Automatic Verification of Embedded Systems Using Horn Clause Solvers

Anoud Alshnakat
Abstract

Automatic Verification of Embedded Systems Using Horn Clause Solvers

Anoud Alshnakat

Recently, an increase in the use of safety-critical embedded systems in the automotive industry has led to a drastic up-tick in vehicle software and code complexity. Failure in safety-critical applications can cost lives and money. With the ongoing development of complex vehicle software, it is important to assess and prove the correctness of safety properties using verification and validation methods.

Software verification is used to guarantee software safety, and a popular approach within this field is called deductive static methods. This methodology has expressive annotations, and tends to be feasible even on larger software projects, but developing the necessary code annotations is complicated and costly. Software-based model checking techniques require less work but are generally more restricted with respect to expressiveness and software size.

This thesis is divided into two parts. The first part, related to a previous study where the authors verified an electronic control unit using deductive verification techniques. We continued from this study by investigating the possibility of using two novel tools developed in the context of software-based model checking: SeaHorn and Eldarica. The second part investigated the concept of automatically inferring code annotations from model checkers to use them in deductive verification tools. The chosen contract format was ACSL, derived from Eldarica and used later in Frama-C.

The results of the first part showed that the success of verification was tool-dependent. SeaHorn could verify most of the safety functional requirements and reached a higher level of abstraction than Eldarica. The comparison between this thesis and the related work showed that model checkers are faster, scaled to the code size, required less manual work, and less code overhead compared to deductive verification tools.

The second part of the thesis was implemented as an extension of TriCera, using Scala. The ACSL annotations were tested with Frama-C using verification tasks from SV-COMP benchmark, and the results showed that it was possible to generate adequate contracts for simple functions.

This thesis was conducted at Scania CV AB in Södertälje, Sweden.
Acknowledgements

I would like to express my deepest gratitude to my supervisors Philipp Rümmer and Christian Lidström for their patient guidance, valuable discussions and useful critique of this thesis. My earnest thanks to Mohamed Faouzi Atig, my reviewer, for providing his support and valuable comments which led to a better quality of the report.

Finally, I wish to thank my family, and especially my parents, for their enthusiastic encouragement throughout my studies.
# Contents

1 Introduction .................................................. 1
  1.1 Motivation .................................................. 1
  1.2 Industrial Software Verification .......................... 2
  1.3 Thesis Goals ............................................... 4

2 Background .................................................... 6
  2.1 Requirements and Specifications ......................... 6
  2.2 Verification and Validation ............................... 7
    2.2.1 Verification Objectives ............................ 8
    2.2.2 Verification Categories ............................ 8
  2.3 Deductive Verification .................................... 9
    2.3.1 Frama-C .............................................. 11
    2.3.2 VCC .................................................. 12
  2.4 Software Model Checking ................................. 13
    2.4.1 Eldarica ............................................. 14
    2.4.2 SeaHorn .............................................. 17

3 Related Work .................................................. 19

4 System of Interest: Steering ECU .......................... 22
  4.1 Abstract Module Architecture ............................ 22
  4.2 MISRA-C Code Correspondence ........................... 23
    4.2.1 Function-Call Hierarchy ............................ 24
# Verification Process Overview

## Verification Process

6.1 Case Study Setup

6.1.1 Default Arguments for Eldarica and SeaHorn

6.1.2 Preprocessed Code

6.2 Mapping Requirements to the Code

6.3 From Semi-Formal to First-Order Logic

6.4 Unit Verification Harness

6.5 Integration with Database Verification

6.5.1 Input to Output Using Functional Substitution

6.6 Violation Handling Techniques

6.7 Mutation Testing and Sanity Check

## Empirical Results of Verification

7.1 Results for Eldarica

7.2 Results for SeaHorn

7.2.1 Single Requirement Verification Results

7.2.2 Unit Verification Results

7.2.3 Integration Verification Results

7.2.4 I/O Verification Results

7.3 Mutation testing and Sanity Check Outcomes

7.3.1 Mutation Test Results
Chapter 1

Introduction

Currently, the automotive industry systems are becoming intricate, imposing the necessity to confirm the safety and surety of the whole vehicle, to prevent the economical and human costs from system failures. In this chapter, the motivation, problem formalisation and the thesis goals are discussed.

1.1 Motivation

The automotive industry is constantly aiming to change the mechanical systems to electronic ones, where the future vision for most of them is to reach an autonomous fully automated driving system. SAE International classified the driving automation to six well-defined technical levels rather than normative and legal [1]:

- Level 0 (No Automation): Human handles all aspects of the dynamic driving, such as, steering, braking and monitoring the vehicle.
- Level 1 (Driver Assistance): It subsumes Level 0 in addition to a driver assistance system of either steering or acceleration/deceleration, where the human is expected to handle the rest of the dynamic driving tasks.
- Level 2 (Partial Automation): It subsumes Level 0 in addition to one or more driver assistance systems of both steering and acceleration/deceleration, where the human is expected to handle the rest of the dynamic driving tasks.
- Level 3 (Conditional Automation): All aspects of dynamic driving are handled by an automated driving system; the human is expected to respond appropriately to intervention requests or feedback.
• Level 4 (High Automation): All aspects of dynamic driving are handled by an automated driving system even if the human does not respond to intervention in requests or feedbacks.

• Level 5 (Full Automation): All aspects of the dynamic driving tasks are handled by a full-time automated system, under all roadway and environmental conditions that manageable by a human driver.

SAE level classification shows that the human effort decreases whenever the automation level increases in automotive, resulting in an increase of embedded system units within the vehicles. This, in turn, leads to more software code, increasing the complexity of the vehicle system; today, modern vehicles may have approximately 100 million lines of code [2]. Size and complexity of the code also make it prone to errors, possibly introducing faults in the embedded system units, which can create a failure in the vehicle.

Vehicle failure costs lives or can cause a serious injury, such a system is called safety-critical or life-critical system. In order to decrease the cost of such a failure, the safety-critical system design should adhere to adequate and appropriate requirements and specifications. A safety-critical system must also be verified to either prove its correctness with respect to its specification, or prove that it is bug-free [3].

1.2 Industrial Software Verification

Proving program correctness is a non-trivial task, as it may have many aspects that make the process complicated for the developers to perform.

Requirements are usually vague and informal, but there are a few standards that clarify the confusion that might occur, such as the IEEE Recommended Practice for Software Requirements Specifications. There are also more specific standards for vehicle functional safety features, such as the ISO 26262 standard requirements, that apply to road vehicles. ISO 26262 discusses hazards management caused by malfunctioning behaviour of safety-related electronic or electronic systems and the interaction between them [4].
To validate the requirements, it is expected from the developers to interpret the specifications and translate them to logical formulas. The requirements, therefore, must be well written and unambiguous in order to eliminate any misconceptions that might arise in the verification procedure. Requirement interpretation might also differ from person to person. A simple example for illustration:

“It is not the case that the engine is in state ON and brake\_switch is in state SET if sensor B428 is sending ERROR values.”

The statement “it is not the case” can create confusion as it can be interpreted as a negation that includes both propositions status of the engine and the brake\_switch:

\[ B428 == ERROR \implies !(\text{engine} == \text{ON} && \text{brake\_switch} == \text{SET}) \]

It also might be interpreted to include the negation of only the state of the engine, excluding the brake\_switch:

\[ B428 == ERROR \implies (!\text{engine} == \text{ON} && \text{brake\_switch} == \text{SET}) \]

The developer must have an excellent command of mathematical formal logic, and must also be a domain experts in order to translate the informal sentences. The developers must reason about the predicates as accurately as possible, especially when it is applied to complex and sophisticated embedded systems that interact with system busses or other sensor units.

To guarantee that the requirements hold while considering all input values for industrial software, they should be verified statically or dynamically. One of the common ways to prove correctness in the safety-critical systems is using deductive verification tools; they usually achieve a high level of certainty and reliability.

Deductive verification tools analyse all possible inputs of a program, therefore they add further confidence to the system, but are quite troublesome to use, utilise and adapt to. The most challenging issue that faces developers when using deductive verification tools is that they demand an excessive manual effort to identify and develop the verification annotations, which might slow down the verification phase in the software development cycle and increase the overall financial cost.

On the other hand, software model checking can formally prove the correctness
of the program automatically. The concept of automation means it does not demand additional annotations to be specified, so it is not complicated to use. This makes the verification experience focused on correctly translating the requirements only, rather than adding the overhead of figuring out the missing annotations or assumptions that deductive verification requires. Model checking techniques have the possibility of returning a counter-example, making error tracing faster and more efficient.

One of the limitations that face model checking is that it is prone to the state-space explosion. Model checking uses finite state models to represent the program, it also uses algorithms that exhaustively search the entire state-space to determine whether the software satisfies the specifications. In complex systems, the number of states per process can grow exponentially, known as state-space explosion problem [5]. State-space explosion leads to prohibitively long run-times and exceeding the amount of the available memory [6].

1.3 Thesis Goals

The thesis is highly inspired by a previous study “Deductive Functional Verification of Safety-Critical Embedded C-Code: An Experience Report” [7] that was conducted at Scania, where the authors carried out an experiment that aimed to apply deductive verification methods on an embedded system software written in MISRA-C, using VCC (Verifier for Concurrent C) [8][7].

This thesis will address a case study which investigates the possibility of automating the verification of real-life embedded systems, and verify the specifications imposed on industrial scale safety-critical software using software model checking techniques, instead of deductive verification. The overall goal of this thesis is to explore approaches to automatically infer code annotations, using novel methods developed in the context of software-based model checking, in order to reduce the manual work required for deductive verification. The case study will be carried out at Scania, a heavy vehicle manufacturing company.

This thesis is a continuation work of [7], where the analysed and examined embedded module is the same, therefore this case study is limited solely to it. For the software model checking verification part, two academic tools were chosen, SeaHorn [9] and Eldarica [10]. Both tools are functioning as software model checkers
based on Horn solvers in [11] and [12].

The goal of the thesis can be partitioned into three objectives, each one shall be implemented individually while following a suitable time plan:

1. Translate the case study from [7], and in particular the formalised requirements of the case study, to a form that can be processed and accepted by the state-of-the-art model checkers; SeaHorn and Eldarica.

2. Investigate whether model checkers are able to verify and prove the correctness of the case study, without providing any code annotations (or adding any extra lines of codes), in addition to the translated requirements from goal 1. Alternatively, investigate how many of the code annotations from the study can be removed.

3. Extend the model checkers to output computed code annotations [7] in a format that can be fed back to the Frama-C system, this way obtaining a complete tool-chain to automate Frama-C based deductive verification.
Chapter 2

Background

This chapter starts with a brief description of requirements and specifications definitions. Then it explains the overall differences between validation and verification. The scope will be focused on verification, listing the analysis methods, concise theories and software tools used in this thesis.

2.1 Requirements and Specifications

Most industrial software systems have a set of requirements which are conditions and capabilities that determine the behaviour of the resulting system, and what the product is expected to do. Requirements are extracted from a series of discussions between the stakeholders, i.e., the clients, end users, and system developers. The requirements are usually collected via use cases or documented statements obtained from the client, and they can be functional or non-functional. Functional requirements list the features, functionality, ability and security of the software product. The requirements can also be non-functional in the sense that they describe the overall attributes and interactions of the software product with the environment, in various ways, such as usability needs.

The number of requirements can grow dramatically even considering simple small systems, so it is useful to categorise them with the FURPS+ Requirement Model. FURPS+ was created by Robert Grady [13], and it stands for Functionality, Usability, Reliability, Performance and Support. The ‘+’ means other checkpoints that can be included, such as design constraints, implementation requirements, physical requirements, etc. The requirements will later be analysed with technical details and documented, after which they are referred to as specifications.

ASTM (American Society for Testing and Materials) International standards
defines the specifications as an explicit set of requirements to be satisfied by a material, product, system, or service [14]. Specifications can be considered as a connection layer located between the requirements and software engineering since they are written for the software developers. It is necessary to understand the requirements, because it will assist the developers to build the end product as accurate as possible with respect to the customer’s needs.

2.2 Verification and Validation

Verification and validation are terms commonly used interchangeably, however, they can described them as follows:

The validation procedure is targeted towards evaluating the final software product and to inspect whether it associates with the client’s stated demands and expectations. During the validation procedure, the actual system will be tested, which can be considered a high-level process since all the code will be evaluated.

The verification procedure is targeted towards evaluating the software program’s internal functions, design and documents with respect to the technical specifications. It provides an indication of the quality of the software and its correctness rather than the correctness of the system as a whole. During the verification, the software units will be evaluated to judge whether they meet the specifications, which can be considered a low-level process compared to validation. Barry W. Boehm defined the verification and validation processes [15]:

- “Verification - to establish the truth of the correspondence between a software product and its specification. Am I building the product right?”

- “Validation - to establish the fitness or worth of a software product for its operational mission. Am I building the right product?”

In other words, verification precedes validation, because the hardware and software that are related to the product will be built and developed first and then verified, based on the specifications. Afterwards, the completed product will be validated, based on the requirements.
2.2.1 Verification Objectives

The main purpose of verification is to prove program correctness. It can emphasise and add to the quality of the software product. It can also be useful in detecting program defects in the early stages of development rather than the later ones, reducing the cost of accumulative errors and faults. As Barry W. Boehm describes it: verification and validation are not only a way to save the cost, there are also clear payoffs in improving reliability, maintainability, and human engineering of the resulting software product [15].

2.2.2 Verification Categories

The verification categories vary between simple and sophisticated techniques. They can be straightforward, such as the manual effort to check the software, or they can be more systematic, such as dynamic or static analysis.

Dynamic Analysis

The dynamic analysis approach requires the software program to be concretely executed, to acquire the dynamic behaviour of the code of the software. The methodologies under this category are testing and simulation. Testing is considered the standard evaluation method, where the test engineers need to apply either white box or black box testing suits. These will follow one or more of the coverage criteria, for instance, statement coverage, control flow-graph coverage or branch coverage [16]. The test case contains the initialisation of the test suite, the invocation of the function that should be verified, and the oracle that returns true if the test succeeds.

Static Analysis

The static analysis approach does not require the program to be executed, meaning that the static analysis assumes that the compiler, the operating system and the hardware are working correctly [16]. The static analysis uses mathematical proofs along with all the possible inputs in order to prove the correctness of the program.
There are multiple techniques that comply with static analysis, for instance, deductive verification, abstract interpretation and model checking techniques [17][18]. Those techniques are an adequate path to prove correctness and/or proves that no bugs are left in the software.

### 2.3 Deductive Verification

Deductive verification is a manual formal technique and can be considered static because the code will not be executed. It adopts logic-based semantic languages to express properties and formally reason about them. Compared to other static verification techniques, deductive verification is more expressive. The expressiveness feature derived from the annotations that are necessary to verify a piece of software. They contain the properties that shall be proved as correct, as well as other characteristics related to the program internal execution correctness, for example, loop invariants, pointers aliasing. The annotations also contain multiple built-in labels that specify the old state or the result of the functions, they are usually described in one of the annotation languages and scripted directly in the software.

Deductive verification has advanced abilities. By way of illustration, it is possible to verify the software program without internally creating an abstraction of the original program’s data structures, loops or recursion functions.

Deductive verification tools are generally built as distinct and independent programs, for example, Frama-C [19] and VCC (Verifier for Concurrent C) [20]. They use automatic provers, such as Z3, CVC3 or CVC4, to solve the logical and mathematical formulas. The formulas were automatically produced by the annotated original software program using ACSL (ANSI/ISO C Specification Language), JML, or similar, as shown in Figure 2.1.

In spite of all the advantages that the deductive verification tools provide, they require effort from the programmers to develop the annotations that are necessary for such verification. It is not only to just write the Boolean expressions throughout the code, but it is also necessary to understand how to reason about the Boolean expressions, like simplifying them, prove that one follows from another.
The theorem provers use the logical axioms and inference rules to formally reason about the whole program’s functions, thus proving their partial and total correctness. The most common approach to formally reason about the correctness of software programs is Hoare logic by Tony Hoare [21], and weakest precondition calculus by Dijkstra [22].

Hoare logic is based on the triple \( \{S\}C\{Q\} \), where S is a logical expression, representing the acceptable initial state of the program; C is the program execution; Q is a logical expression, representing the acceptable final state. In other words, S can be considered as a precondition for the program, while Q can be the postcondition.

There are two main categories of correctness that Hoare logic is aiming to prove: partial correctness and total correctness. Partial correctness verification states that if the program C starts with an execution satisfying S, and if C terminates then the program will be in a state satisfying Q. Total correctness is similar to partial correctness and in addition proves that the program successfully terminates in a state satisfying Q.

An example of this is to prove the invariant for a simple iterative program, as shown in the inference rule below [21]; P is a loop invariant, which means its Boolean expression evaluates to true before the iterative statement ever runs, during the loop execution, and after the loop terminates.

\[
\left\{P \land b\right\}C\{P\} \Rightarrow \left\{P\right\}\text{while } b \text{ do } C\{\neg b \land P\}
\]

The symbol b is a Boolean expression that states the Boolean condition that keeps the iterative statements running. If the loop terminates the condition must be false, therefore a not-b appears in the postcondition. The complicated part is to find the invariant P, which might be tricky and indirect in complicated programs.
2.3.1 Frama-C

Frama-C (Framework for Modular Analysis of C programs) is an open-source static analysis platform, devoted to verifying and analysing the functional specification of C programs. Frama-C can demonstrate that there is no occurrence of run-time errors and no variation between the functional specification and the program [19]. The verification analysis that Frama-C is classified under is static analysis, and specifically deductive verification. It performs an exhaustive effort to prove the absence of bugs.

The specifications in Frama-C are written as annotations using the ACSL language. The annotations are initially comments scripted directly in the C program, as shown in the example code in Listing 1. The main reason behind writing the annotations as comments in Frama-C is that the source-code can still be compiled unchanged; the C compiler ignores them while Frama-C reads them.

```
/*@ requires -2147483648<=val && val<=2147483642;
   ensures \result == \old(val) + 5; */

int add_5_val(int val) {
    /*@ loop invariant 0 <= i <= 5;
    loop invariant val = \old(val) + i;
    loop assigns i, val;
    loop variant 5 - i; */
    for (int i = 0; i < 5; ++i)
        ++val;
    return val;
}
```

Listing 1: ACSL Annotated C Program Example

The annotations start with /*@, end end with */, and each relevant statement ends with a semicolon. The logical statements specify the conditions required to verify the function.

The precondition is written next to requires, where the caller of the function should fulfil it. The postcondition is written next to ensures and the function needs to fulfil the requirement after it terminates. The loop invariant deter-
mines whatever condition that holds before, during and after terminating the loop. The loop assigns specifies whichever variables are being modified during the loop execution. The loop variant can be considered as the decrease function that decreases while the loop is running, and it becomes zero when the loop terminates.

The conditions can also include auxiliary variables. These represent many aspects for verifying the software efficiently. For example, the old state of a variable val represented by \texttt{old(val)}.

The plug-ins that Frama-C’s implementation depends on can collaborate and interact with each other. The plug-ins can also be extended by the user as Frama-C has a modular architecture [23]. Some common plug-ins are Eva (Evolved Value Analysis), Jessie, WP (Weakest precondition), Impact Analysis, Slicing and Mthread. The Eva plug-in automatically computes variation domains for C variables and constructs using the Abstract Interpretation technique [24][23].

The deductive verification plug-ins are Jessie and WP; the difference between them is that Jessie relies on the Why3 back-end, and employs a memory model inspired by separation logic. This logic is an extension of Hoare logic to reason about pointer data structures and modular reasoning between concurrent modules. WP focuses on memory parametrisation, that works well with low-level memory manipulation, making it a complementary plug-in for Jessie. The Impact Analysis plug-in can be used in the GUI interface, and it highlights the source-code lines that are impacted by the modification of a selected statement. The slicing plug-in can also be used in the GUI; it slices the source-code and produces an output program based on the developer criterion. For the concurrent programs, the plug-in Mthread is widely used along with Eva. It monitors all possible threads, and reports on their approximate behaviour along with information about all shared variables [23].

2.3.2 VCC

VCC (Verified Concurrent C) is an open-source tool, developed by Microsoft Research in Software Engineering, that formally prove the correctness of C code. It also supports the verification of concurrent programs [8]. VCC reasons about the software programs using deductive verification techniques, where the reasoning is done by the tool Boogie.
In Boogie, the annotations are converted into verification conditions, and then passed to the Z3 solver [25]. VCC uses an annotation language that were specifically created for it [26], similar to the ACSL annotations used in Frama-C. The contracts are enclosed by parentheses, preceded by an underscore, and the placements are below the function name as shown in Listing 2.

```c
int add_5_val(int val)
_-\text{(requires - 2147483648 <= val && val <= 2147483642)}
_-\text{(ensures\ result == \old(val) + 5)}
{
  for (int i = 0; i < 5; ++i)
    _\text{(decreases 5 - i)}
    _\text{(invariant 0 <= i <= 5 && val = \old(val) + i)}
    {
      ++val;
    }
  return val;
}
```

Listing 2: VCC Annotated C Program Example

### 2.4 Software Model Checking

Software model checking is an automatic algorithmic verification technique. It proves the correctness or finds the violation for both properties and specifications. Ideally, it uses finite state-space exploration algorithms to verify the program [27].

Software Model checking exhaustively checks all the possible inputs with respect to the specifications, by setting it up with simple functions, such as `assert()` and `assume()`. It does not require a large amount of annotations as in the deductive verification technique, which makes it more automatic checking. The specification are written as a propositional logic formula, without adding additional annotation regarding loop invariants, variants, etc. Model checking therefore reduces the human intervention, as can shown in Listing 3.
```c
int add_5_val(int val) {
    assume(val >= INT_MIN && val < INT_MAX - 5);
    int x = val;
    for (int i = 0; i < 5; ++i)
        x++;
    assert(x == val + 5);
    return x;
}
```

Listing 3: Assertions in Software Model Checking Example

Software Model checking, also called property checking, is known to be efficient [16] as well as capable of providing counter-example for the unsatisfied properties. A general view of the software model checking approach is shown in Figure 2.2.

![Figure 2.2: General View of Model Checking Techniques](image)

2.4.1 Eldarica

Eldarica is a state-of-the-art open-source software model checker, specialised in solving Horn clauses module over integer arithmetic. Eldarica as a Horn solver was extended to support algebraic data types and bit vectors. It also supports theories that are applied in verification but not supported yet in most other Horn solvers. Eldarica can be used on any platform with JVM and it depends only on Scala and Java libraries [28][12].
Horn clauses are a disjunction of literals, with at most one non-negated literal. The first logician who suggested their importance was Alfred Horn [29]. The disjunction of the literals form can be transformed into an implication form, which can be formally proved by applying De Morgan’s law and implication definition.

An example of a definite Horn Clause:

\[ \neg a_0 \lor \neg a_1 \lor \ldots \lor \neg a_n \lor b \]

This can be rewritten to the implication form, where b is called the head and \( a_0 \land a_1 \land \ldots \land a_n \) is called the body:

\[ b \leftrightarrow a_0 \land a_1 \land \ldots \land a_n \]

Horn clauses form a proper basis from which one can verify a program, and are used in multiple software model checking tools, for example, Eldarica, SeaHorn [30], and JayHorn[31].

In Eldarica, a Control-Flow Graph (CFG) is constructed from the input program, where the transitions of the CFG will be encoded using Horn clauses, and solved later by the theorem prover. Horn clauses are constructed in a way that its satisfiability is equivalent to the program safety [12].

Eldarica accepts software programs in the form of C, Prolog and SMT-LIB, as shown in Figure 2.3. Input Programs will be sent to a preprocessing phase where the code will be transformed by the forward slicing operations, reachability analysis, clause inlining, etc. Afterwards, Eldarica will check satisfiability using a combination of Lazy Cartesian Predicate Abstraction and CEGAR (Counterexample-Guided Abstraction Refinement). The CEGAR engine loads Princess as a library to represent terms, formulas and background theories in order to speed up the SMT queries [12].

A classical problem with software model checkers is the need for refining the abstraction because it is failing to discover the right predicates. Eldarica resolves this problem by using two methods. The first one is based on an acceleration using the FLATA tool to boost the Craig Interpolator that the Princess solver provides. The second method Interpolation Abstraction which controls Craig Abstraction’s computed results [28].
An illustrative example of Eldarica in Listing 4, where the return value for this execution is SAFE.

```c
void f1(void) {
    int x = 1;
    int y = 0;
    while (y < 3) {
        x = x + y;
        y = y + 1;
    }
    assert(x >= y);
}
```

Listing 4: Eldarica Code Verification Simple Example

Currently, an extension of Eldarica is under development, called TriCera. TriCera uses Eldarica’s Horn encoder for C programs and extends it to support structs, pointers, and heap pointers. TriCera uses Eldarica’s theorem prover to solve the Horn clauses.
2.4.2 SeaHorn

SeaHorn is an open-source software model checker, specialised in verifying safety properties in software programs written in C. The quality that makes SeaHorn different is the modularity and the reusable verification components. Its design creates an environment with an extensible and customisable framework. SeaHorn is implemented in the LLVM compiler infrastructure and built using C++ [30].

The main advantages of using SeaHorn over other software model checking tools are the possibilities to reason about pointer addresses, scalars, bit-vectors and memory contents. The disadvantage of SeaHorn is that it can not reason about dynamic linked data structures or concurrency [30].

The SeaHorn algorithm starts with a preprocessing phase, where the initial analysis of the C program with CIL initialises the local variables and defines missing functions. After that llvm-gcc will translate it to LLVM IR (Intermediate Representation) and after an optimisation step, it will do other preprocessings like inlining and dead code elimination, as can be seen in Figure 2.4.

![Figure 2.4: Overview of SeaHorn Architecture [30]](image)

In the end, a DSA (Data Structure Analysis) will be run to analyse the alias in the memory. The result will be further handled in the invariant generation phase and
Horn clauses encoding phase. The invariant generation phase will compute inductive numerical invariants using the IKOS library. The Horn clause encoding phase will translate the byte code to constrained Horn clauses with either small step semantics or large block encoding. This will be fed to the last phase, which will attempt to solve the Horn clauses with an SMT solver using PDR/IC3 algorithms [30].

Verifying a piece of code using SeaHorn requires the variables to be initialised, otherwise the results returned from it will be undefined. SeaHorn uses Clang and LLVM to preprocess the code, both of these tools apply different heuristics to deal with undefined behaviour, thus the result is completely unpredictable. For example, in listing 5, integer \( n \) is initialised to a non-deterministic value using the function \( \text{nondet()} \), which returns arbitrary integer value to eliminate the undefined behaviour.

If SeaHorn successfully verifies the assertions in the C code, the terminal output result will be UNSAT, which means that the error is not reachable, thus safe behaviour. Correspondingly, if SeaHorn returns SAT, it means that the error is reachable, or in other words, unsafe behaviour. For example, in the simple Listing 5, the terminal output is UNSAT.

```c
int nondet();
int main(void) {
  int x = 0;
  int n = nondet();
  while(x<n) {
    x++;
  }
  if(n>0)
    sassert(x==n);
}
```

Listing 5: SeaHorn Code Verification Simple Example
Chapter 3

Related Work

Verification of safety-critical systems has been an important topic for many years. Future trends are evolving rapidly, and currently, the transportation industry trends are directed towards autonomous driving that depends heavily on safety-critical embedded systems. Those embedded systems interact with each other, thus will force the sharing of resources between them. “This interaction will eliminate the physical separation that provides confidence in correct operation”, as John C. Knight mentioned [32]. He also proposed “Verification by testing is impossible for systems that have to operate in what has been called the ultra-dependable range. Yet, in practice, there are few choices. Formal verification and model checking are desirable technologies but are limited in their applicability”.

In the early eighties, NASA conducted a case study to apply formal methods to the SIFT control system, a safety-critical flight control system written in Pascal using the SRI specifications. The process was divided into two parts, verifying the I/O specifications, and verifying the transition and fault model specifications [33]. NASA concluded that it was possible to use hierarchical verification techniques to demonstrate that the design satisfied its requirements. NASA had thus established one of the early real applications of safety-critical systems verification using formal methods.

Another safety-critical system is the interlocking control tables of railways. These were verified using NuSMV and SPIN model checkers in the “Model Checking Interlocking Control Tables” paper [34]. The study was performed on different functions with various input sizes. The results state that model checking tools can verify a small scale interlocking system, however it can not verify the medium or large scale systems.

Another experiment performed was on seL4, the first formally verified micro OS kernel [35] in 2009. The source-code of the seL4 project consists of 8700 lines of C99 and 600 lines of ARM assembly. The formal verification prover that the authors used
is Isabelle/HOL theorem prover, and it requires human intervention to construct and

guide the proofs, which was preferable over model checking techniques, because of

potential problems like being constrained by either specific properties or finite state

space. The properties verified here are out of reach for software model checkers.

Recently, the study “Deductive Functional Verification of Safety-Critical Em-

bedded C-Code: An Experience Report” was conducted in Scania by Gurov et al

[7]. They experimented with formalising the functional requirements, and apply-

ing formal methods, specifically deductive verification using VCC, on an embedded

safety-critical module. The study results were positive, but there were drawbacks

with developing the annotations of VCC, and their proposed solution was to auto-

mate this annotating process.

Their implementation required analysing the requirements and forming a combi-
national logic circuit out of them. Afterwards, the code had to be transformed into

a compatible form that could be passed to the verifier, VCC. This was managed by

rewriting or removing any compiler-specific language extensions or header files.

To verify the code in [7], it was required to use suitable code annotations. The

code annotation of the single top-level function contained the contracts of require-

ments that were selected for the verification process. Some of them were written

in a way that described the memory read and write, and others were written to

describe the actual variables of the code.

There was excessive use of ghost variable assignments to represent the local

function variables and their modifications. There were two methods used to ensure

that the ghost variables were correctly assigned to the correct values during the

execution. The first approach was to simply assign the ghost variables the values

that are local to the program variables. This made VCC verification time fast

because the ghost variables were continuously synchronised with the actual code.
The second method was to specify a separate ghost program that computed the

combinational logical circuit that they created out of the requirements. This was

slower than the first method because the number of represented variables was very

large in the steering ECU.

The results of paper [7] were that the run-time to verify the whole module was

165 seconds, and the annotations overhead was almost 50% or roughly 700 lines of

annotations.
In general, verifying a safety-critical system that consists of non-trivial functions is a challenging task. The formal methods were successful in verifying the SIFT system of NASA and seL4 micro kernel. Deductive verification can be successful in verifying functional requirements, but it is hard to develop annotations for temporal features. Some model checkers provide the feature of formalising temporal properties, but it was confirmed to be working better on small scale systems.
Chapter 4

System of Interest: Steering ECU

This chapter gives a concise overview of the system used in this case study. It describes general hardware and software architecture, the relationship between MISRA-C guidelines with the available source-code, and briefly discusses the requirements of the steering unit. Abiding by Scania privacy policy, none of the function names, requirements and examples are real.

4.1 Abstract Module Architecture

The safety-critical system that was considered for this case study is the steering module, which is in charge of powering the steering of the vehicle and allows the driver to steer effortlessly. The system contains two main separated circuits, primary and secondary. The ECU (Electronic Control Unit) circuits are activated, functioning, deactivated alternatively based on an algorithm, which makes the decisions for the steering unit behaviour, depending on the state of both primary and secondary circuits along with the input signals that arrive to the ECU.

![Coordinator Unit and ECUs](image)

*Figure 4.1: Coordinator Unit and ECUs*

Every single ECU is connected and controlled by the proper coordinator unit, as shown in Figure 4.1, and considered as part of a bigger system. The ECUs are connected via a CAN bus, which allows them to initiate the communication, without
using a host computer. The coordinator is a task scheduler, that calls the functions of the ECU every time unit, or when an event interrupt occurs.

In general, the steering module interacts with a real-time database, that stores the values it reads and writes to the steering ECU. The real-time database also transacts and receives data through a CAN bus to other units, such as, other ECUs, sensors and actuators. The overall simplified architecture I/O is visualised in Figure 4.2.

![Simplified System Architecture](image)

*Figure 4.2: Simplified System Architecture*

### 4.2 MISRA-C Code Correspondence

The code is written under Scania internal programming rules most of which are similar to MISRA-C (Motor Industry Software Reliability Association) guidelines [36]. MISRA-C guidelines govern the embedded control software inserted in the vehicle, to ensure the safety and quality of it.

The similarities between Scania programming rules and MISRA-C can be found in multiple aspects, such as, memory allocations, recursions and control flow.

The steering module program does not contain any dynamic memory allocations, as they often introduce unpredictable behaviours while allocating the heap memory.
or the stack, thus it is harmful to safety-critical software. An example of dangerous
behaviour includes; forgetting to free the memory which might cause an attempt
to step out of the dynamic memory boundaries. To prevent this, the malloc,
calloc, realloc and free represented in Rule 20.4 in MISRA-C 2004 [36] are
excluded from the program.

Recursions are eliminated from the program, due to the possibility of accidentally
exceeding the available stack space. This rule is documented clearly in MISRA-C
[36], Rule 16.2.

The control flow of the program is very clear as it does not contain any goto
statements or loops. However, it contains many conditionals, switch cases, pointers
and structs. The absence of the loops make the verification process of the deductive
verification tools easier, i.e., Frama-C and VCC do not require finding loop invariants
and the decrease functions in this case.

### 4.2.1 Function-Call Hierarchy

![Approximated Function Call Graph of the System](image)

The backbone file contains 10 functions, approximately 1400 lines of code that
depend on other macros and typedef. One of the functions acts as the top-level
function and is called periodically. Figure 4.3 illustrates the function hierarchy.
It also calls all the other functions to perform the obligatory assigned tasks that
operate the steering ECU. The execution of the functions occur sequentially within

24
each other, while the whole program itself runs in parallel to other ECUs in the vehicle software execution.

### 4.3 System Requirements

The total number of requirements are 33, and they represent only the functional safety requirements of the system. Principally, the requirements document is not fully formal nor informal, but a combination of both, which can be contemplated as semi-formal. The documentation of the requirements is well structured. Each requirement specifies the behaviour of the system in two manners; in an informal descriptive way, and a roughly logical explanation of it, i.e semi-formal.

<table>
<thead>
<tr>
<th>REQ97012: If the vehicle is moving and the primary circuit cannot provide power steering then the vehicle is moving without primary power steering.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF VehicleMoving==TRUE AND PrimCkt==TRUE MovingWithoutPrim == TRUE</td>
</tr>
</tbody>
</table>

The steering requirements variables, such as VehicleMoving, do not resemble the variables names in the steering code. They can be one of the following:

1. A single independent variable that could represent a similar variable or a state in the actual code. For example, the requirement variable MovingWithoutPrim refers to the variable Mov_Not_Prim_Sat in the C code, i.e. Mov_Not_Prim_Sat == True.

2. A representative variable that describes the behaviour when a combination of two or more conditions and variables in the code hold. For example, the requirement variable VehicleMoving point out the condition where the following arguments are satisfied: SensorX voltage is above 5V, the state of the brakes, the state of the electric engine speed. VehicleMoving implies the atomic satisfaction of the condition:

   SensorX > 0x05 && Park_Br.state== NotSet && Engine_Speed.state==Sat
The functional safety requirements adhere to the ISO 26262 standard [4], which recommends enforcing the feature of requirement traceability and tracking to other artefacts of the project. These include tests, issues, and code, which serves to make the requirements verifiable.
Chapter 5

Verification Process Overview

To fulfil the goals of this thesis, a case study was conducted on one ECU only, with one set of requirements, resulting in a case study more qualitative than quantitative. The selected ECU was the steering unit, and the requirements were the safety functional requirements that contained both liveness and safety properties. The selection is the same as the previous experiment that was carried out in Scania [7]. The software model checking tools used in this case study were SeaHorn and Eldarica.

The verification process method was a bottom-up approach, where the initial steps were intended to form a link between the safety-critical requirements and the code. Afterwards, the verification process steps were formalised in a higher level of abstraction, as viewed in Figure 5.1.

![Figure 5.1: Verification Method Outline](image-url)
First of all, the requirements sheet was analysed and investigated. The requirements were well sectioned and partitioned, which eased the understanding of the overall purpose of the steering unit. Each requirement was inspected to gain an insight into the related variables in the requirements sheet, and how they were connected. The requirements were classified according to three criteria: safety-critical, outputs from other ECUs and input to steering, and real-time.

Second, the code was reviewed and inspected for a better insight of the steering unit’s comprehensive task. During this stage, the 10 functions that formulated the concept of the steering unit were considered. The inputs and the outputs of each function were fully understood, thus the data flow and the code structure became more clear. The purpose of each function was written down on a paper sheet, as it would help deriving and mapping the requirements to them later.

Third, mapping the requirements to the related function. After getting the appropriate understanding of each function, data flow, flags purposes and the requirements sheet, it was necessary to associate the requirements to the proper function, to be able to start the verification process. Here, a function-call tree was constructed, and the selected requirements were placed below each related function. This was important to keep track of the overall implementation of the case study.

Fourth, interpreting the requirements to a non-abstract level. Each mapped requirement was translated to the appropriate logical formula in the complementary line in the function. Each requirement variable was mapped to the legitimate code variable or the set of variables that represented it. Here, the requirement assertions were inside of the function’s scope, to ensure that the formalisation was correct.

Fifth, unit and component verification. In this stage, the functions were individually verified in a separate main(), that called all the functions and assigned respective preconditions and postconditions to their proper placements.

Sixth, system integration verification. The steering ECU is not an isolated unit; it is a part of a bigger system and it interacts with other units like the real-time database, error diagnostic unit, etc. The scope of the requirements was narrowed down to include only the real-time database relevant functions, simply because this was the available code.

Seventh, document the findings. A comparison between model checking and deductive verification was done at this point. The main correlated comparison value
was the CPU run-time of the user mode and the time spent in the kernel executing
the call for all deductive verification and model checking tools.

If step four, five or six failed, then the following techniques were followed; gen-
erate a counter-example, instrument slice-back and code inspection, simplify the
logical statement, change the placement of the assertions and go back to the tool’s
software documentation to understand the reason.
Chapter 6

Verification Process

This chapter describes the verification process steps. The process starts by associating the requirements to the correct piece of code, and then introducing the translation technique from semi-formal to logical formulas. Throughout the chapter, the steps that have been carried out will be further elaborated, until the implementation reaches the input-output relationship of the ECU.

The sections are also supported with an unpretentious example, to acquire a better understanding of the overall process. The example is not realistic, solely an approximated requirement and does not represent any of Scania’s requirements, due to the non-disclosure agreement.

6.1 Case Study Setup

This section previews the general setup of both Eldarica and SeaHorn, and a short discussion of the preprocessed code, and its importance concerning Eldarica.

6.1.1 Default Arguments for Eldarica and SeaHorn

Eldarica v2.0 and SeaHorn v3.8 are available online. The installation was straightforward for Eldarica, as it required JRE (Java Runtime Environment) installed on the machine to run Eldarica’s binary downloaded from GitHub. It is also possible to compile Eldarica’s source-code, as it only requires JDK (Java Development Kit) and sbt (Scala build tool) to be installed. SeaHorn installation was slightly more tricky, and in the end, the best alternative was to run it in a container with Docker[37].

There are various arguments to control in both Eldarica and SeaHorn. In El-
darica, the most significant one is changing the arithmetic mode to 32-bit integers rather than mathematical, in order to catch the overflow and underflow.

On the other hand, SeaHorn commands can check the array-bound instrumentation, NULL deference, simple memory safety check and create an executable file for the counter-example. The arguments that were used in the implementation of the test cases were enabling the non-deterministic behaviour and the compiler optimisation flag to be set once on and off. Other than that, both SeaHorn and Eldarica arguments were left to default values in the terminal, and they were altered only when a verification violation or error occurred.

6.1.2 Preprocessed Code

Eldarica’s input accepts mainly Horn clauses written in SMT-LIB 2 and Prolog. It supports a fragment of the C language, and this fragment is to some extent limited, although the tool is still under development. Eldarica also runs using its own parser, although it requires some adjustments for the input C source-code currently. The adjustments are necessary, especially for non-trivial programs that contain macros, which is common in industrial software as a matter of fact, but not for simple and short programs.

Preparing the input to Eldarica’s parser is possible through preprocessing the C source-code. This textually involves replacing the macros in the source-code with their actual definitions and representations. C preprocessing also adds the contents of the files that are included in the source-code and removes the comments.

6.2 Mapping Requirements to the Code

Requirement mapping was done after considering the code structure and each function’s objective, where each requirement was traced to the correct code line by human insight. It was also important to ascertain that each requirement cannot be related to any other lines of code, i.e. distinctive.

The steering requirements had been written to describe the overall implemen-
tation of the system, and accordingly most requirements were not function-based. That is, they did not state the precondition and postcondition to the functions, so they did not correlate to the function’s arguments or returned values most of the time. Instead, they indicated specific lines of code inside the functions and their local variables. As a result, code inspection was done multiple times throughout the process.

The overall number of requirements associated with the case study was 33, and they were classified into three main categories: 19 requirements determined the safety functional behaviour of the steering ECU, 6 requirements determined the behaviour of other ECU modules that were related to the steering ECU, 5 requirements depended on outputs from other ECUs, 3 requirements were identified as real-time and held some temporal properties. Out of these, only the safety-critical category was under the scope of this study, because the other categories described irrelevant attributes to the steering ECU. For example, they described certain settings, and the way they were stored in other modules preceding the activation of the steering ECU. In addition, neither the specification sheet nor the code for those were available.

All of the requirements were based on IF statements, and most of them also included ELSE IF and ELSE, which meant that conceptually any single requirement, in fact, contains its own additional requirement. For example, consider a safety-critical requirement that contains the variable ElectricMotorSensor, that will be set to ON, OFF, or FAULT, depending on the value of the following variables $\alpha$, $\beta$ and $\gamma$. The requirement thus represents more than one requirement; it states three distinct preconditions and three distinct postconditions. The electric motor sensor is ON when the precondition $\alpha=T \land \beta=F$ is satisfied, while the electric motor sensor is OFF when $\alpha=F \lor \gamma=T$; further the electric motor sensor is FAULT when both of the previous preconditions fail.

$$ElectricMotorSensor = \begin{cases} 
ON, & \text{if } \alpha=T \land \beta=F \\
OFF, & \text{else if } \alpha=F \lor \gamma=T \\
FAULT, & \text{else}
\end{cases}$$
6.3 From Semi-Formal to First-Order Logic

Requirements were translated from the semi-formal to formal logical formulas that included the relevant variable name in the software, thus ensuring that the translation was correct. The logical formulas had to be compatible with the model checker’s compilers; SeaHorn used the LLVM compiler with Clang front-end, and Eldarica used its own parser.

Eldarica and SeaHorn did not support the implication operation, $A \implies B$ as in Frama-C and VCC, hence the proper technique to translate them to disjunctions. The disjunction logical equivalence of implication is the statement $\neg A \lor B$, where $A$ and $B$ are propositions similar to the ones in the original implication statement.

The requirements statements were in some cases very long and could be highly correlated to an IF or SWITCH CASE statements in the steering ECU code. Requirements partitioning was the easiest solution to adjust them to the source-code. The requirements were segmented into multiple assertions where each assertion satisfied the result of the IF statement in the code.

The following Hoare logic rule states the inference for conditionals in the code:

$$
\begin{align*}
\{B \land P\} & C_1\{Q\}, \{\neg B \land P\} C_2\{Q\} \\
\{P\} & \text{if } B \text{ } C_1 \text{ else } C_2\{Q\}
\end{align*}
$$

In the previous Hoare rule $B$ is the relational conditional of the IF statement, $P$ is precondition, $Q$ is postcondition, $C_1$ is the statement result executed if $B \land P$ is true, and $C_2$ is the statement result executed if $\neg B \land P$ is true. The rule states that if the initial state satisfies $B \land P$ and passes through the statements $C_1$ it will end in a state that satisfies $Q$.

Likewise, if the initial state satisfies $\neg B \land P$ and passes through the statement $C_2$ it will end in a state that satisfies $Q$.

Consequently, the verification of the requirements can be partitioned into multiple sections, i.e. multiple assertions, thus prove correctness. For illustration, consider the following requirement:
REQ97018: If the alternative circuit is providing power steering and the parking brake is not set then the electric motor must be activated. If the alternative circuit is not providing power steering or Switch1 is sending “set” then the electric motor must be deactivated.

IF AlternativeCircuitPower == TRUE AND StopBrake == NotSet
ElectricMotor = Activate
ELSE IF AlternativeCircuitPower == FALSE OR Switch1 == Set
ElectricMotor = Deactivate

The requirement contains both logical equivalences and assignment operators. The logical equivalence denoted by “==” returns the value true if and only if both operands are equal. The assignment operator denoted by “=” assigns the right side operand to the operand on the left side.

The logical interpretation of the previous statement can vary from person to person. One interpretation that could be correct is the following:

\[(\text{AlternativeCircuitPower} == T \land \text{StopBrake} == \text{NotSet}) \implies \text{ElectricMotor} == \text{Activate}\] \land
\[(\text{AlternativeCircuitPower} == F \lor \text{Switch1} == \text{Set}) \implies \text{ElectricMotor} == \text{Deactivate}\]

The interpretation of this statement could be translated in a C compatible format as shown in Listing 6, where the requirement variables are mapped to their equivalent local and global variables, structs, pointers, etc. For example, the variable AlternativeCircuitPower is associated with the struct Elec_Mo.Ac in the code, which contains multiple members, whereas ElectricMotor is associated with Elec_Mo.Ac in the code, which is a local variable that is initiated inside the scope of the function. MAXI is basically a macro function that determines whether the alternative circuit is providing power based on a suitable range.

A single requirement variable can be associated with more than one variable in the code, for example, in Listing 6 it is shown that the requirement variable StopBrake is actually a logical formula that contains two code variables: master cylinder and parking brake switch number four. The stop brake satisfies the state
NotSet whenever both of the following conditions hold; the master cylinder and the parking brake switch labelled by number four are not set.

```c
int Setting_1(){

    ...

    assert(!(MAXI(Alter_Ckt_P.boolvalue) == TRUE
             && (Park_Br_sw4.state == NotSet && Master_cylin == NotSet))
             || (Elec_Mo_Ac == Activate));

    assert(!(MAXI(Alter_Ckt_P.boolvalue) == FALSE
             || switch_bool_1.state == Set)
             || (Elec_Mo_Ac == Deactivate));

    ....
}
```

*Listing 6: Requirement Interpretation and Translation for Requirement REQ97018*

### 6.4 Unit Verification Harness

Proving the unit correctness required creating a main function that called every single function, i.e. harness, and specifying the safety properties below accordingly. The requirements used both global variables and local variables in the function scope. In the case of local variables, few minor modifications to the code were necessary. In order to access the local variables in C, a representation of them needed to be declared in the global scope, thereafter assigning the local variables to the global inside the scope of the function. To eliminate confusion, the global scope variables had the same exact names of the locals, with the prefix `SH_`.

The implementation in SeaHorn was in a format similar to Listing 7, where the global variable `SH.Elec_Mo_Ac` represents the local variable `Elec_Mo_Ac` in the main.

On the other hand, verification in Eldarica is slightly different. Requirements were written in the C preprocessed code, where the simple macros and function-like
macros that are in the source-code or other header files that are included in it got expanded, in a similar manner to Listing 8. The macros MAXI, SET and NOTSET were replaced.

```c
#define MAXI(a) ((a) > (0xF0) ? (a) : (0xF0))
#define SET 0x01
#define NOTSET 0x00
int SH_Elec_Mo_Ac;
int setting_1(){
    //settings Function Implementation
    SH_Elec_Mo_Ac = Elec_Mo_Ac;
    ...
}
void main () {
    ...
    setting_1();
    sassert(!(MAXI(Alter_Ckt_P.boolvalue) == TRUE
     && (Park_Br_sw4.state == NOTSET && Master_cylin == NOTSET))
     ||(SH_Elec_Mo_Ac == Activate));
    sassert(!(MAXI(Alter_Ckt_P.boolvalue) == FALSE
     || switch_bool_1.state == SET)
     ||(SH_Elec_Mo_Ac == Deactivate));
    ...
}
```

*Listing 7: Requirement REQ97018 Translation in SeaHorn*
Listing 8: Requirement REQ97018 Translation in Eldarica

### 6.5 Integration with Database Verification

The integration with the real-time database imposed that the steering ECU was verified on an abstract level. The unit verification process examined the correctness of every single function, as specified in the requirement sheet, however it was restricted to the function itself. It did not guarantee that the whole ECU was working properly or the fact that it was satisfied globally with respect to reading and writing to the database. Out of the 19 requirements, only 10 were related to integration with the real-time database and were considered in this stage of implementation.

To perform the integration verification task, a `main()` function was created as a harness, along with one function call only; the top-level function that called all the other functions. The variables that were being retrieved or stored in the database had been altered to the genuine signal names, which could be fetched from the different source-codes and header files that initiated them.

The read and write processes to the real-time database were not trivial. The database contained many arrays, functions, macros and structs scattered in different header files. Accessing the content of the proper signal value required the knowledge
of which disk chunk it referred to as well as other factors like the exact struct name and its members. For example, an approximated way of implementing it shown in Listing 9.

```c
void main ()
{
    Periodic_func(); //Single-entry Top-level Function
    sassert(!(MAXI(DB_Chunk_1[ALTER_POWER_CKT].StructX.SignalY)==TRUE
            && (DB_Chunk_2[BREAK_S_N4_VOLT].StructX.StateY==NOTSET
                && DB_Chunk_2[MASTER_CYL_CALC].StructX.StateY==NOTSET))
            ||(SH_Elec_Mo_Ac == Activate));
    sassert(!(MAXI(DB_Chunk_1[ALTER_POWER_CKT].StructX.signalY)==FALSE
            || (DB_Chunk_2[BREAK_S_PARK_VOLT].StructX.StateY== SET))
            ||(SH_Elec_Mo_Ac == Deactivate));
}
```

Listing 9: Requirements REQ97018 Translation in SeaHorn using Database Signals

6.5.1 Input to Output Using Functional Substitution

The local variables still appeared in the integration verification at this point, which indicated the dependencies of the safety-critical requirements on the local values of functions. The requirements were implementation specific rather than functional or unit specific.

The local variables that were introduced by the requirements as a postcondition, were actually reused again by other requirements as a part of the precondition. For example, the following requirement shows that the ElectricMotor is responsible for transmitting the voltage signal to the corresponding vehicle port to activate it.
REQ97022: If the electric motor is activated, the voltage signal should be send to the port 3 switch.

\[
\text{IF ElectricMotor == Activate} \\
\text{SendVoltage = Port3} \\
\text{IF ELSE ElectricMotor == Deactivate} \\
\text{SendVoltage = NA}
\]

The requirement variable ElectricMotor is a local variable in the code and it is not reading from or writing to the database. In REQ97018 it appears as a postcondition, while in REQ97022 it appears clearly as a precondition. The requirement variable SendVoltage is a representation of a value that is being written to the real-time database.

Consider the following mathematical equations, that present both REQ97022 and REQ97018.

\[
SendVoltage = \begin{cases} 
\text{Port3,} & \text{if ElectricMotor=Activate} \\
\text{NA,} & \text{else if ElectricMotor=Deactivate}
\end{cases}
\]

\[
ElectricMotor = \begin{cases} 
\text{Activate,} & \text{if AlternativeCircuitPower=T} \land \text{StopBreak=F} \\
\text{Deactivate,} & \text{else if AlternativeCircuitPower=F} \lor \text{Switch1=T}
\end{cases}
\]

Applying a functional substitution will produce the following mathematical equation that traces the database related variables only.

\[
SendVoltage = \begin{cases} 
\text{Port3,} & \text{if AlternativeCircuitPower=T} \land \text{StopBreak=F} \\
\text{N/A,} & \text{else if AlternativeCircuitPower=F} \lor \text{Switch1=T}
\end{cases}
\]

It can be concluded that it is possible to get rid of the local variables in the verification process, by utilising functional substitution of the requirements in the local variables; through finding a path that connects the requirement variables that represents data fetched from the database, without using the requirement variables that represent local variables. A graphical illustration is shown in Figure 6.1 and Figure 6.2.
The path should be simplified and added in a logical form in the assertions. When local variables disappear from the assertions, it will no longer be required to modify the code to retrieve them, and the requirements turn into a relationship between the input and the output of the steering ECU. The implementation at this point was similar to Listing 10.

```c
void main () {
    Periodic_func(); //Single-entry Top-level Function
    sassert(!(MAXI(DB_Chunk_1[ALTER_POWER_CKT].StructX.SignalY)==TRUE
            && (DB_Chunk_2[BREAK_S_N4_VOLT].StructX.StateY==NOTSET
            && DB_Chunk_2[MASTER_CYL_CALC].StructX.StateY==NOTSET))
        || (DB_Chunk_3[VOLTAGE_TRANSMIT].StructX.signalY == Port3));
    sassert(!(MAXI(DB_Chunk_1[ALTER_POWER_CKT].StructX.signalY)==FALSE
            || (DB_Chunk_2[BREAK_S_PARK_VOLT].StructX.StateY== SET)
            || (DB_Chunk_3[VOLTAGE_TRANSMIT].StructX.signalY == FALSE));
}
```

**Listing 10: Requirement REQ97018 Functionally Substituted in REQ97022 in SeaHorn**

At this stage in the steering module, only two local variables were found in
multiple requirements, and the suitable functional substitutions were performed accordingly.

6.6 Violation Handling Techniques

Unsatisfied properties were handled by utilising multiple techniques, usually starting with generating a counter-example from SeaHorn or Eldarica. The counter-example gave the reasoning about when a specific assertion could be unsatisfied and explained why the requirement was identified as not safe.

In Eldarica, counter-examples are generated by adding to the execution command the argument \(-\text{log}\), which generates an output that presents all the variables in the code and displays the mapping for those variables in the unsafe state.

To generate a counter-example in SeaHorn, more intricate steps are followed; LLVM creates a bit-code of the violated C program, i.e. test harness, as an object file. This step was implemented by adding to the run command the argument \(-\text{cex=/host/harness.ll}\). The object file \text{harness.ll} was connected to the original source-code to create the executable file. The LLDB debugger [38] was then used to acquire the counter-example values. LLDB operates by setting breakpoints in the code, running the executable file and printing out the counter-examples.

The generation of counter-examples is quite convenient, because these can detect whether the error resulted from either the interpretation of the requirement or a coding error. Both cases are reached through an analysis of the code and the assertions.

The unsatisfied requirements were reviewed again. They were logically simplified and refined with their relationship to the code. The related code variables that affected the requirement were examined manually by backward slicing. It was useful to divide the unsatisfied requirements into multiple subsections, and implement the backward slicing on a smaller subset of the requirements variables.
6.7 Mutation Testing and Sanity Check

After the implementation of the verification process, each assertion in the unit verification and the integration verification was tested to validate the verification result.

The first test was a mutation test, which required modifying certain statements in the code, in order to assess the quality of both requirement formalisation and the software model checkers being used. The mutation test cases were given by Scania, and they contained 10 statements that expressed faulty conditions. The faulty statements were achieved through removing, adding and changing values. Those alterations represented changing the signal names that are being read from the database on the verification output, extending the range of variables in the control statements, and altering the variables that are critical to the implementation.

The second test was a basic sanity check, where the verified statements were altered to have a stronger guarantee of the software model checkers’ results. It was necessary to implement a quick sanity check along with the mutation test, due to the poor coverage of the mutation test over the assertions of the requirements. The sanity check was conducted through falsifying each requirement and investigating the result, if it indicated an unsafe behaviour.
Chapter 7

Empirical Results of Verification

The aim of this case study was to investigate the feasibility of proving the correctness of embedded systems using Horn clause solvers. The result was positive in most cases.

The available requirement sheet contained 33 requirements, 19 were translated to first-order predicate logic. A subsection of those was simplified as well. The verification process was enforced on multiple levels; low non-abstract where the system was treated as a white box, and high abstract where the system was treated as a black box. This was done to investigate how far the Horn clause solvers could reach in terms of verifying such embedded systems. The experiment was successfully implemented and well-scaled to the size of the code as presented in this chapter.

The result chapter is divided into three sections. The first section discusses the verification outcomes using Eldarica. The second section addresses the results using SeaHorn, over multiple levels of verification. The verification levels contains the categories discussed in Chapter 6: single requirement, unit verification, integration verification and I/O verification. The third section addresses the results of the mutation test and sanity check.

The results are measured by the run-time and the ability to verify the requirements. The run-time includes three categories; real, user and system. The real-time is the duration from start to finish of the verification procedure, the user-time is the actual CPU time used in the verification process, and system-time is the amount of CPU time spent in the kernel space within the verification process. The run-times were captured using the built-in time function in the bash shell.

The case study was conducted using Ubuntu 17.10 running inside a virtual machine. The host machine had an Intel Core i5-7500 CPU @ 3.40 GHz. Three CPU cores were allocated to the virtual machine, and 3.8 GiB memory.
7.1 Results for Eldarica

Run-time results of the verification process using Eldarica is shown in Table 7.1. The results are measured in seconds where the preprocessing stage is enabled. The preprocessing contains transformations on the code such as forward slicing, reachability analysis, constant propagation and clauses inlining.

Overall, the tool succeeded in proving the correctness of two requirements only, which are satisfied in the same function. When both requirements are verified together, the run-times increase as can be seen in the row *Full Implementation*. The CPU user-time is higher compared to the time spent in CPU kernel space and real-time. This can be due to the multiple cores that the program is running on.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Enable Eldarica Preprocessing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Real-time[s]</td>
</tr>
<tr>
<td>REQ11</td>
<td>3.19</td>
</tr>
<tr>
<td>REQ12</td>
<td>2.70</td>
</tr>
<tr>
<td>Full Implementation</td>
<td>5.10</td>
</tr>
</tbody>
</table>

*Table 7.1: Verification Run-times of Requirements Using Eldarica*

7.2 Results for SeaHorn

The verification procedure results using SeaHorn are adequate comparing to Eldarica. The outcomes are divided into sections, where each section presents the verification level. Each verification level contains the run-times for both the verification process and the sanity check. Some sections also contain the run-times with LLVM optimisations enabled, to investigate the effect of it.

7.2.1 Single Requirement Verification Results

Table 7.2 presents the run-times of the verification procedure for each individual requirement. It was found that the number of requirements that are relevant to this case study was 19 safety functional properties. In Table 7.2 each requirement was placed alongside the appropriate function it should satisfy. The table is missing
two requirements due to implementations issues, which will be further discussed in Chapter 8.

When the compiler optimisations are enabled, the real-times are very close. They are all less than 0.4 seconds. Disabling the compiler optimisation will increase the real-times by at least a factor of two.

Fields marked with the symbol X indicate that SeaHorn froze during the verification process. The reason of the freezes were most likely due to halting and unresolvable problems that occurred with optimisations disabled. It increased the amount of code that was transformed into Horn clauses and evaluated by the SeaHorn theorem prover, as evidenced by Table 7.2.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Real-time[s]</td>
<td>User-time[s]</td>
</tr>
<tr>
<td>Setting 1()</td>
<td>REQ4</td>
<td>0.25</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>REQ9</td>
<td>0.33</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td>REQ15</td>
<td>0.23</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>REQ13</td>
<td>0.25</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td>REQ18</td>
<td>0.25</td>
<td>0.20</td>
</tr>
<tr>
<td></td>
<td>REQ16</td>
<td>0.26</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td>REQ17</td>
<td>0.21</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>REQ19</td>
<td>0.24</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td>REQ14</td>
<td>0.20</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>REQ20</td>
<td>0.21</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>REQ30</td>
<td>0.22</td>
<td>0.15</td>
</tr>
<tr>
<td>Setting 2()</td>
<td>REQ11</td>
<td>0.18</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>REQ12</td>
<td>0.18</td>
<td>0.90</td>
</tr>
<tr>
<td>Setting 3()</td>
<td>REQ21</td>
<td>0.22</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td>REQ32</td>
<td>0.21</td>
<td>0.13</td>
</tr>
<tr>
<td>Setting 3()</td>
<td>REQ5</td>
<td>0.28</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>REQ6</td>
<td>0.35</td>
<td>0.24</td>
</tr>
</tbody>
</table>

Table 7.2: Verification Run-times of Each Requirement Using SeaHorn

7.2.2 Unit Verification Results

The unit verification run-times are shown in Table 7.3, where all the safety-critical requirements related to every single function were verified concurrently. For function Setting_1() it was noticed in Table 7.2 that five of the assertions froze, and those were discarded from the overall unit verification process at this point.

The Full Implementation row indicates that the four listed functions were veri-
ified concurrently. The real-time for each function is less than 0.4 seconds with compiler optimisations, and less than 4 seconds without optimisations. The full implementation appears higher in both cases of the optimisations; which is a result of calling multiple functions at the same moment.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Real-time[s]</td>
<td>User-time[s]</td>
</tr>
<tr>
<td>Setting_1()</td>
<td>0.31</td>
<td>0.23</td>
</tr>
<tr>
<td>Setting_2()</td>
<td>0.17</td>
<td>0.13</td>
</tr>
<tr>
<td>Setting_3()</td>
<td>0.22</td>
<td>0.16</td>
</tr>
<tr>
<td>Setting_4()</td>
<td>0.23</td>
<td>0.15</td>
</tr>
<tr>
<td>Full Implementation</td>
<td>0.72</td>
<td>0.49</td>
</tr>
</tbody>
</table>

*Table 7.3: Verification Run-times of Each Function Using SeaHorn*

### 7.2.3 Integration Verification Results

The run-times of the integration verification are shown in Table 7.4. Each requirement was reading non-deterministic signals and/or writing them back to the database. The requirements were verified separately with a real-time variation between 2 and 5 seconds. The *Full Implementation* row shows that all the requirements together can be verified at a real-time of 4.62 seconds.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Enable Compiler Opt.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Real-time[s]</td>
</tr>
<tr>
<td>REQ4</td>
<td>3.32</td>
</tr>
<tr>
<td>REQ10</td>
<td>2.55</td>
</tr>
<tr>
<td>REQ15</td>
<td>3.60</td>
</tr>
<tr>
<td>REQ18</td>
<td>2.41</td>
</tr>
<tr>
<td>REQ19</td>
<td>2.78</td>
</tr>
<tr>
<td>REQ16</td>
<td>2.64</td>
</tr>
<tr>
<td>REQ17</td>
<td>3.65</td>
</tr>
<tr>
<td>REQ30</td>
<td>3.85</td>
</tr>
<tr>
<td>REQ21</td>
<td>3.61</td>
</tr>
<tr>
<td>REQ32</td>
<td>3.34</td>
</tr>
<tr>
<td>Full Implementation</td>
<td>4.62</td>
</tr>
</tbody>
</table>

*Table 7.4: Verification Run-times of Requirements with respect to Database Signals*
7.2.4 I/O Verification Results

The requirements that could be identified as containing a valid path to map a relationship between the signal inputs and outputs were gathered in two groups, as was described in Section 6.5.1. The overall real-time from start to finish was 3.42 seconds, the CPU user-time was 0.99 seconds, and the time spent in the kernel was 2.43 seconds. The results are shown in Table 7.5.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Group 1</td>
<td></td>
<td>2.74</td>
<td>1.13</td>
<td>1.61</td>
</tr>
<tr>
<td>Requirements Group 2</td>
<td></td>
<td>2.86</td>
<td>0.98</td>
<td>1.88</td>
</tr>
<tr>
<td>Full Implementation</td>
<td></td>
<td>3.42</td>
<td>0.99</td>
<td>2.43</td>
</tr>
</tbody>
</table>

*Table 7.5: Verification Run-times of Requirements Functionally Substituted*

7.3 Mutation testing and Sanity Check Outcomes

The mutants creation and code modifications were provided by Scania testing department, and the test was conducted on the SeaHorn implementation only. The sanity check was informal and it was the only rapid method available to investigate the validity of the results. It worked primarily by falsifying each safety property in the source-code and then observing the output of SeaHorn.

7.3.1 Mutation Test Results

The mutation test sheet contained ten various alterations of the source-codes. Table 7.6 shows the fault type and the outcome of the detection in the mutant version.

Eight out of ten faults were detected. Faults 1 and 4 are marked as Not Applicable because the resulting mutant will falsify two requirements that are not included in the verification process at this point. Those are related to the safety requirements that had complications with the implementation mentioned earlier.
<table>
<thead>
<tr>
<th>Fault No.</th>
<th>Type</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Change boundary values</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>Change boundary values</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>Change boundary values</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>Change boundary values</td>
<td>N/A</td>
</tr>
<tr>
<td>5</td>
<td>Mix DB signal values</td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td>Mix DB signal values</td>
<td>✓</td>
</tr>
<tr>
<td>7</td>
<td>Mix DB signal values</td>
<td>✓</td>
</tr>
<tr>
<td>8</td>
<td>Change condition values</td>
<td>✓</td>
</tr>
<tr>
<td>9</td>
<td>Change condition values</td>
<td>✓</td>
</tr>
<tr>
<td>10</td>
<td>Change condition values</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 7.6: Mutation Testing Results Using SeaHorn

7.3.2 Requirement Sanity Check

The instant sanity check run-times results are presented in Table 7.7. The times were captured in seconds, both with compiler optimisation and without. The table demonstrates that all of the requirements were falsifiable, and they passed the sanity check test.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Real-time[s]</td>
<td>User-time[s]</td>
</tr>
<tr>
<td>Setting_1()</td>
<td>REQ1</td>
<td>0.23</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>REQ10</td>
<td>0.23</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td>REQ15</td>
<td>0.24</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>REQ13</td>
<td>0.32</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>REQ18</td>
<td>0.25</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>REQ16</td>
<td>0.22</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td>REQ17</td>
<td>0.19</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td>REQ19</td>
<td>0.24</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td>REQ14</td>
<td>0.21</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>REQ20</td>
<td>0.21</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>REQ30</td>
<td>0.20</td>
<td>0.14</td>
</tr>
<tr>
<td>Setting_2()</td>
<td>REQ11</td>
<td>0.16</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>REQ12</td>
<td>0.15</td>
<td>0.07</td>
</tr>
<tr>
<td>Setting_3()</td>
<td>REQ21</td>
<td>0.20</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td>REQ32</td>
<td>0.18</td>
<td>0.12</td>
</tr>
<tr>
<td>Setting_3()</td>
<td>REQ5</td>
<td>0.21</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td>REQ6</td>
<td>0.22</td>
<td>0.15</td>
</tr>
</tbody>
</table>

Table 7.7: Sanity Check Run-times of Each Requirement Using SeaHorn
Chapter 8

Discussion

This chapter discusses a general overview of the empirical results listed in chapter 7. It also presents the comparison between this thesis implementation with related work in [7] and [39].

8.1 Eldarica’s Limitations

The number of requirements Eldarica could verify was limited. The tool is fairly new and under active development. It does not currently support some aspects that were necessary to operate on the steering ECU source-code, such as heap, pointers and arrays.

Eldarica’s SMT-solver, Princess, could verify two requirements for one function. The special characteristics of those two succeeded properties are that the function arguments were not pointers. To make this verification work, the source-code was modified excessively. For example, any reads or writes from arrays were altered with variables that read a non-deterministic value.

The presence of Eldarica’s own parser created some challenges with the verification. It was required to use the preprocessed code in this case study to be able to verify the steering unit. It was a substantial task to verify the code with the macros as the expansion of them were confusing, and those were frequently used in the logical statements.
8.2 SeaHorn as a Verification Tool

The verification outcomes using SeaHorn were positive, especially when the LLVM compiler optimisations were enabled. The verification process was adequate; it could go far beyond unit verification. SeaHorn succeeded with retrieving and transmitting the source-code variables from its original source as signals from the real-time database functions. The source-code required a few modifications in order to continue the verification on an abstract level of the system. The alterations included the following:

1. Declaring global variables to represent the function’s local variables.
2. Declaring extra variables and structs to represent arguments to the functions.
3. All newly declared variables from 1 and 2 to be initialised to a non-deterministic function of a compatible return data type and value.

8.2.1 Compiler Optimisation Effect

The default compiler optimisation level was 3, i.e flag -O3. This flag performs code size and speed optimisations by setting another group of attributes to proper values. For example, it enables flags for inlining functions, basic alias analysis, simplifying the control flow graph and dead code elimination. The output of the optimised code is semantically equivalent to the original one, but the execution time is faster compared to the original code. This was clearly demonstrated in the run-time tables in Chapter 7, where the compiler optimised code executions were significantly faster than the non-optimised ones.

8.2.2 Issues with the Results

There were two requirements that suffered from being verified in SeaHorn. They were expected to be safe, however SeaHorn provided unsafe as a result. There can be many potential reasons for such behaviour for these two requirements, for instance, there might be a problem in SeaHorn, the LLVM compiler optimisation level might have effected the result, or the data dependencies.
It could potentially be a problem in the SeaHorn framework. The indication of the presence of a bug in SeaHorn was that each requirement consisted of two assertions, and no individual assertion reach the error, i.e. safe. On the other hand, when the two assertions were tested at the same time, the result pointed out that the error was reachable, while this should have been a safe state because the two assertions were nothing but a logical conjunction, and if both states were true, the outcome had to be true. This problem occurred only with compiler optimised code, for both requirements 7 and 8, as Figure 8.1 and Figure 8.2 demonstrate.

![Figure 8.1: Requirements Failing when Compiler Optimisation Enabled](image)

![Figure 8.2: Requirements Passing when Compiler Optimisation Disabled](image)

Conceivably, the level of the LLVM optimisation might have been quite high, so it could have eliminated some of the code which was essential for the verification processes to produce a safe state. SeaHorn is a software model checker that proves the correctness of the Horn clauses being generated from the LLVM bit-code, not the original source-code.

The data dependencies could also affect the result. The variables in the code that reflect the requirement variables could have been modified somewhere in the source-code and they were considered by SeaHorn as a possible path to the error state.

The interpretations for these two requirements were most likely correct, and the first-order logical translation or the later simplifications of the semi-formal requirements were not mistaken. The interpretations and simplifications of the original statements were tested again inside the function scope rather than the main har-
ness, where SeaHorn produced a safe state result. Additionally, they were verified as correct without the compiler optimisations. Therefore, the interpretation was correct.

8.3 SeaHorn Comparison to VCC and Frama-C

The steering module had previously been verified using deductive verification programs, namely VCC in [7] and Frama-C in [39]. This section contains multiple comparison criteria that are relevant to the case study, with data from the authors.

8.3.1 Requirements Statistics

The encountered requirements in VCC and Frama-C were in total 10. 5 requirements were verified in a functional modular approach that used the program variables, while the other 5 were implemented with respect to the read and write operations to the database that used the signal names. On the other hand, the implementation with SeaHorn encountered 19 requirements in the functional modular approach, and 10 of those were verifiable with respect to memory implementation, as the Venn diagram shows in Figure 8.3.

Figure 8.3: Total Number of Requirements Verified Using SeaHorn, VCC and Frama-C
8.3.2 Annotations Statistics

The contracts for each function in VCC and Frama-C naturally demanded extra annotations, and these were considerably different from the contracts of the steering ECU. The annotations included a large amount of assigns, valid_read and ghost code. The added ghost code represented the intermediate values that the requirements introduced, and there was a notable redundancy of the contracts throughout the code to be able to verify the top level function. The code overhead in VCC was around 50%, as in 700 lines of code were added to be able to verify 10 requirements. This was also the case with the verification using Frama-C.

The contracts were added in the main function in the SeaHorn implementation accompanied by the single-entry top-level function. Because of the nature of the steering module code and its requirements, it was necessary to add global variables to represent the intermediate variables that the requirement introduced. The code overhead was about 190 lines, around 14% of the code.

8.3.3 Required Persons Workload

After formalising the requirements and understanding the code, the annotation development workload in VCC was roughly estimated to be between 1 to 1.5 person-months [7]. The translation from VCC annotations to Frama-C ACSL required roughly 1 month. The amount of time taken to develop the contracts using SeaHorn was roughly 1 month, and this time does not include the mapping to the code variables and or logical formalisation of the requirements.

8.3.4 Scalability and Speed

The general consensus comparison for this type of case study is the scalability to the code and speed.

VCC could scale to the code, and it could prove the top level entry function, though it required a very long run time, approximately 165 seconds. The verification using Frama-C was interesting because it could not verify the single-entry top-level function. The reason was that it ran out of memory while verifying. It was not
clear whether it can scale to such a code size or it required more memory. Frama-C could potentially have scaled to the code size if the translation from VCC to ACSL annotations was not naive.

SeaHorn could scale to the code, with fast run-times as demonstrated in Table 7.3 and Table 7.4 in Section 7.2.

8.3.5 Mutation Test Result Comparison

The information regarding the mutation test results was available for VCC only, where it succeeded in detecting all the faults in Table 7.6. On the other hand, the time spent catching those faults often took more than 15 seconds. Comparatively, SeaHorn needed less than 2 seconds.

In conclusion, the comparison criteria confirmed that the verification process was efficient in SeaHorn compared to VCC and Frama-C. SeaHorn could verify approximately double the number of requirements verified in VCC and Frama-C, with faster run-times and less code overhead. VCC and Frama-C required slightly similar workload effort to SeaHorn. In the case of VCC and Frama-C workloads, both implementations were done by experienced users with the tools, while in SeaHorn’s case the implementation was done by a first-time user. The mutation test cases were detected by VCC, however it took a long time to detect them compared to SeaHorn.
Chapter 9

Automatic Inference of Code Contracts

This chapter presents the additional goal of the thesis, which is to automatically infer contracts from Horn-based software model checkers that are compatible with deductive verification tools. Contracts indicate the preconditions and postconditions of the functions.

The main reason to accomplish this goal is that deductive verification guarantees higher assurance of the code correctness than model checkers, but model checkers require less effort to formalise the safety requirements. Model checking is also faster compared to deductive verification, as was confirmed by the results in chapter 7.

For this experiment, TriCera was picked as the software model checker. TriCera is the extended Horn encoder of Eldarica, it can model structs and pointers. Frama-C v16 Phosphorus was the used deductive verification tool in this experiment. The annotations of Frama-C are scripted in ACSL format, therefore the inferred precondition and postcondition followed the same format. The automatically generated contracts were fully implemented using Scala, an object-oriented and functional programming language. The output was considered as an extension of TriCera’s output.

The approach was to transition from simple assertions that represented requirements and or safety properties in the C code to ACSL annotations that were compatible with Frama-C. In general, the procedure was a proof of concept; to demonstrate the possibility of automatically inferring annotations from Horn clause solvers. An example of the assertions in C is shown in Listing 11 and the generated contracts of the code in Listing 12.
/*@ contract @*/

int max(int x, int y, int z) {
    int max;
    if (x > y) max = x;
    else max = y;
    if (z > max) max = z;
    return max;
}

int nondet();

int main() {
    int num1 = nondet();
    int num2 = nondet();
    int num3 = nondet();
    int ans = max(num1, num2, num3);
    assert(ans >= num1 && ans >= num2 &&
           ans >= num3);
}

Listing 11: Verification Using TriCera’s Assertions Example

/*@ Function: max */

requires \true ;
ensures \result >>= \old(z) &&
\result >>= \old(y) &&
\result >>= \old(x) ;

int max(int x, int y, int z) {
    int max;
    if (x > y) max = x;
    else max = y;
    if (z > max) max = z;
    return max;
}

Listing 12: ACSL Contracts Adequate for Frama-C

9.1 Fetching Solutions of Horn Clauses

TriCera generated a program model of the input C code, and encoded the model with Horn clauses. Horn clauses were expressed by either SMT-LIB or Prolog, and in this case, Prolog will be the utilised expression for the purpose of inferring contracts in this case study. The generated Horn clauses were abstracted using CEGAR loop technology and refined using Craig interpolation [40] located in Princess, which is the SMT-solver and theorem prover.

In order to automatically generate the precondition and postconditions, it was required to fetch the solution of the Horn clauses from Princess. It was a first-order logic statement scripted in Prolog notation, where each statement declared the state transitions of the functions, accompanied by the compatible number of arity at that transition moment. To generate the solution section, an addition of
the function-preceding comment /*@ contract @*/ is necessary in the C code. This comment was not ignored by TriCera.

In the solution, the notations of _0, _1 ... _n-1 represented the program variables, for arity n. Each one of those was traced back to its original variable in the input C code. The associated variables could be found in the background axioms class. Appendix A contains an example of the solution and background axioms layout.

9.2 Ordering Contracts and Tracing to Variables

The generated solution contained multiple irrelevant logical formulas to preconditions and postconditions. For example, it listed conditions of the functions at different states and main invariants, so it was necessary to extract the relevant statements.

The extracted contracts were not in the appropriate order with respect to the function name, neither syntactically i.e. precondition precedes the postcondition, thus acquiring the correct ACSL contracts structure. To eliminate ambiguity, each function name was kept on top of the produced contracts.

The background axioms contained the genuine code variable name, that outline the arity symbols _0, _1 ... _n-1 in the solution statements, and were used to substitute the arities with the variable names.

9.3 Formatting the Contracts

To create an output compliant with ACSL notation the following lexical adjustments were applied:

1. The logical symbols (∧, |, =) were exchanged with (&&, ||, ==).
2. The literals (true, false) were exchanged with (\true, \false).
3. The built in construct old (value_old) was exchanged with (\old(value)).
4. The keywords (func_pre, func_post) were exchanged with (requires, ensures).
In addition to the previous alterations, parenthesis elimination made the contract more readable. Some logical simplifications were also made to the result. For instance \((-1 \times x + y) = 0\) was converted into \(x = y\).

The alterations were done in two possible ways; using a pretty printer file and regular expressions. The pretty printer file was inspired by an existing code lineariser of the Princess solver; it was greatly utilised to produce most of the needed alterations. Regular expressions were used to fix the construct old, where each old got an id, to refer to the correct variable name, which was replaced with the new syntax of old.

### 9.4 SV-COMP Benchmarks Results

SV-COMP (Competition on Software Verification) is an annual competition for evaluating state-of-the-art fully automated software verifiers based on effectiveness and efficiency [41]. The verification tasks are constructed and collected as a benchmark in [42]. 18 verification tasks were chosen from the repository to test their inferred contracts from TriCera. The contracts were labelled with the function names, which were placed on top of the relevant functions, and verified using Frama-C. The theorem prover that was used in Frama-C was the Alt-Ergo, performing the weakest precondition calculus. In addition to the SV-COMP benchmark, 6 more examples were tested from various made-up functions to expand the results.

Most of the selected SV-COMP tasks were implemented inside the main function, where the verification mission was to prove a safety property rather than specifying the precondition and postcondition, thus there were slight adjustments on the SV-COMP files; transferring the implementation of the main function to separate functions, and strengthening the properties.

In SV-COMP verification tasks, the safety properties were typically defined in first-order logic assertion statements or conditional IF statements. Most of the conditionals statements specify that the assertions should not happen, for example \(\text{IF}(a=10) \text{assert}(0)\). TriCera succeeded in understanding such properties and generated the ACSL contracts.

The contracts were able to verify the selected functions, and this with or without
adding extra manual annotations. For example, for functions with loops, Alt-Ergo provided \textit{Unknown} as an answer. This is because the contracts do not specify the variant (termination element), invariant and assign of the iteration. If those were manually specified by the user, Alt-Ergo will give either \textit{Surely Valid} or \textit{Valid Under Hypothesis} as an answer. The second one means that the local status is valid, however at least one of the dependencies has a consolidated status unknown [19].

The length of generated contracts varied between clear short statements and long ones, and this is due to the way Princess solved the modelled program. The long precondition or postcondition results could be concatenated or simplified manually. For example, for some recursion function, the precondition was the statement:

\[
/*@ \text{requires } (n==5||n==4||n==3||n==2||n==1||n==0); */
\]

Such a statement could be shortened by the users common sense as the following statement shows:

\[
/*@ \text{requires } 0 <= n <= 5; */
\]

The safety properties demanded integer boundary analysis, i.e. overflow and underflow analysis. Integer value checking required an extremely long time in some cases with recursion functions, so instead those values were manually added in the ACSL annotations, whenever the execution of Princess took more than 30 minutes.

In conclusion, the automatically generated contracts successfully verified verification tasks that contained a single function call, multiple functions calls, and functions with recursions. Verification tasks that contained loops required extra manual work related to loop invariants and variants, however the generated contracts were verified.
Chapter 10

Conclusion

The steering electronic control unit was provided by Scania, with a supplementary requirement document containing 33 functional safety properties. 19 out of the 33 requirements were selected for interpretation and formalised in first-order logic predicates. The overall outcome of the verification procedure varied between the two state-of-the-art tools that operated as software-based model checkers; Eldarica and SeaHorn.

SeaHorn verified most of the selected requirements, with an extremely high speed as shown in Chapter 7, due to many significant reasons regarding the code and tool-based reasons. The code is written according to MISRA-C guidelines and does not contain iteration statements; instead, most of the code structure is based on conditional statements like if and switch cases. SeaHorn makes an abstraction of the code, which makes it faster and decidable in the point of view of Spacer; the back-end theorem solver it depends upon. Although Seahorn produced good results, it suffered from some correctness issues regarding two of the requirements.

Eldarica used the preprocessed version of the implementation, and the solver could verify two of the requirements. This is due to the limitations in the availability of the code modelling the heap, pointers and structs at the time.

Although the overall verification result was tool dependent, the results also varied concerning the run-time and outcome based on the compiler optimisation level.

The verification process scaled to the industrial code size, and it also succeeded beyond proving functional correctness. It achieved a higher level of unit integration with a real-time database, while calling only the top level single-entry function. The mutation and sanity check tests were passed, validating the verification outcome.

The related works to this case study in [7] and [39], were implemented on the same steering electronic unit using deductive verification tools VCC and Frama-
C. The difference between the deductive verification versus software-based model checkers was described in section 8.3. Model checkers achieved faster run-times, demanded minimal human workload, and in general the annotation overhead for a number of selected requirements was lower compared to deductive verification tools.

Eldarica was extended to automatically infer adequate contracts in ACSL format, which is also used by the deductive verification tool Frama-C. The experiment was primarily a proof of concept; it was demonstrated that there is a possibility to automatically generate preconditions and postconditions from software model checkers that performs as sufficient annotations in Frama-C, which in turn gives a higher guarantee of the code correctness.

10.1 Future Work

The automatically generated contracts currently contain the preconditions and postconditions of simple functions with recursions and multiple function calls. The experiment was a proof of concept; the implementation lacks the struct and pointer contract generation. The future work could be taking a step further and generating annotations for loop invariants, and the decrease function in order to prove total correctness rather than partial correctness. The inferred contracts are currently printed out in the terminal as a result of Eldarica, but it could be useful to generate the contracts in the correct placement with respect to the code. The final design could also be tested on embedded software.
References


Appendix A

Generating Contract Process

Appendix A is a clarification of the steps taken to automatically generate contracts with Eldarica, to be used in Frama-C. The steps were implemented using Scala. In this appendix, the figures are explanations for the steps mentioned in Chapter 9.

```c
/*@ contract @*/
int GetID(int x) {
    if (x > 0) {
        x--;  
        return GetID(x) + 1;
    } else
        return 0;
}

/*@ contract @*/
unsigned int addition2(unsigned int x) {
    x = x + 2;
    return x;
}

int nondet(void);  

int main() {
    int y = nondet();
    assume(y > 0);
    int ID = GetID(y);
    assert(ID == y);
    int ID1 = addition2(ID);
    assert(ID1 == ID + 2);
}
```

Listing 13: C Program in Eldarica’s Annotation
Listing 13 shows two simple functions that will be used for demonstrating the process. Function \texttt{GetID} returns the same value that was provided as an input recursively, while the input is positive. Function \texttt{addition2} adds 2 to the input value and returns it. The main function can be considered as a harness, where the function will be called with the related assumptions and assertions. The comment \texttt{/*@ contract @/} is essential for generating the solution section, it is not ignored by Eldarica. The function implementation output from Eldarica will be SAFE. Horn clause solutions can be fetched by the command \texttt{./tri file.c -log - arithMode:ilp32}. The solution looks similar to Figure A.1.

```
solution:
GetID_post/2: (((_1 + -1 * _0) = 0) & (_0 >= 0))
GetID1/3: (((_1 + -1 * _2) = 0) & (_2 >= 0))
GetID2/2: (((_1 + -1 * _0) = 0) & (_0 >= 0))
GetID6/3: (((_2 + -1 * _0) = 0) & ((-1 + (_1 + -1 * _0)) = 0)) &
            (((2147483646 + -1 * _0) >= 0) & (_0 >= 0)))
addition2_pre/1: (((2147483647 + -1 * _0) >= 0) & ((-1 + _0) >= 0))
addition21/3: (((_2 + -1 * _0) = 0) & ((2 + (_1 + -1 * _0)) = 0)) &
             (((2147483649 + -1 * _0) >= 0) & ((-3 + _0) >= 0)))
GetID2/2: false
inv_main6/2: (((_1 + -1 * _0) = 0) & ((-1 + _0) >= 0))
GetID4/2: (((_1 = 0) & (_0 = 0))
addition2_post/2: ((((_2 + -1 * -1 * _0) = 0) & ((-1 + _0) >= 0)) &
            (((-4294967294 + _0) >= 0) | ((2147483647 + -1 * _0) >= 0)))
inv_main10/3: ((((_2 + -1 * -1 * _0) = 0) & ((-1 + _0) >= 0)) &
             (((2147483645 + -1 * _1) >= 0) & ((2 + (_1 + -1 * _2)) = 0) |
             ((2147483646 + 1) >= 0)))
inv_main7/2: (((_1 + -1 * _0) = 0) & ((-1 + _0) >= 0))
addition22/2: false
GetID3/2: (((_1 + -1 * _0) = 0) & ((-1 + _0) >= 0))
inv_main9/3: ((((_2 + -1 * _0) = 0) & ((-1 + -1 * _0) = 0)) &
             ((-1 + _0) >= 0))
inv_main2/0: true
addition23/2: (((2 + (_1 + -1 * _0)) = 0) &
             (((2147483649 + -1 * _0) >= 0) & ((-3 + _0) >= 0)))
addition20/2: (((_0 + -1 * _0) = 0) & ((2147483647 + -1 * _0) >= 0) &
             ((-1 + _0) >= 0))
GetID_pre/1: (_0 >= 0)
GetID5/2: ((((-1 + (_1 + -1 * _0)) = 0) & (_0 >= 0)))
```

Figure A.1: Solution Output From Eldarica

67
In Figure A.2 the pre-state and post-state of every function represent the required contracts, that was extracted and left separately.

```
GetID_post/2: (((-1 + -1 * _0) = 0) & (_0 >= 0))
addition2_pre/1: (((2147483647 + -1 * _0) >= 0) & ((-1 + _0) >= 0))
addition2_post/2: ((((2147483647 + -1 * _0) = 0) & ((-1 + _0) >= 0)) &
(((-4294967294 + _0) >= 0) | ((2147483647 + -1 * _0) >= 0)))
GetID_pre/1: (_0 >= 0)
```

**Figure A.2: Extracting Contracts From Solution Output**

The precondition is supposed to precede the postcondition in ACSL, so the output was ordered accordingly. The result of this step should look like Figure A.3.

The name of the function is printed out as well, in order to eliminate any confusion that might occur if there were more than one function to verify.

```
//function: addition2
addition2_pre/1: (((2147483647 + -1 * _0) >= 0) & ((-1 + _0) >= 0))
addition2_post/2: ((((2147483647 + -1 * _0) = 0) & ((-1 + _0) >= 0)) &
(((-4294967294 + _0) >= 0) | ((2147483647 + -1 * _0) >= 0)))

//function: GetID
GetID_pre/1: (_0 >= 0)
GetID_post/2: (((_1 + -1 * _0) = 0) & (_0 >= 0))
```

**Figure A.3: Adequate Ordering of Contracts**

The actual variable names of the arity numbers are shown in Figure A.4. These called the background axioms, and they can be fetched by adding -p to the command.

The genuine variables were substituted with the arity numbers in the solution, to get the complete format of the contract before adjusting it in the pretty printing process.

The contracts are printed in the terminal in a format that is compatible with
ACSL, after entering a process to pretty print them using a lineariser and regular expressions, as shown in Figure A.6.

At this point, the previous contracts can be generated automatically with the -log flag. The contracts will appear in the terminal, thus it is needed to transfer
Frama-C Contracts:
/*@ */
//Function: addition2
  requires 2147483647 >= x && x >= 1 ;
  ensures \result - \old(x) == 2 && \old(x) >= 1
  && (\old(x) >= 4294967294 || 2147483647 >= \old(x)) ;
 */

/*@ */
//Function: GetID
  requires x >= 0 ;
  ensures \result == \old(x) && \old(x) >= 0 ;
 */

Figure A.6: Pretty Printed ACSL Format Output

them to the actual source-code, and try in in Frama-C. Frama-C used plugin is
the Weakest Precondition with Qed and Alt-Ergo solvers. The code in Frama-C is
normalised automatically, and the green circles next to the contracts mean valid.
Figure A.7: Frama-C Output for GetID is Valid

Figure A.8: Frama-C Output for addition2 is Valid