Degree Project in Computer Science and Engineering
Second cycle, 30 credits

Improving the Synthesis of Annotations for Partially Automated Deductive Verification

HOVIG MANJIKIAN
Improving the Synthesis of Annotations for Partially Automated Deductive Verification

HOVIG MANJIKIAN

Degree Programme in Computer Science and Engineering
Date: July 20, 2023

Supervisor: Christian Lidström
Examiner: Cyrille Artho
School of Electrical Engineering and Computer Science
Host company: Scania CV AB
Swedish title: Att förbättra syntes av funktionsanteckningar för partiellt automatiserad deduktiv verifiering
Abstract

This work investigates possible improvements to an existing annotation inference tool. The tool is part of a toolchain that aims to automate the process of software verification using formal methods. The purpose of the annotations is to facilitate the use of deductive verification, which is the formal method used in this project for proving that a given program meets its specifications. In the project, two categories of annotations are established. The first category is the category of functional annotations. These annotations describe the behavior of a function or a module. The other category is what we call auxiliary annotations. These annotations describe properties that are necessary for proving the correctness of the functional annotations. The tool that this work tries to improve is dedicated to inferring the auxiliary annotations.

To our knowledge, this is the first tool of its kind to automatically infer auxiliary annotations for a complete module given the module’s source code and its interface specifications. The work contributed in four areas: inferring annotations from the interface specifications of a module and propagating these annotations to all the helper functions used in the module; inferring annotations for floating-point constructs; inferring annotations for pointer constructs; and finally, inferring annotations for array constructs.

The improved tool was tested on production embedded code used in the heavy automotive industry. The results demonstrated a considerable improvement and were in line with earlier findings. The work confirms the feasibility and usability of auxiliary annotation inference in this scope.

Keywords

Formal verification, Automated verification, Contract inference.
Sammanfattning


Arbetet bidrog inom fyra områden: att härleda annoteringar från gränsnittsspecifikationerna (interface specifications) för en modul och sprida dessa annoteringar till alla hjälpfunktioner som används i modulen; härleda annoteringar för flyttalskonstruktioner (floating-point constructs); härleda annoteringar för pekarstrukturer; och slutligen, härleda annoteringar för arraystrukturer.

Det förbättrade verktyget testades på produktionsinbyggd kod som används inom fordonsindustrin. Resultaten visade en avsevärd förbättring och var i linje med tidigare resultat. Arbetet bekräftar genomförbarheten och användbarheten av hjälpannoteringshärledning i projektets omfattning.

Nyckelord

Formell verifiering, Automatiserad verifiering, Kontraktgenerering.
iv | Sammanfattning
Acknowledgments

Foremost, I would like to express my sincere gratitude to my supervisor, Christian Lidström, for his patience, motivation, encouragement, and above all, great knowledge. I would also like to thank Fredrik Diffner, Adnan Jamil, and Anders Renström for being such wonderful companions during the five-year journey that we had together at KTH. Lastly, I would like to thank the Swedish people for affording me the unimaginable opportunity to complete my study at one of Sweden’s leading universities. I’m immeasurably grateful for this generosity, and I hope I will be as generous towards Sweden as it was towards me.

Stockholm, July 2023
Hovig Manjikian
# Contents

## 1 Introduction

1.1 Background ........................................... 1
1.2 Problem .................................................. 3
   1.2.1 Original problem and definition ..................... 4
   1.2.2 Scientific and engineering issues ................. 4
1.3 Purpose and Impact on Society ........................ 4
1.4 Goals .................................................. 5
1.5 Research Methodology .................................. 5
1.6 Delimitations .......................................... 6
1.7 Structure of the thesis ................................ 6

## 2 Background

2.1 Formal verification ...................................... 7
   2.1.1 Deductive verification .............................. 7
   2.1.2 Hoare logic ...................................... 8
   2.1.3 Weakest Precondition .............................. 10
   2.1.4 Abstract interpretation ......................... 12
2.2 The C Programming Language ............................ 15
   2.2.1 ISO C99 ........................................... 15
   2.2.2 MISRA C ........................................... 15
2.3 Frama-C and the tool chain .............................. 16
   2.3.1 ANSI/ISO C Specification Language (ACSL) ........ 17
   2.3.2 The WP plugin ................................... 20
   2.3.3 The RTE plugin ................................... 23
   2.3.4 The EVA plugin ................................... 24
   2.3.5 Scania’s plugin ................................... 26
2.4 Related work ........................................... 28
   2.4.1 Synthesis of Annotations ......................... 28
   2.4.2 Automatic Inference of Necessary Preconditions ... 29
2.4.3 Inferring better contracts .................................. 30
2.4.4 Houdini, an Annotation Assistant for ESC/Java ..... 31
2.5 Summary ......................................................... 31

3 Method ......................................................... 33
3.1 Research Process ........................................... 33
3.2 Source code collection ..................................... 33
3.3 Identifying improvements .................................. 34
3.4 Evaluation .................................................... 35

4 Implementation ................................................. 37
4.1 Interface specification inference ......................... 37
  4.1.1 Translating the interface specifications .............. 38
  4.1.2 Propagating the interface specifications ............ 38
4.2 Floating point annotation inference ..................... 40
4.3 Pointer annotation inference ............................. 42
4.4 Array annotation inference ............................... 42
  4.4.1 Delimitation on expressions used as array index ... 42
  4.4.2 Delimitation on annotation simplification ........... 43
4.5 Summary .................................................... 44

5 Results and Analysis .......................................... 45
5.1 Results on Simplified Scania C-modules ................ 45
  5.1.1 Simplified Steering C-modules ....................... 45
  5.1.2 Simplified Accelerator Pedal C-modules ............ 48
5.2 Results on Scania Proprietary C-modules .............. 51
  5.2.1 Proprietary Scania Steering C-modules ............ 52
  5.2.2 Proprietary Scania Main Light Warning C-modules 58
5.3 Reliability Analysis ...................................... 59
5.4 Summary .................................................... 59

6 Discussion ..................................................... 61
6.1 Interface Specification Inference ......................... 61
6.2 Auxiliary Annotations for Floating-Points ................ 62
6.3 Auxiliary Annotations For Arrays ....................... 62
6.4 Auxiliary Annotations for Pointers ..................... 62
6.5 Auxiliary Annotations in General ....................... 62
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A possible abstraction for the set of integers Z</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>WP’s graphical output for floating-point arithmetic</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of the result of running ISP and WP</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Frama-C proof result on unsupported array constructs</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>WP’s output on a contradiction</td>
<td>56</td>
</tr>
<tr>
<td>5.2</td>
<td>ISP’s output on a module with an external import</td>
<td>58</td>
</tr>
<tr>
<td>A.1</td>
<td>A simplified requirements document</td>
<td>74</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Results achieved by Skantz (no annotations)</td>
<td>29</td>
</tr>
<tr>
<td>2.2</td>
<td>Results achieved by Skantz (high-level spec. annotations)</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>An interface specification in natural language</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>The results of ISP on the simplified steering C-module</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>Modified results on simplified steering module</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Results on simplified Scania accelerator pedal C-modules</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>Modified results on simplified accelerator pedal module</td>
<td>51</td>
</tr>
<tr>
<td>5.5</td>
<td>Results on the Scania steering C-modules</td>
<td>52</td>
</tr>
<tr>
<td>5.6</td>
<td>Results on the Scania main light warning C-modules</td>
<td>59</td>
</tr>
</tbody>
</table>
Listings

2.1 An example of global and local ACSL annotations. . . . . . . 18
2.2 An example of an ACSL behavior. . . . . . . . . . . . . . . . 19
2.3 A module demonstrating ghost variable usage. . . . . . . . . 19
2.4 An example of ACSL annotations. . . . . . . . . . . . . . . . 20
2.5 The output of the WP plugin on the file swap.c. . . . . . . . 20
2.6 Simple module for demonstrating WP’s limitation. . . . . . . 21
2.7 WP’s output for floating-point arithmetic. . . . . . . . . . . . 22
2.8 The output of the RTE plugin on file swap.c. . . . . . . . . 23
2.9 The output of WP with RTE preprocessing enabled. . . . . . . 24
2.10 The abstract function abs(). . . . . . . . . . . . . . . . . . 24
2.11 The output of the Eva plugin on the file abs.c . . . . . . . . 25
2.12 The function swap() without any annotations. . . . . . . . . 27
2.13 The output of Scania’s plugin . . . . . . . . . . . . . . . . . 28
4.1 A translation to an ACSL contract . . . . . . . . . . . . . . . . 38
4.2 Simple C module for a car movement state. . . . . . . . . . . . 39
4.3 A C-module with a function argument used as an index. . . . 42
4.4 Repeated ACSL annotation of the same property. . . . . . . . 44
4.5 An ACSL annotation for array cells with identical property. . . 44
5.1 A module to demonstrate EVA’s over-approximation. . . . . . 49
5.2 EVA’s over-approximates result. . . . . . . . . . . . . . . . . 50
5.3 A module to demonstrate the static variable issue. . . . . . . . 53
5.4 ISP’s output with static variable. . . . . . . . . . . . . . . . . 53
5.5 Modified output with static variable. . . . . . . . . . . . . . . 54
5.6 An example of an error in the entry-point contract. . . . . . . 55
5.7 A module importing an external module. . . . . . . . . . . . . 57
5.8 A header of an external module. . . . . . . . . . . . . . . . . 57
5.9 B.1 A simplified steering module. . . . . . . . . . . . . . . . 75
5.10 ISP’s output on simplified steering module. . . . . . . . . . . 79
5.11 B.3 Simplified steering module with functional specifications. . . 83
5.12 B.4 Corrected transformed simp. steering module. . . . . . . . 87
B.5 Corrected output for simp. steering module with func. sepcs. 91
B.6 A simplified accelerator pedal module. 98
B.7 ISP’s output on simplified accelerator pedal 100
B.8 The simplified accelerator pedal with func. spec. 103
B.9 Corrected output for simp. accelerator pedal 106
B.10 Corrected output for simp. accelerator pedal with func. spec. 109
## List of acronyms and abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACSL</td>
<td>ANSI/ISO C Specification Language</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>BISL</td>
<td>Behavioral Interface Specification Language</td>
</tr>
<tr>
<td>CIL</td>
<td>C Intermediate Language</td>
</tr>
<tr>
<td>EVA</td>
<td>Evolved Value Analysis</td>
</tr>
<tr>
<td>Frama-C</td>
<td>Framework for Modular Analyses of C</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>ISP</td>
<td>Interface Specification Propagator</td>
</tr>
<tr>
<td>MISRA</td>
<td>Motor Industry Software Reliability Association</td>
</tr>
<tr>
<td>SMT</td>
<td>Satisfiability Modulo Theories</td>
</tr>
<tr>
<td>WP</td>
<td>Weakest Precondition</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter gives a brief context to the problem of software incorrectness from an economic and historical perspective. It then describes how this problem is relevant to the project’s host company, Scania CV AB, and demonstrates the company’s ongoing effort to ensure high software quality. Finally, the chapter ends by stating the purpose, goals, and limitations of the project.

1.1 Background

It is easy to argue against spending time and resources on ensuring software safety, security, and reliability. This ease makes many higher-level managers in different companies neglect or overlook the software safety aspect of a project. Instead, they prioritize other aspects that they deem to have a greater impact on the business’s growth. Such negligence time and again led to accidents at different companies that caused severe economic damages. In some unfortunate cases, the damage was not limited to a company but extended to individuals or, even worse, whole societies. In 2014, an air-bag software bug forced Nissan Motor Corporation to recall nearly 1 million cars for a software fix, causing major economic losses for the company [1]; in 1999, a software bug put a US military satellite into a useless orbit, causing a loss of USD 1.2 billion [2]; and in 2014, Heartbleed, a security bug in the widely used OpenSSL cryptography library, made a large number of commercial and governmental websites that host sensitive services vulnerable for exploitation [3]. These are just a few examples of the many disastrous accidents caused by software malfunctions. The gravity and seriousness of accidents caused by software incorrectness have made many
companies realize the importance of the software safety aspect and thus adapt rigorous methods for ensuring software correctness to avoid such disastrous consequences. Legislators in many countries have also reacted to this problem by requiring stringent certification processes in many safety-critical industries. For example, in the avionics industry, the DO-178B standard “Software Considerations in Airborne Systems and Equipment Certification” was introduced in 1992 [4]. The document emphasizes the importance of software verification and recommends using software testing to verify the correctness of software.

Since the 1980s, software engineers have been ensuring the correctness of a given piece of software by performing different types of tests on it [5]. Acceptance tests, integration tests, and unit tests are among a long list of tests that are performed to verify that a software does what it is supposed to do. However, achieving 100% coverage of all possible states of a piece of software is rarely possible. Even simple software can have an enormous number of possible states, and covering all these states is often not feasible due to time and processing power limitations. As a result, most software tests fail to exhaustively verify the correctness of software, making it difficult for engineers to ensure the correctness of their software with high confidence.

Formal verification is a more recent alternative for ensuring software correctness. The advantage of formal verification is that it can verify software exhaustively. By using logic and mathematics, this method achieves 100% coverage of all possible states that a piece of software can take. Although new to the industry, formal verification has reached a good level of maturity to become a viable option to replace certain types of software testing. This led many companies to consider it as an alternative for ensuring software quality, and standards like the aforementioned DO-178B have been revised to accommodate this new technique. The new version of the DO-178B specification that emphasizes formal verification is the DO-178C, which was introduced in 2012 [6]. There is also a similar standard which is used in the automotive industry called ISO 26262 and is titled “Road vehicles – Functional safety” [7].

However, most companies today are still using conventional software testing in their quality assurance process, and only a very small number of companies are implementing formal verification. Even those who have implemented formal verification are limiting its use to the verification of the absolutely most safety-critical components of a piece of software. The reason for this limited use is that formal verification is not an easy task. In a paper that studied formal verification in the automotive industry, the authors found
that the process is difficult, very time-consuming, and complex [8]. The complexity of the process also makes it prone to errors, thus requiring highly skilled professionals to take care of it. This makes formal verification an expensive alternative that is less attractive to companies. In a case study on the formally verified OS microkernel seL4, it was concluded that the process of formal verification cost USD 700 on average for each line of code [9].

A solution to this problem is to automate parts of or the entire formal verification process. Scania CV AB, which is one of the leading companies in the automotive industry, is investing in the development of such automated tools that can be used to verify exhaustively the correctness of software in its vehicles. The company highly values its customers. It is not only striving to ensure the customers’ safety, but it also cares about the customers’ satisfaction and aims to eliminate any frustration or economic losses caused by software malfunctions. Furthermore, with Scania’s current goal of manufacturing and selling fully autonomous vehicles, the need for comprehensive verification tools is becoming increasingly important as the software in these vehicles assumes greater responsibility for critical decision making.

Scania’s solution uses Deductive Verification as its formal method to verify that a given module satisfies its functional specifications. Functional specifications are a set of requirements that describe the behavior of a module, and the module needs to implement all these requirements in order to be correct. The functional requirements in this solution are written as annotations in the source code in the ANSI/ISO C Specification Language (ACSL) [10], and the platform used for verifying that the module satisfies its specifications is Frama-C [11]. Nevertheless, in many cases, Frama-C fails to verify these module-level functional specifications without some low-level code specifications and low-level memory specifications. Throughout this report, we call these low-level helper specifications auxiliary specifications. In Scania’s solution, there is already a plugin being developed for the Frama-C platform to achieve an automatic inference of these auxiliary specifications from the source code and its annotations. However, this plugin is still under development and is currently limited to a subset of the C language constructs and a subset of the ACSL constructs [12].

1.2 Problem

The problem can be described on two levels: a higher-level problem that describes the general issue, and a more specific, lower-level engineering issue that lies behind the problem.
1.2.1 Original problem and definition

Current software quality assurance processes used by Scania and the automotive industry in general rely heavily on software testing, which makes it difficult to exhaustively verify software correctness. Although formal verification achieves the goal of exhaustively verifying software correctness, the difficulties associated with it make it less accessible for these quality assurance processes. Consequently, exhaustive verification of software remains a difficult goal to realize, making vehicles vulnerable to software bugs.

1.2.2 Scientific and engineering issues

Scania is putting great effort into developing a solution that automates the process of formal verification to alleviate the difficulties associated with the technique. The solution is, however, still under development and achieves only a partial automation of the verification process. The main issue that still prevents complete automation is a limitation in the solution’s specification inference mechanism. Thus, the research question is:

*How can existing tools for automatic specification inference for the Frama-C platform be improved, specifically when the tool is used on sequential application-level embedded software written in the ANSI C language?*

1.3 Purpose and Impact on Society

The existing tool developed by Scania today is still unable to automatically verify a complete module without manual intervention in the form of writing some extra auxiliary ACSL specifications. Now, the purpose of this project is to improve the existing solution so the extra manual effort is eliminated. This could be achieved mainly by identifying new methods that could improve the existing specification inference mechanism so that all the auxiliary ACSL specifications can be inferred automatically, and thus a complete automatic verification will be possible at least for some of Scania’s modules, if not all.

The results of this project is interesting not only to Scania but to other parties as well. Any organization, company, or individual interested in software robustness may benefit from this work because the concepts that are used in this project are not limited to the embedded software or the C programming language used by Scania.
There is also a theoretical significance to the results of this work. Since the automation of formal verification for arbitrary software is not possible due to the Halting Problem [13] and Gödel’s Incompleteness Theorem [14], it is interesting to see to what extent these limitations affect the automation of formal verification for a restricted subset of software that runs in a well-defined environment.

Finally, the results of this work also have an impact on society. As demonstrated earlier in Subsection 1.1, malfunctioning software could cause enormous damage to society in terms of economic damage and loss of lives. Hence, since the results of this work contribute to reducing bugs in software and improving the correctness of software, the damage caused to society by software safety issues will be reduced.

1.4 Goals

The goal of this project is to improve the inference mechanism implemented in Scania’s current solution. This has been divided into the following two sub-goals:

1. Identify missing functionalities in the current inference mechanism that will improve the inference of auxiliary specifications.

2. Implement the missing functionalities to ultimately achieve a completely automatic formal verification process.

1.5 Research Methodology

The method for this work consists mainly of two parts. The first part is to identify the issues that are hindering the existing tool from achieving complete automatic verification. This is done mainly by studying the C source code to identify C constructs that the tool does not cover and by examining verification cases where verification fails. The second part is finding and implementing a solution to some or optimally all of the identified issues. The implantation of the solutions is done iteratively in short sprints, where each spring a single solution is implemented and evaluated.
1.6 Delimitations

Formal verification could be applied to a wide variety of software. In this project, however, we focus on application-level embedded software that is written specifically in the ANSI C language. Only a subset of the C language is used, which is similar to the MISRA C subset [15]. Moreover, we focus only on sequential software, and thus, parallel software is not considered.

Scania’s project as a whole consists of many components. Nonetheless, this project only focuses on improving the component that is responsible for inferring what we call “auxiliary specifications”.

1.7 Structure of the thesis

Chapter 2 presents relevant background information about formal verification techniques, programming and annotation languages, and related work in the area of contract inference. Chapter 3 presents the methods used to identify improvements and improve the inference mechanism. Chapter 4 describes the implementation of the improvements that were made to the existing solution. Chapter 5 presents the results that were achieved by the implemented method and their analysis. Chapter 6 includes a discussion about the achieved results. Finally, Chapter 7 presents a conclusion and future work.
Chapter 2

Background

This chapter provides basic background information about formal verification, function contracts, and contract inference mechanisms. Additionally, the chapter includes descriptions of some specific tools used in this project, and related work in the area of automated formal verification.

2.1 Formal verification

Formal verification is a process used during software development to exhaustively ensure that a piece of software works reliably. It uses formal methods of mathematics that consist of languages, techniques, and tools that are used to verify that a given software fulfills its specifications and properties. There are many mathematical languages, techniques, and tools used in the area of formal verification. This project uses two languages: a subset of the ANSI C language used for writing source code and the ANSI/ISO C Specification Language (ACSL) used for annotations. These languages are explained in detail in Subsections 2.2, and 2.3.1. The techniques used in this project are deductive verification, abstract interpretation, Hoare logic, and weakest precondition, which are demonstrated in the following subsections. Finally, the only tool used in this project is the Frama-C verification platform, which is demonstrated in Section 2.3.

2.1.1 Deductive verification

Deductive verification is a formal method used to exhaustively verify that all possible behaviors of a given system satisfy a formally defined behavior. The approach in this method is to generate proof obligations based on the
system, the system’s specifications, and the system’s environment. These proof obligations are then discharged, ideally with the help of an automatic theorem prover like an SMT solver. When this method successfully discharges all of its proof obligations, the system can be guaranteed to fulfill the provided specifications.

In this approach, the functional specifications of the system can be annotated using a Behavioral Interface Specification Language (BISL). These annotations can give a formal description of the source code using preconditions, postconditions, invariants, and assertions. The formality of the language eliminates any vagueness by precisely describing the desired behavior of the source code. More importantly, the formalization facilitates the use of the language in conjunction with automated analysis and software verification tools [16]. In Subsection 2.3.1, the specific type of BISL used in this project is explained with some examples.

The difficulty with deductive verification, however, is that the developer needs to have a very good understanding of the system and its environment to be able to generate the necessary proof obligations. This makes this method tedious and difficult when done manually.

2.1.2 Hoare logic

Hoare logic, which is also known as Floyd-Hoare logic, is a framework used for verifying the behavior of software. The framework is based on Hoare’s work [17], which in its turn is based on some concepts introduced by Floyd [18]. The central concept in Hoare logic is the Hoare triple, which is represented by

\[ \langle P \rangle \ C \ \langle Q \rangle \]

where \( P \) is an assertion called the precondition, \( C \) is a statement called the command, and \( Q \) is an assertion called the postcondition. The semantics for partial correctness of the Hoare triples states:

1. If the state of the system satisfies the precondition \( P \) and we execute the command \( C \) and the execution of \( C \) terminates,

2. then the state of the system after \( C \)'s termination satisfies the postconditions \( Q \).

In Hoare logic, there are different rules that are used for proving the Hoare triple, depending on what the command \( C \) is. For example, when the command
$C$ is an assignment instruction, the proof rule for assignment can be used, which is stated as

$$\langle Q[E/x]\rangle \ x := E \ \langle Q \rangle$$

where $Q[E/x]$ means the expression $E$ substitutes $x$ in the assertion $Q$. In the following Hoare triple example,

$$\langle x = 0 \rangle \ x := x + 1 \ \langle x = 1 \rangle$$

the proof rule for assertion can be used, which gives us

$$\langle (x = 1)[(x + 1)/x]\rangle \ x := x + 1 \ \langle x = 1 \rangle$$

$$\Rightarrow \langle x + 1 = 1 \rangle \ x := x + 1 \ \langle x = 1 \rangle$$

$$\Rightarrow \langle x = 0 \rangle \ x := x + 1 \ \langle x = 1 \rangle$$

which proves the validity of the given Hoare triple.

The proof rule for assignment is, however, not sufficient for proving software in general. For this purpose, Hoare logic encompasses three additional rules: a rule for handling multiple commands, a rule for branching instructions, and a rule for loop instructions. We introduce all the three rules now and explain their mechanics, but we skip digging deep into the definition behind these rules to avoid a lengthy explanation that will not have a significant contribution to the understanding of this project.

The rule for handling multiple instructions is the proof rule for composition:

$$\langle \langle P \rangle \ C_1 \ \langle R \rangle \ C_2 \ \langle Q \rangle \rangle$$

$$\Rightarrow \langle P \rangle \ C_1; C_2 \ \langle Q \rangle$$

and in this rule, a new assertion $R$ is introduced, which is used to decompose the set of commands into multiple Hoare triples.

The rule for handling branching instructions is the proof rule for if statements:

$$\langle \langle P \wedge E \rangle \ C_1 \ \langle Q \rangle \rangle \ \langle \langle P \wedge \neg E \rangle \ C_2 \ \langle Q \rangle \rangle$$

$$\langle P \rangle \text{ if } E \text{ then } C_1 \text{ else } C_2 \ \langle Q \rangle$$

where $E$ is an expression representing the branching condition.

Finally, when it comes to loops, a given loop is converted into a while-loop, and the rule for while-loop handling is the proof rule for while:

$$\langle I \wedge E \rangle \ C \ \langle I \rangle$$

$$\langle I \rangle \text{ while } E \text{ do } C \ \langle I \wedge \neg E \rangle$$
where \( I \) represents an assertion called the invariant and \( E \) is an expression that represents the while-loop’s condition. The invariant is an assertion that needs to be true before executing the while-loop, during the while-loop execution, and after the while-loop terminates. For example,

\[
\langle x = 0 \rangle \text{ while } \{ x := 0 \} \langle x = 1 \rangle
\]

is a partially valid Hoare triple, because the command never terminates, thus satisfying the Hoare triple rule. Another example in which the proof rule for while can be used is the Hoare triple

\[
\langle x = 0 \land y = 0 \land n = 4 \rangle \\
\text{while } x < n \text{ do } \{ x := x + 1; y := x + y \} \\
\langle y = 10 \rangle
\]

where the invariant \( I \) can be \( y = \frac{x(x+1)}{2} \land x \leq n \land n = 4 \) and the loop condition \( E \) is \( x < n \). Thus,

\[
\langle I \land E \rangle \ x := x + 1; y := x + y \langle I \rangle \\
\langle y = x(x + 1)/2 \land x \leq n \land n = 4 \land x < n \rangle \\
\text{while } x < n \text{ do } \{ x := x + 1; y := x + y \} \\
\langle y = x(x + 1)/2 \land x \leq n \land n = 4 \land x \geq n \rangle
\]

can prove the original Hoare triple, because conjoining the \( x \leq n \land n = 4 \) in the invariant with the postcondition \( x \geq n \) gives \( x = n = 4 \). This means

\[
y = \frac{x(x+1)}{2} = \frac{4(4+1)}{2} = 10
\]

which is exactly what the postcondition was in the original Hoare triple.

### 2.1.3 Weakest Precondition

Weakest precondition or weakest precondition calculus is a method of formal verification based on the works of Floyd [18], Dijkstra [19], and Hoare [17].

The essence of this approach is to calculate the weakest precondition \( P \) in the Hoare triple \( \langle P \rangle \ C \langle Q \rangle \) that makes the Hoare triple valid in all the states where \( P \) is true.

We say that a condition \( A \) is weaker than a condition \( B \) if the condition \( B \) implies \( A \), i.e., \( B \Rightarrow A \). Correspondingly, condition \( B \) is considered stronger
The weakest precondition can be obtained by substituting the expressions of the command \( C \) in the postconditions \( Q \). Thus, the weakest precondition can be seen as a function of the command and the postcondition, as seen in the following Hoare triple:

\[
\langle \text{wp}(C, Q) \rangle \ C \ \langle Q \rangle
\]

where the function \( \text{wp} \) has a different definition for different commands in \( C \). For example, the definition that is used for the assignment command \( x := E \) is

\[
\text{wp}(x := E, Q) = Q[E/x]
\]

where the notation \( C[B/A] \) means \( B \) substitutes \( A \) in \( C \). For an assignment command like \( x := x + 1 \) that has the postconditions \( x > 0 \), the function \( \text{wp} \) can be used in the following way:

\[
\langle (x > 0)[(x + 1)/x] \rangle \ x := x + 1 \ (x > 0)
\]

\[
\langle x + 1 > 0 \rangle \ x := x + 1 \ (x > 0)
\]

which gives us the weakest precondition \( x + 1 > 0 \).

This technique can be used for verifying software contracts where the contracts represent preconditions, assertions, and postconditions. Given a Hoare triple

\[
\langle P \rangle \ C \ \langle Q \rangle
\]

we can obtain the weakest precondition \( W \) by applying the function \( \text{wp}(C, Q) \) and thus get the Hoare triple

\[
\langle W \rangle \ C \ \langle Q \rangle
\]

which we know to be valid by the definition of \( \text{wp} \). Now, if we can prove that \( P \) implies \( W \), or in other words, that \( W \) is a weaker condition than the \( P \) condition, then we can conclude that when \( P \) is true, \( Q \) will also be true. This proves that the original triple

\[
\langle P \rangle \ C \ \langle Q \rangle
\]

is a valid one.

We can formalise this technique with the following proof rule:

\[
\frac{P \Rightarrow W \quad \langle W \rangle \ C \ \langle Q \rangle}{\langle P \rangle \ C \ \langle Q \rangle}
\]
Proving a piece of software using this technique can be done as follows. First, the assertion that represents the postcondition $Q$ of the software is introduced. This assertion is based on the functional specifications of the software. Then, the weakest precondition of all the statements in the software is calculated, starting from $Q$ and traversing backwards towards the software’s entry point. The precondition which is calculated at the entry point is the weakest precondition $W$ for the software as a whole. Next, the set of conditions $P$ is introduced. $P$ represents the actual state of the software and its context at the entry point. At this point, the verification of the software is reduced to a proof obligation of the property $P \Rightarrow W$, and thus, the last step is to discharge this proof obligation, which can be done either manually or by using an automatic theorem prover. If the proof obligation is discharged, the software is then guaranteed to satisfy its specifications. On the other hand, if the proof obligation fails to be discharged, then we cannot say anything, because there are many possible underlying reasons for such a failure. A possible reason for the failure could be that a precondition that causes a software malfunction has been found. Another possible reason is that the prover is insufficient. A third reason could be that the halting problem has occurred. All of these reasons are possible, and the list is not exhaustive.

This technique is very powerful. Nevertheless, the challenge is that calculating the weakest preconditions throughout the program is tedious work. Introducing the precondition $P$ that represents the condition of the software’s state and its context is also tricky and would require good knowledge of the systems components, such as the memory model in use. However, these challenges could be eliminated if automation were introduced to the process.

\subsection{Abstract interpretation}

Abstract interpretation is another technique used in the area of formal verification. The method of this technique is to create an abstract model that would correspond to the source code of the software, and then evaluate the abstract model to get some insights into the execution of the original source code without actually executing it [20].

The abstract model consists of two components: (i) an abstract set of values that correspond to the concrete values that variables can take; and (ii) abstract functions that correspond to the concrete functions in the source code. As illustrated in Figure 2.1, the abstract set of values (blue) consists of symbolic values rather than actual values, which can be seen in the concrete set (black). The relationship between the concrete set of values and the abstract set of
values is defined by the *abstraction function* $\alpha$, which is a total function from the set of values in the concrete set to the abstract set. Thus, if $L$ is a concrete ordered set of values and $L'$ is an abstract ordered set of values and

$$x \in L \implies \alpha(x) \in L'$$

then we say that $\alpha(x)$ is an abstraction of $x$.

To define the abstraction for the concrete functions $f$, we need to first define the *concretization function* $\gamma$. A function $\gamma$ is said to be a concretization function when

$$x \in L \land \alpha(x) \in L' \implies x \in \gamma(\alpha(x))$$

where $L$ is the concrete set of values, $L'$ is the abstract set of values, and $\alpha$ is the abstraction function. Given the ordered sets $L_1, L_2, L'_1$ and $L'_2$, as well as the function $f$ from $L_1$ to $L_2$, we can say that a function $f'$ from $L'_1$ to $L'_2$ is
an abstraction of $f$ if

$$\forall x' \in L_1', f(\gamma(x')) \subseteq \gamma(f'(x'))$$

A practical example of the abstract interpretation technique, which is inspired by Sintzoff’s work [21], can be demonstrated in the following mathematical equation

$$x = 32 \ast -2334$$

where a possible abstract representation can be created using the symbols $(-)$, $(+)$, and $(\pm)$. The abstract representation is then

$$x = (+) \ast -(+)$$

and evaluating this abstraction gives us

$$x = (+) \ast -(+)$$

$$\Rightarrow x = (+) \ast (-)$$

$$\Rightarrow x = (-)$$

which provides us with the insight that $x \in [-\infty, 0)$.

Abstract interpretation is used to prove the functional specifications of software. However, this technique over-approximates the possible values of a variable, because if $S$ is the set of concrete values that a variable $i$ can take, then $S \subseteq \gamma(\alpha(i))$, which implies that $\alpha(i)$ is an abstraction that represents some additional values in the concrete set that do not necessarily exist in $S$. This makes the technique prone to generating false alarms by discovering faulty states that contain values that are in reality unreachable. Nevertheless, when the technique does not raise any alarm, then the developer can be sure that the program fulfills its functional specification.

Abstract interpretation can be used in conjunction with other formal methods when proving large software with complex contracts that contain many functional specifications. An advantage of abstract interpretation is that it is cheap when it comes to computation complexity. This is mainly due to the fact that the generated abstract model is often much simpler than the original code, and thus requires a lower number of computations during the proof process. As a result, abstract interpretation scales well with the size of software, making it suitable to prove the correctness of arbitrary software of any size.
2.2 The C Programming Language

The C programming language was developed by Bell Laboratories in 1972 [22]. Many of the language's features were taken from the older programming languages B, BCPL, and CPL. For more than a decade, the C language had no standards, which meant that the code was not easily portable between different compilers or re-usable in other projects. The first official standard, commonly referred to as C89, was introduced in 1989 by the American National Standards Institute (ANSI). Since the release of C89, ANSI and the International Organization for Standardization (ISO) have released multiple successor standards for the C programming language, alleviating the problem of portability.

2.2.1 ISO C99

The ISO C99, officially named ISO/IEC 9899:1999, is the standard used in this project for the C programming language. The standard was introduced in 1999 and updated the previous C90 standard [23]. The new features of the standard include, among other things, better hardware support for floating point arithmetic, inline functions, new data types, single-line comments, and some new headers such as `<tgmath.h>` and `<inttypes.h>`.

2.2.2 MISRA C

MISRA C compliance was developed by a collaboration of manufacturers, component suppliers, and engineering consultancies [15]. The initial release was in 2016, and it aimed to achieve better software safety, security, and reliability in both embedded control systems and general application-level software. The compliance promotes some guidelines for development practices that include:

- Training for the use of the chosen programming language so that safety, security, and integrity are taken into consideration. The use of compilers and complex static analysis tools is also included in the training.

- Style guidelines on code layout, indentation, use of braces, naming conventions, and comments.

- Software metrics that will eliminate problems such as unwieldy and untestable code. More source code metrics can be found in Fenton
and Bieman’s work “Software Metrics: A Rigorous and Practical Approach” [24].

This compliance also includes some requirements on the choice of tools. The compiler, for example, needs to comply with a standard (ISO C99 in this project), and the static analysis tool should not raise false alarms but only report genuine violations.

Adherence to the MISRA C Compliance, however, does not guarantee the development of error-free, robust software. does not either entail portability or re-use.

2.3 Frama-C and the tool chain

Frama-C is an open-source code analysis platform dedicated to analyzing source code written in the C programming language [11]. It includes a collection of plugins that use different analysis techniques to perform static or dynamic analysis, and it can be extended with user-developed plugins. The platform offers a collaborative framework that allows plugins to use the results from other plugins in the analysis process. This collaboration simplifies the analysis of sophisticated source code in which many analysis techniques are needed.

The platform has its own modified implementation of the C Intermediate Language (CIL). CIL is a high-level representation of parsed C source code, but it has fewer constructs than the C programming language. Thus, complicated C constructs are broken down into simpler constructs in the CIL representation, which makes the abstract syntax tree (AST) of CIL a lower level representation than a typical AST of a compiler.

Frama-C’s implementation of CIL parses C99 source code and is extended to parse ACSL annotations as well. The ACSL annotations are discussed more thoroughly in Subsection 2.3.1. The created AST is then shared among all plugins, which allows collaboration in a consistent manner.

When discussing code analysis, one might think of measuring the code using different metrics like the proportion of comment lines to the source code, cyclomatic complexity, or the depth of nested control structures. This kind of analysis is called syntactic analysis, and although it is possible to implement this kind of analysis in Frama-C, this is not the purpose of the platform and this project. Frama-C has the following two goals:

• to prove that the source code does not contain any dangerous construct that could cause a run-time error,
• to prove that the source code adheres to its functional specifications that are annotated in the functions’ contract.

Frama-C achieves these goals by (i) performing \textit{static analysis}, which is to compute synthetic information about the source code without executing it, or (ii) performing \textit{dynamic analysis}, which is to execute the source code and watch the state of the program and perform run-time verification. In this project, however, only static analysis is being used. In the following subsections, we introduce the different tools and plugins used in this project.

\subsection*{2.3.1 ANSI/ISO C Specification Language (ACSL)}

The behavioral specification language (BISL) used in Frama-C for C source code is ACSL. ACSL is a \textit{formal} language that can express a wide range of functional specifications \cite{10}. The language is designed to facilitate writing functional specifications as comment annotations and eliminate the need to write run-time assertions for expressing these kinds of specifications.

ACSL annotations have two types: \textit{global annotations} and \textit{local annotations}. Global annotations can be function contracts written before the function declaration, global invariants written next to the global declarations, or logic specifications also written next to the global declarations. Local annotations can be assertions, which can be placed at any place in the source code, loop annotations such as invariant, variant, and assignment, which are written before the loop instructions, or a statement contract, which is written before a statement or a block of statements.

An example of the ACSL annotation, which is taken from ACSL’s documentation \cite{10}, can be seen in Listing 2.1. In this example, lines 2–7 represent a global annotation, which is a function contract. Lines 2 and 3 of the contract are some low-level specifications of the C programming language, and lines 5 and 6 are functional specifications for the function \texttt{set_to_0}. Lines 11–16 are local annotations representing a loop invariant, variant, and assignment.

Moreover, the example in Listing 2.1 demonstrates how the syntax is not only human-readable but also computer-manipulable. These annotations can be generated by software, and in fact, there are some plugins in Frama-C that generate or infer new ACSL annotations when visiting and analyzing the CIL representation of a given source code.

The lexical rules in ACSL mostly follow those of the C Language. ACSL annotations are written as comments that start with the symbols \texttt{\*@} or \texttt{\@} and end as usual in the C language, which means that the annotations do not
affect the compilability of the code. An important difference to note is that the lexical rules of ACSL allow the use of some UTF-8 characters instead of the standard C constructs, e.g. the character $\geq$ can be used instead of $\geq$, and the character $\land$ instead of $\&\&$. The ACSL manual contains the complete list of the available UTF-8 symbols [10]. The manual also lists other differences in the lexical rules compared with the C language.

The most common clauses in the ACSL annotations are requires, assigns, and ensures. Listing 2.1 demonstrates all of these clauses. The semantics in this example is as follows:

- The requires clause means that the caller of the functions needs to guarantee the properties stated in the clause, which in this case requires valid read-write memory addresses in the range of $a[0]..a[n-1]$.

- The assigns clause means that only the memory locations that are listed in the clause might be mutated and nothing else. In this example, only the memory locations in the range $a[0]..a[n-1]$ can be mutated.

- The ensures clause means that the called function returns a result in which the listed properties hold. In this example, the property is that all the memory locations in the range of $a[0]..a[n-1]$ contain the integer value 0.
Listing 2.2: An example of an ACSL behavior named $b$.

Listing 2.3: A module demonstrating ghost variable usage. The ensures specification specifies that the ghost variables value at the functions exit-point, which represents the value of the local variable $input$.

When writing a function contract, these clauses can be grouped into a behavior. Listing 2.2 is an example of a function contract that contains a behavior named $b$. The semantic of such a contract is that the calling function must guarantee that if $A$ is true, then $R$ must also be true, and when the calling function has guaranteed that, the called function must guarantee that $E$ will be true and only the global variable $L$ will be assigned.

Finally, sometimes there will be the need to write some requirements for some local variables that are only accessible inside the scope of a function or a block. ACSL provides ghost variables and ghost statements for this purpose. These variables and statements are like C variables and statements, but they do not affect the logic in the C code. They are only visible in the specification. Listing 2.3 demonstrates a module that contains a local variable. To write a function contract with an ACSL specification for the local variable $input$, a ghost variable, $model\_input$, is introduced, and a ghost statement is added at line 9.
2.3.2 The WP plugin

The WP plugin is used to prove that a source code fulfills its ACSL contract. The tool uses weakest precondition calculus, which was introduced in Subsection 2.1.3. WP proves the ACSL contract using its own prover, Qed, or by delegating the proof obligations to an external automatic SMT prover, such as Alt-Ergo.

We can prove that the swap function, introduced in Listing 2.4, fulfills its ACSL contract by running the instruction:

```bash
$ frama-c -wp swap.c
```

and the output of the execution will be

```
[kernel] Parsing intro_example.c (preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 4 goals scheduled
[wp] [Qed] Goal typed_swap_assigns_part2 : Valid
[wp] [Qed] Goal typed_swap_assigns_part1 : Valid
[wp] [Qed] Goal typed_swap_ensures_B : Valid
[wp] [Alt-Ergo] Goal typed_swap_ensures_A : Valid
[wp] Proved goals: 4 / 4
Qed: 3
Alt-Ergo 2.4.1: 1
```

Listing 2.5: The output of the WP plugin after analyzing the file swap.c which contains the code in Listing 2.4.

The output of WP in Listing 2.5 demonstrates how the Frama-C kernel and the WP plugin analyze and prove the swap function. In the output, the line that begins with [kernel] shows how first Frama-C’s kernel preprocesses the code and creates the CIL representations. After that, the WP plugin visits the CIL tree and discovers four goals. The term goal in the Frama-C context
Listing 2.6: A simple module for demonstrating WP’s limitation with floating-point arithmetic.

```c
int X;
int XMAX;
double Y;
double YMAX;

/*@ requires \valid(&X);
@ requires \valid(&XMAX);
@ requires X >= 0 && X <= 30;
@ requires XMAX == 30;
@
@ requires \valid(&Y);
@ requires \valid(&YMAX);
@ requires Y >= 0.0 && Y <= 30.0;
@ requires YMAX == 30.0;
@ requires \is_finite(\div_double(Y, YMAX));
@ requires \is_finite(\mul_double(\div_double(Y, YMAX), (double)100.0));
@
void main() {
    int i = (X / XMAX) * 100;
    //@ assert i \in {0, 100};
    int i \in {0, 100};
    double j = (Y / YMAX) * 100.0;
    //@ assert j >= 0.0 && j <= 100.0;
    //@ assert j >= 0.0 && j <= 100.0;
}
```

means a proof obligation. Three of the discovered goals are proven by the Qed prover, and the last goal is delegated to the external SMT prover, Alt-Ergo. The last 3 lines of the output give a summary of the results and show that the function `swap` fulfills its ACSL contract for all possible executions.

Notice that the output contained a warning message “missing RTE guards”. This message can be ignored for now, as the RTE plugin is discussed in detail in Subsection 2.3.3. Nevertheless, it is important to mention that the cause of this message is that the WP plugin assumes that the source code does not contain run-time errors, such as division by zero. Consequently, WP will prove that the source code adheres to its contract, but it will not know whether the contract guarantees the absence of run-time errors or not.

WP have many limitations. One limitation which is relevant to this work is WP’s inability to prove goals that involve floating-point arithmetic. Listing 2.6 demonstrates a simple module that includes two statements. The first statement at line 19 is an assignment that contains an expression of integer arithmetic. The second statement at line 22 is a similar assignment but the expression here contains floating-point arithmetic. Two assertions are made, one for each statement.

Executing WP on this module to prove the two assertions, yields the output
in Listing 2.7. The output shows that one of the goals was not proved. The output does not say clearly which one of the assertions is failing to be proved.

Listing 2.7: The output of the WP plugin when applied to the module that contains a floating-point arithmetic

To see more details we can run the graphical version of Frama-C by executing the command \texttt{frama-c-gui} instead of \texttt{frama-c}. The graphical output, see Figure 2.2, shows that the SMT provers have managed to prove the assertion for the integer arithmetic, but failed for the floating-point arithmetic. Frama-C indicates proved goals with a green bullet, while orange bullets indicate that a goal is not proved.

Figure 2.2: The graphical output of the WP plugin when applied to the module that contains a floating-point arithmetic.
Listing 2.8: The output of the RTE plugin after analyzing file swap.c, which contained the code in Listing 2.4. The command-line arguments -then -print were used to output the result into the terminal.

2.3.3 The RTE plugin

The RTE plugin is used for generating ACSL annotations for statements in the source code that can potentially lead to undefined behavior. The plugin is usually used as a preprocessor for other plugins, such as the WP plugin.

Applying the RTE plugin on the previous Listing 2.4 can be done with the command:

```
$ frama-c -rte swap.c -then -print
```

which will return the output seen in Listing 2.8.

Looking at the output in Listing 2.8, one can see that the annotations in lines 1 to 7 are generated by Frama-C. These lines are the Frama-C’s representation of the ACSL contract that was included with the source code. However, lines 10, 12, 13, and 15 are all ACSL annotations that were generated by the RTE plugin. In this example, the RTE plugin detected access to some specific memory addresses, so when a read statement was encountered, the mem_access: \valid _read() annotation was generated, and, respectively, when a write statement was encountered the mem_access: \valid () annotation was generated.

Now that RTE has been introduced, we can redo the WP example in Subsection 2.1.3, but this time by using the RTE plugin as a preprocessor before executing the WP plugin. The command is
$ frama-c -wp -wp-rte swap.c

where the command-line argument \texttt{-wp-rte} tells the WP plugin to call the RTE as a preprocessor before creating the proof obligations. The output of this execution will be

```
[kernel] Parsing intro_example.c (preprocessing)
[rte] annotating function swap
[wp] 8 goals scheduled
[wp] [Qed] Goal typed_swap_assert_rte_mem_access_2 : Valid
[wp] [Qed] Goal typed_swap_assumes_B : Valid
[wp] [Qed] Goal typed_swap_ensures_B : Valid
[wp] [Qed] Goal typed_swap_assigns_part1 : Valid
[wp] [Qed] Goal typed_swap_assumes_A : Valid
[wp] [Qed] Goal typed_swap_ensures_A : Valid
[wp] 8 / 8
[Qed]: 7
Alt-Ergo 2.4.1: 1
```

Listing 2.9: The output of the WP plugin with RTE preprocessing enabled executed on the file \texttt{swap.c} which contains the code in Listing 2.4.

The output in Listing 2.9 shows that, as expected, the RTE plugin has visited the CIL tree and inserted additional annotations that ensure run-time error free source code. After that the WP plugin visited the modified CIL tree and detected eight proof obligations, which is exactly four more proof obligations than the original execution with no RTE. The WP plugin did not emit any warnings this time, because now it knows that the RTE plugin was used and the code is run-time error free.

### 2.3.4 The EVA plugin

EVA, or the Evolved Value Analysis plugin, is a tool that uses abstract interpretation techniques to compute the variation domains\footnote{The term \textit{variation domain} is not a well established term within mathematics or computer science. This term is used in EVA’s documentation and it refers to the \textit{range of a variable} in computer science, i.e., the possible values that may be stored in a variable.} of variables \cite{EVA}. The plugin not only computes the variations domains but also it emits an error if the computed variations domain could cause a run-time error.

```c
int abs(int x) {
    if (x < 0) return -x;
    else return x;
}
```

EVA, or the Evolved Value Analysis plugin, is a tool that uses abstract interpretation techniques to compute the variation domains\footnote{The term \textit{variation domain} is not a well established term within mathematics or computer science. This term is used in EVA’s documentation and it refers to the \textit{range of a variable} in computer science, i.e., the possible values that may be stored in a variable.} of variables \cite{EVA}. The plugin not only computes the variations domains but also it emits an error if the computed variations domain could cause a run-time error.
Listing 2.10 is an example, taken from the Eva documentation [25], of a function `abs` that corresponds to the mathematical absolute function. This function, however, contains a bug that could cause a run-time error: signed overflow. By using the command

```bash
$ frama-c -eva abs.c -main abs
```

Eva analyzes the function and detects the bug. The output of this execution will be

```
[kernel] Parsing eva_example.c (with preprocessing)
[eva] Analyzing a complete application starting at abs
[eva] Computing initial state
[eva] Initial state computed
[eva:initial-state] Values of globals at initialization
[eva:alarm] eva_example.c:3: Warning: signed overflow. assert -x ≤ 2147483647;
[eva] done for function abs
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function abs: __retres ∈ [0..2147483647]
[eva:summary] ====== ANALYSIS SUMMARY ======
--------------------------------------------------
1 function analyzed (out of 1): 100% coverage.
In this function, 8 statements reached (out of 8): 100% coverage.
--------------------------------------------------
No errors or warnings raised during the analysis.
--------------------------------------------------
1 alarm generated by the analysis:
 1 integer overflow
--------------------------------------------------
No logical properties have been reached by the analysis.
```

Listing 2.11: The output of the Eva plugin on the file `abs.c` which contains the code in Listing 2.10.

The output in Listing 2.11 demonstrates how Eva emits an alarm when it detects a variable variation that could cause a signed overflow. In ISO C99, the value of a variable of type `int` is defined to be in the interval $[-2147483648, 2147483647]$. As a result, Eva has computed the variation domain of the return result to be __retres ∈ [0..2147483647]. At the same time, Eva sees that the variable x is also of type `int` and could take the value -2147483648, which will lead to an undefined behaviour.
When the Eva plugin does not emit any alarms, it means that the source code is run-time error free. It is relevant to mention here that in the verification process, if the Eva plugin has already visited the CIL tree and proved the absence of run-time errors, then there is no need for the RTE plugin to visit the tree and generate annotations that will be used to prove the same property again.

2.3.5 Scania’s plugin

Scania is developing a plugin for automatically inferring auxiliary specifications as part of the project to automate the formal verification process. The plugin will annotate in ACSL the source code with the newly inferred specifications. These annotations are then used by other plugins in the successor steps to prove the software’s correctness.

The purpose of the project as a whole is to prove that a given module is

- run-time error free, and
- adheres to its functional specifications.

The goal is that during the software development process, the developers will only need to specify the higher level functional specifications as an ACSL contract at the module level. The plugin then traverse the hierarchy of the module and, for each of the functions and global statements, it creates the ACSL annotations that are necessary to prove the absence of run-time errors and the adherence to the higher level ACSL contract.

The plugin that takes care of inferring the auxiliary specifications relies on two sources for its inference: (i) the source code combined with the specifications of the programming language, which is the ISO C99; and (ii) the interface specifications that are provided with the module.

The source code and the C99 specification provide many restrictions that can be converted into ACSL contracts. For example, when the plugin encounters an arbitrary variable $x$ which has the type `int`, it assumes that $x$ will have the range of $[-2147483648, 2147483647]$ based on the definition of the type `int`. This is, of course, true given that the plugin cannot make any stronger inferences about $x$ from the context.

Interface specifications also provide restrictions that can be converted into ACSL contracts. Interface specifications provide information about the interfaces of other modules that the module under verification might interact with. For example, this could be information about the range of values that a
certain memory location could hold. This memory location can represent any external component, e.g., the state of certain hardware that has four states: 00 (off), 01 (on), 10 (faulty), and 11 (unavailable).

In its inference mechanism, the plugin depends on external plugins like EVA, RTE, and From. The results from these plugins are composed into a complex contract, which not only ensures the absence of run-time errors, but also helps the successor plugins in proving the higher level functional specifications.

The plugin, however, is still under development so currently it only supports a subset of MISRA ISO C99 constructs, and a subset of the ACSL annotations. The following list includes identified constructs that the plugin does not support yet.

- Floating points in C.
- Deep nested pointers in C.
- Static local variables in C.
- Union types in C.
- Interface specifications in ACSL.

To demonstrate how the plugin works, we execute it on the function `swap()` in Listing 2.12, which is the exact same function `swap()` that was introduced in Listing 2.4 but with the annotations stripped away. The command for executing the plugin is

```
$ frama-c -load-module "top/Auto" swap\_clean.c -main swap -ocode "out.c"
```

which transforms the file `swap_clean.c` by annotating it. The result is then stored in the file `out.c`, which is shown in Listing 2.13. In this listing, we can see that the clauses in lines 1 and 2 are inferred using the Eva plugin. The plugin uses the Eva plugin internally to infer some annotations and then hoist these annotations and include them in the function’s contract.

In a similar manner, the plugin used the RTE plugin internally to infer the clauses in lines 3–6, and the From plugin to infer the clause in line 7.
Listing 2.12: The function `swap()` without any annotations.

```c
/*@ requires Eva: mem_access: \valid_read(a); 
requires Eva: mem_access: \valid_read(b); 
requires \valid(b); 
requires \valid(a); 
requires \valid_read(b); 
requires \valid_read(a); 
assigns *a, *b; */
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
    return;
}
```

Listing 2.13: The output of Scania’s plugin when applied on the clean version of the function `swap()`.

### 2.4 Related work

In this section we present related work in the area of annotation inference and the automation of annotation inference.

#### 2.4.1 Synthesis of Annotations for Partially Automated Deductive Verification

This is a master’s thesis that has been conducted at Scania as a part of the same project, and it focuses on implementing the very plugin that we are trying to improve in our thesis. In this work, the author investigates the feasibility of inferring auxiliary annotations from source code and starts the development of a plugin for the Frama-C platform that aims to achieve this goal [12]. The work focuses on inferring and annotating auxiliary specifications for industrial embedded automotive software source code.

For evaluation, the author uses two approaches. The first approach is to execute the plugin on a C-module in which the source code does not contain any annotations at all. Thereafter, the transformed source code is verified using the WP plugin in Frama-C. The quality of the inferred annotations is then quantified in terms of the proportion of annotations that were successfully verified using WP.
Table 2.1: This table demonstrates the results which were achieved by Skantz [12] using C-modules in which the source code does not contain any annotations. The modules that are marked “simplified” are published with the work. The other modules are closed source code that belongs to Scania.

The second method of evaluation was to run the plugin on a C-module with annotations representing high-level functional specifications in the source code. In this approach, the quality of the inferred annotations is also quantified in terms of the proportion of all annotations that were successfully verified using WP, but in this case, the annotations will not only include the auxiliary specifications but also the functional specifications.

Table 2.2: This table demonstrates the results that were achieved by Skantz [12] using a C-module in which the source code contained annotations of high-level functional specifications. The second row represents the results when the plugin was used to transform the source code.

### 2.4.2 Automatic Inference of Necessary Preconditions

The authors in this work present a solution that focuses on inferring precondition annotations using static analysis and the weakest precondition calculus [26]. An important aspect of this work is that the authors differentiate between necessary preconditions and sufficient preconditions, and they advocate the use of necessary preconditions in the context of automatic verification. In this work, the problem of inferring termination preconditions is considered as a separate problem, and therefore it is not examined.

A sufficient precondition is informally defined in the paper as the set...
of preconditions under which the program is correct, and the necessary precondition is the set of preconditions under which, if violated, the program will always be incorrect. What makes sufficient preconditions less suitable for use in the context of automatic verification is the strength of the preconditions. A strong precondition will not only rule out faulty runs but also some good ones, causing false alarms during verification.

As a result, the authors advocate necessary preconditions, which are weak preconditions that only ensure the absence of preconditions that always cause faulty behavior. This eliminates the problem of false alarms completely. At the same time, this implies that these preconditions do not guarantee correctness.

The technique that the authors use for inferring necessary preconditions is based on the work of Cousot et al. [27]. The approach relies on inferring an under-approximated set of preconditions, $P^\flat$. This step is similar to the concretization function $\gamma$, which we introduced in the abstract interpretation Subsection 2.1.4 with the deference of this method being an under-approximation instead of an over-approximation. The inferred set of conditions $P^\flat$ are processed to remove redundant preconditions, and finally, the remaining preconditions become the set of necessary preconditions and are hoisted to the entry point of the code.

Although this work does not present a solution that guarantees the correctness of software, none the less, it manages to eliminate about 60% of the warnings that existed in real-world production code. The authors also succeeded in improving the technique that was introduced by Cousot et al. [27], by refining it with invariants that were inferred using static analysis.

### 2.4.3 Inferring better contracts

In this work, the authors introduce a new annotation inference mechanism that relies on programmer-written simple annotations [28]. The focus of this approach is on inferring the postconditions for commands that change the state of the software. The authors motivate their choice by stating that empirical observations have shown that writing complete preconditions is relatively easy and programmers often succeed in this task. This motivation is based on a comparative study of programmer-written contracts [29].

The inference technique in this solution is partly based on simple annotations written by the programmer and partly on performing a dynamic analysis on the software. The dynamic analysis is done by first generating a random test suite that aims to cover all the commands in the software. The next step is to run the test suite, monitor the changes in the state, and then
create a change profile based on the detected changes. The change profile is then used to infer the postcondition contracts. Finally, the tests are re-executed to validate the inferred contracts.

Since this technique uses dynamic analysis, it has the advantage of flexibility in that it can be applied even when the source code does not exist. The downside of this technique is that, however, no guarantees can be made in terms of soundness and completeness. This is mainly due to the fact that the method relies on the generated test suite, which affects the gathered data on the software’s behaviour. This problem can be witnessed in the results of the authors’ experiment, which contained some unsound inferred postconditions and achieved complete postcondition inference for only 75% of the commands in the experiment.

2.4.4 Houdini, an Annotation Assistant for ESC/Java

The authors of this work present a tool called Houdini, which assists the programmer in annotating Java source code [30]. The annotations in this project are then used by an automatic prover called ESC/Java to prove the correctness of the software. The approach taken here is to try proving, one by one, using the automatic prover, a large number of candidate annotations. The annotations that get proved successfully by the prover are included in the source code.

Although the method uses a naive search for annotations, the generated annotations have reduced the number of false alarms that the automatic prover emits. The method has also reduced the time required by the programmer to find bugs in the unannotated source code.

The disadvantage of this approach is that the number of candidate annotations being tested cannot get very large due to limitations in processing power and time. Thus, using this method, exhaustive search of annotations is not possible.

2.5 Summary

Skantz’s work, which was introduced in Subsection 2.4.1, demonstrates a very high proportion of successfully proven specifications. However, these results do not reflect to what degree the implementation is complete. One reason is that the author has just tried to prove some high-level functional specifications, but there are many more untried possible functional specifications that might fail. Another issue is that even those few proof obligations that
were not proven could necessitate a large number of auxiliary specifications that are not yet covered by this implementation. Furthermore, it is not clear how representative the used module is in this work. There might be other modules that contain different C constructs and have different types of specifications. Nevertheless, the results of this work could serve as a benchmark for measuring the results of our work, especially given that our work is a continuation of this work.

The work presented in Subsection 2.4.3, where the authors used dynamic analysis, demonstrates how this method is not suitable for our project because of the faulty contracts that it might generate. This is especially problematic with regards to the MISRA specifications (presented in Subsection 2.2.2), which require the absence of false alarms in the verification process.

In the work presented in Subsection 2.4.2, the authors introduce the notion of a necessary precondition, which is a condition inferred using under-approximations to eliminate any false alarms. This approach is suitable for our project since it eliminates the problem of false alarms. However, there is the problem of false negatives that needs to be mitigated in this approach.

Finally, the approach used in the Houdini tool presented in Subsection 2.4.4 could be useful for our project. Although in our project we use a more systematic approach for inferring contracts, we can still complement our approach by using the naive approach used by Houdini. For example, by identifying a set of contracts that our plugin cannot infer from the source code and testing these contracts naively, we can generate some new contracts that might be useful for the proof process as a whole.
Chapter 3

Method

This chapter provides an overview of the research method used in this thesis. Section 3.1 describes the research process. Section 3.2 covers the collection of C source code used in this work. Section 3.3 demonstrates the method for identifying improvements. Section 3.4 explains the evaluation method.

3.1 Research Process

This research process generally uses a qualitative approach. Since the solution is not developed enough to be implemented on a large number of C modules, a quantitative method is not practicable, making this approach the only practical choice.

First, a source code analysis was conducted to identify possible improvements in the auxiliary specification inference mechanism. Thereafter, based on the findings of the analysis and previous work, new features were designed and implemented in the plugin. Finally, the improved plugin was tested and benchmarked.

Due to time limitations, it was not possible to implement all the identified improvements. Only a subset of the improvement candidates were chosen for implementation. The criteria for choosing the improvements was to prioritize those that potentially yielded a greater number of successfully proven requirements.

3.2 Source code collection

The source code used for testing and benchmarking purposes was gathered from Scania’s production repository. These modules, however, are not open-
source, and therefore they are not included in this report. Nevertheless, to
give the reader an idea of what the code look like, simplified versions are
included. These simplified modules contain similar constructs and structures
as in the original modules, to ensure that the simplified version is a good
representation of the original. Hence, the results achieved on these modules
are meaningful for the project. An additional benefit of using simplified
modules is that it is easier to understand certain types of issues related to
contract inference. The fact that a simplified module abstracts away a great
deal of complexity associated with the environment, i.e., the operating system
and the middleware, makes pinpointing problems and understanding them
much easier.

Following is a list of the chosen modules and a description of the
motivation for the choice.

- **Steering - real module**: This module was used in previous work, which
  makes result evaluation easier. Additionally, the module has existing
  ACSL functional contracts in place, which saves the time needed to
  create those contracts from scratch.

- **Steering - simplified module**: This module was also used in previous
  work and has the exact same benefits described earlier.

- **Accelerator pedal - simplified module**: This module contains floating-
  point constructs, which was one of the limitations identified in previous
  work. A simplified version is included along with its ACSL functional
  contracts.

- **Main light warning - real module**: This module is an arbitrary one.
  The purpose of this module is to evaluate the tool’s general performance
  and identify potential new issues.

### 3.3 Identifying improvements

The first approach was to analyze the source code manually and try to identify
constructs that were not covered by the implementation. This method is
effective only for detecting simple constructs, such as *Bit Fields*, which were
one of the C constructs that were not covered by the existing implementation.
On the other hand, this method is not suitable for detecting problematic
combinations of constructs. There were combinations where each of the
constructs by itself was covered, but the combination as a whole was not.
The second approach was to execute the existing implementation on the simplified modules and analyze the results. This method is effective for detecting constructs or combinations of constructs that were not covered by the implementation. Additionally, when functional ACSL contracts are available, the method is also effective in identifying the missing auxiliary annotations that would help in proving the functional annotations.

The last approach was to execute the implementation on real Scania modules. Since these modules involve the middleware layer and the operating system layer, they include more complex combinations of constructs. Consequently, this method is very effective for detecting complex problems. However, this complexity makes understanding a problem a challenging task. As a result, it became harder to identify the needed improvement in the implementation to mitigate the detected problem.

It should be mentioned here that although the aim of this project is to improve the implementation by covering all the C constructs found in the target industrial code, this goal can be achieved by other means. Another way to achieve the goal is by restricting the usage of certain constructs or combinations that are difficult to cover. This approach is especially viable when there exist simpler alternative constructs that could express the same semantics as the problematic constructs. This approach exists already in the form of the MISRA standard and could be extended or modified to facilitate formal verification.

### 3.4 Evaluation

The results achieved by executing the new implementation on the C modules listed in Section 3.2 are evaluated. The evaluation does not only consider the numbers of proven and unproven annotations but also the analysis of the underlying reason behind each unproven annotation. This is done for two versions of a given module: one that includes only an interface contract and one that includes both an interface contract and a functional contract.

Additionally, the results are analyzed even without applying the implementation, i.e., by just importing the module with its contracts into Frama-C and executing the SMT solver. This was done to evaluate to what degree the implementation helped in proving the contracts.

This evaluation approach is similar to the approach taken in Skantz’s work, presented in Subsection 2.4.1. Consequently, a part of the evaluation is done by comparing the results with what was achieved in Skantz’s work.
Method
Chapter 4

Implementation

This chapter provides an overview of the implemented improvements. Section 4.1 describes the newly added mechanism for propagating interface specifications. Sections 4.2, 4.3, and 4.4 describe the methods used for inferring auxiliary specifications for selected C constructs. Section 4.5 includes a summary of the implemented improvements.

4.1 Interface specification inference

The first improvement to the implementation was to develop a propagation mechanism for interface specifications. Interface specifications specify the variation domains of different parameters, and different input and output signals on a module level. Since this type of specifications do not specify a behavior of the module, they are considered as high-level help specifications or auxiliary specifications.

The solution for interface specification propagation consists of two parts. The first part is to translate the interface specifications from natural language to ACSL annotations. These annotations are placed at the entry function of the module. However, to deductively prove the correctness of the annotations, similar annotations are required for the helper functions that are used in the module. Thus, the second part of the solution is the implementation of a mechanism for propagating the entry point specifications to the helper functions of the module.
4.1.1 Translating the interface specifications

This step was executed manually. The specifications that described inputs are translated into ACSL requires annotations, and the specifications that described outputs are translated into ACSL ensures annotations. Additionally, the ACSL annotations assigns, valid, and valid_read are created for any global variable that the module interacts with.

To demonstrate the method, an example of an interface specification can be seen in Table 4.1. This specification is translated to the ACSL contract in Listing 4.1. The entry point of the example module is the function steering().

<table>
<thead>
<tr>
<th>Input/Output</th>
<th>Variable Name</th>
<th>Unit</th>
<th>Required Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>ElectricMotorActivated</td>
<td>Binary</td>
<td>On or Off</td>
</tr>
</tbody>
</table>

Table 4.1: An interface specification similar to the interface specifications found in Scania documents.

```c
/*@ behavior interface_spec:
   assumes \true;
   requires \valid(&state[ELECTRIC_MOTOR_ACTIVATED]);
   requires \valid_read(&state[ELECTRIC_MOTOR_ACTIVATED]);
   ensures state[ELECTRIC_MOTOR_ACTIVATED] ∈ {0, 1};
   assigns state[ELECTRIC_MOTOR_ACTIVATED];
*/
void steering() { ... }
```

Listing 4.1: A translation of an interface specification as an ACSL contract.

In this example, the variable ElectricMotorActivated is an output variable, thus the translated ACSL annotation is an ensures annotation. Notice that there are two requires annotations that do not map to any explicit requirement in the interface specifications in Table 4.1. These annotations are implicit requirements that are necessary for the output memory location &state[ELECTRIC_MOTOR_ACTIVATED] to be writable.

4.1.2 Propagating the interface specifications

A Frama-C plugin was developed for propagating the interface specifications into all the helper functions. The design of the plugin is based on the following points:

- A module interacts with a set of memory locations and signals.
• Interface specifications specify the variation domains of these locations and signals at the entry point of the module, i.e., the entry state.

• Using static analysis methods, it is then possible to calculate the variation domains in the subsequent states of the execution. This implies, of course, that the source code for the module is available.

• By identifying all reachable states in which a certain helper function is called, it is possible to calculate the variation domains for that helper function.

The plugin is called Interface Specification Propagator, or ISP. The plugin is designed to be used in the following steps:

1. Import the C-module, including its entry-point interface specifications into Frama-C.

2. Frama-C automatically parses the module and creates a CIL representation of the module.

3. Execute the ISP plugin on the CIL representation. This step requires specifying the entry-point function of the module.

4. The ISP plugin outputs a transformed CIL representation, which contains the newly inferred annotations for the helper functions.

5. Execute a plugin of choice for proving the ACSL contracts in the module. In this work, the WP plugin is used.

To achieve its goal, the ISP plugin relies not only on its own logic but also on the Frama-C plugin, EVA. ISP keeps states that track different memory locations and global variables and uses the EVA plugin to make evaluations of these locations and variables at different states of the module. Based on these evaluations, ACSL annotations are inferred for the helper functions used in the module.

As an example, consider the module in Listing 4.2, which contains an entry-point function main and a helper function update_state.

```c
int WHEEL_SPEED;
int STATE;

typedef enum { STAND_STILL = 0, MOVING = 1 } VEHICLE_STATE;

void update_state() {
    if (WHEEL_SPEED > 0)
```
Executing the following command:

```bash
$ frama-c-gui -isp -isp-entry-point main example.c -then-last -wp -wp-rte -lib-entry
```

starts the Frama-C platform with its graphical interface. Frama-C parses the C-module in the file `example.c` and creates the corresponding CIL representation. Then it calls the plugin ISP, which traverses the CIL starting at the entry point function `main` and outputs a new CIL that contains the inferred annotations. The `-then-last` flag tells Frama-C to call the WP plugin on the latest created CIL, which in this example is the output of ISP. Based on the ACSL annotations, WP creates proof obligations and tries to prove them using its SMT solvers. The result of these operations is illustrated in Figure 4.1.

### 4.2 Floating point annotation inference

The second improvement made to the implementation was adding functionality for inferring annotations for the floating-point constructs in C. The existing implementation did not support annotation inference for expressions that contain a floating-point variable. When an expression contains a variable of type `float`, `double`, or `long double`, the expression is skipped without inferring any annotations for it.

The newly implemented mechanism for inferring floating-point annotations is a general one, i.e., it is used for inferring annotations related to interface specifications as well as any other auxiliary specifications.
Figure 4.1: The output in the Frama-C graphical user interface. The output contains the CIL representation of the C-module, including the entry point annotations and inferred annotations, which are colored in blue. The bullets on the left side mark the results of the WP plugin’s execution. A green bullet means that the property and all its dependencies are valid. A blue bullet means that no proof was attempted because these are properties that need to be guaranteed by the caller.

The mechanism relies heavily on the EVA plugin. It keeps track of the floating-point variables and memory locations, and when a helper function interacts with these entities, the EVA plugin is called to calculate the variation domains of these entities for that helper function. Based on the results, ACSL annotations are created for that helper function.

The EVA plugin always evaluates floating-point entities in interval format. This is true even when the evaluated value is a singleton, e.g., if a floating point is evaluated to the single value of 1.0, it is represented in the result as an interval $[1.0, 1.0]$. The implementation detects this format and simplifies the inferred annotation by avoiding the use of intervals whenever possible.
4.3 Pointer annotation inference

Unlike the floating-point annotation inference mechanism, which manages to cover all cases of floating-point usage, the implementation for pointer annotation inference is limited to the simple case. The implantation can handle the inference of annotations for expressions that contain a pointer to a simple C type, e.g., a pointer to an `int` value.

In contrast, nested pointers, e.g., `**x`, or pointers with manipulated addresses, e.g., `*(x + y - 1)`, are not handled by the implementation. For nested pointers, it is not clear whether the need is there for covering this type of pointer because they were not detected in the examined source code. On the other hand, it is known for a fact that many of the types of address manipulation do not need to be covered by the implementation. This is because the MISRA standard, which was introduced in Subsection 2.2.2, prohibits all kinds of address arithmetic except those used for indexing operations.

Similar to the inference mechanism for floating points, the mechanism for inferring annotations for pointer constructs also relies on the EVA plugin.

4.4 Array annotation inference

An inference mechanism for arrays is implemented for most of the cases. In a similar fashion to the logic used in previous inference mechanisms, this mechanism also keeps track of memory locations, in this case array cells, and uses the EVA plugin for evaluation. However, the mechanism do not cover all the cases of array usage. The following two subsections describe the cases that are not covered.

4.4.1 Delimitation on expressions used as array index

The first case that is not supported is when a function is called within another function, where the inner function has an argument that is used as an index for an array. For example, consider the module in Listing 4.3. Applying the ISP plugin and then the WP plugin to this module gives the results demonstrated in Figure 4.2.

```c
1  int DB[5];
2  void help_function(int idx) {
3      DB[idx] = 3;
4  }
5
```
/**@ behavior interface_spec: 
  assumes \true;
  requires \valid_read(&DB[1]);
  requires \valid(&DB[1]);
  ensures DB[1] == 3;
  assigns DB[1];
*/

void main()
{
  help_function(1);
}

Listing 4.3: A C-module with a function argument used as an index.

int DB[5];
/*@ behavior isp_generated:
  assumes \true;
  requires \valid_read(&DB[1]);
  requires \valid(&DB[1]);
  ensures DB[1] == 3;
  assigns DB[1];
*/

void help_function(int idx)
{
 /*@ assert rte: index_bound: 0 <= idx; */
 /*@ assert rte: index_bound: idx < 5; */
  DB[idx] = 3;
  return;
}

/*@ behavior interface_spec:
  assumes \true;
  requires \valid_read(&DB[1]);
  requires \valid(&DB[1]);
  ensures DB[1] == 3;
  assigns DB[1];
  behavior isp_generated:
  assumes \true;
  requires \valid_read(&DB[1]);
  requires \valid(&DB[1]);
  ensures DB[1] == 3;
  assigns DB[1];
*/

void main(void)
{
  help_function(1);
  return;
}

(a) Lines 1–16.  
(b) Lines 17–35.

Figure 4.2: The results of applying the ISP plugin followed by the WP plugin to a module that contains a help function that uses its argument as an array index.

The inferred contract for help_function() cannot be proven using deductive logic without including annotations for the function argument idx, i.e., the annotation requires idx == 1, which is needed for proving the contract. The mechanism does not process the index of an array. Thus, annotations will not be inferred for variables or expressions that appear only in an array index.

4.4.2 Delimitation on annotation simplification

When inferring annotations for array cells, the implementation creates a separate annotation for each cell in the array. This means that even when
the array has a single property for all its cells, the inference mechanism will create multiple annotations. For example, consider an array of ten cells, \( X[\] \), where all the cells have the property of containing either 0 or 1. Then, consider that in a given module, the first eight cells of the aforementioned array are accessed. The inference mechanism will then create the annotations demonstrated in Listing 4.4. However, the ACSL language has constructs for expressing the same information in a more simplified way, e.g., see the annotation demonstrated in Listing 4.5.

```plaintext
/*@ requires X[0] ∈ \{0, 1\};
 requires X[1] ∈ \{0, 1\};
 requires X[2] ∈ \{0, 1\};
 requires X[3] ∈ \{0, 1\};
 requires X[4] ∈ \{0, 1\};
 requires X[5] ∈ \{0, 1\};
 requires X[6] ∈ \{0, 1\};
 requires X[7] ∈ \{0, 1\};
 */
```

Listing 4.4: Repeated ACSL annotation of the same property.

```plaintext
/*@ requires X[0..7] ∈ \{0, 1\};
 */
```

Listing 4.5: An ACSL annotation for array cells with identical property.

It is not clear what the implications of this limitation are. It is interesting to see whether this limitation will cause any noticeable problems when the number of cells starts getting larger.

### 4.5 Summary

In summary, the implementation consists of four improvements. The first improvement is the added mechanism for inferring annotations based on the interface specifications. The second improvement is the added functionality for inferring annotations for floating-point constructs in C. The last two improvements are the added functionality for inferring annotations for pointers and arrays. These two improvements have some limitations and do not support annotation inference for all types of C constructs involving pointers or arrays.

The functionalities for inferring annotations for floating-points, pointers, and arrays are utilized in both the inference based on the ISO C99 standard and the inference based on interface specifications.
Chapter 5

Results and Analysis

This chapter presents the results obtained by applying the implementation to different C-modules. Section 5.1 covers the results obtained by applying the implementation to simplified C-modules. Section 5.2 covers the results obtained by applying the implementation to real Scania C-modules. Section 5.3 includes an analysis of the reliability and validity of the results. Finally, Section 5.4 contains a summary of the results.

5.1 Results on Simplified Scania C-modules

In this section, the results achieved on two simplified Scania modules are presented. Both modules have interface and functional specifications in ACSL.

5.1.1 Simplified Steering C-modules

This module is a simplified version of a proprietary Scania C-module that manages some of the vehicle’s steering functionalities. The results achieved by applying the implementation, the ISP plugin, to this module are presented in Table 5.1.

To help the reader more easily understand the results and to make the work easier to reproduce, the following versions of this module are included in the appendix.

- A clean version of this module that do not include any inferred annotations, but only the interface specification annotations at the entry-point. The entry-point of this module is the function `steering()`. See Appendix B.1.
• A transformed version, which is the raw output of the ISP plugin. This version includes the annotations that were inferred by ISP. See Appendix B.2.

• A clean version that contains only functional specification annotations at the entry-point. There are no interface specifications in this version. See Appendix B.3.

<table>
<thead>
<tr>
<th>Functional Specifications</th>
<th>Interface Specifications</th>
<th>ISP Applied</th>
<th>Proven Goals</th>
<th>Unproven Goals</th>
<th>Total Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without</td>
<td>With</td>
<td>No</td>
<td>4</td>
<td>29</td>
<td>33</td>
</tr>
<tr>
<td>Without</td>
<td>With</td>
<td>Yes</td>
<td>197</td>
<td>30</td>
<td>227</td>
</tr>
<tr>
<td>With</td>
<td>Without</td>
<td>No</td>
<td>36</td>
<td>14</td>
<td>50</td>
</tr>
<tr>
<td>With</td>
<td>With</td>
<td>No</td>
<td>38</td>
<td>26</td>
<td>64</td>
</tr>
<tr>
<td>With</td>
<td>With</td>
<td>Yes</td>
<td>228</td>
<td>32</td>
<td>260</td>
</tr>
</tbody>
</table>

Table 5.1: The results of ISP obtained on the simplified steering C-module.

The first two rows in Table 5.1 show that the number of goals increased when ISP was applied. The ISP inferred and propagated new ACSL annotations throughout the module. Although the number of unproven goals increased from 29 to 30, a closer examination of these on the Frama-C platform shows that many of the previously unproven goals have been proven. The failing goals are mostly the newly inferred ones. Upon further examination, many faulty annotations were identified. The rest of this subsection is dedicated to pointing out and explaining the issues that are behind the faulty annotations.

Lines 35–46 in the output, see Appendix B.2, reveal a strange approximation. The entire array `state[]` is approximated to have a value in the range $[5, 65280]$. This is clearly an incorrect approximation caused by either a flaw in our implementation or a limitation in the underlying EVA static analysis plugin. Some effort was put into trying to pinpoint the source of the problem without success. A more extensive examination is required.

Another issue in the same output is the over-approximation at lines 53 and 54. This over-approximation is a consequence of the behavior of the underlying EVA plugin. In its static analysis, EVA detected that the function `read()` accesses `state[2]` at line 108. `state[2]` represents the signal `PRIMARY_CIRCUIT_HIGH_VOLTAGE`. Since the interface specifications do not specify this signal, EVA evaluates it according to the set of values
an int variable can take in the ISO C99 standard. Thus, the approximation became the interval \([-2^{31}, 2^{31}-1]\). Neither EVA nor ISP could notice that the value of \(\text{state}[2]\) is not used anywhere in the logic of the module. Consequently, ISP over-approximates the return value of the function.

The rest of the issues are purely limitations in the ISP implementation. The issues are:

- No annotation inference of type requires for global variables and addresses that are only used in an array index. See the contracts for the functions \(\text{write}()\). This limitation was introduced in Subsection 4.4.1.

- No annotation inference of type requires for enums used as function argument. See the contracts for the functions \(\text{write}()\) and \(\text{read}()\).

- No annotation inference for pointers pointing to a struct. See the contract for the function \(\text{get_system_state}()\).

Rows 3–5 in Table 5.1 present the results obtained when functional specifications are also included in the module. There are six functional requirements listed at lines 126 to 152; see Appendix B.3. The SMT solvers used by the WP plugin fails to prove the goals generated from these requirements. Due to the aforementioned issues in ISP, adding the interface specifications and applying the plugin did not make any improvements in proving the functional goals.

By manually correcting the issues in the output of ISP, the results in Table 5.2 are achieved. Two manually corrected versions are included in the appendix. The first corrected version includes only the interface specification; see Appendix B.4. The second corrected version includes both the interface and functional specifications; see Appendix B.5.

It should be mentioned that in the second corrected version, some annotations are manually added to solve some issues that were listed above. These additional annotations are related to the ACSL ghost variables, which are introduced at lines 33–35 and 297–298. However, since ghost annotations are not related to the inference of auxiliary annotations and only related to some functional annotation issues, more details is not provided about these fixes in this work.
### Table 5.2: These results are obtained after fixing the issues in the raw output in Appendix B.2.

<table>
<thead>
<tr>
<th>Functional Specifications</th>
<th>Interface Specifications</th>
<th>ISP Applied</th>
<th>Proven Goals</th>
<th>Unproven Goals</th>
<th>Total Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without</td>
<td>With</td>
<td>Yes</td>
<td>264</td>
<td>0</td>
<td>264</td>
</tr>
<tr>
<td>With</td>
<td>With</td>
<td>Yes</td>
<td>293</td>
<td>5</td>
<td>298</td>
</tr>
</tbody>
</table>

The results in Table 5.2 show that fixing the aforementioned problems helps prove all the interface specification annotations. More importantly, the results show that many of the functional annotations were proven. This confirms that interface specification annotations help prove functional specifications. There are five goals that remain failing. Further examination of the results on the Frama-C graphical user interface shows that these goals are the functional requirements 1, 2, 3, 5, and 6 at the entry function.

#### 5.1.2 Simplified Accelerator Pedal C-modules

This module is a simplification of a proprietary Scania module that manages the behavior of the accelerator pedal. Here we have also included different versions of this module.

- A clean version of this module that do not include any inferred annotations, but only the interface specification annotations at the entry-point. The entry-point of this module is the function `Ac-cped_update_5ms()`. See appendix B.6.

- A transformed version, which is the raw output of the ISP plugin. This version includes the annotations that were inferred by ISP. See appendix B.7.

- A clean version that contains only functional specification annotations at the entry-point. There are in total four functional requirements. Two of the requirements are about internal variables, which required the introduction of ACSL ghost variable annotations. See appendix B.8.

Additionally, a simplified version of the interface and the functional requirements for this module can be found in appendix A.1. The results obtained from this module are presented in Table 5.3.
Table 5.3: The results of ISP obtained on the simplified accelerator pedal C-modules.

In this module, the raw output of the ISP plugin contains the same issues that were discovered in the steering module. However, in addition to the previous issues, two new ones are encountered.

The first issue is in the annotation at line 89 in the output; see Appendix B.7. This annotation specifies that the return value of the function `calculate_pedal_position()` is in the variation domain \([-11.1111111111, 111.1111111111]\]. This is clearly an over-approximation of the real variation domain. Examining the logic of the implementation shows that the real variation domain is the interval \([0.0, 100.0]\). To isolate the problem and identify the underlying reason for this over-approximation, a simple module is created; see Listing 5.1.

```c
int X;
int Y;
/*@ requires valid_read(&Y);
 requires valid_read(&X);
 requires X ∈ (0..10);
 requires Y ∈ (0..20);
 */
int calculate() {
  if (Y > X)
    return 0;
  else
    return X - Y;
}
```

Listing 5.1: A simple function that returns an integer value between 0 and 10 when the function contract is true.

Applying the EVA plugin to this module yields the result presented in Listing 5.2. The result shows clearly the same kind of over-approximation that was detected in the accelerator pedal module. The result shows that the over-approximation occurs in connection with branching statements. This indicates
that the problem is caused by a limitation in the underlying EVA plugin, which is
used internally by ISP.

Listing 5.2: The output of the EVA plugin when applied to function calculate().

The second problem was discovered after manually fixing all the known issues (see the fixed version in Appendix B.9). At the deductive proving step, the WP fails to prove the annotation at line 96. None of the SMT solvers used by the WP plugin manage to prove this annotation. Nevertheless, this behavior is expected. It was introduced in the background (see Subsection 2.3.2) that the WP plugin has some limitations when it came to proving annotations that involve floating-point arithmetic. In this case, the annotation at line 96 is directly related to the expression at line 100, which is a floating-point arithmetic expression. Thus, the problem is not caused by ISP and its annotation inference mechanism but rather because of a limitation in the SMT solvers used by the WP plugin.

Apart from the aforementioned annotation, this limitation affects seven other annotations. These annotations were generated by the RTE plugin, which was introduced in Subsection 2.3.3. The results obtained from the manually corrected ISP output are presented in Table 5.4.
Table 5.4: These results are obtained after fixing the issues in the raw output of the ISP plugin when applied to the accelerator pedal module.

The results in Table 5.4 show that the version which contains functional specifications has additional four unproven goals. This result gives the impression that the additional four goals are the same four functional requirements at lines 161–179 in Appendix B.10. Examining the goals closely on the Frama-C platform shows that the third functional requirement is successfully proven, so that only three of the functional requirements are not proven.

The first functional requirement is expected to fail because it involved the ghost variable `model_pedal_raw_position`, which represents the value of a floating-point arithmetic expression; see line 200. Since this ghost variable also represents the value of the argument of function `calculate_pedal_position()`, the annotation for that function argument at line 95 fails to be proven. Thus the failing consists of three functional annotations and one auxiliary annotation.

The failing auxiliary annotation is interesting because in the version that does not contain functional requirements, this annotation is proven, but it is marked with an `unknowing dependency` bullet. Introducing the ghost variable in the functional version exposed the dependency to the SMT solver. When it discovered that the dependency is floating-point arithmetic, the WP plugin marked the goal as unproven.

## 5.2 Results on Scania Proprietary C-modules

In this section, the results achieved on two proprietary Scania modules are presented. These modules were not open-source. As a result, the code was not included in this report.

The two modules were the steering module and the main light warning module. The steering module had functional specifications in ACSL, and the ISP plugin was used for proving these specifications. The main light warning module, on the other hand, did not have any function specification in ACSL.
Thus, the ISP plugin was only used for proving the interface specifications.

## 5.2.1 Proprietary Scania Steering C-modules

The results for the proprietary Scania steering C-modules are presented in Table 5.5.

<table>
<thead>
<tr>
<th>Functional Specifications</th>
<th>Interface Specifications</th>
<th>ISP Applied</th>
<th>Proven Goals</th>
<th>Unproven Goals</th>
<th>Total Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without</td>
<td>With</td>
<td>Yes</td>
<td>1018</td>
<td>16</td>
<td>1034</td>
</tr>
<tr>
<td>With</td>
<td>Without</td>
<td>No</td>
<td>215</td>
<td>96</td>
<td>311</td>
</tr>
<tr>
<td>With</td>
<td>Without</td>
<td>Yes</td>
<td>1061</td>
<td>35</td>
<td>1096</td>
</tr>
<tr>
<td>With</td>
<td>With</td>
<td>Yes</td>
<td>1074</td>
<td>35</td>
<td>1109</td>
</tr>
</tbody>
</table>

Table 5.5: Results on the proprietary Scania steering C-modules.

The issues discovered in the simplified modules are also present in this module. It is expected to find these issues in the proprietary modules because the simplified modules are a close representation of the real Scania modules. Nevertheless, this module has more complexity than the simplified modules. The complexity is a result of the fact that the proprietary module is operating on top of operating system-level modules and middleware-level modules. This complexity led to the discovery of a number of new issues.

The first issue is related to \#pragma directives. Some of these directives are compiler-specific directives and are not defined in the ISO C99 standard. In the steering module, there are many of these directives. Frama-C supports only the directives that are specified in the ISO C99 standard [11]. Any other directive causes a syntax error. To solve this issue, these directives were deleted. The task of deleting is easy because the problematic directives are intended for some preprocessing tools and have no impact on the logic of the module itself.

Static local variables are the source of the second issue. These variables persist outside the scope of a function and behave in a similar manner to a global variable. When the Frama-C platform creates a CIL representation of a module, it converts all the static local variables into global variables. Frama-C do the conversion in a way that will not affect the logic of the module. At a later point, when the ISP plugin is applied to the CIL representation, it successfully infers the annotations required for these newly created global variables. The issue, however, is that these variables get over-approximated, which in turn do not help in the proof process of some of the goals.
To isolate and understand the issue, a simple module was created; see Listing 5.3. This module includes a function that contains a static local variable and an interface specification contract at its entry-point.

```
int X;
int count() {
    static int c = 0;
    c++;
    return c;
}

/*@ behavior isp_interface:
    assumes \true;
    requires \valid_read(&X);
    requires \valid(&X);
    requires X == 0;
    ensures X == 3;
    assigns X;
*/
void main() {
    X += count();
    X += count();
}
```

Listing 5.3: A simple module with the function count() that contains the static local variable c.

Importing this module into the Frama-C platform and applying the ISP plugin gave the output presented in Listing 5.4. The output demonstrates how the newly created global variable count_c is over-approximated; see lines 6 and 7. This over-approximation hinders the SMT solvers used by the WP plugin from proving deductively the annotation at line 26 in the same output. The over-approximation shows that the underlying static analysis plugin EVA do not take into consideration that count_c is initialized to 0 at line 2, and that the function count() is only called twice; see lines 30 and 31. These facts imply that count_c can only have the values 0 and 1 when the function count() is called. The EVA plugin has instead approximated the variation domain of count_c to be the values an int variable can take in the ISO C99 standard.

```
int X;
static int count_c = 0;
/*@ behavior isp_generated:
    assumes \true;
    requires count_c \geq -2147483648;
    requires count_c \leq 2147483647;
    requires \valid_read(&count_c);
    requires \valid(&count_c);
*/
```
Listing 5.4: The output of the ISP plugin when applied to the module in Listing 5.3. Note that the static local variable c is converted by the Frama-C platform into the global variable count_c.

This issue is solved by manually modifying the output; see Listing 5.5. Note that in order to solve the issue, even the entry-point contract are modified.

```c
int X;
static int count_c = 0;

/*@ behavior isp_generated:
  assumes \true;
  requires count_c \geq 0;
  requires count_c \leq 1;
  requires \valid_read(&count_c);
  requires \valid(&count_c);
  ensures count_c == \old(count_c) + 1;
  ensures \result == \old(count_c) + 1;
  assigns count_c;
*/
int count() {
  count_c++;
  return count_c;
}

/*@ behavior isp_interface:
  assumes \true;
  requires \valid_read(&X);
  requires \valid(&X);
  requires X \equiv 0;
  ensures X \equiv 3;
  assigns X;
*/
void main() {
  X += count();
  X += count();
}
```
26  requires count_c == 0;
27  ensures X == 3;
28  assigns X, count_c;
29  */
30  void main() {
31    X += count();
32    X += count();
33  }

Listing 5.5: The modified output in which the over-approximation issue for static variables was solved.

The solution in Listing 5.5 raises a new question about the architecture of the entire implementation. The ISP plugin does not update entry-point contracts because the approach is to write the annotations for auxiliary and the functional specifications at the entry-point of the module and avoid any modification to these annotations at a later point. The reason behind this approach is to avoid introducing errors into the entry-point contract. Errors in the entry-point contract are complicated and difficult to detect. Listing 5.6 demonstrates an example of an error at the entry-point. At lines 7 and 8, a contradiction is made. Executing the WP plugin on this module gives a result in which all the goals are proven; see Figure 5.1. In fact, the goals at lines 9 and 10 are not possible to prove deductively, but since the contract contains a contradiction in the precondition, the SMT solver concludes that the contract holds in all states, i.e., there are zero states that can fulfill the precondition. If the ISP plugin is allowed to mutate the entry-point contract, this kind of problem might be introduced. The source of the error introduction might be an incorrect logic in the module’s implementation or an error in the implementation of the ISP plugin itself.

int X, Y;
/*@ behavior isp_interface:
  assumes \true;
  requires \valid_read(&X);
  requires \valid(&X);
  requires X == 0;
  requires X != 0;
  ensures X == 6;
  ensures Y == 18;
  assigns X; */
void main() {
  X = 1;
}

Listing 5.6: A module that contains a contradiction in its entry-point contract.
Finally, there is the issue of importing the massive number of modules into Frama-C, which includes the operating system modules as well as some middleware modules. Although the steering module itself is not a large module, nevertheless it imports some operating system modules and some middleware modules, and these modules in turn import other modules, and so on. It was difficult to set up the Frama-C platform in the right way so that it imports all these modules correctly. This problem is solved by limiting the import to only the header files of the modules that were used directly by the steering module. In the header files, function contracts are introduced for functions that are used directly by the steering module. These contracts describe the behavior of the function so that no further imports are necessary.

Listings 5.7 and 5.8 demonstrate an example of the solution. In this example, a module imports another module’s header file. Executing the ISP plugin on this example worked mostly fine, except an issue emerged with the introduction of the ghost array. The issue can be seen in the output presented in Figure 5.2. At line 6 in the output, the inferred annotation cannot be proven. The reason is a missing annotation about $\text{ghost_db}[2]$ in the same contract. When the annotation \texttt{requires ghost_db[2] == 6;} was manually added, the issue was solved, and all the goals were proven. This demo shows that the issue is a consequence of the ISP plugin not being designed to infer annotations for ghost variables.
#include "external.h"

int X;

void calculate_x() {
    X = read_db(2);
}

/*@ behavior isp_interface:
    assumes \true;
    requires \valid_read($X$);
    requires \valid($X$);
    requires ghost_db[2] == 6;
    ensures X == 6;
    assigns X;
*/

void main() {
    calculate_x();
}

Listing 5.7: A module that imports the header of another module.

//@ ghost int ghost_db[10];

/*@ requires \valid_read(ghost_db + (0 .. i));
    requires i >= 0 && i <= 9;
    ensures \result == ghost_db[i];
    assigns \nothing;
*/

int read_db(int i);

Listing 5.8: A header file external.h for an external C-module. The header is modified with a function contract and the introduction of a ghost array. This is done to avoid importing additional modules.
The results in Table 5.5 show that adding the interface specifications do not help in proving any of the functional specifications for this module. However, the auxiliary annotations that are inferred by the ISP plugin has helped in proving some of the goals that are necessary to prove the functional specifications.

### 5.2.2 Proprietary Scania Main Light Warning C-modules

The results for the proprietary Scania main light C-modules are presented in Table 5.6. This module did not aid in the discovery of any new issues beyond
those discovered in previous modules. In this module, a single goal fails to prove, which is due to an over-approximation that happens at a branching instruction in a similar manner to what was demonstrated in Listing 5.1.

<table>
<thead>
<tr>
<th>Functional Specifications</th>
<th>Interface Specifications</th>
<th>ISP Applied</th>
<th>Proven Goals</th>
<th>Unproven Goals</th>
<th>Total Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without</td>
<td>With</td>
<td>Yes</td>
<td>88</td>
<td>1</td>
<td>89</td>
</tr>
</tbody>
</table>

Table 5.6: Results on the proprietary Scania main light warning C-modules.

### 5.3 Reliability Analysis

The tools used in this work, and the environment where the experiment took place are all well defined. Given the correct version of the Frama-C platform, which in this work is version 24.0 (Chromium), and the correct versions for the dependencies that are required by Frama-C 24.0, the output of the implementation can be reproduced consistently. The ISP implementation and its environment operate in a deterministic fashion, i.e., for a given C-module, the ISP plugin always produces the same output. However, this is not true if a human error occurs in calling the plugin with different flags in different executions. In such a case, the output may become inconsistent between executions.

The results achieved by the ISP implementation are not perfect. The implementation is suffering from some bugs. Due to limited time, unit tests and system tests are not implemented thoroughly with a suitable coverage criterion. Consequently, some of the issues in the output are difficult to understand, and the root cause is not easy to pin down. This difficulty was noticeable in the first issue with the simplified steering module in Subsection 5.1.1.

### 5.4 Summary

In summary, the implementation was benchmarked, and results were obtained on four different modules. Two of the modules are simplified modules, with their source code included in the report. The other two modules are Scania’s proprietary modules. The source code for these two modules is confidential and consequently not included in the report.
The results revealed many limitations and issues. A considerable portion of the limitations are a direct result of missing functionalities in the implemented plugin. Limitations in the underlying tools, such as the EVA plugin and the WP plugin, were also the reason behind some of the issues. Finally, there is a single issue that might be caused by a bug in the implementation. Further examination is needed to understand the issue.
Chapter 6

Discussion

This chapter discusses the results presented in Chapter 5. The first four sections cover the results achieved by each improvement. The last section is dedicated to discussing auxiliary annotation inference in general.

6.1 Interface Specification Inference

Despite the numerous bugs and limitations in the implemented mechanism for propagating interface specifications, the results show that using interface specifications as high-level auxiliary annotations helps in proving functional specifications. This is specifically seen in the results achieved on the simplified Steering module and the simplified Accelerator Pedal module. The most clear example in this work is the interface specification for the dead_zone parameter. Without including and propagating this specification to the helper function calculate_pedal_position(), the variation domain for the parameter would have been over-approximated, leading to the false positive alarm of division by zero at line 62 in Appendix B.6.

The results expose an issue related to interface specification documents, which is one of the required inputs for this solution. As it was discovered in the simplified Steering module (see Subsection 5.1.1), a missing specification for one of the signals causes an over-approximation in two of the inferred annotations. Over-approximations are not desirable outcomes since they could potentially lead to false positive alarms, see Subsection 2.1.4. This issue highlights the importance of having complete and accurate interface specifications.
6.2 Auxiliary Annotations for Floating-Points

The results achieved on inferring contacts for floating-point constructs are satisfactory. The simplified Accelerator Pedal module is a use case where the implementation handled floating-point constructs correctly, see Subsection 5.1.2. ISP manages to infer all the relevant annotations for the floating-point signals and parameters used in the module.

However, the inferred annotations are susceptible to over-approximation due to EVA’s intrinsic design of approximating floating-points always with intervals. Finally, it should not be forgotten that the results are achieved on a simplified module with a low complexity. Results on real modules with higher complexity is still a task that needs to be investigated.

6.3 Auxiliary Annotations For Arrays

The results in Subsection 5.1.1 show that the implementation for handling arrays are not sufficient. The fact that the implementation can not process expressions used as an array index, causes many over-approximations in the inferred annotations. Additionally, the issue with approximating all the cells in an array to a single value when that is not the case in reality still needs to be investigated.

6.4 Auxiliary Annotations for Pointers

The implementation for handling pointers is also insufficient. Pointers pointing to special types such as custom structs or enums are not handled, see Subsection 5.1.1. This is a substantial limitation as there are many such cases in the proprietary modules.

On the other hand, the implementation’s lack of support for nested pointers and expressions containing pointer arithmetic went unnoticed in the results. No issues arose from this limitation since none of the modules used in this project contained the mentioned constructs.

6.5 Auxiliary Annotations in General

Although the results achieved in this work agree with what was achieved in Skantz’ work [12], there are some differences in the presented numbers. The simplified Steering module is used in both works, but the results achieved
in this work (see Table 5.1) cannot be compared directly with the results in Skantz’ work. In this work, the total number of high-level functional goals are 50, of which 36 are proven without the help of auxiliary annotations. The exact same case in Skantz’ work is 31 goals, of which 13 are proven. It is difficult to say why the numbers differ because in Skantz’ work there is only one version of the module included. This version contains both the functional and the inferred annotations, making it difficult to know which of the annotations were inferred and which were manually written. Nevertheless, the results in general are inline and confirm that auxiliary annotations help in the verification process for functional specifications.

Another interesting point is that the proprietary modules helped revealing issues that affect the whole auxiliary annotation inference solution. The first issue is related to the import process of these modules into Frama-C. Importing these modules turned out to be a difficult task because of their dependencies, especially since the dependencies consist of a large number of middleware and operating-system modules. To solve the issue, function contracts and ghost variables are used to abstract away this large number of modules.

Another issue is that the modules contain compiler and preprocessor directives that are not part of the ISO C99 standard. These directives cause a problem for the Frama-C platform, as it does not recognize these custom directives and considers them to be a syntax error. To solve the issue, all these directives are manually removed from the modules.

Both the solutions, the one for importing a large number of modules and the one for removing the directives, are not optimal. A much better solution would be to implement a pre-processing stage where the module and its dependencies are automatically organized in a way suitable for a Frama-C import. The pre-processing stage should also automatically remove all the special directives from the module and its dependencies.

There is an important aspect to consider when tackling the issue of the large number of dependencies in the middleware and the operating system. The goal of this project is to find a solution for verifying automatically application-level embedded software. Importing dependencies from the middleware and the operating system into Frama-C will force the project into the territory of system-level software verification, which is not the goal of the project. On the other hand, abstracting away the middleware and the operating system by introducing ghost variables and function contracts requires a great deal of implementation work. This is because the current implementation does not have many necessary mechanisms for inferring annotations based on ghost variables.
Chapter 7

Conclusion and Future work

In this work, the aim was to find ways for improving existing tool for annotation inference for application-level embedded software. The results showed that using interface specifications for inferring auxiliary annotations improved the results. Another improvement was the addition of a mechanism for floating-point annotation inference. Moreover, some steps were taken in the implementation of a mechanism for supporting pointers and arrays. The combination of all these additions has made the tool more capable of proving high-level functional specifications. There is, however, a considerable amount of pending work for the tool to reach a good level of maturity.

Finally, the results achieved in this work were in line with previous work. This confirms the earlier findings and gives more confidence in the technology. The combination of different deductive verification methods such as abstract interpretation, Hoare logic, and weakest precondition seems to be a very promising approach for solving the problem of software verification automation.

7.1 Limitations

For this technology to work, it is essential that the interface specifications document do not contain any ambiguity. Since these documents are currently written by humans, errors or missing information in the documents are unavoidable. This might lead the tool to fail to prove a given module by creating a false-positive alarm. Furthermore, in some very unfortunate but less likely cases, such human error might make the tool successfully verify a faulty module, making the vehicle unsafe. This limitation entails that, although one of the goals of automating the verification process is to eliminate human error,
any human error that occurs in the stage of creating the interface specification
document will still affect the results of the solution. As a conclusion, this tool
will minimize possible human errors in the process of software verification
but it will not completely eliminate them.

7.2 Future work

In this section, a list of improvements and future work is presented. The
difficulty of each of the tasks in the list is evaluated and indicated in
parenthesis. The grades are interpreted as follows:

- **easy** means the solution is relatively obvious and not much investigation
  is needed.
- **medium** means the problem has no obvious solution; thus, some
  investigation is needed.
- **hard** means the problem has no obvious solution, and finding a solution
  might require some theoretical research.

Notice that this grading does not take the time aspect into consideration.
A task can be graded as easy yet be very time-consuming. Also, the order of
the tasks does not entail any prioritization or importance.

7.2.1 What has been left undone?

- Processing mechanism for array indices, e.g., the expression \texttt{index} in
  \texttt{array[index]} must be processed. \textit{(easy)}

- Inference mechanism for pointers of custom types, e.g., a pointer for a
  struct type. \textit{(medium)}

- Considering that MISRA allows pointer arithmetic only for indexing
  purposes, investigate whether such use exists in Scania’s embedded
code. \textit{(easy)}

- Investigate a solution for the over-approximation caused by EVA
  in connection with branching instructions. For more details see
  Subsection 5.1.2. \textit{(medium/hard)}

- Investigate an optimal solution for the issue with dependencies from the
  middleware and the operating system. \textit{(medium/hard)}
• Increase the reliability of the solution by establishing a test coverage strategy, and implementing the necessary unit-tests and integration tests to achieve a good coverage. *easy*

• Designing and implementing a preprocessing stage in which the following tasks are done. For more details see Section 6.5. *medium/hard*
  
  – Remove all C *Pragma* directives which are not defined in the ISO C99 standard.
  
  – Create a suitable project structure for the module and its dependencies so that all the dependencies of the module are automatically imported into the Frama-C platform.

• Investigate an optimal way for solving the issue with static local variables. For details see Subsection 5.2.1. *hard* *(medium)*

• In this work, we considered modules with a single entry-point function. Investigate the viability of the current implementation for modules with multiple entry-point functions. *medium/hard*

The task of implementing a processing mechanism for array indices is an easy task that has a substantial impact on the tool’s performance. Other than this task, all the pending tasks are equally important.

### 7.3 Reflections

The main reflection that we have from this work is a technical one. If someone or some party at Scania AB, the host company for this thesis work, aims to complete the development of the auxiliary annotation inference tool, then we recommend that the task gets assigned to a team of software developers. We believe it is difficult to complete a reliably working tool of this size by a single developer or by multiple thesis workers that do not work together in a coordinated fashion.
References


Appendix A

Figures

A.1 Simplified requirements for accelerator pedal

Interface Specifications

<table>
<thead>
<tr>
<th>Name</th>
<th>Unit</th>
<th>Value Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCPED_POSITION</td>
<td>percent</td>
<td>[0.0, 100.0]</td>
<td>The accelerator pedal position</td>
</tr>
</tbody>
</table>

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Unit</th>
<th>Value Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCPED_DEAD_ZONE</td>
<td>percent</td>
<td>[0.0, 10.0]</td>
<td>Deadzone at the beginning of the pedal position to avoid unintended acceleration</td>
</tr>
<tr>
<td>ACCPED_SENSOR_MAX_VALUE</td>
<td>volt</td>
<td>24.5</td>
<td>The maximum value that the accelerator pedal sensor can output</td>
</tr>
</tbody>
</table>

U/O

<table>
<thead>
<tr>
<th>Name</th>
<th>Unit</th>
<th>Value Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCPED_SENSOR</td>
<td>volt</td>
<td>[0.0, 24.5]</td>
<td>Accelerator pedal sensor</td>
</tr>
</tbody>
</table>

Functional Specifications

Requirement 1

The intermediate value pedal, raw, position have a value between 0.0 and 100.0 percent.

Requirement 2

The value of pedal, raw, position has a linear relation with the value of ACCPED_SENSOR according to the following

\[
\text{pedal}\_\text{raw, position} = \frac{\text{ACCPED}\_\text{SENSOR}}{\text{ACCPED}\_\text{SENSOR}\_\text{MAX, VALUE}}
\]

Requirement 3

The output signal ACCPED, POSITION have a value between 0.0 and 100.0 percent.

Requirement 4

The output signal ACCPED, POSITION relates linearly to the value of pedal, raw, position as in the following table.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pedal_raw_position = ACCPED_DEAD_ZONE</td>
<td></td>
</tr>
<tr>
<td>pedal_raw_position + ACCPED_DEAD_ZONE</td>
<td>ACCPED_POSITION = 0.0</td>
</tr>
<tr>
<td>pedal_raw_position + ACCPED_DEAD_ZONE</td>
<td>pedal_raw_position = 100.0 * (pedal_raw_position - ACCPED_DEAD_ZONE) / (100.0 - ACCPED_DEAD_ZONE)</td>
</tr>
</tbody>
</table>

Figure A.1: A simplified requirements document for the accelerator pedal module. The document contains both interface requirements and functional requirements.
Appendix B

C Source Code

B.1 Simplified steering module with interface specifications

```c
/*
   Interface specifications

   the format is:
   <Specification Number>
   <Interface>
   <Signal Name>
   <Unit>
   {<Possible values>}

   (The information here is fictional and do not reflect any original data.)

Spec. 1
CruiseControlVehSpeed
WheelBasedVehicleSpeed
km/h
{
    s : where s belongs [5..300], 0xFE00 = Error, 0xFF00 = NotAvailable
}

Spec. 2
CruiseControlVehSpeed
ParkingBrakeSwitch
binary
{
    0x0 = ON,
    0x1 = OFF,
    0x2 = Error,
    0x3 = NotAvailable
}

Spec. 3
Input
```
FlowSwitch
binary
{
  0x0 = ON,
  0x1 = OFF
}
Spec. 4
Output
ElectricMotor
binary
{
  0x0 = ON,
  0x1 = OFF
}
Spec. 5
Input
PositionSensor
binary
{
  0x0 = Flow,
  0x1 = NoFlow
} */
#define VEH_MOVING_LIMIT 5
#define TRUE 1
#define FALSE 0
typedef enum {
  WORKING,
  NO_FLOW,
  SHORT_CIRCUIT,
} SENSOR_STATE;
typedef struct {
  int wheelSpeed;
  int parkingBrake;
  int primLowFlow;
  int primHighVoltage;
  int secondCircHandlesStee;
  int electricMotorAct;
} VEHICLE_INFO;
typedef enum {
  PARKING_BRAKE_APPLIED,
  PRIMARY_CIRCUIT_LOW_FLOW,
  PRIMARY_CIRCUIT_HIGH_VOLTAGE,
  WHEEL_BASED_SPEED,
  SECONDARY_CIRCUIT_HANDLES_STEERING,
  ELECTRIC_MOTOR_ACTIVATED,
  NUM_SIGNALS
} SIGNAL;
int state[NUM_SIGNALS]; // Global state
/*
Reads the specified signal from the state.

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

Writes the specified signal to the state.

```c
void write(SIGNAL idx, int val) {
    if (idx < NUM_SIGNALS) {
        state[idx] = val;
    }
}
```

Reads the current state of the system.

```c
void get_system_state(VEHICLE_INFO* veh_info) {
    veh_info->wheelSpeed = read(WHEEL_BASED_SPEED);
    veh_info->parkingBrake = read(PARKING_BRAKE_APPLIED);
    veh_info->primLowFlow = read(PRIMARY_CIRCUIT_LOW_FLOW);
    veh_info->primHighVoltage = read(PRIMARY_CIRCUIT_HIGH_VOLTAGE);
    veh_info->secondCircHandlesStee = read(SECONDARY_CIRCUIT_HANDLES_STEERING);
    veh_info->electricMotorAct = read(ELECTRIC_MOTOR_ACTIVATED);
}
```

Evaluates the state of the primary steering circuit sensors.

```c
void eval_prim_sensor_state(VEHICLE_INFO* veh_info, SENSOR_STATE* sensor_state) {
    if (veh_info->primHighVoltage == TRUE) {
        *sensor_state = SHORT_CIRCUIT;
    } else if (veh_info->primLowFlow == TRUE) {
        *sensor_state = NO_FLOW;
    } else {
        *sensor_state = WORKING;
    }
}
```

Evaluates whether steering should be handled by the secondary circuit.

```c
void secondary_steering(VEHICLE_INFO* veh_info, SENSOR_STATE* sensor_state) {
    char vehicleIsMoving;
    char vehicleIsMovingWithoutPrimaryPowerSteering;
    // Check whether the vehicle is moving.
    if (veh_info->wheelSpeed > VEH_MOVING_LIMIT) {
        vehicleIsMoving = TRUE;
    } else {
        vehicleIsMoving = FALSE;
    }
    // Check whether vehicle is moving without primary power steering.
    // (Code snippet continues here)
```
if (vehicleIsMoving == TRUE &&
    (*sensor_state == NO_FLOW || *sensor_state == SHORT_CIRCUIT)) {
    vehicleIsMovingWithoutPrimaryPowerSteering = TRUE;
} else {
    vehicleIsMovingWithoutPrimaryPowerSteering = FALSE;
}

// Let secondary circuit handle steering if necessary.
if (vehicleIsMovingWithoutPrimaryPowerSteering == TRUE) {
    veh_info->secondCircHandlesStee = TRUE;
}

// Activate the electric motor.
if (veh_info->secondCircHandlesStee == TRUE &&
    veh_info->parkingBrake == FALSE) {
    veh_info->electricMotorAct = TRUE;
}

/*
Module entry point function.
*/
/*@ behavior interface_spec:
assumes \true;

// Spec. 1
requires (state[WHEEL_BASED_SPEED] >= 5
    && state[WHEEL_BASED_SPEED] <= 300) 
    || state[WHEEL_BASED_SPEED] == 0xFE00 
    || state[WHEEL_BASED_SPEED] == 0xFF00;

// Spec. 2
requires state[PARKING_BRAKE_APPLIED] >= 0
    && state[PARKING_BRAKE_APPLIED] <= 3;

// Spec. 3
requires state[SECONDARY_CIRCUIT_HANDLES_STEERING] == 0
    || state[SECONDARY_CIRCUIT_HANDLES_STEERING] == 1;

// Spec. 4
requires state[ELECTRIC_MOTOR_ACTIVATED] == 0
    || state[ELECTRIC_MOTOR_ACTIVATED] == 1;

// Spec. 5
requires state[PRIMARY_CIRCUIT_LOW_FLOW] == 0
    || state[PRIMARY_CIRCUIT_LOW_FLOW] == 1;
ensures state[SECONDARY_CIRCUIT_HANDLES_STEERING] \in \{0, 1\};
ensures state[ELECTRIC_MOTOR_ACTIVATED] \in \{0, 1\};
assigns state[SECONDARY_CIRCUIT_HANDLES_STEERING],
    state[ELECTRIC_MOTOR_ACTIVATED];
*/
void steering() {
    VEHICLE_INFO veh_info;
    SENSOR_STATE prim_sensor;

    get_system_state(&veh_info);
    eval_prim_sensor_state(&veh_info, &prim_sensor);
Listing B.1: A simplified steering module.

B.2 ISP output on simplified steering module

```c
#define VEH_MOVING_LIMIT 5
#define TRUE 1
#define FALSE 0

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

typedef struct {
    int wheelSpeed;
    int parkingBrake;
    int primLowFlow;
    int primHighVoltage;
    int secondCircHandlesStee;
    int electricMotorAct;
} VEHICLE_INFO;

typedef enum {
    PARKING_BRAKE_APPLIED,
    PRIMARY_CIRCUIT_LOW_FLOW,
    PRIMARY_CIRCUIT_HIGH_VOLTAGE,
    WHEEL_BASED_SPEED,
    SECONDARY_CIRCUIT_HANDLES_STEERING,
    ELECTRIC_MOTOR_ACTIVATED,
    NUM_SIGNALS
} SIGNAL;

int state[NUM_SIGNALS]; // Global state

/*@ behavior isp_generated:
    assumes \true;
    requires state[4] \geq 5;
    requires state[4] \leq 65280;
    requires state[0] \geq 5;
    requires state[0] \leq 65280;
    requires state[1] \geq 5;
    requires state[1] \leq 65280;
    requires state[2] \geq 5;
    requires state[2] \leq 65280;
    requires state[5] \geq 5;
*/
```
80 | Appendix B: C Source Code

```c
requires state[5] <= 65280;
requires state[3] >= 5;
requires state[3] <= 65280;
requires valid_read(&state[4]);
requires valid_read((int *)state);
requires valid_read(state[1]);
requires valid_read(state[2]);
requires valid_read(state[5]);
ensures result >= -2147483648;
ensures result <= 2147483647;
assigns nothing;
*/
int read(SIGNAL idx) {
    if (idx < NUM_SIGNALS) {
        return state[idx];
    }
}
/*@ behavior isp_generated:
assumes \true;
requires val \in \{0, 1\};
requires valid_read(&state[4]);
requires valid_read(state[4]);
requires valid_read(state[5]);
ensures state[4] >= 5;
ensures state[4] <= 65280;
ensures state[5] >= 5;
ensures state[5] <= 65280;
assigns state[5], state[4];
*/
void write(SIGNAL idx, int val) {
    if (idx < NUM_SIGNALS) {
        state[idx] = val;
    }
}
/*@ behavior isp_generated:
assumes \true;
requires state[4] >= 5;
requires state[4] <= 65280;
requires state[0] >= 5;
requires state[0] <= 65280;
requires state[1] >= 5;
requires state[1] <= 65280;
requires state[2] >= 5;
requires state[2] <= 65280;
requires state[5] >= 5;
requires state[5] <= 65280;
requires valid_read(&state[4]);
requires valid_read((int *)state);
requires valid_read(state[1]);
requires valid_read(state[2]);
requires valid_read(state[5]);
requires valid_read(state[3]);
assigns nothing;
```
void get_system_state(VEHICLE_INFO* veh_info) {
    veh_info->wheelSpeed = read(WHEEL_BASED_SPEED);
    veh_info->parkingBrake = read(PARKING_BRAKE_APPLIED);
    veh_info->primLowFlow = read(PRIMARY_CIRCUIT_LOW_FLOW);
    veh_info->primHighVoltage = read(PRIMARY_CIRCUIT_HIGH_VOLTAGE);
    veh_info->secondCircHandlesStee = read(SECONDARY_CIRCUIT_HANDLES_STEERING);
    veh_info->electricMotorAct = read(ELECTRIC_MOTOR_ACTIVATED);
}

/*@ behavior isp_generated:
assumes \true;
requires veh_info->primHighVoltage \geq -2147483648;
requires veh_info->primHighVoltage \leq 2147483647;
requires veh_info->primLowFlow \in \{0, 1\};
requires \valid_read(sensor_state);
requires \valid(sensor_state);
requires \valid_read(sensor_state);
requires \valid(sensor_state);
requires \valid_read(veh_info);
requires \valid_read(veh_info);
ensures *sensor_state \in \{0, 1, 2\};
ensures *sensor_state \in \{0, 1, 2\};
ensures *sensor_state \in \{0, 1, 2\};
assigns *sensor_state;
*/
void eval_prim_sensor_state(VEHICLE_INFO* veh_info,
SENSOR_STATE* sensor_state) {
    if (veh_info->primHighVoltage == TRUE) {
        *sensor_state = SHORT_CIRCUIT;
    } else if (veh_info->primLowFlow == TRUE) {
        *sensor_state = NO_FLOW;
    } else {
        *sensor_state = WORKING;
    }
}

/*@ behavior isp_generated:
assumes \true;
requires veh_info->wheelSpeed \geq 5;
requires veh_info->wheelSpeed \leq 65280;
requires *sensor_state \in \{0, 1, 2\};
requires *sensor_state \in \{0, 1, 2\};
requires veh_info->secondCircHandlesStee \in \{0, 1\};
requires veh_info->parkingBrake \in \{0, 1, 2, 3\};
requires \valid_read(veh_info);
requires \valid_read(veh_info);
requires \valid_read(veh_info);
requires \valid_read(veh_info);
requires \valid_read(veh_info);
requires \valid_read(sensor_state);
requires \valid_read(sensor_state);
requires \valid_read(sensor_state);
requires \valid_read(sensor_state);
requires \valid_read(veh_info);
requires \valid_read(veh_info);
requires \valid_read(veh_info);
requires \valid_read(veh_info);
requires \valid_read(veh_info);
ensures veh_info->secondCircHandlesStee \in \{0, 1\};
ensures veh_info->electricMotorAct \in \{0, 1\};
assigns veh_info->electricMotorAct, veh_info->secondCircHandlesStee;
void secondary_steering(VEHICLE_INFO* veh_info, SENSOR_STATE* sensor_state) {
    char vehicleIsMoving;
    char vehicleIsMovingWithoutPrimaryPowerSteering;

    // Check whether the vehicle is moving.
    if (veh_info->wheelSpeed > VEH_MOVING_LIMIT) {
        vehicleIsMoving = TRUE;
    } else {
        vehicleIsMoving = FALSE;
    }

    // Check whether vehicle is moving without primary power steering.
    if (vehicleIsMoving == TRUE &&
        (*sensor_state == NO_FLOW || *sensor_state == SHORT_CIRCUIT)) {
        vehicleIsMovingWithoutPrimaryPowerSteering = TRUE;
    } else {
        vehicleIsMovingWithoutPrimaryPowerSteering = FALSE;
    }

    // Let secondary circuit handle steering if necessary.
    if (vehicleIsMovingWithoutPrimaryPowerSteering == TRUE) {
        veh_info->secondCircHandlesStee = TRUE;
    }

    // Activate the electric motor.
    if (veh_info->secondCircHandlesStee == TRUE &&
        veh_info->parkingBrake == FALSE) {
        veh_info->electricMotorAct = TRUE;
    }
}

/*@ behavior interface_spec:
   assumes \true;
   requires
       (state[WHEEL_BASED_SPEED] ≥ 5 ∧ state[WHEEL_BASED_SPEED] ≤ 300) ∨
       state[WHEEL_BASED_SPEED] ≡ 0xFE00 ∨
       state[WHEEL_BASED_SPEED] ≡ 0xFF00;
   requires
       state[PARKING BRAKE_APPLIED] ≥ 0 ∧
       state[PARKING BRAKE_APPLIED] ≤ 3;
   requires
       state[SECONDARY_CIRCUIT_HANDLES_STEERING] ≡ 0 ∨
       state[SECONDARY_CIRCUIT_HANDLES_STEERING] ≡ 1;
   requires
       state[ELECTRIC MOTOR_ACTIVATED] ≡ 0 ∨
       state[ELECTRIC MOTOR_ACTIVATED] ≡ 1;
   requires
       state[PRIMARY CIRCUIT LOW FLOW] ≡ 0 ∨
       state[PRIMARY CIRCUIT LOW FLOW] ≡ 1;
   ensures state[SECONDARY CIRCUIT HANDLES STEERING] ∈ (0, 1);
   ensures state[ELECTRIC MOTOR ACTIVATED] ∈ (0, 1);
   assigns state[SECONDARY CIRCUIT HANDLES STEERING],
          state[ELECTRIC MOTOR ACTIVATED];
   */

void steering() {
    VEHICLE_INFO veh_info;
    SENSOR_STATE prim_sensor;
}
get_system_state(&veh_info);

eval_prim_sensor_state(&veh_info, &prim_sensor);

secondary_steering(&veh_info, &prim_sensor);

write(SECONDARY_CIRCUIT_HANDLES_STEERING, veh_info.secondCircHandlesStee);
write(EFFECTIVE_MOTOR_ACTIVATED, veh_info.electricMotorAct);
}

Listing B.2: ISP’s output on simplified steering module.

B.3 Simplified steering module with functional specifications

#define VEH_MOVING_LIMIT 5
#define TRUE 1
#define FALSE 0
typedef enum {
  WORKING,
  NO_FLOW,
} SENSOR_STATE;

typedef struct {
  int wheelSpeed;
  int parkingBrake;
  int primLowFlow;
  int primHighVoltage;
  int secondCircHandlesStee;
  int electricMotorAct;
} VEHICLE_INFO;

typedef enum {
  PARKING_BRAKE_APPLIED,
  PRIMARY_CIRCUIT_LOW_FLOW,
  PRIMARY_CIRCUIT_HIGH_VOLTAGE,
  WHEEL_BASED_SPEED,
  SECONDARY_CIRCUIT_HANDLES_STEERING,
  ELECTRIC_MOTOR_ACTIVATED,
} SIGNAL;

// ghost variables representing model_variables in requirements
//@ ghost int model_vehicleIsMoving;
//@ ghost int model_vehicleMovingWithoutPrimaryPowerSteering;
//@ ghost int model_primaryCircuitProvidingPowerSteering;

int state[NUM_SIGNALS]; // Global state
Appendix B: C Source Code

```c
/*
 * Reads the specified signal from the state.
 */
int read(SIGNAL idx) {
    if (idx < NUM_SIGNALS) {
        return state[idx];
    }
}

/*
 * Writes the specified signal to the state.
 */
void write(SIGNAL idx, int val) {
    if (idx < NUM_SIGNALS) {
        state[idx] = val;
    }
}

/*
 * Reads the current state of the system.
 */
void get_system_state(VEHICLE_INFO* veh_info) {
    veh_info->wheelSpeed = read(WHEEL_BASED_SPEED);
    veh_info->parkingBrake = read(PARKING_BRAKE_APPLIED);
    veh_info->primLowFlow = read(PRIMARY_CIRCUIT_LOW_FLOW);
    veh_info->primHighVoltage = read(PRIMARY_CIRCUIT_HIGH_VOLTAGE);
    veh_info->secondCircHandlesStee = read(SECONDARY_CIRCUIT_HANDLES_STEERING);
    veh_info->electricMotorAct = read(ELECTRIC_MOTOR_ACTIVATED);
}

/*
 * Evaluates the state of the primary steering circuit sensors.
 */
/*@ behavior functional_spec:
 * requires \valid_read(veh_info);
 * requires \valid(sensor_state);
 * ensures \old(veh_info)->primHighVoltage == TRUE
 *     ==> *sensor_state == SHORT_CIRCUIT;
 * ensures \old(veh_info)->primHighVoltage == TRUE
 *     && veh_info->primLowFlow == TRUE
 *     ==> *sensor_state == NO_FLOW;
 * ensures veh_info->primHighVoltage == FALSE
 *     && veh_info->primLowFlow == FALSE
 *     ==> *sensor_state == WORKING;
 * ensures \old(veh_info->primHighVoltage) == TRUE
 *     || \old(veh_info->primLowFlow) == TRUE
 *     ==> *sensor_state != WORKING;
 * ensures \old(sensor_state) == NO_FLOW
 *     || \old(sensor_state) == SHORT_CIRCUIT
 *     || \old(sensor_state) == WORKING
 * assigns *sensor_state;
 */
void eval_prim_sensor_state(VEHICLE_INFO* veh_info, SENSOR_STATE* sensor_state) {
    if (veh_info->primHighVoltage == TRUE) {
        *sensor_state = SHORT_CIRCUIT;
    } else if (veh_info->primLowFlow == TRUE) {
        *sensor_state = NO_FLOW;
    }
    ...
```
Appendix B: C Source Code

```c
98 } else {
99     *sensor_state = WORKING;
100 }
101 */
102 /*
103 Evaluates whether steering should be
104 handled by the secondary circuit.
105 */
106/*@
107
108 behavior functional_spec:
109   requires \valid(veh_info);
110   requires \valid_read(sensor_state);
111
112   // Req. 2
113   ensures \old(veh_info)->wheelSpeed > VEH_MOVING_LIMIT
114     ==> model_vehicleIsMoving == TRUE;
115
116   // Req. 3
117   ensures model_vehicleIsMoving == TRUE
118     && \old(*sensor_state) != WORKING
119     ==> model_vehicleMovingWithoutPrimaryPowerSteering == TRUE;
120
121   // Req. 4
122   ensures model_vehicleMovingWithoutPrimaryPowerSteering == TRUE
123     ==> veh_info->secondCircHandlesStee == TRUE;
124
125   // Req. 5
126   ensures veh_info->secondCircHandlesStee == TRUE
127     && \old(veh_info)->parkingBrake == FALSE
128     ==> veh_info->electricMotorAct == TRUE;
129
130   assigns model_vehicleIsMoving,
131     model_vehicleMovingWithoutPrimaryPowerSteering,
132     veh_info->secondCircHandlesStee,
133     veh_info->electricMotorAct;
134 */
135 void secondary_steering(VEHICLE_INFO* veh_info, SENSOR_STATE* sensor_state) {
136     char vehicleIsMoving;
137     char vehicleIsMovingWithoutPrimaryPowerSteering;
138     // Check whether the vehicle is moving.
139     if (veh_info->wheelSpeed > VEH_MOVING_LIMIT) {
140         vehicleIsMoving = TRUE;
141     } else {
142         vehicleIsMoving = FALSE;
143     }
144
145     // Check whether vehicle is moving without primary power steering.
146     if (vehicleIsMoving == TRUE &&
147         (*sensor_state == NO_FLOW || *sensor_state == SHORT_CIRCUIT)) {
148         vehicleIsMovingWithoutPrimaryPowerSteering = TRUE;
149     } else {
150         vehicleIsMovingWithoutPrimaryPowerSteering = FALSE;
151     }
152
153     // Let secondary circuit handle steering if necessary.
154     if (vehicleIsMovingWithoutPrimaryPowerSteering == TRUE) {
155         veh_info->secondCircHandlesStee = TRUE;
156     }
```
// Activate the electric motor.
if (veh_info->secondCircHandlesStee == TRUE &&
    veh_info->parkingBrake == FALSE) {
    veh_info->electricMotorAct = TRUE;
}

//@ ghost model_vehicleMovingWithoutPrimaryPowerSteering =
vehicleIsMovingWithoutPrimaryPowerSteering;
//@ ghost model_vehicleIsMoving = vehicleIsMoving;

/* Module entry point function. */
/*@
behavior functional_spec:
// Req. 1-4 without intermediary variables *
ensures (∀old(state[PRIMARY_CIRCUIT_HIGH_VOLTAGE]) == TRUE
   || ∀old(state[PRIMARY_CIRCUIT_LOW_FLOW]) == TRUE)
   && ∀old(state[WHEEL_BASED_SPEED]) > VEH_MOVING_LIMIT
   ==> state[SECONDARY_CIRCUIT_HANDLES_STEERING] == TRUE;
// Req. 1 *
ensures (∀old(state[PRIMARY_CIRCUIT_HIGH_VOLTAGE]) == TRUE
   || ∀old(state[PRIMARY_CIRCUIT_LOW_FLOW]) == TRUE)
   ==> model_primaryCircuitProvidingPowerSteering == FALSE;
// Req. 2
ensures ∀old(state[WHEEL_BASED_SPEED]) > VEH_MOVING_LIMIT
   ==> model_vehicleIsMoving == TRUE;
// Req. 3
ensures model_vehicleIsMoving == TRUE
   && model_primaryCircuitProvidingPowerSteering == FALSE
   ==> model_vehicleMovingWithoutPrimaryPowerSteering == TRUE;
// Req. 4
ensures model_vehicleMovingWithoutPrimaryPowerSteering == TRUE
   ==> state[SECONDARY_CIRCUIT_HANDLES_STEERING] == TRUE;
// Req. 5
ensures state[SECONDARY_CIRCUIT_HANDLES_STEERING] == TRUE
   && ∀old(state[PARKING_BRAKE_APPLIED]) == FALSE
   ==> state[ELECTRIC_MOTOR_ACTIVATED] == TRUE;
*/

void steering() {
    VEHICLE_INFO veh_info;
    SENSOR_STATE prim_sensor;
    get_system_state(&veh_info);
    eval_prim_sensor_state(&veh_info, &prim_sensor);
    //@ ghost model_primaryCircuitProvidingPowerSteering = (prim_sensor ==
    WORKING);
    secondary_steering(&veh_info, &prim_sensor);
Listing B.3: Simplified steering module that only includes the functional specification annotations.

### B.4 Corrected ISP output for simplified steering module

```c
#define VEH_MOVING_LIMIT 5
#define TRUE 1
#define FALSE 0
typedef enum {
    WORKING,
    NO_FLOW,
    SHORT_CIRCUIT,
} SENSOR_STATE;

typedef struct {
    int wheelSpeed;
    int parkingBrake;
    int primLowFlow;
    int primHighVoltage;
    int secondCircHandlesStee;
    int electricMotorAct;
} VEHICLE_INFO;

typedef enum {
    PARKING_BRAKE_APPLIED,
    PRIMARY_CIRCUIT_LOW_FLOW,
    PRIMARY_CIRCUIT_HIGH_VOLTAGE,
    WHEEL_BASED_SPEED,
    SECONDARY_CIRCUIT_HANDLES_STEERING,
    ELECTRIC_MOTOR_ACTIVATED,
    NUM_SIGNALS
} SIGNAL;

int state[NUM_SIGNALS]; // Global state

/*@ behavior isp_generated:
  assumes \true;
  requires idx \in (0..5);
  requires state[4] \geq 0;
  requires state[4] \leq 1;
  requires state[0] \geq 0;
  requires state[0] \leq 3;
  requires state[1] \geq 0;
  requires state[1] \leq 1;
  requires state[2] \geq -2147483648;
*/
```
requires state[2] <= 2147483647;
requires state[5] >= 0;
requires state[5] <= 1;
requires state[3] >= 5;
requires state[3] <= 65280;
requires \valid_read(&state[4]);
requires \valid_read((int *)state);
requires \valid_read(&state[1]);
requires \valid_read(&state[2]);
requires \valid_read(&state[3]);
ensures \result == state[idx];
assigns \nothing;
*/
int read(SIGNAL idx) {
if (idx < NUM_SIGNALS) {
  return state[idx];
}
}
/*@ behavior isp_generated:
assumes \true;
requires state[4] >= 0;
requires state[4] <= 1;
requires state[5] >= 0;
requires state[5] <= 1;
requires state[0] >= 0;
requires state[0] <= 3;
requires state[1] >= 0;
requires state[1] <= 1;
requires state[2] >= -2147483648;
requires state[2] <= 2147483647;
requires state[5] >= 0;
requires state[5] <= 1;
requires state[3] >= 5;
requires state[3] <= 65280;
requires \valid_read(&state[4]);
*/
void write(SIGNAL idx, int val) {
if (idx < NUM_SIGNALS) {
  state[idx] = val;
}
}
requires \valid_read((int *)state);
requires \valid_read(state[1]);
requires \valid_read(state[2]);
requires \valid_read(state[5]);
requires \valid_read(state[3]);
requires \valid_read(&veh_info->wheelSpeed);
requires \valid(&veh_info->wheelSpeed);
requires \valid_read(&veh_info->parkingBrake);
requires \valid(&veh_info->parkingBrake);
requires \valid_read(&veh_info->primLowFlow);
requires \valid(&veh_info->primLowFlow);
requires \valid_read(&veh_info->primHighVoltage);
requires \valid(&veh_info->primHighVoltage);
requires \valid_read(&veh_info->secondCircHandlesStee);
requires \valid(&veh_info->secondCircHandlesStee);
requires \valid_read(&veh_info->electricMotorAct);
requires \valid(&veh_info->electricMotorAct);
ensures veh_info->wheelSpeed ∈ (5..65280);
ensures veh_info->parkingBrake ∈ {0, 1, 2, 3};
ensures veh_info->primLowFlow ∈ {0, 1};
ensures veh_info->primHighVoltage ∈ (-2147483648..2147483647);
ensures veh_info->secondCircHandlesStee ∈ {0, 1};
ensures veh_info->electricMotorAct ∈ {0, 1};
assigns veh_info->wheelSpeed,
veh_info->parkingBrake,
veh_info->primLowFlow,
veh_info->primHighVoltage,
veh_info->secondCircHandlesStee,
veh_info->electricMotorAct;

*/
void get_system_state(VEHICLE_INFO* veh_info) {
    veh_info->wheelSpeed = read(WHEEL_BASED_SPEED);
    veh_info->parkingBrake = read(PARKING_BRAKE_APPLIED);
    veh_info->primLowFlow = read(PRIMARY_CIRCUIT_LOW_FLOW);
    veh_info->primHighVoltage = read(PRIMARY_CIRCUIT_HIGH_VOLTAGE);
    veh_info->secondCircHandlesStee = read(SECONDARY_CIRCUIT_HANDLES_STEERING);
    veh_info->electricMotorAct = read(EFFECTIVE_MOTOR_ACTIVATED);
}

/*@ behavior isp_generated:
assumes \true;
requires veh_info->primHighVoltage ≥ -2147483648;
requires veh_info->primHighVoltage ≤ 2147483647;
requires veh_info->primLowFlow ∈ {0, 1};
requires \valid_read(sensor_state);
requires \valid(sensor_state);
requires \valid_read(sensor_state);
requires \valid(sensor_state);
requires \valid_read(sensor_state);
requires \valid(read(veh_info));
requires \valid_read(veh_info);
ensures *sensor_state ∈ {0, 1, 2};
ensures *sensor_state ∈ {0, 1, 2};
ensures *sensor_state ∈ {0, 1, 2};
assigns *sensor_state;
*/
void eval_prim_sensor_state(VEHICLE_INFO* veh_info,
SENSOR_STATE* sensor_state) {
if (veh_info->primHighVoltage == TRUE) {
    *sensor_state = SHORT_CIRCUIT;
} else if (veh_info->primLowFlow == TRUE) {
    *sensor_state = NO_FLOW;
} else {
    *sensor_state = WORKING;
}

/*@ behavior isp_generated:
assumes \	true;
requires veh_info->wheelSpeed ≥ 5;
requires veh_info->wheelSpeed ≤ 65280;
requires veh_info->secondCircHandlesStee ∈ {0, 1};
requires veh_info->electricMotorAct ∈ {0, 1};
requires *sensor_state ∈ {0, 1, 2};
requires *sensor_state ∈ {0, 1, 2};
requires veh_info->secondCircHandlesStee ∈ {0, 1};
requires veh_info->parkingBrake ∈ {0, 1, 2, 3};
requires \valid_read(veh_info);
requires \valid(veh_info);
requires \valid_read(sensor_state);
requires \valid_read(veh_info);
requires \valid(veh_info);
requires \valid_read(veh_info);
requires \valid_read(veh_info);
requires \valid_read(sensor_state);
requires \valid_read(veh_info);
requires \valid_read(veh_info);
requires \valid_read(veh_info);
ensures veh_info->secondCircHandlesStee ∈ {0, 1};
ensures veh_info->electricMotorAct ∈ {0, 1};
assigns veh_info->electricMotorAct, veh_info->secondCircHandlesStee;
*/
void secondary_steering(VEHICLE_INFO* veh_info, SENSOR_STATE* sensor_state) {
    // Check whether the vehicle is moving.
    if (veh_info->wheelSpeed > VEH_MOVING_LIMIT) {
        vehicleIsMoving = TRUE;
    } else {
        vehicleIsMoving = FALSE;
    }

    // Check whether vehicle is moving without primary power steering.
    if (vehicleIsMoving == TRUE &&
        (*sensor_state == NO_FLOW || *sensor_state == SHORT_CIRCUIT)) {
        vehicleIsMovingWithoutPrimaryPowerSteering = TRUE;
    } else {
        vehicleIsMovingWithoutPrimaryPowerSteering = FALSE;
    }

    // Let secondary circuit handle steering if necessary.
    if (vehicleIsMovingWithoutPrimaryPowerSteering == TRUE) {
        veh_info->secondCircHandlesStee = TRUE;
    }

    // Activate the electric motor.
    if (veh_info->secondCircHandlesStee == TRUE &&
        veh_info->parkingBrake == FALSE) {
veh_info->electricMotorAct = TRUE;
}

/*@ behavior interface_spec:
assumes \true;
requires
(state[WHEEL_BASED_SPEED] ≥ 5 ∧ state[WHEEL_BASED_SPEED] ≤ 300) ∨
(state[WHEEL_BASED_SPEED] ≡ 0xFE00 ∨
state[WHEEL_BASED_SPEED] ≡ 0xFF00);
requires
state[PARKING_BRAKE_APPLIED] ≥ 0 ∧
state[PARKING_BRAKE_APPLIED] ≤ 3;
requires
state[SECONDARY_CIRCUIT_HANDLES_STEERING] ≡ 0 ∨
state[SECONDARY_CIRCUIT_HANDLES_STEERING] ≡ 1;
requires
state[ELECTRIC_MOTOR_ACTIVATED] ≡ 0 ∨
state[ELECTRIC_MOTOR_ACTIVATED] ≡ 1;
requires
state[PRIMARY_CIRCUIT_LOW_FLOW] ≡ 0 ∨
state[PRIMARY_CIRCUIT_LOW_FLOW] ≡ 1;
ensures state[SECONDARY_CIRCUIT_HANDLES_STEERING] ∈ {0, 1};
ensures state[ELECTRIC_MOTOR_ACTIVATED] ∈ {0, 1};
assigns state[SECONDARY_CIRCUIT_HANDLES_STEERING],
state[ELECTRIC_MOTOR_ACTIVATED];
*/
void steering() {
VEHICLE_INFO veh_info;
SENSOR_STATE prim_sensor;
get_system_state(&veh_info);
eval_prim_sensor_state(&veh_info, &prim_sensor);
secondary_steering(&veh_info, &prim_sensor);
write(SECONDARY_CIRCUIT_HANDLES_STEERING, veh_info.secondCircHandlesStee);
write(EFFECTIVE_MOTOR_ACTIVATED, veh_info.electricMotorAct);"
typedef struct {
  int wheelSpeed;
  int parkingBrake;
  int primLowFlow;
  int primHighVoltage;
  int secondCircHandlesStee;
  int electricMotorAct;
} VEHICLE_INFO;

typedef enum {
  PARKING_BRAKE_APPLIED,
  PRIMARY_CIRCUIT_LOW_FLOW,
  PRIMARY_CIRCUIT_HIGH_VOLTAGE,
  WHEEL_BASED_SPEED,
  SECONDARY_CIRCUIT_HANDLES_STEERING,
  ELECTRIC_MOTOR_ACTIVATED,
  NUM_SIGNALS
} SIGNAL;

int state[NUM_SIGNALS]; // Global state

/*@ behavior isp_generated:
  assumes \true;
  requires idx ∈ {0..5};
  requires state[4] ≥ 0;
  requires state[4] ≤ 1;
  requires state[0] ≥ 0;
  requires state[0] ≤ 3;
  requires state[1] ≥ 0;
  requires state[1] ≤ 1;
  requires state[2] ≥ -2147483648;
  requires state[2] ≤ 2147483647;
  requires state[5] ≥ 0;
  requires state[5] ≤ 1;
  requires state[3] ≥ 5;
  requires state[3] ≤ 65280;
  requires \valid_read(xstate[4]);
  requires \valid_read((int *)state);
  requires \valid_read(xstate[1]);
  requires \valid_read(xstate[2]);
  requires \valid_read(xstate[5]);
  requires \valid_read(xstate[3]);
  ensures \result == state[idx];
  assigns \nothing;
*/
int read(SIGNAL idx) {
if (idx < NUM_SIGNALS) {
    return state[idx];
}

/*@ behavior isp_generated:
    assumes \true;
    requires state[4] \geq 0;
    requires state[4] \leq 1;
    requires state[5] \geq 0;
    requires state[5] \leq 1;
    requires idx \in \{4, 5\};
    requires val \in \{0, 1\};
    requires \valid_read(&state[4]);
    requires \valid_read(&state[5]);
    ensures state[4] \geq 0;
    ensures state[4] \leq 1;
    ensures state[5] \geq 0;
    ensures state[5] \leq 1;
    assigns state[5], state[4];
*/
void write(SIGNAL idx, int val) {
    if (idx < NUM_SIGNALS) {
        state[idx] = val;
    }
}

/*@ behavior isp_generated:
    assumes \true;
    requires state[4] \geq 0;
    requires state[4] \leq 1;
    requires state[0] \geq 0;
    requires state[0] \leq 3;
    requires state[1] \geq 0;
    requires state[1] \leq 1;
    requires state[2] \geq -2147483648;
    requires state[2] \leq 2147483647;
    requires state[5] \geq 0;
    requires state[5] \leq 1;
    requires state[3] \geq 5;
    requires state[3] \leq 65280;
    requires \valid_read(&state[4]);
    requires \valid_read((int *)state);
    requires \valid_read(&state[1]);
    requires \valid_read(&state[2]);
    requires \valid_read(&state[3]);
    requires \valid_read(&veh_info->wheelSpeed);
    requires \valid(&veh_info->wheelSpeed);
    requires \valid_read(&veh_info->parkingBrake);
    requires \valid(&veh_info->parkingBrake);
*/
void get_system_state(VEHICLE_INFO* veh_info) {
    veh_info->wheelSpeed = read(WHEEL_BASED_SPEED);
    veh_info->parkingBrake = read(PARKING_BRAKE_APPLIED);
    veh_info->primLowFlow = read(PRIMARY_CIRCUIT_LOW_FLOW);
    veh_info->primHighVoltage = read(PRIMARY_CIRCUIT_HIGH_VOLTAGE);
    veh_info->secondCircHandlesStee = read(SECONDARY_CIRCUIT_HANDLES_STEERING);
    veh_info->electricMotorAct = read(ELECTRIC_MOTOR_ACTIVATED);
}
```c
void eval_prim_sensor_state(VEHICLE_INFO* veh_info,
                          SENSOR_STATE* sensor_state) {
  if (veh_info->primHighVoltage == TRUE) {
    *sensor_state = SHORT_CIRCUIT;
  } else if (veh_info->primLowFlow == TRUE) {
    *sensor_state = NO_FLOW;
  } else {
    *sensor_state = WORKING;
  }
}
```

```plaintext
/* Evaluates whether steering should be handled by the secondary circuit. */

/* @@ behavior isp_generated: */
```
requires \valid_read(sensor_state);

// Req. 2
ensures \old(veh_info)->wheelSpeed > VEH_MOVING_LIMIT
  ==> model_vehicleIsMoving == TRUE;

// Req. 3
ensures model_vehicleIsMoving == TRUE
  && \old("sensor_state") != WORKING
  ==> model_vehicleMovingWithoutPrimaryPowerSteering == TRUE;

// Req. 4
ensures model_vehicleMovingWithoutPrimaryPowerSteering == TRUE
  ==> veh_info->secondCircHandlesStee == TRUE;

// Req. 5
ensures veh_info->secondCircHandlesStee == TRUE
  && \old(veh_info)->parkingBrake == FALSE
  ==> veh_info->electricMotorAct == TRUE;

assigns model_vehicleIsMoving,
  model_vehicleMovingWithoutPrimaryPowerSteering,
  veh_info->secondCircHandlesStee, veh_info->electricMotorAct;
*/
void secondary_steering(VEHICLE_INFO* veh_info, SENSOR_STATE* sensor_state) {
  char vehicleIsMoving;
  char vehicleIsMovingWithoutPrimaryPowerSteering;

  // Check whether the vehicle is moving.
  if (veh_info->wheelSpeed > VEH_MOVING_LIMIT) {
    vehicleIsMoving = TRUE;
  } else {
    vehicleIsMoving = FALSE;
  }

  // Check whether vehicle is moving without primary power steering.
  if (vehicleIsMoving == TRUE &&
      (*sensor_state == NO_FLOW || *sensor_state == SHORT_CIRCUIT)) {
    vehicleIsMovingWithoutPrimaryPowerSteering = TRUE;
  } else {
    vehicleIsMovingWithoutPrimaryPowerSteering = FALSE;
  }

  // Let secondary circuit handle steering if necessary.
  if (vehicleIsMovingWithoutPrimaryPowerSteering == TRUE) {
    veh_info->secondCircHandlesStee = TRUE;
  }

  // Activate the electric motor.
  if (veh_info->secondCircHandlesStee == TRUE &&
      veh_info->parkingBrake == FALSE) {
    veh_info->electricMotorAct = TRUE;
  }

  //@ ghost model_vehicleMovingWithoutPrimaryPowerSteering =
  // vehicleIsMovingWithoutPrimaryPowerSteering;
  //@ ghost model_vehicleIsMoving = vehicleIsMoving;
}
Module entry point function.

// Interface Specifications
behavior interface_spec:
  assumes \true;

  // Spec. 1
  requires (state[WHEEL_BASED_SPEED] >= 5
    && state[WHEEL_BASED_SPEED] <= 300)
    || state[WHEEL_BASED_SPEED] == 0xFE00
    || state[WHEEL_BASED_SPEED] == 0xFF00;

  // Spec. 2
  requires state[PARKING_BRAKE_APPLIED] >= 0
    && state[PARKING_BRAKE_APPLIED] <= 3;

  // Spec. 3
  requires state[SECONDARY_CIRCUIT_HANDLE_STEERING] == 0
    || state[SECONDARY_CIRCUIT_HANDLE_STEERING] == 1;

  // Spec. 4
  requires state[ELECTRIC_MOTOR_ACTIVATED] == 0
    || state[ELECTRIC_MOTOR_ACTIVATED] == 1;

  // Spec. 5
  requires state[PRIMARY_CIRCUIT_LOW_FLOW] == 0
    || state[PRIMARY_CIRCUIT_LOW_FLOW] == 1;

  ensures state[SECONDARY_CIRCUIT_HANDLE_STEERING] \in {0, 1};
  ensures state[ELECTRIC_MOTOR_ACTIVATED] \in {0, 1};
  assigns state[SECONDARY_CIRCUIT_HANDLE_STEERING],
    state[ELECTRIC_MOTOR_ACTIVATED],
    model_primaryCircuitProvidingPowerSteering,
    model_vehicleMovingWithoutPrimaryPowerSteering,
    model_vehicleIsMoving;

// Functional Specifications
behavior functional_spec:
  // Req. 1-4 without intermediary variables
  ensures \old(state[PRIMARY_CIRCUIT_HIGH_VOLTAGE]) == TRUE
    || \old(state[PRIMARY_CIRCUIT_LOW_FLOW]) == TRUE
    && \old(state[WHEEL_BASED_SPEED]) > VEH_MOVING_LIMIT
    ==> state[SECONDARY_CIRCUIT_HANDLE_STEERING] == TRUE;

  // Req. 1
  ensures \old(state[PRIMARY_CIRCUIT_HIGH_VOLTAGE]) == TRUE
    || \old(state[PRIMARY_CIRCUIT_LOW_FLOW]) == TRUE
    ==> model_primaryCircuitProvidingPowerSteering == FALSE;

  // Req. 2
  ensures \old(state[WHEEL_BASED_SPEED]) > VEH_MOVING_LIMIT
    ==> model_vehicleIsMoving == TRUE;

  // Req. 3
  ensures model_vehicleIsMoving == TRUE
    && model_primaryCircuitProvidingPowerSteering == FALSE
Listing B.5: Manually corrected ISP output of a simplified steering module that not only includes interface specifications but also functional specifications.

B.6 Simplified accelerator pedal module with interface specifications

```c
typedef unsigned char tB;
#define TRUE ((tB)1)
#define FALSE ((tB)0)
typedef enum {
    ACCPED_DEAD_ZONE_PARAM,
    ACCPED_SENSOR_MAX_VALUE,
    NUMBER_OF_PARAMS
} tF64_Params_E;

typedef enum {
    STEERING_M,
    BRAKE_PEDAL_M,
    ACCELERATOR_PEDAL,
    CLUTCH_PEDAL,
    LIGHTING_M,
    NUMBER_OF_MODULES
```
typedef enum {
  BRAKEPED_SENSOR_E,
  ACCPED_SENSOR_E,
  ACCPED_POSITION,
  STEERING_SENSOR_E,
  CLUTCH_SENSOR_E,
  NUMBER_OF_SIGNALS
} tF64_Signals_E;

double DB_TF64_PARAMS[NUMBER_OF_PARAMS];
tB DB_ENABLED_MODULES[NUMBER_OF_MODULES];
double DB_TF64_SIGNALS[NUMBER_OF_SIGNALS];

double Read_tF64_param(tF64_Params_E tf64_param_E) {
  return DB_TF64_PARAMS[tf64_param_E];
}

tB Is_module_enabled(Modules_E module_E) {
  return DB_ENABLED_MODULES[module_E];
}

double Read_tF64_signal(tF64_Signals_E tf64_signal_E) {
  return DB_TF64_SIGNALS[tf64_signal_E];
}

void Write_tF64_signal(tF64_Signals_E tf64_signal_E, double value) {
  DB_TF64_SIGNALS[tf64_signal_E] = value;
}

double calculate_pedal_position(double raw_position) {
  double pedal_position;
  double dead_zone;  // A dead zone at the begin of the pedal position to avoid unintended acceleration.

  dead_zone = Read_tF64_param(ACCPED_DEAD_ZONE_PARAM);

  if (raw_position < dead_zone) {
    pedal_position = 0.0;
  } else {
    pedal_position = ((raw_position - dead_zone) / (100.0f - dead_zone)) * 100.0f;
  }

  return pedal_position;

/*@ behavior interface_spec:
assumes \true;
requires \valid_read(&DB_ENABLED_MODULES[ACCELERATOR_PEDAL]);
requires \valid(&DB_TF64_SIGNALS[ACCPED_POSITION]);
requires \valid_read(&DB_TF64_SIGNALS[ACCPED_POSITION]);
requires \valid_read(&DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM]);
requires \valid_read(&DB_TF64_PARAMS[ACCPED_SENSOR_MAX_VALUE]);
requires DB_ENABLED_MODULES[ACCELERATOR_PEDAL] \in {TRUE, FALSE};
*/
Listing B.6: A simplified accelerator pedal module.

B.7 ISP output on simplified accelerator pedal module

typedef unsigned char tB;
#define TRUE ((tB)1)
#define FALSE ((tB)0)
typedef enum {
    ACCPED_DEAD_ZONE_PARAM,
    ACCPED_SENSOR_MAX_VALUE,
    NUMBER_OF_PARAMS
} tF64_Params_E;

typedef enum {
    STEERING_M,
    BRAKE_PEDAL_M,
    ACCELERATOR_PEDAL,
    CLUTCH_PEDAL,
    LIGHTING_M,
    NUMBER_OF_MODULES
} Modules_E;

typedef enum {
    BRAKEPED_SENSOR_E,
    ACCPED_SENSOR_E,
    ACCPED_POSITION,
    STEERING_SENSOR_E,
    CLUTCH_SENSOR_E,
    NUMBER_OF_SIGNALS
} tF64_Signals_E;

double DB_TF64_PARAMS[NUMBER_OF_PARAMS];
tB DB_ENABLED_MODULES[NUMBER_OF_MODULES];
double DB_TF64_SIGNALS[NUMBER_OF_SIGNALS];

/*@ behavior isp_generated:
 assumes \true;
 requires DB_TF64_PARAMS[0] \equiv 24.5;
 requires DB_TF64_PARAMS[1] \equiv 24.5;
 requires \valid_read((double *)DB_TF64_PARAMS);
 requires \valid_read(&DB_TF64_PARAMS[1]);
 ensures \result \geq -0. \wedge \result \leq 24.5;
 assigns \nothing;
 */
double Read_tF64_param(tF64_Params_E tf64_param_E) {
    return DB_TF64_PARAMS[tf64_param_E];
}

/*@ behavior isp_generated:
 assumes \true;
 requires DB_ENABLED_MODULES[2] \in \{0, 1\};
 requires \valid_read(&DB_ENABLED_MODULES[2]);
 ensures \result \in \{0, 1\};
 assigns \nothing;
 */
tB Is_module_enabled(Modules_E module_E) {
    return DB_ENABLED_MODULES[module_E];
}

/*@ behavior isp_generated:
 assumes \true;
 requires DB_TF64_SIGNALS[1] \geq -0. \wedge DB_TF64_SIGNALS[1] \leq 24.5;
 requires \valid_read(&DB_TF64_SIGNALS[1]);
 ensures \result \geq -0. \wedge \result \leq 24.5;
 assigns \nothing;
double Read_tF64_signal(tF64_Signals_E tf64_signal_E) {
    return DB_TF64_SIGNALS[tf64_signal_E];
}

/*@ behavior isp_generated:
    assumes \true;
    requires value ≥ -11.1111111111 ∧ value ≤ 111.111111111;
    requires valid(DB_TF64_SIGNALS[2]);
    assigns DB_TF64_SIGNALS[2];
*/
void Write_tF64_signal(tF64_Signals_E tf64_signal_E, double value) {
    DB_TF64_SIGNALS[tf64_signal_E] = value;
}

/*@ behavior isp_generated:
    assumes \true;
    requires DB_TF64_PARAMS[0] ≡ 24.5;
    requires DB_TF64_PARAMS[1] ≡ 24.5;
    requires raw_position ≥ -0. ∧ raw_position ≤ 100.;
    requires raw_position ≥ -0. ∧ raw_position ≤ 100.0;
    requires valid(DB_TF64_PARAMS);
    requires valid(DB_TF64_PARAMS[1]);
    ensures \result ≥ -11.1111111111 ∧ \result ≤ 111.1111111111;
    assigns \nothing;
*/
double calculate_pedal_position(double raw_position) {
    double pedal_position;
    double dead_zone;  // A deadzone at the begining of the pedal
                        // position to avoid unintended acceleration.
    dead_zone = Read_tF64_param(ACCPED_DEAD_ZONE_PARAM);
    if (raw_position < dead_zone) {
        pedal_position = 0.0;
    }
    else {
        pedal_position = ((raw_position - dead_zone) / (100.0 - dead_zone)) * 100.0f;
    }
    return pedal_position;
}

/*@ behavior interface_spec:
    assumes \true;
    requires valid(DB_ENABLED_MODULES[ACCELERATOR_PEDAL]);
    requires valid(DB_TF64_SIGNALS[ACCPED_POSITION]);
    requires valid(DB_TF64_SIGNALS[ACCPED_SENSOR_E]);
    requires valid(DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM]);
    requires valid(DB_TF64_PARAMS[ACCPED_SENSOR_MAX_VALUE]);
    requires DB_ENABLED_MODULES[ACCELERATOR_PEDAL] \in \{TRUE, FALSE\};
    requires DB_TF64_SIGNALS[ACCPED_POSITION] >= 0.0 ∧
    DB_TF64_SIGNALS[ACCPED_POSITION] <= 100.0;
    requires DB_TF64_SIGNALS[ACCPED_SENSOR_E] >= 0.0 ∧
    DB_TF64_SIGNALS[ACCPED_SENSOR_E] <= 24.5;
void Accped_update_5ms() {
  typedef unsigned char tB; // Boolean value indicating if module is activated.
 +#define TRUE ((tB)1)
  +#define FALSE ((tB)0)
  tB module_enabled; // Sensore max value in volts.
  double sensor_max_value; // Sensor value in volts.
  double sensor_reading; // Pedal raw position excluding the deadzone
  double pedal_raw_position; // Pedal position including the deadzone
  double pedal_position; // in percent.

  module_enabled = Is_module_enabled(ACCELERATOR_PEDAL);
  if (module_enabled) {
    sensor_reading = Read_tF64_signal(ACCPED_SENSOR_E);
    sensor_max_value = Read_tF64_param(ACCPED_SENSOR_MAX_VALUE);
    pedal_raw_position = (sensor_reading / sensor_max_value) * 100.0f;
    pedal_position = calculate_pedal_position(pedal_raw_position);
    Write_tF64_signal(ACCPED_POSITION, pedal_position);
  } else {
    // Do nothing.
  }
}

B.8 Simplified accelerator pedal module with functional specifications

typedef unsigned char tB;
#define TRUE ((tB)1)
#define FALSE ((tB)0)
typedef enum {

Listing B.7: The ISP output for the simplified accelerator pedal module that includes only interface specifications.
ACCPED_DEAD_ZONE_PARAM,
ACCPED_SENSOR_MAX_VALUE,
NUMBER_OF_PARAMS
} tF64_Params_E;

typedef enum {
    STEERING_M,
    BRAKE_PEDAL_M,
    ACCELERATOR_PEDAL,
    CLUTCH_PEDAL,
    LIGHTING_M,
    NUMBER_OF_MODULES
} Modules_E;

typedef enum {
    BRAKEPED_SENSOR_E,
    ACCPED_SENSOR_E,
    ACCPED_POSITION,
    STEERING_SENSOR_E,
    CLUTCH_SENSOR_E,
    NUMBER_OF_SIGNALS
} tF64_Signals_E;

double DB_TF64_PARAMS[NUMBER_OF_PARAMS];
tB DB_ENABLED_MODULES[NUMBER_OF_MODULES];
double DB_TF64_SIGNALS[NUMBER_OF_SIGNALS];
//@ ghost double model_pedal_raw_position;

double Read_tF64_param(tF64_Params_E tf64_param_E) {
    return DB_TF64_PARAMS[tf64_param_E];
}

tB Is_module_enabled(Modules_E module_E) {
    return DB.Enabled_MODULES[module_E];
}

double Read_tF64_signal(tF64_Signals_E tf64_signal_E) {
    return DB_TF64_SIGNALS[tf64_signal_E];
}

void Write_tF64_signal(tF64_Signals_E tf64_signal_E, double value) {
    DB_TF64_SIGNALS[tf64_signal_E] = value;
}

double calculate_pedal_position(double raw_position) {
    double pedal_position;
    double dead_zone; // A deadzone at the begining of the pedal
    // position to avoid unintended acceleration.

    dead_zone = Read_tF64_param(ACCPED_DEAD_ZONE_PARAM);

    if (raw_position < dead_zone) {
        pedal_position = 0.0;
    } else {
        pedal_position =
            ((raw_position - dead_zone) / (100.0f - dead_zone)) * 100.0f;
    }
Appendix B: C Source Code | 105
Listing B.8: The simplified accelerator pedal module that includes only functional specifications.

B.9 Corrected ISP output for simplified accelerator pedal module
39 requires DB_TF64_PARAMS[1] ≡ 24.5;
40 requires \valid_read((double *)DB_TF64_PARAMS);
41 requires \valid_read(&DB_TF64_PARAMS[1]);
42 //ensures \result ≥ 0.0 ∧ \result ≤ 24.5;
43 ensures tf64_param_E == ACCPED_DEAD_ZONE_PARAM ==> \result ==
44 DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM];
45 ensures tf64_param_E == ACCPED_SENSOR_MAX_VALUE ==> \result ==
46 DB_TF64_PARAMS[ACCPED_SENSOR_MAX_VALUE];
47 assigns \nothing;
48 */
49 double Read_tF64_param(tF64_Params_E tf64_param_E) {
50 return DB_TF64_PARAMS[tf64_param_E];
51 }
52
53 /*@ behavior isp_generated:
54 assumes \true;
55 requires module_E == ACCELERATOR_PEDAL;
56 requires DB_ENABLED_MODULES[2] ∈ {0, 1};
57 requires \valid_read(&DB_ENABLED_MODULES[2]);
58 ensures \result ∈ {0, 1};
59 assigns \nothing;
60 */
61 tB Is_module_enabled(Modules_E module_E) {
62 return DB_ENABLED_MODULES[module_E];
63 }
64
65 /*@ behavior isp_generated:
66 assumes \true;
67 requires tf64_signal_E == ACCPED_SENSOR_E;
68 requires DB_TF64_SIGNALS[1] ≥ -0. ∧ DB_TF64_SIGNALS[1] ≤ 24.5;
69 requires \valid_read(\DB_TF64_SIGNALS[1]);
70 ensures \result ≥ -0. ∧ \result ≤ 24.5;
71 assigns \nothing;
72 */
73 double Read_tF64_signal(tF64_Signals_E tf64_signal_E) {
74 return DB_TF64_SIGNALS[tf64_signal_E];
75 }
76
77 /*@ behavior isp_generated:
78 assumes \true;
79 requires tf64_signal_E == ACCPED_POSITION;
80 requires value ≥ 0.0 ∧ value ≤ 100.0;
81 requires \valid_read(\DB_TF64_SIGNALS[2]);
82 ensures DB_TF64_SIGNALS[2] ≥ 0.0 ∧ DB_TF64_SIGNALS[2] ≤ 100.0;
83 assigns DB_TF64_SIGNALS[2];
84 */
85 void Write_tF64_signal(tF64_Signals_E tf64_signal_E, double value) {
86 DB_TF64_SIGNALS[tf64_signal_E] = value;
87 }
88
89 /*@ behavior isp_generated:
90 assumes \true;
91 requires DB_TF64_PARAMS[0] >= 0.0 &&
92 DB_TF64_PARAMS[0] ≤ 10.0;
93 requires DB_TF64_PARAMS[1] ≡ 24.5;
94 requires raw_position ≥ -0. ∧ raw_position ≤ 100.;
95 requires \valid_read((double *)DB_TF64_PARAMS);
96 requires \valid_read(\DB_TF64_PARAMS[1]);
ensures \result \geq 0.0 \land \result \leq 100.0;
assigns \nothing;
*/

double calculate_pedal_position(double raw_position) {
    double pedal_position;
    double dead_zone; // A deadzone at the begining of the pedal
    // position to avoid unintended acceleration.

    dead_zone = Read_tF64_param(ACCPED_DEAD_ZONE_PARAM);

    if (raw_position < dead_zone) {
        pedal_position = 0.0;
    } else {
        pedal_position = ((raw_position - dead_zone) / (100.0 - dead_zone)) * 100.0f;
    }

    return pedal_position;
}

/*@ behavior interface_spec:
assumes \true;
requires \valid_read(&DB_ENABLED_MODULES[ACCELERATOR_PEDAL]);
requires \valid(&DB_TF64_SIGNALS[ACCPED_POSITION]);
requires \valid_read(&DB_TF64_SIGNALS[ACCPED_SENSOR_E]);
requires \valid_read(&DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM]);
requires \valid_read(&DB_TF64_PARAMS[ACCPED_SENSOR_MAX_VALUE]);
requires DB_ENABLED_MODULES[ACCELERATOR_PEDAL] \in \{TRUE, FALSE\};
requires DB_TF64_SIGNALS[ACCPED_POSITION] >= 0.0 &&
DB_TF64_SIGNALS[ACCPED_POSITION] <= 100.0;
requires DB_TF64_SIGNALS[ACCPED_SENSOR_E] >= 24.5;
requires DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM] >= 0.0 &&
DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM] <= 10.0;
requires DB_TF64_PARAMS[ACCPED_SENSOR_MAX_VALUE] == 24.5;
ensures DB_ENABLED_MODULES[ACCELERATOR_PEDAL] \in \{TRUE, FALSE\};
ensures DB_TF64_SIGNALS[ACCPED_POSITION] >= 0.0 &&
DB_TF64_SIGNALS[ACCPED_POSITION] <= 100.0;
ensures DB_TF64_SIGNALS[ACCPED_SENSOR_E] >= 24.5;
ensures DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM] >= 0.0 &&
DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM] <= 10.0;
ensures DB_TF64_PARAMS[ACCPED_SENSOR_MAX_VALUE] == 24.5;
assigns DB_TF64_SIGNALS[ACCPED_POSITION];
*/

void Accped_update_5ms() {
    tB module_enabled; // Boolean value indicating if module
    // is activated.
    double sensor_max_value; // Sensor max value in volts.
    double sensor_reading; // Sensor value in volts.
    double pedal_raw_position; // Pedal raw position in percent.
    double pedal_position; // Pedal position excluding the deadzone
    // in percent.

    module_enabled = Is_module_enabled(ACCELERATOR_PEDAL);
if (module_enabled) {
    sensor_reading = Read_tF64_signal(ACCPED_SENSOR_E);
    sensor_max_value = Read_tF64_param(ACCPED_SENSOR_MAX_VALUE);
    pedal_raw_position = (sensor_reading / sensor_max_value) * 100.0f;
    // assert pedal_raw_position >= 0.0 && pedal_raw_position <= 100.0;
    pedal_position = calculate_pedal_position(pedal_raw_position);
    Write_tF64_signal(ACCPED_POSITION, pedal_position);
} else {
    // Do nothing.
}

Listing B.9: Manually corrected ISP output for the simplified accelerator pedal module that includes only interface specifications.

B.10 Corrected ISP output for simplified accelerator pedal module with functional specifications

typedef unsigned char tB;
#define TRUE ((tB)1)
#define FALSE ((tB)0)
typedef enum {
    ACCPED_DEAD_ZONE_PARAM,
    ACCPED_SENSOR_MAX_VALUE,
    NUMBER_OF_PARAMS
} tF64_Params_E;
typedef enum {
    STEERING_M,
    BRAKE_PEDAL_M,
    ACCELERATOR_PEDAL,
    CLUTCH_PEDAL,
    LIGHTING_M,
    NUMBER_OF_MODULES
} Modules_E;
typedef enum {
    BRAKEPED_SENSOR_E,
    ACCPED_SENSOR_E,
    ACCPED_POSITION,
    STEERING_SENSOR_E,
    CLUTCH_SENSOR_E,
    NUMBER_OF_SIGNALS
}
//@ ghost double model_pedal_raw_position;

/*@ behavior isp_generated:
  assumes \true;
  requires tf64_param_E \in \{ACCPED_DEAD_ZONE_PARAM, ACCPED_SENSOR_MAX_VALUE\};
  requires DB_TF64_PARAMS[0] \geq 0.0 &&
    DB_TF64_PARAMS[0] \leq 10.0;
  requires DB_TF64_PARAMS[1] \equiv 24.5;
  requires \valid_read((double *)DB_TF64_PARAMS);
  requires \valid_read(&DB_TF64_PARAMS[1]);
  //ensures \result \geq 0.0 \land \result \leq 24.5;
  ensures tf64_param_E == ACCPED_DEAD_ZONE_PARAM ==> \result == DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM];
  ensures tf64_param_E == ACCPED_SENSOR_MAX_VALUE ==> \result == DB_TF64_PARAMS[ACCPED_SENSOR_MAX_VALUE];
  assigns \nothing;
  */
double Read_tF64_param(tF64_Params_E tf64_param_E) {
  return DB_TF64_PARAMS[tf64_param_E];
}

/*@ behavior isp_generated:
  assumes \true;
  requires module_E == ACCELERATOR_PEDAL;
  requires DB_ENABLED_MODULES[2] \in \{0, 1\};
  requires \valid_read(DB_ENABLED_MODULES[2]);
  ensures \result \in \{0, 1\};
  assigns \nothing;
  */
tB Is_module_enabled(Modules_E module_E) {
  return DB_ENABLED_MODULES[module_E];
}

/*@ behavior isp_generated:
  assumes \true;
  requires tf64_signal_E == ACCPED_SENSOR_E;
  requires DB_TF64_SIGNALS[1] \geq -0.0 \land DB_TF64_SIGNALS[1] \leq 24.5;
  requires \valid_read(DB_TF64_SIGNALS[1]);
  ensures \result \geq -0.0 \land \result \leq 24.5;
  assigns \nothing;
  */
double Read_tF64_signal(tF64_Signals_E tf64_signal_E) {
  return DB_TF64_SIGNALS[tf64_signal_E];
}

/*@ behavior isp_generated:
  assumes \true;
  requires tf64_signal_E == ACCPED_POSITION;
  requires value \geq 0.0 \land value \leq 100.0;
  requires \valid_read(DB_TF64_SIGNALS[2]);
  requires \valid_read(DB_TF64_SIGNALS[2]);
  ensures DB_TF64_SIGNALS[2] \geq 0.0 \land DB_TF64_SIGNALS[2] \leq 100.0;
```c
void Write_tF64_signal(tF64_Signals_E tf64_signal_E, double value) {
    DB_TF64_SIGNALS[tf64_signal_E] = value;
}

/*@ behavior isp_generated:
  assumes \true;
  requires DB_TF64_PARAMS[0] >= 0.0 &&
  DB_TF64_PARAMS[0] <= 10.0;
  requires DB_TF64_PARAMS[1] == 24.5;
  requires \valid_read((double *)DB_TF64_PARAMS);
  requires \valid_read(&DB_TF64_PARAMS[1]);
  ensures \result >= 0.0 && \result <= 100.0;
  assigns \nothing;
*/

double calculate_pedal_position(double raw_position) {
    double pedal_position;
    double dead_zone; // A deadzone at the begining of the pedal
    // position to avoid unintended acceleration.
    dead_zone = Read_tF64_param(ACCPED_DEAD_ZONE_PARAM);
    if (raw_position < dead_zone) {
        pedal_position = 0.0;
    } else {
        pedal_position = ((raw_position - dead_zone) / (100.0f - dead_zone)) * 100.0f;
    }
    return pedal_position;
}

/*@ behavior interface_spec:
  assumes \true;
  requires \valid(&model_pedal_raw_position);
  requires \valid_read(&DB_ENABLED_MODULES[ACCELERATOR_PEDAL]);
  requires \valid_read(&DB_TF64_SIGNALS[ACCPED_POSITION]);
  requires \valid_read(&DB_TF64_SIGNALS[ACCPED_SENSOR_E]);
  requires \valid_read(&DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM]);
  requires \valid_read(&DB_TF64_PARAMS[ACCPED_SENSOR_MAX_VALUE]);
  requires DB_ENABLED_MODULES[ACCELERATOR_PEDAL] \in \{TRUE, FALSE\};
  requires DB_TF64_SIGNALS[ACCPED_POSITION] >= 0.0 &&
  DB_TF64_SIGNALS[ACCPED_POSITION] <= 100.0;
  requires DB_TF64_SIGNALS[ACCPED_SENSOR_E] >= 0.0 &&
  DB_TF64_SIGNALS[ACCPED_SENSOR_E] <= 24.5;
  requires DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM] >= 0.0 &&
  DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM] <= 10.0;
  requires DB_TF64_PARAMS[ACCPED_SENSOR_MAX_VALUE] == 24.5;
  ensures DB_ENABLED_MODULES[ACCELERATOR_PEDAL] \in \{TRUE, FALSE\};
  ensures DB_TF64_SIGNALS[ACCPED_POSITION] >= 0.0 &&
  DB_TF64_SIGNALS[ACCPED_POSITION] <= 100.0;
  ensures DB_TF64_SIGNALS[ACCPED_SENSOR_E] >= 0.0 &&
  DB_TF64_SIGNALS[ACCPED_SENSOR_E] <= 24.5;
  ensures DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM] >= 0.0 &&
```
DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM] <= 10.0;
ensures DB_TF64_PARAMS[ACCPED_SENSOR_MAX_VALUE] == 24.5;

assigns DB_TF64_SIGNALS[ACCPED_POSITION],
model_pedal_raw_position;

behavior functional_spec:
assumes \true;
requires \valid(&model_pedal_raw_position);
requires \valid_read(&model_pedal_raw_position);
requires \valid_read(&DB_ENABLED_MODULES[ACCELERATOR_PEDAL]);
requires \valid(&DB_TF64_SIGNALS[ACCPED_POSITION]);
requires \valid_read(&DB_TF64_SIGNALS[ACCPED_POSITION]);
requires \valid_read(&DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM]);
requires \valid_read(&DB_TF64_PARAMS[ACCPED_SENSOR_MAX_VALUE]);

// Requirement 1
ensures model_pedal_raw_position >= 0.0 &&
model_pedal_raw_position <= 100.0;

// Requirement 2
ensures model_pedal_raw_position ==
(DB_TF64_SIGNALS[ACCPED_SENSOR_E] /
DB_TF64_PARAMS[ACCPED_SENSOR_MAX_VALUE]) * 100.0;

// Requirement 3
ensures DB_TF64_SIGNALS[ACCPED_POSITION] >= 0.0 &&
DB_TF64_SIGNALS[ACCPED_POSITION] <= 100.0;

// Requirement 4
ensures model_pedal_raw_position < DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM] ==>
DB_TF64_SIGNALS[ACCPED_POSITION] == 0.0;
ensures model_pedal_raw_position >= DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM] ==>
DB_TF64_SIGNALS[ACCPED_POSITION] == 100 *
\{
(model_pedal_raw_position - DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM]) /
(100.0 - DB_TF64_PARAMS[ACCPED_DEAD_ZONE_PARAM])\};
assigns DB_TF64_SIGNALS[ACCPED_POSITION],
model_pedal_raw_position;

*/
void Accped_update_5ms() {
    tB module_enabled; // Boolean value indicating if module
    // is activated.
    double sensor_max_value; // Sensor max value in volts.
    double sensor_reading; // Sensor value in volts.
    double pedal_raw_position; // Pedal raw position in percent.
    double pedal_position; // Pedal position excluding the deadzone
    // in percent.

    module_enabled = Is_module_enabled(ACCELERATOR_PEDAL);

    if (module_enabled) {
        sensor_reading = Read_tF64_signal(ACCPED_SENSOR_E);
        sensor_max_value = Read_tF64_param(ACCPED_SENSOR_MAX_VALUE);

        pedal_raw_position = (sensor_reading / sensor_max_value) * 100.0f;
Listing B.10: Manually corrected ISP output for the simplified accelerator pedal module that includes both interface and functional specifications.

B.11 Just for testing KTH colors

You have selected to optimize for print output

- Primary color
  - kth-blue
  - kth-blue80

- Secondary colors
  - kth-lightblue
  - kth-lightred
  - kth-lightred80
  - kth-lightgreen
  - kth-coolgray
  - kth-coolgray80

black
For DIVA

Join the community and help improve this document!

{ "Author": { "Last name": "Manjikian", "First name": "Hovig", "Local User Id": "ujynzdr", "E-mail": "hoftime@kth.se", "organisation": { "L1": "School of Electrical Engineering and Computer Science", } }, "Degree": { "Educational program": "Degree Programme in Computer Science and Engineering", "programcode": "CDATE", "Degree": "Degree of Master of Science in Engineering", "subjectArea": "Computer Science and Engineering" } }, "Title": { "Main title": "Improving the Synthesis of Annotations for Partially Automated Deductive Verification", "Language": "eng" }, "Alternative title": { "Main title": "Att förbättra syntes av funktionsanteckningar för partiellt automatiserad deduktiv verifiering", "Language": "swe" }, "Supervisor1": { "Last name": "Lidström", "First name": "Christian", "Local User Id": "u1tvsmx", "E-mail": "clid@kth.se", "organisation": { "L1": "School of Electrical Engineering and Computer Science", "L2": "Computer Science" } }, "Examiner1": { "Last name": "Artho", "First name": "Cyrille", "Local User Id": "u1d35c2", "E-mail": "artho@kth.se", "organisation": { "L1": "School of Electrical Engineering and Computer Science", "L2": "Computer Science" } }, "Cooperation": { "Partner_name": "Scania CV AB", "National Subject Categories": "10201, 10205, 10206", "Other information": { "Year": "2023", "Number of pages": "2xvi,113" }, "Series": { "Title of series": "TRITA-EECS-EX", "No. in series": "2022:00" }, "Opponents": { "Name": { "A. B. Normal & A. X. E. Normalè" } }, "Presentation": { "Date": "2022-03-15 13:00" }, "Language": "eng", "Room": "via Zoom https://kth-se.zoom.us/j/ddddddddddd", "Address": "Isafjordsgatan 22 (Kistagången 16)", "City": "Stockholm", "Number of lang instances": "2", "Abstract[eng]": "This work investigates possible improvements to an existing annotation inference tool. The tool is part of a chain that aims to automate the process of software verification using formal methods. The purpose of the annotations is to facilitate the use of deductive verification, which is the formal method used in this project for proving that a given program meets its specifications. In the project, two categories of annotations are established. The first category is the category of functional annotations. These annotations describe the behavior of a function or a module. The other category is what we call auxiliary annotations. These annotations describe properties that are necessary for proving the correctness of the functional annotations. The tool that this work tries to improve is dedicated to inferring the auxiliary annotations. To our knowledge, this is the first tool of its kind to automatically infer auxiliary annotations for a complete module given the module's source code and its interface specifications. The work contributed in four areas: inferring annotations from the interface specifications of a module and propagating these annotations to all the helper functions used in the module; inferring annotations for floating-point constructs; inferring annotations for pointer constructs; and finally, inferring annotations for array constructs. The improved tool was tested on production embedded code used in the heavy automotive industry. The results demonstrated a considerable improvement and were in line with earlier findings. The work confirms the feasibility and usability of auxiliary annotation inference in this scope.

Keywords[eng]: Formal verification, Automated verification, Contract inference.

Abstract[eng]: This work investigates possible improvements to an existing annotation inference tool. The tool is part of a chain that aims to automate the process of software verification using formal methods. The purpose of the annotations is to facilitate the use of deductive verification, which is the formal method used in this project for proving that a given program meets its specifications. In the project, two categories of annotations are established. The first category is the category of functional annotations. These annotations describe the behavior of a function or a module. The other category is what we call auxiliary annotations. These annotations describe properties that are necessary for proving the correctness of the functional annotations. The tool that this work tries to improve is dedicated to inferring the auxiliary annotations. To our knowledge, this is the first tool of its kind to automatically infer auxiliary annotations for a complete module given the module's source code and its interface specifications. The work contributed in four areas: inferring annotations from the interface specifications of a module and propagating these annotations to all the helper functions used in the module; inferring annotations for floating-point constructs; inferring annotations for pointer constructs; and finally, inferring annotations for array constructs. The improved tool was tested on production embedded code used in the heavy automotive industry. The results demonstrated a considerable improvement and were in line with earlier findings. The work confirms the feasibility and usability of auxiliary annotation inference in this scope.

Keywords[swe]: Formalverifiering, Automatisk verifiering, Contrakt-inference.


Arbetet bidrog inom fyra områden: att härdla annoteringar från gränssnittsspecifikationerna (interface specifications) för en modul och sprida dessa annoteringar till alla hjälpfunktioner som används i modulen; härdla annoteringar för flyttalgskonstruktioner (floating-point constructs); härdla annoteringar för pekarkonstruktioner; och slutligen, härdla annoteringar för arraykonstruktioner. Det förbättrade verktyget testades på produktionsnyttgjord kod som används inom fordonsindustrin. Resultatet visade en avsevärd förbättring och var i linje med tidigare resultat. Arbetet bekräftar genomförbarheten och användbarheten av hjälpannoteringen häradling i projektets omfattning.

Keywords[swe]: bildning, bildning, bildning.
Formell verifiering, Automatiserad verifiering, Kontraktgenerering.