its own specification language [21], with annotations enclosed by parenthesis preceded by an underscore, and thus requires a separate header file to be included that removes the annotations during the preprocessing step when compiled. Listing 2.3 shows how a small function contract may look in VCC.

```
#include <vcc.h>
void swap(int * a, int * b)
    _(writes a, b)
    _(ensures *a == \old(*b)  && *b == \old(*a))
    _(decreases 0);
```

Listing 2.3: Example of function contract in VCC.

As in Frama-C `\old` refers to the value of the expression before execution, with similar rules for when an expression is evaluated when the keyword is not present. The postcondition then states the value of the pointers should be swapped. Here there are no explicit precondition (which, again, the keyword `requires` is used for), although `writes`, apart from letting the calling function know that the memory location may be changed, has the implicit requirements that the object is mutable (and thus valid memory) and locally owned.

Much like Frama-C, VCC supports annotations for invariants over loops and data, ghost state manipulation, and global lemmas, and other features like natural arithmetic. Since VCC is mainly focused on verification of concurrent code, addi-

tional annotations are provided that are used to prove ownership and mutability of memory locations in order to establish atomicity.

VCC does not prove termination by default, but requires an annotation with the keyword `decreases` in order to do that. Its usage is similar to that in Frama-C, and used with the argument 0 as in listing 2.3 VCC will will prove that simple loop-free functions halt. VCC also has another keyword, `recursive`, that is used instead when proving termination of recursive functions.

2.5 Functional Requirements for Embedded C Code

The requirements relating to the code base of the case study are mostly written on the module or even system level, and do not include any implementation details, and they are often expressed in something closer to natural language rather than a programming language. Thus, there is a need to identify how these relate to the actual variables of our module's interface, and finally break them down to function level.

In [24] the authors present a framework for modeling of individual components and the architecture of a CPS at all levels of design. This framework supports the structuring and specification of requirements on components based on the idea of contracts, that splits the responsibilities between components and their environment into guarantees which express properties of the components under the assumption that the environment fulfills its responsibilities. All entities of the system, those in software as well as hardware, are considered as elements in the framework, and these elements are then connected by structuring them in a hierarchical manner to form the architecture. The elements are connected by modeling the interaction with other elements, in the form of expressions over the shared variables of their interfaces.

The above framework is shown in [25] to enable compositional verification of sequential but complex C-programs. A C-program or module is modeled as an architecture by parallel decomposition of specifications to a hierarchy of functions as elements with contracts. A case study within an automotive embedded system is also presented.

In [15] the notion of strongest postconditions, another predicate transformer, is examined. A strongest postcondition $\mathrm{sp}(S, P)$ is the strongest predicate Q such that $\{P\}S\{Q\}$ holds, i.e. for all such Q it must be true that $\mathrm{sp}(S, P) \Rightarrow Q$. The paper discusses how symbolic execution can be used to formulate calculation of strongest postconditions, which they show provide a framework that can combine deductive methods for correctness proofs with automatic property checking. This could then provide a basis for automation of the process of annotating C programs from requirements on only the highest level functions, which can then be verified using established tools.

Chapter 3

Related Work

This chapter presents similar studies that have been performed within formal verification.

Both of the examined tools have been the subject of other case studies, which have resulted in a number of more or less successfully verified programs. In fact, VCC was built with the goal of of verifying the Microsoft Hyper-V hypervisor, a virtualiser for Windows, and many decisions in the development of VCC were based on their experiences in this project [6, 8].

In [12] it is examined whether formal specification and verification are suitable to be used in an industrial context. This is done by performing a case study using Frama-C (with WP and Alt-Ergo) to verify safety-critical avionics software. Several aspects are evaluated, such as whether notation and method is within range of an average software engineer, whether the complexity of the formal language is at the same level as the programming language, amount of information available, maturity and stability of the tools, and the kind of feedback provided by the tools. Some obstacles encountered include functions with different behaviour over multiple executions (e.g. by relying on local static variables), formalising input values from subroutines outside the scope of the specification, and redundancy in specifications caused by parts of specifications for lower level functions proliferating to higher level functions. The authors conclude that formalisation of requirements into logic notation is an excellent way to "debug" specifications. They recommend using formal verification only for highly safety-critical code, and only in a team supported by someone with expertise within the area. They also suggest a set of best practices, and workarounds for some of the problems encountered.

Within the same avionics research project another case study was performed [4], in which VCC is used to verify system calls in the microkernel-based operating PikeOS, which is targeted at safety-critical embedded systems. The authors give an overview of their verification process, and describe their model for the underlying hardware, as well as explain how assembly code can be semantically specified so that functions which contains it can be integrated in the verification process. Specifically they show that formal verification tools can be used to prove that large parts of

system calls are exclusively non-concurrent, through the properties of how interrupts are handled.

In [5] a case study utilising Frama-C to verify a critical module of a secure microkernel and virtualisation hypervisor is performed. As the software uses parallel execution, the authors describe a way to simulate this in sequential code, in order to be able to use the chosen tool. They conclude that deductive verification based on automatic theorem provers is a cost effective way to achieve correctness, and also discuss the benefits of combining this approach with interactive proof assistants.

Another case study is [1], where VCC is used to verify a small hypervisor, the results of which were used during the development of VCC. The hypervisor is much simpler than the one the VCC project aimed at verifying. The study presents techniques and frameworks for verifying system correctness with automated methods. An approach to modeling the underlying system architecture in first-order logic is presented. The study concludes that functional systems verification by applying automated methods is a feasible enterprise.

Chapter 4

Tool Comparison: VCC and Frama-C

This chapter discusses the relevant differences found between the two examined verification tools.

One of the first steps in the project was to do a preliminary evaluation of the verification tools, since there was not time to try to verify the full code base in both of the relevant tools. This evaluation was done by verifying a small subset of the real code base, only containing a few functions which had also had much of their code removed from them. While the size of this reduced code base was many times smaller than the original, it was still representative of the full code base in terms of functionality and coding style. The requirements that were verified in this preliminary test were simple functional conditions that could be inferred from examining what the remaining code actually calculated. This was necessary since this subset of the code no longer fulfilled any of the real requirements. From this preliminary test no great functional deficiency could be identified in either tool compared to the other, leading to the conclusion that the type of code that is possible to verify will not depend on the choice of tool. This is supported by the fact that both tools have been used in several previous case studies of large and complex code bases.

Neither was any great difference in annotation overhead or complexity between the two tools found. Frama-C and VCC use similar syntax for specifying functions contracts, invariants and ghost code. One slight advantage for Frama-C is that the function contracts can be organised into what is called behaviours, which can make the annotations more readable. Consider a VCC contract specification for a function that returns its largest parameter, as is shown in listing 4.1. Here `==>` is a regular logical implication. The straight-forward equivalent contract in Frama-C is shown in listing 4.2.

But the same contract could be instead formalised using behaviours, as can be seen in listing 4.3, which makes the function contract more human-readable, especially as the contracts grow larger. This also has the benefit of enabling of the keywords `disjoint` and `complete` in the contract, which can be used to verify that for all inputs that satisfy the preconditions at most one, and at least one behaviour

```
int max(int a, int b)
    _(ensures \old(a) > \old(b)
        ==> \result == \old(a))
    _(ensures \old(a) <= \old(b)
        ==> \result == \old(b));
```

Listing 4.1: VCC contract for a function returning the largest of two values.

```
/*@ ensures \old(a) > \old(b)
        ==> \result == \old(a));
    ensures \old(a) <= \old(b)
        ==> \result == \old(b));
 */
int max(int a, int b);
```

Listing 4.2: Frama-C contract for a function returning the largest of two values.

```
/*@ behavior a_is_larger:
        assumes a > b;
        ensures \result == \old(a);
    behavior b_is_larger_or_equal:
        assumes b >= a;
        ensures \result == \old(b);
 */
int max(int a, int b);
```

Listing 4.3: Frama-C contract equivalent to that in listing 4.2, using behaviours.

is applied, respectively. These properties would otherwise require complex logical expressions to formalise.

When it comes to reasoning about the state of memory, the tools have their own unique but similar challenges. VCC, because of its assumption that all code is concurrent, introduces some additional work in proving that memory cannot be changed by other threads during execution of a function, and that the executing has ownership or access to the data it is trying to read and write. For a sequential code base such as the one that was the subject of the case study, this is mostly accomplished with keywords such as `\thread_local`, `\mutable` and their equivalents for arrays and structures. In Frama-C on the other hand, some extra effort is required

in proving that memory is not aliased, because of its weaker memory model not taking advantage of types in the same way that VCC does. This usually entails annotating contracts with the `\separated`-keyword and pointer inequalities.

The small subset of the code base consisted of roughly 200 lines of code and 6 functions, 2 of which where simple mutator functions. After verifying the same requirements in both tools, the Frama-C version of the code contained 60 lines of annotations and the VCC version 35 lines. This difference was mostly due to the fact that behaviours were used to specify contracts in Frama-C, and that the behaviours were verified to be complete and disjoint (that is, slightly stronger requirements were actually verified in Frama-C). If the Frama-C contracts were written in a style similar to those in VCC, there would be about 15 lines of annotations less. In this small test verification there is still about 10 lines of annotation advantage for VCC, which is because the extra requirements needed in Frama-C's weaker memory model are slightly more than those needed in VCC's concurrent model. For larger projects with more complex functions the difference in annotation overhead will grow smaller, as a larger proportion of the annotation will specify functionality. Coupled with the fact that, just like regular code, readability is often more important than brevity, then the difference is negligible.

Unlike VCC, Frama-C does not report the time required for verification, which makes it difficult to compare the performance of the two tools. For the code base used in the comparison VCC verified each function in under fifteen milliseconds, although most of them much faster. All that can be said about Frama-C is that it too verified every function in a small fraction of a second. One difference in the tools which might affect the time required for verification of complex functions is that in VCC only an entire file or a single function contract can be verified at a time, whereas in Frama-C individual conditions within function contracts can also be verified by themselves. Changes can thus be made to the code or annotations without having to verify entire functions in every step of the process.

The main differences in the tools are then the non-technical aspects, such as their supported platforms and whether commercial support is available. Depending on the kind of system one may want to verify, the concurrency aspect must of course also be taken into consideration when choosing which tool to work with. One of the related studies showed that parallel execution can be simulated in sequential code and consequently verified in Frama-C, but a concurrent system will still likely be much simpler to verify using VCC. Eventually it was decided to continue the full verification effort with VCC, only because it fit the platforms and workflow of the principal company better. They primarily use Windows and Visual Studio for development, and as such VCC was simpler to incorporate.

Chapter 5

The Case Study

This chapter gives a high-level description of the module that was examined in the case study, and the requirements on it which were attempted to formally verify.

5.1 Code Base

The software that was the focus of the case study was a module controlling the steering systems in vehicles. This module is considered to be safety-critical, and is part of a larger embedded system. The module and all related code was written in the C language.

In general the concept of a module is not well defined, but for the purposes of this project it was considered to be a single C implementation file. This module consisted of 10 functions spanning roughly 1400 lines of code, including a few type definitions and macros specific to the module. One function acted as the entry point from which all other functions were called, forming a hierarchy of functions. The entry point function was called at a specific interval from the surrounding system. As is common in safety-critical systems the program flow of the software was simple, consisting only of `if`- and `switch`-statements, and function calls.

The module controlling the steering primarily interacted with two other modules. One of them was a diagnostics module whose implementation was unavailable during the case study, which meant that computations that depended on output of this module could not be considered for verification. Among other things, this module examined and logged the status of signals, and in a few cases reported results back, such as whether a given signal value was valid for use in the verified module.

The other module was performing all I/O to the global state of the system. For this module all source code was available, and contracts had to be annotated in order for the verification to be possible. In total 9 functions performing reads, writes, and status checks of different types of signals had to be involved in the verification process in this module. Some of the signals come from sensors directly mapped to memory, but the memory read through this module has been written to by an intermediate module which is called sequentially with the module of the

case study, ensuring that the memory does not change during execution. As such, even though the system itself is not strictly sequential, this does not affect the computations performed by the verified functions.

The global state of the module consisted of several sets of signals of various types, each with a value and status. Disregarding the diagnostics module, the module of the case study could mostly be viewed as a functional relation between the signals of the global state, i.e. a mapping from states before execution to states after, with a few exceptions where signals instead depended on previous executions of the module. Signals were not exclusively input or output, in some cases their usage overlapped.

Apart from the two aforementioned interacting modules many other header files were included in the project, containing mostly type definitions and macros for the entire system. This means that in reality several thousand more lines of code were involved in the verification.

In appendix A an example module is given, which is similar to the module of the case study in structure and program flow complexity. However, the handling of signals and computations performed is completely fictional and not realistic. It is also only a fraction of the size, and does not depend on any other modules.

5.2 Requirements

The document of requirements for the relevant module that was available for the project consisted of 27 separate requirements. The requirements were given both in natural and a more formal language. They were written to be used by engineers at the company with great knowledge of the embedded system, and not to be understandable by outsiders. They were also not written with either formal verification or any kind of functional model in mind.

The following is an example of what an informally specified requirement might look like: *if the vehicle is moving without primary power steering then the secondary circuit should handle power steering.* This requirement is taken from the example module in appendix A and might not be functionally realistic, but shows how a requirement might be specified in natural language. Requirements specified in this way could be preferable for other purposes, such as during implementation or testing.

Because of the nature of the requirements they lacked any implementation details, including how they related to memory within the embedded system.

5.3 Method

To perform the verification the type of requirements that it is possible to verify with the chosen tool had to be identified. The given requirements were classified into groups of different types, and the ones deemed verifiable were chosen for verification. A functional model was formalised, under which the verifiable requirements could

be interpreted. A discussion of verifiable requirements and the proposal for the formal model that was used can be found in chapter 6.

In addition, variables in the requirements had to be mapped to memory locations within the software since no such relation existed.

The code base was initially annotated to verify memory safety and termination of all operations, before any verification of requirements was performed. Requirements were then chosen to be verified, and annotated at the top level of the module according to the model and mapping.

The requirements at the top level had to be decomposed to lower levels of the function hierarchy. Two strategies for doing this were formulated and applied in the annotation process. The annotation process is described in more detail in chapter 7, with some additional suggestions on how to annotate the code for easier verification when encountering common obstacles.

From the experience of having learnt the tools and performed the verification of requirements, guidelines for writing software that is easier to verify were conceived. These can be found in chapter 8.

Chapter 6

Verifiable Requirements

This chapter discusses what type of requirements can be verified, and proposes a model for specifying such requirements in a way that can be easily translated to annotations.

6.1 Functional Requirements

The tools that have been examined in this case study can only verify conditions that are strictly functional, i.e. they must be able to be expressed as mathematical functions.

In performing the case study many other types of requirements were discovered. One type of requirement stated that certain conditions should hold on the state before any execution took place. This cannot be translated to a functional condition, which can only state that if certain conditions hold in the initial state, then some other conditions will be fulfilled after the execution. In other words, only requirements that relate states from before and after execution are valid.

Another encountered type was temporal requirements, which state that a condition should hold for a certain time. This, by itself, cannot be expressed as a mathematical function, however there are ways to get around this. If the system is structured such that functions can be assumed to run at a specific interval, then time elapsed can be deduced simply by counting the number of executions. As such, temporal requirements can be written functionally, as seen in listing 6.1, assuming the appropriate logic exists within the code to handle the temporal condition.

Moreover, a functional specification can not contain any contradictions, as two such requirements can never be proven true simultaneously. However, a specification may contain partial functions, and as such it is allowed for the value of variables to be unspecified under certain conditions. For example, a requirement that states that if the vehicle is moving then the steering should be activated, but makes no mention of whether the steering should be activated when the vehicle is not moving, poses no problem for formal verification.

```
_(ghost \int time_elapsed)
int state;

void run_10ms()
    _(ensures time_elapsed > 100 ==> state == 0)
{
    ...
    _(ghost time_elapsed += 10;)
    ...

}
```

Listing 6.1: C function with temporal requirement translated to functional.

6.2 Mapping of Variables

It is important that the verified properties correspond to the actual requirements, for the verification to hold any value. For this reason, a clear mapping between requirement variables and program memory must exist. Say a requirement might mention the *speed* of the vehicle, then there must also be a mention of what this means within the code, for example `state[SPEED]` which might be a global memory location. Lacking such a mapping will make the translation from requirements to annotations code much harder, if not impossible, and introduce additional assumptions under which the verification is performed.

For the verification to be sound the mapping must be precise, and it is not enough to state what requirement variables correspond to what program variables. In embedded systems, apart from signal values, it is common to also have a separate parameter for the status of a signal, which could take on values such as, say, *initialised*, *faulty* or *good*. A reference to vehicle speed in the requirements must then not only be mapped to the speed value measured by a sensor, but also to the status threshold for which there is enough confidence to use it in the computation.

6.3 Module-specific Requirements

A specification of requirements for a module should as far as possible impose conditions upon memory which the corresponding code has access to, or it is not possible to translate the requirement into annotations within that module, as they follow all the normal rules of scoping in C. For example, if a result is computed and then stored by calling a function in another module which writes it to its local memory, it is not possible within the first module to reason about the function's effect on the state, unless that module also has access to the memory (e.g. by declaring it in a header which can be included in other modules). To some extent this can be worked

around with ghost code, by having the relevant functions in other modules return specific ghost variables, but it will introduce many complexities to the verification process.

It is also important that requirements are specified at the appropriate level in the program hierarchy. In section 6.1 an example of a non-functional condition which specified that certain conditions should hold before execution was used. Such a requirement would be more appropriate at the level of the program that initialises the state in which the module is executed.

Similarly, conditions on how one module's execution affects the result of another module's execution should preferably be stated on a level above both of them. Such a requirement should then be decomposed with only the relevant parts stated at each of the modules' top level. For example, say a system consists of modules A, B, and C, where B and C are called consecutively from A. A requirement which states that if a certain error occurs in module B, then this event should be logged in module C, can not be annotated at the top level of either module B or C, but must be a requirement on module A. This requirement can then be decomposed to B and C depending on how their respective executions affects it.

6.4 A Requirements Model

In this section a model for formalising requirements, which incorporates the hardware and software of an embedded system, is proposed. It is then discussed how to use it to translate requirements to annotations at module level. The material here is just a brief summary, and the full detailed description can found in the thesis *Formal Requirement Models for Automotive Embedded Systems* by John Eriksson [13].

When defining this model the assumption is made that it is the functional behaviour of a module that is of interest, that is how an execution of the module relate initial values to final values (input and output). The requirements model consists of two views, an external and an internal, both of which is based around sets of variables. Two sets of variables are defined:

state variables comprise the module's interface, together forming the state over which the module operates. A state value is considered an input if its value affects the final value of a state variable, and an output if its value changes during execution of the module.

model variables are local or intermediary variables.

The external view is then the collection of functions defining final values of outputs as mathematical expressions of inputs. The internal view introduces model variables, and consists of functions corresponding to decomposition of the external view. For example, say that the state variables are s_1, s_2, s_3 (and let primed variables denote their values after execution), then an external view for some functions f, g may be:

$$s_1' = f(s_1, s_2), \quad s_2' = g(s_3, f(s_1, s_2))$$

Here, all s_i are inputs and s_1, s_2 are outputs. The internal view could introduce the model variable m_1 such that:

$$m = f(s_1, s_2), \quad s_1' = m, \quad s_2' = g(s_3, m)$$

This model should be used to write the requirements for the module, and function as a tool for deriving the annotations in the C code. The external view would conceptually be sufficient for writing requirements and deriving annotations, but the introduction of model variables may help make them more readable, while also simplifying the annotation process since they can directly correspond to local variables anywhere in the module.

The translation to annotations at the top level in this model is straightforward: every requirement can be annotated as a postcondition consisting of an implication from the function input variables to the output variables. State variables can be referred to directly as program memory, whereas model variables will need to have ghost code added to the function.

Chapter 7

Code Annotation

This chapter describes a process for annotating code in order to verify its requirements. All of the examples in this chapter are taken from the example module given in appendix A, where the full annotated source code is given. Some of the examples have been slightly rewritten here, in order to be easier to understand on their own.

7.1 Preparatory Annotation

When verifying an advanced system some preparatory work in annotating the code will have to be expected, before it is possible to perform any verification of actual requirements. Both of the tested tools had trouble handling some of the preprocessor directives correctly, such as conditional inclusion of platform specific headers and compiler specific language extension. Such code had to be rewritten or removed in order to get the tools to even parse the program.

Another issue that will have to be taken care of to get the code base into a workable state is that of memory validity, otherwise the verification will fail every read and write to memory that exists in the code. Thus, annotations have to be inserted concerning the validity of all memory locations used. For example, consider the simple function in listing 7.1 that just reads global memory. Without any annotations VCC will fail to verify this, with a message saying that `state[idx]` is not thread local, which is required to safely read the memory. It would need to be annotated with both preconditions of the `read`-function in listing 7.2, which all callers then need to fulfill.

7.2 Decomposing Requirements

In describing the process of annotating the code for verification of requirements it is assumed that there already exists a model for translating the requirements to functions over states on the level of the module that will be verified, similar to the one described in section 6.4. How to break these requirements down to a functional level will depend on what the code base looks like. At the top level it is preferable

```
int state[NUM_SIGNALS]; // Global state

int read(int idx)
{
    if(idx < NUM_SIGNALS)
    {
        return state[idx];
    }
}
```

Listing 7.1: C function that reads global memory.

to be able to state conditions exactly as they appear in the formal requirements, both for the sake of readability and to minimise the risk of formalising incorrect requirements in the annotations. In the case study, the module only has a single entry point at the top level, from which all computation is performed. This makes it easy to use the requirements directly, translated into conditions over the state of memory at module level. A module might of course have several entry points, and then it is necessary already at the top level to divide the requirements into the relevant entry point functions. It is also preferable to perform the verification only one or a few requirements at a time, since it will be easier to understand why the verification has failed with fewer possible sources of error.

The next step is to break down the requirements from module level to function level. When doing this, two different approaches have primarily been employed, which will be called top-down and bottom-up. In the top-down approach, a single requirement is chosen for verification, and its trail of execution is followed through the functions which it is affected by. These functions are then annotated with contracts that will ensure that the requirement will be satisfied at the top level. The annotations will differ between the functions depending on how a specific function affects the requirement. The resulting contracts will be an incomplete specification of the function, that is sufficient for verification of the considered requirements.

In the bottom-up approach annotation is instead performed strictly at the function level, starting at the bottom of the call hierarchy. Each function is annotated with contracts stating everything that the function does, until the top level is reached and all requirements should then be fulfilled. In other words, contracts resulting from this approach will be complete specifications of their corresponding functions. Because of this, the annotation overhead will be larger when using the bottom-up approach.

These two approaches have been found to work well together, when using each for different type of functions. The bottom-up approach is well suited for small functions that many requirements will depend on. Examples of this could be mutator functions (often referred to as getters and setters), that is functions that simply

reads or writes values to memory. Listing 7.2 shows how two such complete specifications could look.

```
int state[NUM_SIGNALS]; // Global state

int read(int idx)
    _(requires \thread_local_array(state, NUM_SIGNALS))
    _(requires 0 <= idx && idx < NUM_SIGNALS)
    _(ensures \result == state[idx]);

void write(int idx, int val)
    _(writes &state[idx])
    _(requires 0 <= idx && idx < NUM_SIGNALS)
    _(ensures state[idx] == val);
```

Listing 7.2: VCC contracts for simple mutator functions, annotated using bottom-up.

Top-down is more suited to larger functions containing many logical branches, where specifying the entire function at once is neither simple nor effective. Listing 7.3 contains the contract of a top level function, where reasoning over the variables occurs in the global state. Say that function (`steering`, in the example) reads the relevant values into a local data structure, calls another function (`secondary_steering`) which transforms the structure, and then the top level function writes the values back. The contract in listing 7.4 would then have been the result of using the top-down approach to annotate the second function, where the logic of each requirement has been followed and annotated on its own.

7.3 Ghost Code

Something that is often helpful when trying to specify the outcome of large conditionals, such as deeply nested `if`-clauses, is to introduce ghost variables that specify partial functions between states. For example, many times a function contains local variables that depend on several of its inputs, and which in turn affects several of its outputs. Such local variables can then be specified as partial results by ghost variables corresponding to their values. In listing 7.4, the ghost variable `model_vehicleIsMoving` could have a corresponding local variable, which is used to simplify the specification. It is of course important to then update these ghost variables accordingly in the logic of the function, which introduces another potential source of error. This is similar to the concept of model variables described in the proposal for a requirements model in section 6.4. Listing 7.5 shows another example where the ghost variable is updated according to the logic. Of course, in cases where model variables do not strictly correspond to a local variable within

```
_(ghost \bool model_vehicleIsMoving) // Intermediary ghost variable

void steering()
    _(writes \array_range(state, NUM_SIGNALS))
    _(writes &model_vehicleIsMoving)
    ...
    // Req. 2
    _(ensures \old(state[WHEEL_BASED_SPEED]) > VEH_MOVING_LIMIT
            ==> model_vehicleIsMoving == \true)
    ...
    // Req. 4
    _(ensures model_vehicleIsMoving == \true
            ==> state[SECONDARY_CIRCUIT_HANDLES_STEERING] == \true);
```

Listing 7.3: Partial VCC contract for a top level function. Uses intermediary ghost variable to simplify the specification.

```
void secondary_steering(VEHICLE_INFO * veh_info)
    _(writes \extent(veh_info))
    _(writes &model_vehicleIsMoving)
    ...
    // Req. 2
    _(ensures \old(veh_info->wheelSpeed) > VEH_MOVING_LIMIT
            ==> model_vehicleIsMoving == \true)
    ...
    // Req. 4
    _(ensures model_vehicleIsMoving == \true
            ==> veh_info->secondCircHandlesStee == \true);
```

Listing 7.4: Partial VCC contract for a function that has been annotated using the top-down approach. Also uses intermediary ghost variable to simplify the specification.

the program, the ghost code used to update it will be more complex than a simple assignment.

7.4 Limited Variable Domains

It is common that the expected domain of a variable within a program is much smaller than its set of actual valid values. Particularly this occurs in C code when using enumerable types, where only a handful of values are of interest. However,

in reality an `enum` variable is backed by the integer type and as such can take the value of any valid integer. This introduces some complexity to the annotation, since the verification tool will check that the postconditions hold for all possible inputs. Consider the example given in listing 7.5. Here, the postcondition would not be possible to verify if the contract lacked the last precondition. This is because there are thousands of valid input values apart from the three listed in the type definition for which the postcondition would not be true. In cases where such preconditions are needed, other functions might then also need to be proven to only assign values within the expected domain.

```
typedef enum { WORKING, NO_FLOW, SHORT_CIRCUIT } SENSOR_STATE;

void secondary_steering(SENSOR_STATE * sensor_state)
    _(writes &model_vehicleWithoutPowerSteering)
    _(requires \thread_local(sensor_state))
    _(requires *sensor_state == NO_FLOW
            || *sensor_state == SHORT_CIRCUIT
            || *sensor_state == WORKING)
    ...
    // Req. 3
    _(ensures \old(*sensor_state) != WORKING
            ==> model_vehicleWithoutPowerSteering == \true)
{
    int vehicleWithoutPowerSteering;
    if(*sensor_state == NO_FLOW || *sensor_state == SHORT_CIRCUIT)
    {
        vehicleWithoutPowerSteering = TRUE;
    }
    _(ghost model_vehicleWithoutPowerSteering
                = vehicleWithoutPowerSteering)
}
```

Listing 7.5: Partial VCC contract for a function requiring precondition on the domain

7.5 Failing Verification

Annotating code correctly is not always an easy task, and it is often hard to understand exactly why the verification of some condition fails. The tools do not offer much assistance other than pointing out the specific condition that failed to verify, but there are many reasons for why that may have happened. One reason could of course be that the code does not fulfill the requirements and a specific fault in the source code has been found. But it might also just be a bug in the annotation,

or a lack of understanding of the internals of the verification tool that may have led to a misunderstanding of what exactly an annotation does. Similarly, a lack of understanding of the code itself may lead to wrongly specified annotations, when working from the bottom-up approach, even though the code does the correct thing. The only way to counteract such bugs is to make sure the annotations are actually correctly specified.

One final reason that a verification might fail, even though both the code and annotations are correct, is that the verification tool and its accompanying theorem prover are not powerful enough. In this case, one solution would be to give the tool some additional help, either by introducing assertions in the function body that helps proving the contract, or by reformulating the contract in an equivalent way but that is easier for the tool to prove, which again requires a thorough understanding of the internals of the tool. There is also the possibility that switching to another theorem prover than the tool's default one will give better results, or incorporating the use of interactive proof assistants. As with the contracts, the code itself may also be rewritten such that it performs the same computation, but is easier to handle for the verification tool. If all else fails, parts of the code may instead be manually reviewed. Isolating small sections of the software for manual review while formally verifying everything around it also fits well with the modular nature of the verification process.

Because of the expertise required in formal verification and the tools themselves, which often differs from that of a regular developer, there is also the pitfall of having successfully verified something different than what was intended.

7.6 Automating Annotation

From having performed this case study and manually annotated all functions of the relevant code base, some observations on how parts of this process could be automated have been made.

First of all a large portion of the annotations will simply pertain to the use of memory. Functions will need preconditions for memory it needs to be writable or readable. In a multi-threaded context this issue would be more complex, but since in the context of this case study only sequential code is of interest, the relevant annotations could simply be inferred from the reads and writes that are present in the code. Any calling function should then easily be able to prove that it fulfills these preconditions, either from having allocated the memory or its own similar automatically generated annotations.

The specification of requirements as annotations at the top level of the module could also be automated, assuming the requirements are stated in a machine readable manner, and are functional with the appropriate mapping for all requirements variables to program memory available. In that case, the mathematical conditions of the requirements only need to be translated into the corresponding notation of the annotation language, with all requirement variable names substituted for their

program counterparts. If the requirements use a model similar to the one described in section 6.4, then intermediary variables that do not exist in the global state of the system can just as easily be substituted for the expressions they represent. Alternatively, they may be inserted as ghost variables to be more readable for humans, as described in section 7.3, but then the appropriate logic still has to be handled manually.

Thus remains the issue of decomposing the requirements to all the functions in the hierarchy. No complete solution has been identified for this yet, but much like the bottom-up approach of the manual decomposition described in section 7.2, it could be possible to automatically infer a full specification of all functions from their implementations, using symbolic execution to calculate strongest postconditions as discussed in section 2.5. This is especially true when the program flow complexity is low, which is commonly true for safety-critical systems.

Chapter 8

Writing Verifiable Code

This chapter discusses what type of code can be verified, and outlines recommendations for how to write code that will be easy to verify, and what pitfalls to avoid to minimise the amount of obstacles encountered.

The way that a code base is written can greatly affect how easy it is to verify it. For example, as mentioned in chapter 7, the tools have problems with platform and compiler specific language extensions. Such code should be avoided to the furthest extent possible, in order to produce more easily verifiable code. This chapter discusses additional guidelines for writing verifiable code.

8.1 Type-safety

One of the most important things to be aware of when writing code that is supposed to be continuously verified is the underlying memory model of the verifier. Even though C:s own type system is weak, verification tools can take advantage of well written code and elevate it to stronger type models. To enable the verifier to do this it is important that the code is written in a type-safe manner. This includes using well-defined data types, avoiding implicit conversion between types, as well as aliasing of distinct memory objects. In listing 8.1 one example of type unsafe code is shown. Not only will the example code complicate verification, but it actually invokes undefined behaviour according to the C standard's strict aliasing, so the result depends on the specific compiler and environment. This means that the code will likely compile and work fine in most situations, and as such methods other than formal verification, such as unit testing, might never find the possible error. Even without the goal of verifying the code, safety-critical systems should adhere to these practices.

8.2 Modular Code

Because of the compositional nature of the verification tools, writing modular code is another way to simplify the verification process. Isolating distinct functionality

```
typedef struct {
    int i;
    char c;
} SensorStatus;

typedef struct {
    int i;
    char c;
} SystemInfo;

void foo(SystemInfo * a, SensorStatus * b) {
    SystemInfo * c = (SystemInfo *) b;
    *a = *c;
}
```

Listing 8.1: A C function that is not type safe.

into separate modules means that there is less room for error during the entire process. There is also less overhead at the beginning of the verification process, as it can be focused on a smaller set of requirements at a time. It is also important that there is as little interdependence between the different modules as possible, since otherwise all the modules that one specific module is dependent on will have to be involved in the process, and there is little to gain by modularisation. In particular, it is useful to separate safety-critical and non-safety-critical functionality. This allows the verification effort to be focused on where it matters the most.

The same strategy is just as useful when structuring a single module. Writing small and specific functions allows for easier decomposition of the requirements, and decreases the complexity and annotation overhead as the verification can be focused on fewer requirements at a time.

8.3 No Dependence on Previous Executions

One big obstacle encountered when trying to verify certain requirements was that because the verification tools only allow specifying conditions over a single execution, functions that depend on previous executions complicate the verification. For example, a function might perform some extra error checking for a certain amount of executions after having encountered a bad signal. This mostly occurs with the use of local static variables. Since they are not part of the global state there is no simple way to reason about the behaviour of the function depending on their values, as a function contract operates outside the scope of the function body. There are ways to get around this, such as introducing additional ghost variables which correspond to the values of local static variables. The drawback is that these ghost

variables then must be updated within the body of the function with the appropriate logic, and the specification then becomes closely tied to the implementation. A better solution would be to think about this when writing the code, and store such values to globally accessible memory so that contracts can be specified as functions from one state to another. Listing 8.2 shows how a function depending on previous executions can also be expressed by only relying on the globally accessible state.

```
int bar(int n) {
    static int counter = 0; // local static counter
    if(counter >= 10) counter = 0;
    ++counter;
    return n;
}

int counter = 0; // global counter
int baz(int n) {
    if(counter >= 10) counter = 0;
    ++counter;
    return n;
}
```

Listing 8.2: Two equivalent C functions. The top one depends on previous executions, while the bottom one only depends on the current state.

8.4 Boolean Variables

Requirements and programs will commonly contain boolean variables, and often will specify conditions both for when they hold and when the inversions hold. In section 7.4 the problem of limited variable domains was discussed, and the same problem occurs with boolean variables, where the domain is expected to only have two values. Since a boolean data type was not part of the C standard for a long time, a common way to handle boolean variables in C is to define the type using some other backing data type. Listing 8.3 shows one example of how this might be done, and subsequently how a simple inversion of a postcondition can then not be verified.

The C99 standard introduced a real boolean type, which is called `_Bool` and requires the `stdbool.h` header. Assuming a relatively modern compiler is used, this problem can easily be avoided by using the new type. Since this type only has two values in its domain, both of the postconditions in the example can easily be verified.

```
typedef unsigned char bool;
#define FALSE ((bool)0)
#define TRUE ((bool)1)

bool qux(bool b)
    _(ensures b == TRUE ==> \result == TRUE)
    _(ensures b != TRUE ==> \result == FALSE) // cannot be verified
{
    if(b) {
        return TRUE;
    } else {
        return FALSE;
    }
}
```

Listing 8.3: A common way to handle boolean variables in C.

Chapter 9

Case Study Results and Analysis

The code base that was the focus of the case study had 27 requirements associated with it. Out of these, 6 requirements were identified as not specific to the relevant module. This was because they were conditions that should hold prior to the module running, specification on how data should be stored in another module, or otherwise not exclusively related to the module of the case study. All of these are believed to be verifiable if the scope of the project was not limited to a single module.

Another 3 requirements were identified as being temporal requirements. They stated that certain conditions should hold for a certain time, and were therefore not strictly functional. In another model it is possible that these requirements could have been verified as well.

Finally, 5 more requirements were discarded, since they depended on output from other modules for which the specification was unavailable in this project. With more time, an attempt at verifying these could have been made, under certain assumptions on how to interpret the output of the other modules.

Out of the 13 remaining requirements, 10 were considered to have been successfully verified. The last 3 requirements were not verified only because of lack of time.

Since no mapping from requirement variables to memory in the application was available, assumptions had to be made of the relation between program state and requirements. Because of the naming of variables, this was often straight-forward.

Some of the 10 verified requirements did not impose conditions on the state after running the module, but rather intermediary results during execution. These were verified under assumptions on what the intermediary results would mean with respect to the actual code.

Apart from the explicitly stated requirements, many other properties were implicitly verified during this process, such as memory safety of the relevant state during execution of the module, termination of all functions in the module, and that parts of the code that were supposed to be unreachable would never be executed.

Under the interpretation of the requirements as functional conditions before and

after execution, two were also discovered to be contradictory to each other. This is a downside of the functional model, as the order of evaluation is not taken into account.

In total the verification effort required about 700 lines of annotations, which is close to half the number of lines of code of the implementation of the module. However, considering that more code than the implementation of the functions were involved, this ratio is in practice smaller.

Not counting functions called in other modules, for some of which the source code was not available for the purpose of the project, a total of 10 functions were annotated and verified. Performing the verification in VCC took on average 165 seconds for the entire module, or just short of 3 minutes. The hardest function for VCC to verify took 65 seconds.

Chapter 10

Conclusion

The results of the case study show that it is currently possible to use the available state-of-the-art tools for formal verification to verify functional requirements on embedded code, specifically safety-critical code as is used in the real world right now. Several of the specified requirements on the module that was the focus in this project were successfully verified, while for many others the reasons for not having been verified within the scope of this project were clearly identified.

A process for performing the verification has been described, from specifying verifiable requirements to annotating the code base. Specific methods for decomposing requirements on the module level to annotations at the different function levels have been formalised and shown to work.

A collection of guidelines for writing code that can easily and continuously be verified has been proposed, most of which stem from problems encountered during the case study.

Formal verification of a complex code base is a large undertaking, so automation of parts of the process is desirable. Some ways in which this can be done have been identified, but whether this can be achieved for the process of decomposing requirements remains unsolved, and could be a possible future extension to the project.

Bibliography

[1] Eyad Alkassar, Mark A. Hillebrand, Wolfgang Paul, and Elena Petrova. *Verified Software: Theories, Tools, Experiments: Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, chapter Automated Verification of a Small Hypervisor, pages 40–54. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. Available at `http://www-wjp.cs.uni-saarland.de/publikationen/AHPP10.pdf`.

[2] Patrick Baudin, François Bobot, Loïc Correnson, and Zaynah Dargaye. WP plug-in manual, 2015. Available at `http://frama-c.com/download/frama-c-wp-manual.pdf`.

[3] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto1. ACSL: ANSI/ISO C specification language, 2015. Available at `http://frama-c.com/download/frama-c-acsl-implementation.pdf`.

[4] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Formal verification of a microkernel used in dependable software systems. In *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP '09, pages 187–200, Berlin, Heidelberg, 2009. Springer-Verlag. Available at `http://formal.iti.kit.edu/beckert/pub/safecomp2009.pdf`.

[5] Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéeric Loulergue. *Formal Methods for Industrial Critical Systems: 20th International Workshop, FMICS 2015 Oslo, Norway, June 22-23, 2015 Proceedings*, chapter A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C, pages 15–30. Springer International Publishing, Cham, 2015. Available at `http://www.springer.com/cda/content/document/cda_downloaddocument/9783319194578-c2.pdf`.

[6] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *Theorem Proving in Higher*

Order Logics (TPHOLs 2009), volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Munich, Germany, 2009. Springer. Available at `http://research.microsoft.com/en-us/um/people/moskal/pdf/tphol2009.pdf`.

[7] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, February 2009. Available at `http://research.microsoft.com/en-us/um/people/moskal/pdf/concurrency3.pdf`.

[8] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A precise yet efficient memory model for C. In *Workshop on Systems Software Verification (SSV 2009)*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 85–103. Elsevier Science B.V., 2009. Available at `http://research.microsoft.com/en-us/um/people/moskal/pdf/ssv2009.pdf`.

[9] Loïc Correnson, Pascal Cuoq, Florent Kirchner, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. Frama-C user manual, 2015. Available at `http://frama-c.com/download/frama-c-user-manual.pdf`.

[10] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag. Available at `http://pathcrawler-online.com/pubs/final_sefm12.pdf`.

[11] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. Available at `http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD472.PDF`.

[12] Frank Dordowsky. An experimental study using ACSL and Frama-C to formulate and verify low-level requirements from a DO-178C compliant avionics project. In *Proceedings Second International Workshop on Formal Integrated Development Environment, F-IDE 2015, Oslo, Norway, June 22, 2015.*, pages 28–41, 2015. Available at `http://arxiv.org/pdf/1508.03894v1.pdf`.

[13] John Eriksson. Formal requirement models for automotive embedded systems. Master's thesis, School of Computer Science and Communication, Royal Institute of Technology, 2016.

[14] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967. Available at `http://www.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf`.

[15] Mike Gordon and Hélène Collavizza. *Reflections on the Work of C.A.R. Hoare*, chapter Forward with Hoare, pages 101–121. Springer London, London, 2010. Available at `http://www.cl.cam.ac.uk/~mjcg/Hoare75/paper.pdf`.

[16] Steve Heath. *Embedded Systems Design*. Butterworth-Heinemann, Newton, MA, USA, 1st edition, 1997.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. Available at `https://www.cs.cmu.edu/~crary/819-f09/Hoare69.pdf`.

[18] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[19] S.K. Khaitan and James Mccalley. Design techniques and applications of cyberphysical systems: A survey. *IEEE Systems Journal*, 9(2):1–16, July 2014.

[20] K. Rustan M. Leino. Efficient weakest preconditions. Technical Report MSR-TR-2004-34, Microsoft Research, April 2004. Available at `http://research.microsoft.com/en-us/um/people/leino/papers/krml114a.pdf`.

[21] The VCC manual, August 2012. Available at `https://www.codeplex.com/Download?ProjectName=VCC&DownloadId=476508`.

[22] International Standardization Organization. ISO 26262-1 - road vehicles – functional safety, 2011.

[23] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[24] Jonas Westman and Mattias Nyberg. Contracts for specifying and structuring requirements on cyber-physical systems, 2015. Available at `https://www.diva-portal.org/smash/get/diva2:786267/FULLTEXT01.pdf`.

[25] Jonas Westman and Mattias Nyberg. Formal architecture modeling of sequential C-programs, 2016.

Appendix A

Example C Module

This appendix contains the requirements as well as the annotated source code for a complete example module. It has been written specifically for this thesis, but is in some ways inspired by the real module that was examined in the case study.

A.1 Requirements

The requirements for the example module are all functional and specified in an informal language. They can be seen in table A.1. The requirements were written to be easy to understand, and their contents are not realistic.

The example requirements were written in informal language with no mapping to program variables, to highlight how such requirements can be interpreted in a formal manner when being converted to annotations. For example, looking at the first requirement and the code, it has been assumed that *primary circuit has no flow* corresponds to a value of 1 (or `true`) of the program variable `state[PRIMARY_CIRCUIT_LOW_FLOW]`.

The specification contains only requirements that are in some way interdependent, and they show all the types of variables discussed in section 6.4. Input variables would be the sensors for circuit voltage and wheel speed, while an output variable would be the actuator for activating the electric motor. The boolean value of whether the vehicle is considered to be moving is a model variable, affected by the speed and affecting the power steering. Depending on the interpretation, the variable for whether the secondary circuit should handle power steering could be considered an input and an output, or an output and a model variable.

A.2 Annotated Source Code

In the example, a module is a single C source file. The annotated code of the example module can be seen in listing A.1. As with the requirements for the module, the actual computations are not realistic. However, the low complexity of the program flow and the structure is similar to the module examined in the case study. The

#	Requirement Text
1	If the primary circuit has no flow or a short circuit is detected then the primary circuit cannot provide power steering.
2	The vehicle is considered to be moving if the wheel based speed signal is greater than 3 km/h.
3	If the vehicle is moving and the primary circuit cannot provide power steering then the vehicle is moving without primary power steering.
4	If the vehicle is moving without primary power steering then the secondary circuit should handle power steering.
5	If the secondary circuit is providing power steering and the parking brake is not set then the electric motor must be activated.

Table A.1: Requirements for the example module.

example module is also only a fraction of the size of the real module, and there is little to no code not directly related to the given requirements.

Many other things that might occur in a real system have been simplified, such as only having one type for signals, disregarding the quality of signals and having no dependence on other modules.

The function `steering` is the entry point function from which the rest of the module is called, and the top level at which all requirements have been specified as annotations. The example showcases the use of ghost code that corresponds to intermediary values, as well as how the first four requirements would have been specified without the intermediary values, which is by combining them into a single postcondition by substitution. In this case combining the requirements is manageable, but it is easy to see how requirements containing many intermediary variables could become complex enough to be hard to understand when annotated this way.

The example also shows the use of both annotation strategies proposed in chapter 7: the function `secondary_steering` has been annotated using the top-down approach, while bottom-up has been used for all other functions.

Listing A.1: Annotated source code for the full example module.

```
1  /*
2      This module controls the steering system of vehicles.
3
4      The following is a list of requirements it should adhere to.
5
6      Req. 1:
7      If the primary circuit has no flow
8      or a short circuit is detected
```

```
 9      then the primary circuit cannot provide power steering.
10
11      Req. 2:
12      The vehicle is considered to be moving
13      if the wheel based speed signal is greater than 3 km/h.
14
15      Req. 3:
16      If the vehicle is moving
17      and the primary circuit cannot provide power steering
18      then the vehicle is moving without primary power steering.
19
20      Req. 4:
21      If the vehicle is moving without primary power steering
22      then the secondary circuit should handle power steering.
23
24      Req. 5:
25      If the secondary circuit is providing power steering
26      and the parking brake is not set
27      then the electric motor must be activated.
28   */
29
30  #include <vcc.h>
31
32  #define VEH_MOVING_LIMIT 3
33
34  #define TRUE 1
35  #define FALSE 0
36
37  typedef enum
38  {
39      WORKING,
40      NO_FLOW,
41      SHORT_CIRCUIT,
42  } SENSOR_STATE;
43
44  typedef struct
45  {
46      int wheelSpeed;
47      int parkingBrake;
48      int primLowFlow;
49      int primHighVoltage;
50      int secondCircHandlesStee;
51      int electricMotorAct;
52  } VEHICLE_INFO;
53
54  typedef enum
55  {
56      PARKING_BRAKE_APPLIED,
57      PRIMARY_CIRCUIT_LOW_FLOW,
```

```
58      PRIMARY_CIRCUIT_HIGH_VOLTAGE,
59      WHEEL_BASED_SPEED,
60      SECONDARY_CIRCUIT_HANDLES_STEERING,
61      ELECTRIC_MOTOR_ACTIVATED,
62      NUM_SIGNALS
63  } SIGNAL;
64
65  // ghost variables representing model_variables in requirements
66  _(ghost \bool model_vehicleIsMoving;)
67  _(ghost \bool model_vehicleMovingWithoutPrimaryPowerSteering;)
68  _(ghost \bool model_primaryCircuitProvidingPowerSteering;)
69
70  int state[NUM_SIGNALS]; // Global state
71
72  /*
73      Reads the specified signal from the state.
74   */
75  int _(pure) read(SIGNAL idx)
76      _(reads &state[idx])
77      _(requires \thread_local_array(state, NUM_SIGNALS))
78      _(requires 0 <= idx && idx < NUM_SIGNALS)
79      _(ensures \result == state[idx])
80      _(decreases 0)
81  {
82      if(idx < NUM_SIGNALS)
83      {
84          return state[idx];
85      }
86  }
87
88  /*
89      Writes the specified signal to the state.
90   */
91  void write(SIGNAL idx, int val)
92      _(writes &state[idx])
93      _(requires 0 <= idx && idx < NUM_SIGNALS)
94      _(ensures state[idx] == val)
95      _(decreases 0)
96  {
97      if(idx < NUM_SIGNALS)
98      {
99          state[idx] = val;
100     }
101 }
102
103 /*
104     Reads the current state of the system.
105  */
106 void get_system_state(VEHICLE_INFO * veh_info)
```

```
107     _(requires \thread_local_array(state, NUM_SIGNALS))
108     _(writes \extent(veh_info))
109     _(ensures \extent_mutable(veh_info))
110     _(ensures veh_info->wheelSpeed == \old(state[WHEEL_BASED_SPEED]))
111     _(ensures veh_info->parkingBrake == \old(state[PARKING_BRAKE_APPLIED]))
112     _(ensures veh_info->primLowFlow ==
        \old(state[PRIMARY_CIRCUIT_LOW_FLOW]))
113     _(ensures veh_info->primHighVoltage ==
        \old(state[PRIMARY_CIRCUIT_HIGH_VOLTAGE]))
114     _(ensures veh_info->secondCircHandlesStee ==
        \old(state[SECONDARY_CIRCUIT_HANDLES_STEERING]))
115     _(ensures veh_info->electricMotorAct ==
        \old(state[ELECTRIC_MOTOR_ACTIVATED]))
116     _(decreases 0)
117 {
118     veh_info->wheelSpeed = read(WHEEL_BASED_SPEED);
119     veh_info->parkingBrake = read(PARKING_BRAKE_APPLIED);
120     veh_info->primLowFlow = read(PRIMARY_CIRCUIT_LOW_FLOW);
121     veh_info->primHighVoltage = read(PRIMARY_CIRCUIT_HIGH_VOLTAGE);
122     veh_info->secondCircHandlesStee =
        read(SECONDARY_CIRCUIT_HANDLES_STEERING);
123     veh_info->electricMotorAct = read(ELECTRIC_MOTOR_ACTIVATED);
124 }
125
126 /*
127     Evaluates the state of the primary steering circuit sensors.
128  */
129 void eval_prim_sensor_state(VEHICLE_INFO * veh_info, SENSOR_STATE *
        sensor_state)
130     _(requires \extent_mutable(veh_info))
131     _(writes sensor_state)
132     _(ensures veh_info->primHighVoltage == \true
133             ==> *sensor_state == SHORT_CIRCUIT)
134     _(ensures veh_info->primHighVoltage != \true
135             && veh_info->primLowFlow == \true
136             ==> *sensor_state == NO_FLOW)
137     _(ensures veh_info->primHighVoltage == \false
138             && veh_info->primLowFlow == \false
139             ==> *sensor_state == WORKING)
140     _(ensures *sensor_state == NO_FLOW
141             || *sensor_state == SHORT_CIRCUIT
142             || *sensor_state == WORKING)
143     _(decreases 0)
144 {
145     if(veh_info->primHighVoltage == TRUE)
146     {
147         *sensor_state = SHORT_CIRCUIT;
148     }
149     else if(veh_info->primLowFlow == TRUE)
```

```
150     {
151         *sensor_state = NO_FLOW;
152     }
153     else
154     {
155         *sensor_state = WORKING;
156     }
157 }
158
159 /*
160     Evaluates whether steering should be
161     handled by the secondary circuit.
162  */
163 void secondary_steering(VEHICLE_INFO * veh_info, SENSOR_STATE *
        sensor_state)
164     _(writes \extent(veh_info))
165     _(writes &model_vehicleIsMoving,
166              &model_vehicleMovingWithoutPrimaryPowerSteering)
167     _(requires \thread_local(sensor_state))
168     _(requires *sensor_state == NO_FLOW
169             || *sensor_state == SHORT_CIRCUIT
170             || *sensor_state == WORKING)
171
172     // Req. 2
173     _(ensures \old(veh_info->wheelSpeed) > VEH_MOVING_LIMIT
174             ==> model_vehicleIsMoving == \true)
175
176     // Req. 3
177     _(ensures model_vehicleIsMoving == \true
178             && \old(*sensor_state) != WORKING
179             ==> model_vehicleMovingWithoutPrimaryPowerSteering == \true)
180
181     // Req. 4
182     _(ensures model_vehicleMovingWithoutPrimaryPowerSteering == \true
183             ==> veh_info->secondCircHandlesStee == \true)
184
185     // Req. 5
186     _(ensures veh_info->secondCircHandlesStee == \true
187             && \old(veh_info->parkingBrake) == \false
188             ==> veh_info->electricMotorAct == \true)
189
190     _(ensures \extent_mutable(veh_info))
191     _(decreases 0)
192 {
193     char vehicleIsMoving;
194     char vehicleIsMovingWithoutPrimaryPowerSteering;
195
196     // Check whether the vehicle is moving.
197     if(veh_info->wheelSpeed > VEH_MOVING_LIMIT)
```

```
198     {
199         vehicleIsMoving = TRUE;
200     }
201     else
202     {
203         vehicleIsMoving = FALSE;
204     }
205
206     // Check whether vehicle is moving without primary power steering.
207     if(vehicleIsMoving == TRUE &&
208       (*sensor_state == NO_FLOW || *sensor_state == SHORT_CIRCUIT))
209     {
210         vehicleIsMovingWithoutPrimaryPowerSteering = TRUE;
211     }
212     else
213     {
214         vehicleIsMovingWithoutPrimaryPowerSteering = FALSE;
215     }
216
217     // Let secondary circuit handle steering if necessary.
218     if(vehicleIsMovingWithoutPrimaryPowerSteering == TRUE)
219     {
220         veh_info->secondCircHandlesStee = TRUE;
221     }
222
223     // Activate the electric motor.
224     if(veh_info->secondCircHandlesStee == TRUE
225         && veh_info->parkingBrake == FALSE)
226     {
227        veh_info->electricMotorAct = TRUE;
228     }
229
230     _(ghost model_vehicleMovingWithoutPrimaryPowerSteering
231             = vehicleIsMovingWithoutPrimaryPowerSteering;)
232     _(ghost model_vehicleIsMoving = vehicleIsMoving;)
233 }
234
235 /*
236     Module entry point function.
237  */
238 void steering()
239     _(writes \array_range(state, NUM_SIGNALS))
240     _(writes &model_vehicleIsMoving,
241              &model_vehicleMovingWithoutPrimaryPowerSteering,
242              &model_primaryCircuitProvidingPowerSteering)
243
244     // Req. 1-4 without intermediary variables
245     _(ensures (\old(state[PRIMARY_CIRCUIT_HIGH_VOLTAGE]) == \true ||
246                \old(state[PRIMARY_CIRCUIT_LOW_FLOW]) == \true)
```

```
247             && \old(state[WHEEL_BASED_SPEED]) > VEH_MOVING_LIMIT
248             ==> state[SECONDARY_CIRCUIT_HANDLES_STEERING] == \true)
249
250     // Req. 1
251     _(ensures (\old(state[PRIMARY_CIRCUIT_HIGH_VOLTAGE]) == \true ||
252                \old(state[PRIMARY_CIRCUIT_LOW_FLOW]) == \true)
253                ==> model_primaryCircuitProvidingPowerSteering == \false)
254
255     // Req. 2
256     _(ensures \old(state[WHEEL_BASED_SPEED]) > VEH_MOVING_LIMIT
257             ==> model_vehicleIsMoving == \true)
258
259     // Req. 3
260     _(ensures model_vehicleIsMoving == \true
261             && model_primaryCircuitProvidingPowerSteering == \false
262             ==> model_vehicleMovingWithoutPrimaryPowerSteering == \true)
263
264     // Req. 4
265     _(ensures model_vehicleMovingWithoutPrimaryPowerSteering == \true
266             ==> state[SECONDARY_CIRCUIT_HANDLES_STEERING] == \true)
267
268     // Req. 5
269     _(ensures state[SECONDARY_CIRCUIT_HANDLES_STEERING] == \true
270             && \old(state[PARKING_BRAKE_APPLIED]) == \false
271             ==> state[ELECTRIC_MOTOR_ACTIVATED] == \true)
272
273     _(decreases 0)
274 {
275     VEHICLE_INFO veh_info;
276     SENSOR_STATE prim_sensor;
277
278     get_system_state(&veh_info);
279
280     eval_prim_sensor_state(&veh_info, &prim_sensor);
281     _(ghost model_primaryCircuitProvidingPowerSteering
282             = (prim_sensor == WORKING));
283
284     secondary_steering(&veh_info, &prim_sensor);
285
286     write(SECONDARY_CIRCUIT_HANDLES_STEERING,
        veh_info.secondCircHandlesStee);
287     write(ELECTRIC_MOTOR_ACTIVATED, veh_info.electricMotorAct);
288 }
```

Listing A.1: Annotated source code for the full example module.

A.3 Result of Verification in VCC

The output from VCC when verifying the example module can be see in listing A.2. As can be seen, the annotated contracts of all functions were successfully verified.

```
=== VCC started. ===
Command Line: vcc steering.c

Verification of read succeeded. [0,20]
Verification of write succeeded. [0,01]
Verification of get_system_state succeeded. [0,7]
Verification of eval_prim_sensor_state succeeded. [0,02]
Verification of secondary_steering succeeded. [0,03]
Verification of steering succeeded. [0,19]
Verification of read#reads succeeded. [0,00]

=== Verification succeeded. ===
```

Listing A.2: Output when verifying the example module in VCC.